Francisco Cavalheiro Mariani

Project Petalite:

A proposal for a post-quantum Trusted Platform Module

São Paulo, SP

2025

Francisco Cavalheiro Mariani

Project Petalite:

A proposal for a post-quantum Trusted Platform Module

Undergraduate thesis submitted to the Department of Computer Engineering and Digital Systems of the Polytechnic School of the University of São Paulo, in partial fulfillment of the requirements for the degree of Bachelor of Engineering.

University of São Paulo – USP

Polytechnic School

Department of Computer Engineering and Digital Systems (PCS)

Supervisor: Professor Bruno Albertini, Ph.D.

São Paulo, SP

2025

Gerar a ficha catalográfica em https://www.poli.usp.br/bibliotecas/servicos/catalogacao-na-publicacao
Salvar o pdf e incluir na monografia

Acknowledgements

Os agradecimentos principais são direcionados ...

S'io credesse che mia risposta fosse
A persona che mai tornasse al mondo,
Questa fiamma staria senza più scosse.
Ma perciocché giammai di questo fondo
Non tornò vivo alcun, s'i'odo il vero,
Senza tema d'infamia ti rispondo.

Dante Alighieri, Inferno XXVII, 61–66

Abstract

This work designs a Trusted Platform Module (TPM) with dedicated hardware support for post-quantum cryptography; specifically, for the operations in the CRYSTALS-Dilithium digital signature scheme. A pre-existing, permissively-licensed RTL core implementing Dilithium is corrected, refactored, and validated. The overall architecture for the TPM is designed as a RISC-V SoC platform. Both the TPM base specification, and its reference software implementation, are expanded to accomodate these new services. The software implementation specifically is also adapted into a firmware that can be executed in a bare-metal environment. The final design is then implemented in an FPGA, where final performance metrics are taken.

Keywords: Post quantum cryptography. Lattice cryptography. Crystals suite. Dilithium. Trusted Platform Module. Hardware design. FPGA.

List of Figures

List of Tables

List of abbreviations and acronyms

RTL	Register Transfer Level
CPU	Central Processing Unit
ISA	Instruction Set Architecture
SoC	System on a Chip
FPGA	Field Programmable Gate Array
TPM	Trusted Platform Module
RoT	Root of Trust
CRHF	Collision Resistant Hash Function
ZKPoK	Zero Knowledge Proof of Knowledge
RNG	Random Number Generator
TRNG	True Random Number Generator
PRNG	Pseudo Random Number Generator
PUF	Physical Unclonable Function
PQC	Post-Quantum Cryptography
SVP	Shortest Vector Problem
SIS	Short Integer Solution
LWE	Learning With Errors
NTT	Number Theoretic Transform

Contents

1 Introduction

This document provides the current specification for the Trabalho de Conclusão de Curso (TCC), to be completed in the academic year of 2025, as part of the Computer Engineering Bachelor's Degree at the Polytechnic School of the University of São Paulo.

1.1 Context

Modern public key cryptography has traditionally been built upon problems from the field of Number Theory; that is, for a given cryptographic primitive, there exists a security reduction proving that breaking that primitive is as computationally hard as solving some particular Number Theory problem. For example, there are systems based on the hardness of factoring large integers (such as RSA (IMAM et al., 2021)), and others based on the difficulty of calculating discrete logarithms over finite fields (such as the Diffie-Hellman key exchange (MISHRA; KAR, 2017)).

More recently, however, methods such as Shor's algorithm have been discovered, which are capable of solving these problems efficiently (that is, in polynomial time), given a quantum computer with sufficient *qubits* (SHOR, 1997). Although there currently is no quantum computer mature enough to execute Shor's algorithm for the magnitudes commonly employed by the aforementioned systems, there have been implementations for simpler cases, showing its overall viability (VANDERSYPEN et al., 2001; MARTíN-LóPEZ et al., 2012).

This fact has sparked a great interest in developing cryptographic systems that are resistant to attacks made by quantum computers: the central aim of the field of *post-quantum cryptography* (PQC). One specific type of PQC that has garnered a lot of attention is lattice-based cryptography (PRADHAN; RAKSHIT; DATTA, 2019), which is based on problems such as the Shortest Vector Problem (SVP) and the Learning With Errors (LWE) problem. At the moment, there is no known algorithm (quantum or otherwise) capable of solving these lattice problems efficiently, making them strong candidates for the foundations of PQC systems.

In 2016, the american National Institute of Standards and Technology (NIST) launched the Post Quantum Cryptography Standardization competition, in which proposals for quantum-safe cryptosystems designed for key encapsulation and digital signing were submitted for evaluation (National Institute of Standards and Technology, 2016). After three rounds of the competition, the original 69 complete proposals were narrowed down to 7 finalists and 8 'alternative candidates'. Of these 15 proposals, roughly half (7) of them

were lattice-based. In 2024, NIST officialized some of these finalists as standards, including CRYSTALS-Kyber for key encapsulation (now formally designated as ML-KEM), and CRYSTALS-Dilithium for digital signatures (formally designated as ML-DSA).

As the need for quantum-safe security systems becomes ever more apparent, computing platforms must start implementing these schemes for a variety of goals currently performed by traditional cryptography, such as data ciphering, Transport Layer Security (TLS) communication, and secure booting.

Both for latency and for security reasons, the modern architectural paradigm is to offload such cryptographic tasks to security-focused co-processors. These *cryptoprocessors* are a common component of modern computers ranging from embedded systems to high-performance cloud-computing platforms. One of the most prevalent examples of cryptoprocessors are Trusted Platform Modules (TPMs), which serve as a foundational Root of Trust (RoT) for the host platform, and are most often soldered onto its mainboard.

A TPM provides both general cryptographic services, as well as the secure management of cryptographic keys in a tamper-resistant environment. Given the specific purpose of a TPM, many of its tasks are commonly accelerated at the hardware level: commerical TPMs typically include hardware cores for performing common cryptographic algorithms, such as AES, SHA, ECC, and RSA. Examples of that can be seen in STMicroelectronics (2021), Microchip Technology Inc. (2018), and Infineon Technologies AG (2021).

At its present iteration, the official TPM specification does not include any PQC capabilities (Trusted Computing Group, 2025c). However, while this project was in development, the entity responsible for the specification — the Trusted Computing Group, or TCG — issued a brief report stating that they "are working on updating [their] specifications to prepare for the post-quantum era" and "are dependent on the [...] algorithms and parameter sets published by key agencies such as NIST" (Trusted Computing Group, 2025h).

1.2 Goal

The goal of this work is to develop an open-source, post-quantum Trusted Platform Module, with the inclusion of dedicated hardware support for lattice-based cryptographic operations, namely the ones defined by the CRYSTALS-Dilithium digital signature scheme.

This device can then be connected to a host platform to provide the services expected of a TPM — data signing, state attestation, secure booting, etc — using either traditional or post-quantum cryptography.

The TPM is then implemented in an FPGA board, in order to validate its functionalities and overall correctness.

1.3 Motivation

As mentioned in Section 1.1, although Trusted Platform Modules form the security backbone of modern computing platforms, they are (at the moment) exclusively based on traditional cryptography, and are therefore vulnerable to attacks made by quantum computers. Developing a post-quantum TPM based on lattice cryptography provides an extra security guarantee, as long as lattice problems retain their computational intractability.

Moreover, by using dedicated hardware to accelerate these operations, both computational latency — a botttleneck for lattice schemes, when compared to the traditional alternatives — and the hypothetical cryptographic attack surface are decreased.

In addition, although quantum computers are still incipient and not widely available at the moment, there is still a problem related to 'store now, decrypt later' strategies; that is, the malicious harvesting of present data so that, when developed, a sufficiently strong quantum computer can break the encryption used. In this context, a post-quantum TPM presents a way of future-proofing current computer systems, and facilitating future transitions.

Finally, by making it fully open-source, the proposed TPM provides a solution that can be used by anyone free of charge, and also makes itself available to be analyzed and expanded upon by other members of the security community.

1.4 Related works

In recent years, a plethora of work has been conducted towards implementing hardware accelerated lattice-based cryptography in different contexts, ranging from designs with small footprints and high energy efficiency aimed at embedded applications (BOS; RENES; SPRENKELS, 2022; GUPTA et al., 2022), to high-performance designs that can be used more generally (ZHAO et al., 2021; BECKWITH; NGUYEN; GAJ, 2024).

For this work, as will be described in **??**, designs concerning Dilithium implementations in System on a Chip (SoC) platforms are especially interesting. The most relevant works accelerating Dilithium for a SoC platform are here summarized:

- Wang et al. (2024) implements both Dilithium and Kyber schemes on a FPGA SoC by following a co-design approach, that accelerates some operations on hardware modules, and offloads others (such as hashing and sampling polynomials) to a RISC-V multi-core hard processor. Although this implementation is faster than pure software approaches, according to the study, it still presents significant worse performance compared to more hardware intensive approaches.

- Banerjee, Ukyab and Chandrakasan (2019) was one of the first comprehensive attempts at building a SoC core to execute Dilithium using hardware acceleration. Due to its earliness, however, it implemented the (now obsolete) NIST Round 2 specification of the scheme, and also lacks hardware optimization strategies later discovered by other studies. Moreover, the study specifically targets ASICs for the core developed.
- Wang et al. (2022), also by the same authors as Wang et al. (2024), proposes a complete hardware FPGA implementation of Dilithium as a SoC core, that communicates with the hard processor through a memory-mapped interface.

Furthermore, there are also works that utilize the Dilithium cores developed for providing security features associated with hardware security modules:

- Gupta, Jati and Chattopadhyay (2023) proposes an architecture for an SoC that utilizes a Dilithium hardware core, and an additional Code Authentication Unit (CAU), to provide secure boot features for a RISC-V soft processor. This architecture is functionally closer to that of a hardware security module, since it provides a more complex feature than that of simply signing or verifying signatures.
- Fiolhais and Sousa (2023) details the high-level architecture and the computational requirements of a quantum-resistant trusted platform module (TPM), another type of cryptoprocessor, which can be used to provide cryptographic services to a host platform. However, the implementation is done using a software TPM; that is, avoiding any kind of hardware design for the functionalities involved.

1.5 Organization

This manuscript is organized as follows. The overall theoretical background related to this work is presented in Chapter 2, along with recent developments in the field of PQC. Chapter 3 details the methodology of the work undertaken. The functional and non functional requisites for the cryptoprocessor designed are specified in Chapter 4. Finally, **??** motivates the architectural framework used for the project.

2 Theoretical Background

2.1 Cryptography

The field of cryptography concerns itself with the secure storage and exchange of information between parties (i.e. *legitimate users*), in spite of the presence of others (i.e *adversaries*) who may attempt to gain unauthorized access to, or tamper with, that information. This goal is typically divided into more specific security services:

- **Confidentiality**: information should only be accessible to legitimate users;
- **Integrity**: information should not be able to be tampered with in a manner that is imperceptible to the legitimate users;
- **Authentication**: legitimate users should be able to provide and verify proofs of each other's identities, while adversaries should not be able to replicate such proofs;
- **Non-repudiation**: legitimate users should not be able to deny having sent, received, or possessing a message.

In order not to arbitrarily assume any specific strategy that may be employed by the attacker, proper cryptographic schemes must be based on mathematical guarantees for these security requirements, instead of ad-hoc approaches (GOLDREICH, 1999). For example, schemes that rely on the attacker having no familiarity with them — what is often called security by obscurity — are generally easily broken.

Here, one of the oldest known cryptographic schemes serves as an educational example. Caesar's Cipher is a scheme that encrypts (or *ciphers*) the original message (the *plaintext*) into an unreadable format (the *ciphertext*) by substituting each letter by another that is shifted N positions on the alphabet.

The central idea is that the specific N used (the *secret key*) should only be known by legitimate users, thus providing confidentiality even if the attacker has access to the ciphertext. For $N = 3$, the message `HELLO, WORLD` becomes:

Plaintext:	H	E	L	L	O	,	W	O	R	L	D
Ciphertext:	K	H	O	O	R	,	Z	R	U	O	G

However, if the attacker knows that the ciphertext was generated using Caesar's Cipher (i.e. there is no more obscurity), they may discover the plaintext by simply trying values of N on the ciphertext until one produces readable output.

This example also serves to illustrate the importance of having the secret key be an element obtained from a very large set (the *keyspace*): since $N \in [0, 26)$, the attacker can feasibly break the scheme by exhaustively computing (*brute-forcing*) the output for all possible elements of the set.

Suppose now that N could be uniformly drawn from a larger set $N \in [0, 2^{64})$ that all produce different ciphertexts. Brute-forcing the scheme by searching the keyspace would then require a much larger number of computations. Assuming that the attacker tries sequential values of N (which is the optimal strategy for the attacker in this case), they will on average have to perform $\frac{N}{2} = 2^{63}$ computations until they find the correct value for N. If each computation takes time on the order of one millisecond, discovering N would take, on average, almost three hundred million years. That is, even if the scheme is not *unbreakable* per se, it is computationally unfeasible to do so; a characteristic which is common to many modern cryptographic schemes.

The worst-case difficulty of breaking a given cryptographic scheme through brute-forcing can be generalized by the concept of a *security factor*, which is a measure of how secure a system is in units of bits. If an attacker must perform p computations to break the scheme in the worst-case scenario with non negligible probability, that system has $log_2 p$ bits of security.

As such, the only reasonable assumption to make about the attacker is concerning their computational abilities: a secure cryptographic scheme must be one that legitimate users can efficiently perform the necessary computations, but attackers looking to violate its security features cannot do it feasibly.

The notions of *efficiency* and *feasibility* — used extensively throughout this work — are formalized in terms of time complexity with respect to the security parameter n: the time required for legitimate computations is upper bounded by the class of functions $T_1(n)$, while adversarial efforts must exceed a time bound defined by the class $T_2(n)$, where $T_2(n) \geq T_1(n)$ (GOLDREICH, 1999). In traditional modern cryptography, $T_2(n)$ is usually chosen to be the class of polynomial functions on n.

Finally, for almost all cryptographic schemes, there is no formal proof stating that they are impossible to be feasibly broken; there is only the empirical observation that a method for it has not yet been found, and the assumption that finding it is at least as hard as some well-studied, hard (NP) mathematical problem.

2.2 Cryptographic constructions

In this section, a brief introduction to common modern cryptographic constructions is given, along with practical examples.

2.2.1 Cryptographic primitives

Cryptographic primitives are fundamental building blocks used in the construction of more complex cryptographic schemes. One of the most important cryptographic primitive are *one-way* functions: a function $f(\cdot)$ is called one-way if the image $f(x)$ is efficiently computed given x, but the preimage x is unfeasible to compute given only $f(x)$ (GOLDREICH, 1999).

Moreover, a special subset of one-way functions are *trap-door* functions, for which x can be efficiently computed given $f(x)$ and an additional trap-door information. Both these concepts are foundational to asymmetric key schemes, as will be shown later.

Another important cryptographic primitive are Collision Resistant Hash Functions (CRHFs), which extend the notion of mathematical hash functions to include additional cryptographic properties. A CRHF is defined as a function $h : \{0,1\}^* \to \{0,1\}^n$ with fixed n, such that:

1. h is a one-way function (*preimage resistance*);
2. it is unfeasible to, given x, find $x' \neq x$ with $h(x') = h(x)$ (*second preimage resistance*);
3. it is unfeasible to find a pair (x, x') with $h(x) = h(x')$ (*collision resistance*).

CRHFs are useful for some reasons. In particular, they can provide integrity: if the *digest* $h(x)$ is known in advance and assumed to be trusted, an attacker cannot feasibly modify a message x into x' such that $h(x') = h(x)$. In other words, a legitimate user can verify message integrity by computing its hash and comparing it to the known value.

One of the most well-known and widely-used families of CRHFs is the Secure Hash Algorithm 3 (SHA-3) set of functions, standardized by NIST in 2015 (National Institute of Standards and Technology, 2015), and part of a larger family of cryptographic primitives called Keccak (BERTONI et al., 2011).

There are several different functions in the Keccak family. Most notably, SHA3-256 and SHA3-512 are CRHFs with digests of sizes 256 and 512 bits (respectively), while SHAKE128 and SHAKE256 are Extendable Output Functions (XOFs) that produce arbitrarily long digests; that is, $h : \{0,1\}^* \to \{0,1\}^*$.

Strictly speaking, XOFs are different kind of cryptographic primitive than CRHFs, since these are not defined for arbitrarily long digests. Nevertheless, one can turn an XOF into a CRHF by simply fixing its output to a size n.

2.2.2 Random Number Generators

The process of generating *truly* random numbers is of critical importance in many fields of computer science, not least of which cryptography. For example, if cryptographic keys are not composed of random values, an attacker can exploit known patterns to reduce the subset of the keyspace over which they must search. Cryptographic schemes therefore rely on Random Number Generators (RNGs) capable of producing these random values.

Computers — and, more generally, Turing Machines — are deterministic in nature and cannot produce truly random values by themselves. They instead depend on an external *source of entropy*, itself assumed to be truly random, and that acts as a True Random Number Generator (TRNG) when sampled.

The usage of an external source of entropy merely shifts the domain of the original problem, since very few physical systems are truly random, and even fewer can also be reliably sampled. That said, a *good* TRNG should (I) have high enough entropy to be statistically indistinguishable from true randomness, and (II) rely on a physical system that is difficult for an adversary to observe or influence. Common sources of entropy include electronic noise, system events such as process timing variations, or user inputs such as mouse movements.

In order to maintain statistical randomness, most TRNGs have low sampling rates. One way of increasing the throughput of a platform's RNG is to supply the output of the TRNG into a Pseudo Random Number Generators (PRNG). These are constructs which, given an input, produce a deterministic, but uniformly random, output. The advantage of such coupling is that PRNGs are generally designed to have a higher sampling rates than TRNGs. Thus, the TRNG provides an initial *seed* over which the PRNG will continuously yield new random values, until it samples the TRNG again for another seed value.

Another use for the CRHFs and XOFs mentioned in Section 2.2.1 is that they can be used as PRNGs since, while not truly random, their pseudo-randomness can be computationally undistinguishable from such.

2.2.3 Asymmetric key schemes

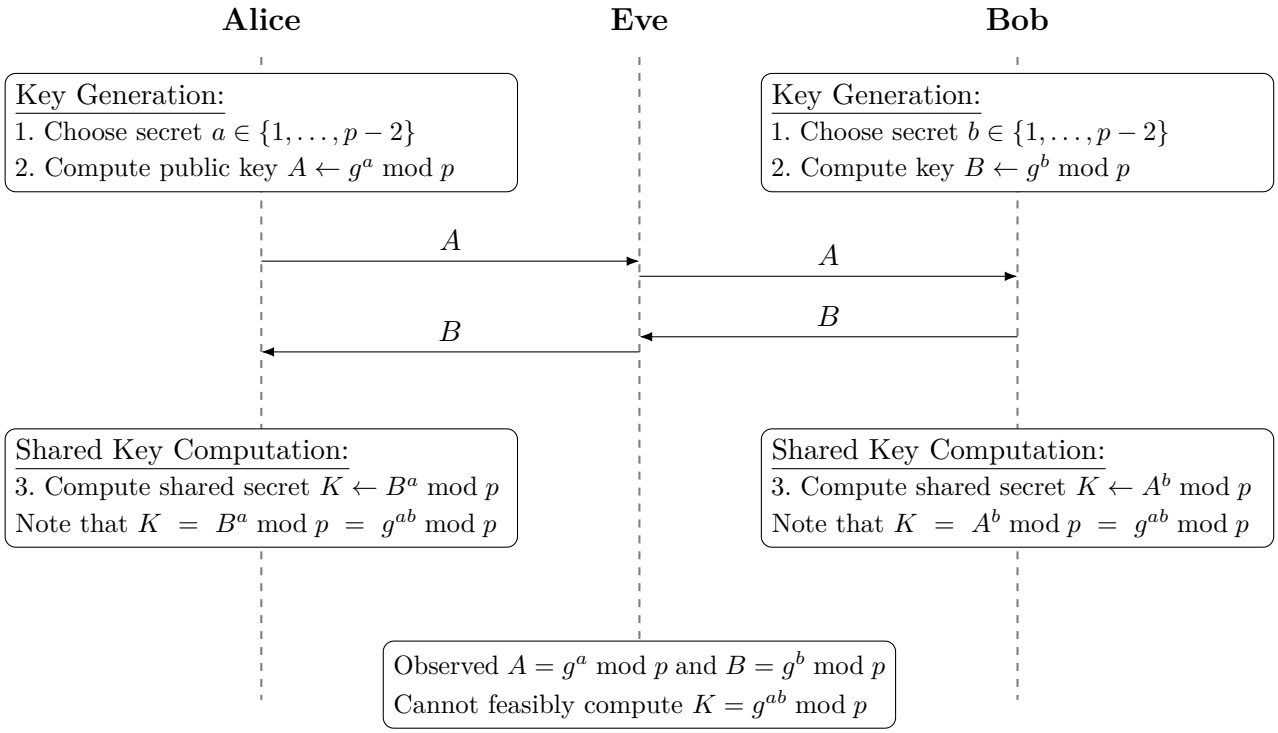
The first cryptographic schemes that used secret keys to provide security features were known as symmetric key schemes, since they used one single key shared among all legitimate users. One limitation of such schemes is that its security guarantees are derivative. That is, it assumes the existence of an already secure (often physical) channel through which the secret key can be shared among legitimate users; otherwise, an attacker could *eavesdrop* on the channel and directly gain access to the key. Sharing the key among large numbers of users then presented a logistical problem.

Such limitations were eventually solved by the introduction of asymmetric (or

public key) cryptography by Diffie and Hellman (1976). In these schemes, each user has a pair of keys, one of which is *secret*, and the other of which is *public* and can be sent over insecure channels. A legitimate user can then securely share information by using their secret key and other users' public keys; an arrangement which is made possible by means of either one-way or trap-door functions.

In the case of the key agreement proposed by Diffie and Hellman (1976), the one-way function is the discrete logarithm problem: given a large prime modulo p and a base g that generates $\mathbb{Z}_p^*$, choosing x and computing $y \equiv g^x \mod p$ is efficient, but recovering x from y is unfeasible. As shown in Figure 1, this allows legitimate users Alice and Bob to agree on a new shared secret key K over an insecure channel, without it being revealed to passive attacker Eve.

Figure 1 – Diffie-Hellman Key Agreement: exchanges between Alice, Bob, and Eve



Source: created by the author

One of the main uses of public key schemes is for providing confidentiality. The core idea is that of a trapdoor function: after the sender ciphers a message using the public key of the receiver, deciphering it should be unfeasible, unless in possession of the private key of the receiver (which corresponds to the trapdoor information).

The first public key encryption scheme was the one proposed by Rivest, Shamir and Adleman (1978). As can be seen in Algorithm 1, although both encryption and decryption steps perform a similar exponentiation operation to Diffie and Hellman (1976),

the underlying mathematical problem in this case is the difficulty of factorizing large numbers into prime factors. That is, if an attacker could feasibly factor the public parameter n into its prime factors, finding (p, q) and then recomputing (e, d) also becomes feasible. Thus, the attacker would have access to the private key, and the scheme would be broken.

Algorithm 1 – RSA Protocol

Require: large distinct primes p, q

Key generation:

1: Compute modulus $n \leftarrow pq$
2: Compute Euler's totient $\phi(n) \leftarrow (p-1)(q-1)$
3: Choose exponent e such that $1 < e < \phi(n)$ and $\gcd(e, \phi(n)) = 1$
4: Compute exponent $d \leftarrow e^{-1} \bmod \phi(n)$

Result:
Public key: (n, e)
Private key: (n, d)

Encryption:

Require: Message $m \in \mathbb{Z}_n$, private key (n, d)
5: Compute ciphertext $c \leftarrow m^e \bmod n$
6: Send c to recipient

Decryption:

Require: Ciphertext c, public key (n, d)
7: Compute plaintext $m \leftarrow c^d \bmod n$

Result: Recipient recovers original message m

Source: created by the author

2.2.4 Digital signature schemes

Another widely used type of asymmetrical cryptography are digital signature schemes. Such schemes allow for a legitimate user (i.e. the *signer*) to *sign* a message, and other users (i.e. *verifiers*) can then check the validity of the signature, thus providing the communication with integrity, authentication, and non-repudiation. For these schemes, it should be unfeasible for an attacker to forge a valid signature for a new message, and for them to recover the private key after observing valid signatures.

Generically, signature schemes are composed of three steps:

$$\begin{aligned} \textbf{Key Generation:} \quad & (\text{pk}, \text{sk}) \leftarrow \texttt{KeyGen}(1^n) \\ \textbf{Signing:} \quad & \sigma \leftarrow \texttt{Sign}(\text{sk}, m) \\ \textbf{Verification:} \quad & \texttt{Verify}(\text{pk}, m, \sigma) \in \{\text{accept}, \text{reject}\} \end{aligned}$$

The **Key Generation** step, performed by the signer, produces the key pair (pk, sk) (respectively, public key and private key), according to the security parameter n. The **Signing** step, also performed by the signer, takes sk and the message m as inputs, and outputs a unique signature σ. The verifier can then check the validity of the signature by performing the **Verification** step, which either accepts the signature or rejects it. Each step performs a corresponding function — `KeyGen()`, `Sign()`, and `Verify()` — whose implementation is dependent on the specific signature scheme.

Asymmetric encryption schemes can be adapted to perform signatures; for example, Rivest, Shamir and Adleman (1978) already presents a way of transforming RSA into a digital signature scheme. Digital signature schemes are typically based on the same mathematical hard problems as asymmetric encryption, such as the discrete logarithm problem and the integer factorization problem.

2.2.5 Zero Knowledge Proofs of Knowledge

A cryptographic construction that is related and yet distinct to signature schemes are Zero Knowledge Proofs of Knowledge (ZKPoKs). In a ZKPoK, a party (the *prover*) convinces another (the *verifier*) that they have knowledge of some secret information, without revealing anything about such secret.

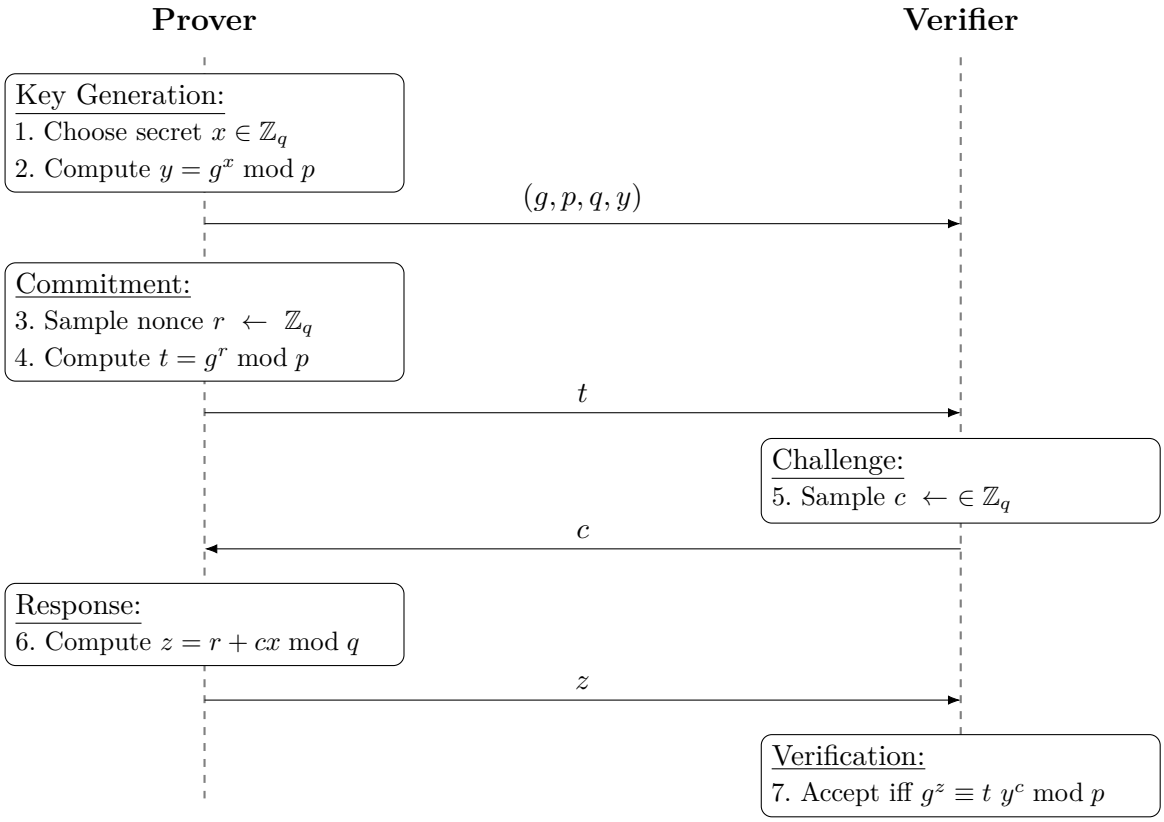
These schemes are usually based on an interactive structure called a Σ-protocol, which is composed of three steps: after generating the key pair, the prover *commits* to the proof by presenting a value computed from a random sample, the verifier then *challenges* him by presenting another random sample, and finally the prover *responds* with a value that is dependent on both previous samples, and also the secret key. The protocol then ends with the verifier ascertaining the validity of the computed value.

One such construction is Schnorr's identification scheme (SCHNORR, 1990), described in Figure 2, and also based on the discrete logarithm problem. In this scheme, given a large prime modulo p and a base g that generates $\mathbb{Z}_p^*$, the prover convinces the verifier of their knowledge of x without leaking any information about it.

Notice that the value r serves the purpose of *masking* the secret x, to avoid information leakage. If there was no r, or more generally if it was reused in multiple communication sessions, an attacker in the role of the verifier could observe $z_1 = r + c_1 x \mod q$ and $z_2 = r + c_2 x \mod q$, and therefore recover the secret by computing $x = \frac{z_1 - z_2}{c_1 - c_2} \mod q$.

Both signature schemes and ZKPoK schemes are based on a user having knowledge of some private secret. Signature schemes can then be seen as a non-interative version of a ZKPoK which is also dependent on a given message. Indeed, the Fiat-Shamir transform takes a ZKPoK based on a Σ-protocol, and transforms it into a signature scheme (FIAT; SHAMIR, 1987) using the CRHF digest of the message. After the transform, all steps of the protocol are performed by the prover alone, except for the verification step, which is still performed by the verifier.

Figure 2 – Schnorr identification scheme: exchanges between prover and verifier



Source: created by the author

2.3 Quantum vulnerabilities

The development of quantum computing over the the last decades has also brought forth proposals for quantum algorithms, many of which can be used for cryptographic applications. In general, these algorithms use the intrinsic advantages of quantum platforms to speed-up performance in ways not possible for classical computers.

Most famously, the algorithm proposed by Shor (1997), can be used to solve the integer factorization problem in subpolynomial $O(\ log(n)^3\ )$ time, as opposed to the best known classical algorithms which take subexponential $O(\ exp(\ log(n)^{\frac{1}{3}}\ log(log(n))^{\frac{2}{3}}\ )\ )$ time (MONTANARO, 2016). It does so by leveraging quantum superposition for computing

multiple values of an expression at the same time, and a quantum version of the Fourier Transform — aptly called the Quantum Fourier Transform — to extract the relevant value.

More generally, Shor's algorithm can be used to efficiently solve a class of mathematical problems called the hidden subgroup problem, of which both the integer factorization problem and the discrete logarithm problem are special cases (MONTANARO, 2016). As such, the cryptographic schemes that were introduced in Section 2.2, and which are based on these problems from the field of Number Theory, become computationally insecure.

Although Shor's algorithm still requires a number of quantum gates which currently prohibits it from being applied to break modern cryptographic schemes, its theoretical viability is well-founded. Therefore, *post-quantum* cryptographic schemes are being built using problems for which no algorithms — quantum or otherwise — have thus been discovered to feasibly solve. Many such proposals are based on geometric lattice constructions, which are introduced in Section 2.4.

2.4 Lattices

The mathematical study of lattices began in the 18th century, with contributions made by names such as Gauss, Lagrange, and latter Minkowski. The field quickly found theoretical applications in Number Theory problems, but also practical applications such as in crystallography, exemplified by the Bravais lattices.

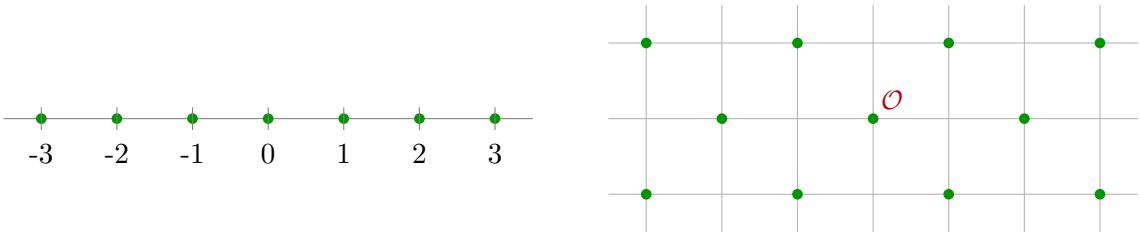
Fundamentally, a lattice is a discrete subgroup of an additive group, usually $(\mathbb{Z}^n, +)$ or $(\mathbb{R}^n, +)$. Geometrically, it can be thought of as a set of points that produce a periodic structure in space (see Figure 3). Similarly to vector spaces, a lattice can be described by a *basis*. Specifically, a lattice Λ in $\mathbb{Z}^n$ can be described by the basis $\mathbf{B} \in \mathbb{Z}^{n \times n}$:

$$\Lambda = \mathcal{L}(\mathbf{B}) = \{v \in \mathbb{Z}^n \ : \ \exists z \in \mathbb{Z}^n \ \ s.t. \ \ \mathbf{B}z = v\} \tag{2.1}$$

A lattice Λ in $\mathbb{R}^n$ could be analogously defined by using a basis $\mathbf{B} \in \mathbb{R}^{n \times n}$.

Despite its apparent similarity — and even simplicity — when compared to vector spaces, the very definition of lattices raises questions that would not exist in its continuous counterpart. For example, a widely studied class of problems for lattices relates to the complexity of finding the shortest vector (Shortest Vector Problem), for a given lattice, up to a certain approximation factor γ.

Figure 3 – Examples of lattices (in green) in the coordinate spaces $\mathbb{Z}$ (left) and $\mathbb{Z}^2$ (right).



Source: created by the author

The previously given definition of lattices references the $\mathbb{Z}^n$ and $\mathbb{R}^n$ additive groups. However, it is possible to define lattices in other groups, such as ones with modulo q arithmetic $\mathbb{Z}_q^n = \mathbb{Z}^n/q\mathbb{Z}^n$, as described in Equation 2.2.

$$\Lambda = \mathcal{L}_q^{\perp}(\mathbf{A}) = \{v \in \mathbb{Z}^n \ : \ \mathbf{A}v \equiv 0 \pmod{q}\} \tag{2.2}$$

2.5 Lattice problems

In this section, a quick overview of the most common lattice problems is given, together with a brief history of lattice-based cryptography.

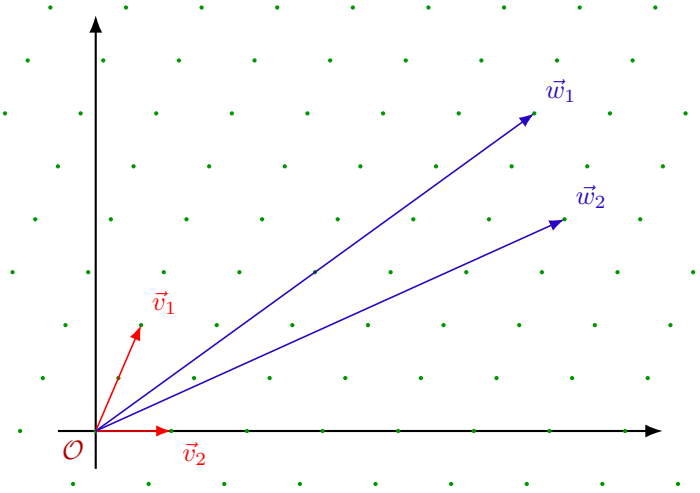
2.5.1 Shortest Vector Problem

The Shortest Vector Problem (SVP) asks to find the shortest non-zero vector in a lattice, using a given norm (usually l_1 or l_∞). The length of this vector is sometimes called the *first successive minimum*, denoted by λ_1. Intuitively, the hardness of this problem is dependent on the basis given for defining the lattice: a basis composed of large, highly dependent vectors is worse for finding the shortest vector when compared to another (equivalent) basis composed of shorter, nearly orthogonal vectors. See Figure 4.

Early algorithms for solving this problem were based on *enumeration*: simply doing an exhaustive search over the integer linear combinations of basis vectors. The best such algorithms perform in $2^{O(n)}$ time and space (MICCIANCIO; WALTER, 2014).

A relaxed version of this problem, called SVPγ, asks to find a vector of length $\gamma \cdot \lambda_1$, for $\gamma \geq 1$. Algorithms for solving this problem usually compute a new (*reduced*) basis in which finding the short vector proves easier. The most common such algorithms are the Lagrange-Gauss algorithm for two dimensions, or the LLL (LENSTRA; LENSTRA; LáSZLó, 1982) and BKZ algorithms for n dimensions, which all find approximations with γ exponential in n (GALBRAITH, 2012). That means it is possible to choose a small enough γ such that no known algorithm can feasibly solve them.

Figure 4 – Examples of "good" (v_1, v_2) and "bad" (w_1, w_2) choices of basis.



Source: created by the author

2.5.2 Short Integer Solution

The Short Integer Solution (SIS) problem asks to find a vector v belonging to a q-ary lattice (as in Equation 2.2), with the added constraint that every element v_i of the vector must be in the range $[-\beta, \beta], \beta \in \mathbb{Z}$. Intuitively, the problem gets easier for larger values of β. The algorithms used for solving this problem also involve finding a short vector in the lattice, making it closely related to the SVP and its variants (LYUBASHEVSKY, 2024).

The SIS problem was first proposed by Ajtai (1996), which also described a family of one-way functions based on its hardness, thus introducing the first lattice-based cryptographic primitive. Interestingly, the work also showed that solving the average instance of the SIS problem is at least as hard as solving *any* given instance of the SVPγ problem, for $\gamma = n^c$ and $c > 1$. This property, known as *worst case hardness*, provides a stronger security guarantee than *average case hardness*, and is a common feature of lattice-based cryptographic schemes.

These findings were then extended by Goldreich, Goldwasser and Halevi (1996) to build lattice-based CRHFs, and also by Ajtai and Dwork (1997) to build the first lattice-based asymmetric cryptographic scheme. These works, although ground-breaking from a theoretical perspective, still proved too inefficient for practical cryptographic applications (LYUBASHEVSKY; PEIKERT; REGEV, 2012).

2.5.3 Learning with Errors

The Learning With Errors (LWE) problem is a more recent problem, introduced by Regev (2009). The problem asks to recover a secret $\mathbf{s} \in \mathbb{Z}_q^n$, from a series of random linear

equations on $\mathbf{s}$ that have been "corrupted" by an error term $\mathbf{e} \in \mathbb{Z}_q^n$; that is, $\mathbf{t} = \mathbf{A}\mathbf{s} + \mathbf{e}$. The error term must be drawn from one of a small number of well-studied distributions, such as the uniform, binomial, or rounded Gaussian (i.e. continuous Gaussian rounded to the nearest integer) distributions.

Interestingly, while solving for $\mathbf{s}$ when there is no error term is incredibly simple (one can use well-known polynomial-time tools such as Gaussian elimination), introducing $\mathbf{e}$ makes the problem significantly more difficult. Using Gaussian elimination, for example, would amplify the error at every step of the algorithm, resulting in loss of information. Similarly to the other lattice problems, all currently known algorithms for solving LWE run in exponential time (REGEV, 2010).

2.6 Lattices over polynomial rings

As mentioned in Section 2.5, a limitation of the first lattice-based cryptographic constructions were its practical inefficiency, both related to their computational complexity, and to their key and ciphertext sizes. These obstacles were then solved by considering constructions using lattices with additional algebraic structure, such as the one given by polynomial rings.

Consider $(\mathbb{Z}_q[x],\ +,\ \cdot)$ as the ring given by all polynomials on x that have coefficients in $\mathbb{Z}_q$, and take $f(x) \in \mathbb{Z}[x]$ as a cyclotomic polynomial of degree n. The quotient ring $\mathcal{R}_{q,f} = \mathbb{Z}[x]\ /\ f(x)$ then represents all integer polynomials of degree less than n, with coefficients modulo q. Finally, considering that $\mathcal{R}_{q,f}$ and $\mathbb{Z}_q^n$ are isomorphic using, for example, the embedding ϕ:

$$\phi : \mathcal{R}_{q,f} \to \mathbb{Z}_q^n, \quad \phi\left(\sum_{i=0}^{n-1} a_i x^i\right) = (a_0, a_1, \ldots, a_{n-1}) \tag{2.3}$$

This then means that the definition of lattices given in Section 2.4 can be made even more general by, instead of the finite group $(\mathbb{Z}_q^n, +)$, taking the ring $(\mathcal{R}_{q,f},\ +,\ \cdot)$, using the usual polynomial addition and multiplication with modular reductions.

Moreover, for a certain choice of lattices called *ideal* lattices — those that correspond to ideals in $\mathcal{R}_{q,f}$, for $f(x) = x^n + 1$, and n as a power of 2 — extensions of the SIS and LWE (appropriately named Ring-SIS and Ring-LWE) have been shown to be similarly hard and provide similarly strong security guarantees for cryptographic schemes.

The cryptographic advantages of using such algebraic structure is two-fold. Firstly, while constructions based on standard lattices typically rely on matrix-vector multiplications with $O(n^2)$ computational complexity, ones based on ideal lattices can leverage methods such as the Number Theoretic Transform (NTT) to accelerate polynomial multiplication and bound its complexity to $O(n\ log(n))$.

Secondly, in the case of the LWE problem, each sample (corresponding to one vector multiplication) in a standard lattice based construction typically encodes a single bit of data, while ideal lattice based constructions can encode on the order of n bits per sample. This improved packing efficiency means that both key and ciphertext sizes can be reduced by a factor of n when using ideal lattices, and provide the same security guarantees (LYUBASHEVSKY; PEIKERT; REGEV, 2012).

2.7 CRYSTALS-Dilithium

Dilithium is a lattice-based digital signature scheme, recently standardized by NIST as ML-DSA. It is a part of the CRYSTALS suite of cryptography, which also includes Kyber, a key encapsulation scheme. Dilithium is based both on the hardness of the Ring-LWE and Ring-SIS problems (LYUBASHEVSKY, 2024); specifically, it uses the ring $\mathcal{R}_{q,f}$ where $f(x) = x^n + 1$, $n = 256$, and $q = 2^{23} - 2^{13} + 1$.

The pseudo-code for a simplified version of Dilithium's functions — `KeyGen()`, `Sign()`, and `Verify()` — is shown in Algorithm 2. This version does not include optimizations present in the official specification of the scheme, such as public key compression. See Appendix A for the official pseudo-code for Dilithium and its subroutine functions. Furthermore, for an in-depth description of the algorithm, its notation, and its proof of security, reading the official specification Bai et al. (2021) in full is highly recommended.

On a theoretical level, Dilithium follows the Fiat-Shamir approach of turning a ZKPoK into a signature scheme (see Subsection 2.2.5). However, it must also perform an additional rejection step to further avoid leaking information about the secret. That is, since it is known that the secret key (s_1, s_2) is uniformly drawn from a range $[-\eta, \eta], \eta \in \mathbb{Z}$, a passive attacker could identify biases in the distribution of signatures over multiple signing instances, and obtain information on the secret key. This inclusion of a rejection step was first proposed by Lyubashevsky (2009), and it implies that the `Sign()` function is not executed in fixed-time (however, it is still independent from (s_1, s_2)).

The other innovation proposed in Dilithium is reducing the size of the public key and the signature through the use of *quantization*. Intuitively, instead of the signer transmitting the computed response z of the Σ-protocol, they can instead transmit its highest order bits, coupled with a *hint* h that describes the summation carries that are needed to recover z. If the computed values lie within certain pre-determined bounds, transmitting rounded z and h is more efficient than transmitting the original z.

Finally, for the reasons described in Section 2.6, most polynomial operations are done in the NTT domain; indeed, the matrix A is also itself sampled in the NTT domain, to avoid conversion operations.

Algorithm 2 – Pseudo-code for a simplified CRYSTALS-Dilithium

KeyGen()

$A \leftarrow \mathcal{R}_q^{k \times \ell}$
$(s_1, s_2) \leftarrow S_\eta^\ell \times S_\eta^k$
$t := As_1 + s_2$
return $(\text{pk} = (A, t),\ \text{sk} = (A, t, s_1, s_2))$

Sign(sk, M)

$z :=\perp$
while $z =\perp$ **do**
    $y \leftarrow S_{\gamma_1 - 1}^\ell$
    $w_1 := \text{HighBits}(Ay,\ 2\gamma_2)$ ▷ HighBits performs quantization
    $c \in \mathcal{B}_\tau := H(M \| w_1)$ ▷ H is a CRHF
    $z := y + cs_1$
    if $\|z\|_\infty \geq \gamma_1 - \beta$ **or** $\|\text{LowBits}(Ay - cs_2,\ 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ **then**
        $z :=\perp$
    end if
end while
return $\sigma = (z, c)$

Verify(pk, $M, \sigma = (z, c)$)

$w_1' := \text{HighBits}(Az - ct,\ 2\gamma_2)$
return $[\![\|z\|_\infty < \gamma_1 - \beta]\!]$ and $[\![c = H(M \| w_1')]\!]$

Source: Bai et al. (2021)

The developers of the CRYSTALS suite provide a reference implementation for the Dilithium algorithm in the C language, together with a set of Known-Answer Test (KAT) vectors provided by NIST for validation purposes[1].

2.8 Trusted Platform Module

A Trusted Platform Module (TPM) is a component with a state that is inaccessible to the host platform to which it is connected. The purpose of the TPM is to evaluate the *trustworthiness* of other components of said platform, and attest it to verifiers (Trusted Computing Group, 2025d).

For example, a TPM can attest to the trustworthiness of the first-stage bootloader (platform component) to the second-stage bootloader (verifier), in order to provide safe-booting for the overall system. The particular entities that act as the platform component and as the verifier are dependent on the attestation being executed.

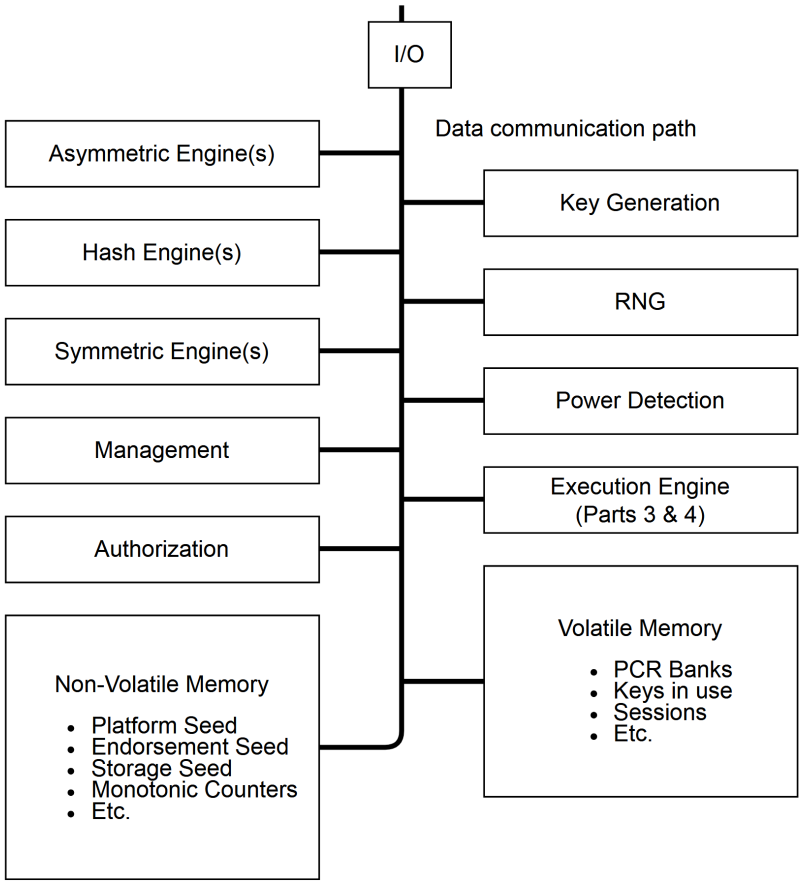
[1] <https://pq-crystals.org/dilithium/software.shtml>

The meaning of *trust* in this context is that of its expected behaviour. By measuring trust, the TPM can evaluate if the platform has been compromised by attackers. Since the TPM is a separate component, it is capable of making these assertions even if the platform itself (i.e. the CPU, the OS, etc.) is acting maliciously.

Hence, the TPM's own trustworthiness is the fundamental axiom from which other components' trustworthiness are derived: a concept which is called *Root of Trust* (RoT) in the TPM specification. As such, the TPM must implement mechanisms that protect it from being compromised, such as tamper-resistant packaging and protected memory regions for storing cryptographic keys. The TPM is also, more generally, intended to act as a cryptoprocessor that provides cryptographic services to the host platform.

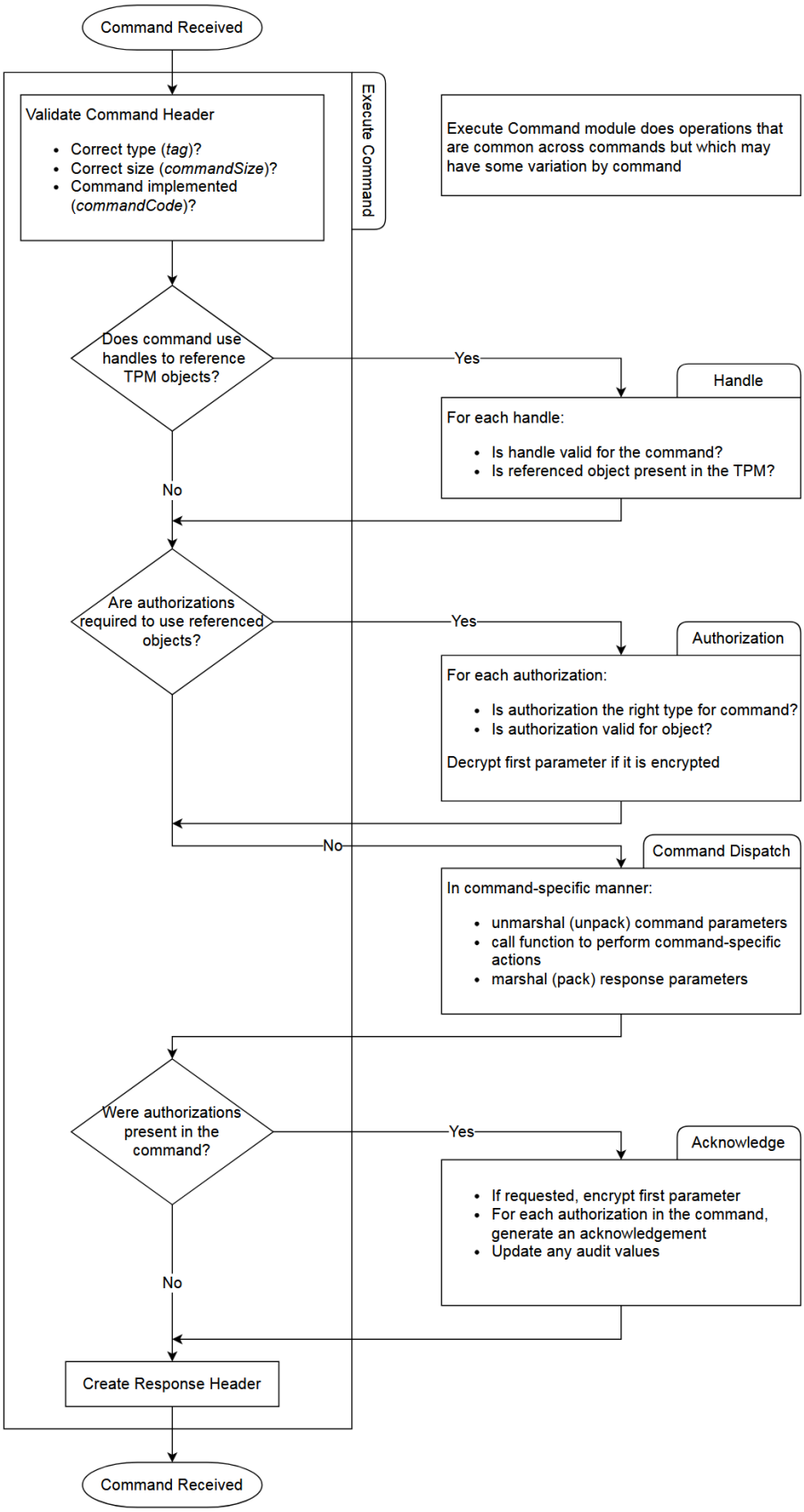
The TPM is defined by a collection of specification documents produced by the Trusted Computing Group, which are consolidated in the TPM 2.0 Library (hereafter also referred to as 'the TPM Library', or more simply, 'the Library') available at Trusted Computing Group (2025c). An architectural overview of a TPM and its execution flow, both taken from the Library, can be seen in Figure 5 and Figure 6, respectively.

Figure 5 – Conceptual overview of the architecture of a TPM.



Source: Trusted Computing Group (2025e)

Figure 6 – TPM 2.0 command execution flow.



Source: Trusted Computing Group (2025e)

The TPM 2.0 Library determines many characteristics of the TPM device, such as authentication methods, session behaviour, and key management hierarchies. It includes the algorithms that are supported for each kind of cryptographic operation (a compiled list can be seen in Appendix A). Most importantly, it also defines the *command codes*, *attributes*, and *handles* that comprise the set of commands the host can send, and the set of responses that can be returned by the TPM.

An example of the format of TPM commands and responses is given in Table 1 and Table 2 for `TPM2_GetRandom`. This operation returns a sequence of random bytes, whose size is determined as a parameter in the input command.

Table 1 – `TPM2_GetRandom` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	TPM_ST_SESSIONS if an audit or encrypt session is present; otherwise, TPM_ST_NO_SESSIONS
UINT32	commandSize	size of the command buffer
TPM_CC	commandCode	TPM_CC_GetRandom
Parameters		
UINT16	bytesRequested	number of octets to return

Source: Trusted Computing Group (2025g), revised by the author.

Table 2 – `TPM2_GetRandom` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	see Clause 6 of Trusted Computing Group (2025g)
UINT32	responseSize	size of the response buffer
TPM_RC	responseCode	command result code
Parameters		
TPM2B_DIGEST	randomBytes	the random octets returned by the TPM

Source: Trusted Computing Group (2025g), revised by the author.

TPMs can be implemented as separate chip components that connect to the host platform via a specific interface, and the host must only be able to affect the TPM via this interface. For example, the host cannot directly change values in the TPM's private memory, other than through the I/O buffer that is part of the interface (Trusted Computing Group, 2025d).

TPMs can also be implemented as software components that are executed by the host CPU in a protected execution mode, such as ARM TrustZone. In this case, the host still needs to allocate resources for the TPM process, such as private memory regions that are inaccessible by other processes, or even the CPU itself, if it is not in the special execution mode. Some open-source projects exist that implement software-based TPMs, such as the community-driven swtpm[2] and IBM's ibmswtpm2[3].

The TCG also maintains its own reference implementation of a software TPM[4], intended to showcase the TPM 2.0 Library functionalities.

[2] <https://github.com/stefanberger/swtpm>
[3] <https://github.com/kgoldman/ibmswtpm2>
[4] <https://github.com/TrustedComputingGroup/TPM>

3 Methodology

The development of this project took place throughout the whole academic year of 2025. The following tasks were performed as part of the project:

a) A bibliographical study of lattices, lattice-based cryptography, and the mathematics behind the CRYSTALS cryptographic suite. This proves essential to understand all steps of the cryptographic routines to be used.

b) A further study into the current state-of-the-art implementations of hardware accelerators for the Dilithium scheme.

c) The design of the overall architecture for the TPM using, when possible, already built (and permissively-licensed) implementations of hardware cores. This allows for a quicker development of a baseline system, without having to 'reinvent the wheel' at certain, already well-studied and well-documented, stages.

d) The integration of a Dilithium hardware core into the platform, as well as the development of Hardware Abstraction Layer (HAL) functions that can be used by the TPM's CPU to invoke the core's services. The core itself can be either a novel implementation, or a validated (and possibly updated) pre-existing design.

e) The development of a firmware library that provides all the cryptographic services — both traditional and post-quantum — required of the TPM, to be executed on the TPM's CPU, leveraging its other cores when needed.

f) An FPGA implementation of this system, followed by an evaluation of the correctness of the implementation, as well as its performance and its security against practical cryptographic attacks.

g) The elaboration of the final documentation of the thesis, detailing every step of the project's development.

This set of tasks prioritizes the aspects of the project which are the most innovative and represent the core of its functionality: that is, providing post-quantum cryptographic functionalities to a host platform. Additionally, some of the tasks were chosen to facilitate an agile development process, such as reusing pre-existing cores which have been validated.

The bulk of the implementation work for this project are contained in tasks (d) to (f). Although the tasks were listed as a sequence, most of the development process was performed in 'cycles', as illustrated in AAAAA.

AAAAA Talk about continuous development, validation, simulation, and implementation.

4 Project specifications

In this Chapter, the functional and non-functional requirements for the TPM platform are given, followed by a brief explanation of the design strategy for the platform.

4.1 Requirements

The TPM to be designed must have the following functional requirements, concerning its fundamental tasks:

a) **Specification compliance:** provide all required services of a TPM, per its official specification in Trusted Computing Group (2025c). Moreover, internal functionalities need to be implemented following the guidelines detailed in the specification.

b) **Signing:** provide Dilithium's digital signature operations — `KeyGen()`, `Sign()`, and `Verify()` — as additional TPM services that can be called by the host platform, using TPM's standard communication protocols and conventions.

c) **Cryptographic correctness:** perform all implemented cryptographic primitives and routines correctly, as per their NIST specifications. This also applies to Dilithium's own specification, available at Bai et al. (2021).

d) **Secure key management:** securely store private keys and certificates to be used in the cryptographic schemes.

Moreover, its non-functional requirements are:

a) **Security:** the system must minimize the device's attack surface against practical cryptographic attacks, by employing a variety of strategies, ranging from hardware architectural decisions, to secure firmware implementation, etc.

b) **Responsiveness:** The system must perform the necessary cryptographic routines fast enough so as not to excessively impact the execution time of the services required by the host.

4.2 Architecture design strategy

A TPM, like network cards and video encoder/decoder engines, is a co-processor that services the host platform. Unlike those other co-processors, however, a TPM is typically not implemented in a pure hardware approach; instead, a TPM is often itself a standalone computing platform. That is, it has its own CPU, that executes instructions stored in its own internal (and private) ROM, and accesses its own accelerators to perform certain tasks.

There are many advantages to designing a TPM as such, the first of which is ease of development. The TPM official specification is extremely long, has very specific behavioural rules, and includes all forms of computations. Trying to implement all functionalities of a TPM in hardware is a herculean effort, with negligible advantages to performance for many of the operations.

The second advantage is maintainability. The Trusted Computing Group regularly announces new revisions of the TPM official specification, and updating software is generally much easier than refactoring a hardware core.

The third advantage is price. TPMs are supposed to be inexpensive components of commercial computers. It is preferable to implement them using comparatively cheap microcontrollers — coupled with modules for cryptographic operations and key storage — as opposed to using custom built ASICs.

For these reasons, the TPM in this project is designed following a software-centric approach. However, given that the original goal is to provide hardware acceleration for the new post-quantum services, it is unfortunately not possible to simply use an off-the-shelf micro-controller — that has no built-in PQC engine — for this project.

Therefore, the ideal architecture for this TPM is a self-contained execution environment, which is also capable of accommodating new custom hardware cores, and directly interacting with them via the CPU. That, in short, is the architecture of a System on a Chip (SoC), which was thus chosen for designing the TPM in this project.

For this project, a RISC-V SoC is specifically targeted, both due to its ease of development and simulation in FPGA platforms, and also due to the open-source nature of the RISC-V ISA.

Finally, it is important to note that it is often preferable to execute the TPM code as bare-metal firmware, as opposed to as a user process in an operating system. One reason for that is that it keeps the SoC simple: even lightweight OSes might impose minimal requirements in terms of memory availability or CPU capabilities. Another reason is that a vulnerability in the OS can provide additional attack vectors through which the TPM could be compromised.

5 Project development

This chapter covers the overall development of the project, conducted throughout the second half of the academical year of 2025. Section X talks about X, ...

5.1 Tools used

In this section, some of the main tools, both in terms of software and hardware, used during the development of the project are presented and contextualized.

5.1.1 Development board

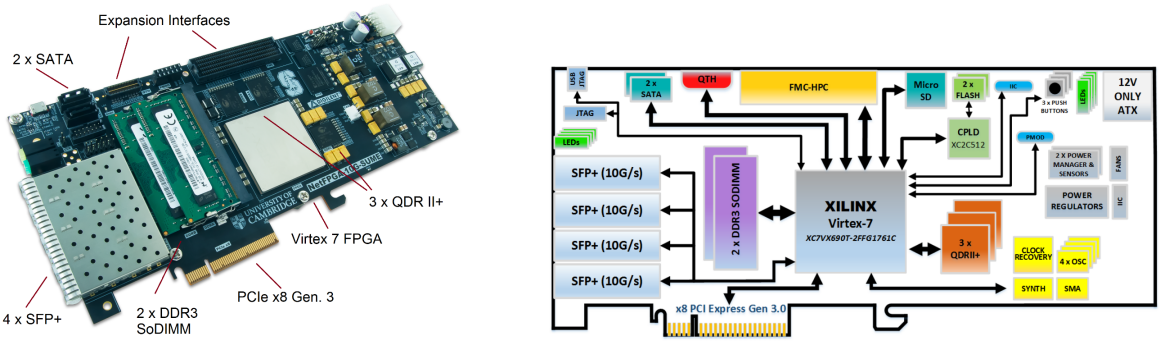
Since the project entailed the development of custom designed hardware — be it for the Dilithium accelerator, or the top-level SoC and its other components — a way of implementing the design into physical hardware was needed. Typically, the options for such are two: Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs). Although both are integrated circuits, FPGAs are composed of basic hardware logic elements which can be programmed at build time to implement arbitrary designs, while ASICs are chips which have been manufactured with a custom digital circuit design by a semiconductor foundry.

Deciding between one or the other is a trade-off between price and flexibility, but both are commonly used at different stages of the same project. FPGAs allow for fast prototyping of new designs, and require a (comparatively) low financial investment; ASICs generally have better performance metrics (be it in terms of latency or power consumption), but are only cost-effective when produced in large volumes.

Considering both the small scale and the proof-of-concept nature of this project, an FPGA was used. Specifically, a NetFPGA Sume (ZILBERMAN et al., 2014) board was targeted, due to it being readily available to the developers, and also due to it having more than sufficient resources for implementing the final design, as will be shown later.

The NetFPGA SUME is a PCIe-based board, developed by Digilent in 2014, and features a Xilinx Virtex-7 XC7VX690T FPGA. Its general appearance and architecture are shown in Figure 7, while a summary of its characteristics can be seen in Table 3.

Figure 7 – NetFPGA SUME board and block diagram.



(a) NetFPGA SUME Board

(b) NetFPGA SUME Block Diagram

Source: Zilberman et al. (2014)

Table 3 – Resources on NetFPGA SUME relevant for this project.

Resource Type	**Quantity**
Logic Cells	693,120
Slices	108,300
Flip Flops	866,400
DSP Slices	3,600
On-chip block RAM (bits)	52,920
Clock Management Tiles	20
MicroSD card slots	1

Source: AMD (2021) and Zilberman et al. (2014)

5.1.2 SoC development framework

In theory, when designing a SoC, one could do so with the tools typically employed in hardware design: an open-source Hardware Description Language (HDL) for developing custom components, and licensed IP blocks for more standard (albeit complex) components, such as CPUs, on-chip interconnects, and memory blocks.

In practice, doing so is rarely ideal. One reason is that different IPs are developed with different interfaces in mind: some may expose a custom register interface, others may expose a specification-compliant interface such as AMBA AXI. Connecting these IPs to a CPU over a system bus (which may also be different than the one originally intended by the IP) is burdensome and requires some additional layer of compatibility.

SoC development frameworks are designed to solve this issue by working at a higher degree of abstraction, in which the developer seldom needs to wire components

manually. Typically, the framework's API allows for declaring an SoC by specifying its CPU architecture and the corresponding system bus; then, any additional component can be connected to the SoC by other API calls. These frameworks also aim for streamlining the process of prototyping, validating and implementing the SoC, by including ways of simulating and synthesizing the design.

With that in mind, there are two main alternatives for SoC development frameworks: Chipyard and LiteX.

Chipyard (AMID et al., 2020) is an open-source framework proposed and maintained by UCB's Berkeley Architecture Research Group, which also created the RISC-V ISA, the Rocket and BOOM CPUs, and the Chisel HDL. Chipyard is based on Chisel, which itself a Scala DSP, and is intended to leverage the RISC-V ecosystem of components. It also includes a tool called FireSim, which allows for building designs in cloud-hosted FPGA platforms.

On the other hand, LiteX (KERMARREC et al., 2020) is an open-source, community-driven framework based on Python and the Migen HDL. LiteX itself boasts a wide range of cores (Ethernet, PCIe, SPI, DRAM, SATA, SD), as well as compatibility layers for other open-source designs, as is the case with the CPU offerings, that include Rocket, VexRiscv, PicoRV32, etc.

The (post-hoc) opinion of the author is that the two frameworks offer a trade-off. LiteX has a faster initial development process, with one-liners for building simple SoCs and integrating new cores. Go beyond that, however, and one will discover a codebase that has several unsolved bugs, and a mostly non-existent documentation. Chipyard — true to its academical origin — has extremely detailed and up-to-date documentation, but can be obtuse for developers inexperienced with Scala or Chisel.

That said, the two tools also have many similarities: both rely on a backend EDA toolchain for performing design analysis, synthesis, and implementation; both offer simulation tooling for running functional simulations of its SoCs using Verilator[5].

Throughout this project, LiteX was used as the framework for developing the TPM.

[5] <https://www.veripool.org/verilator/>

5.1.3 Electronic Design Automation toolchain

The final output of the frameworks described in Section 5.1.2 is a set of design files that describes the entire SoC and its components, written in a HDL such as Verilog or VHDL. In order to actually implement this design on an FPGA or an ASIC, an Electronic Design Automation (EDA) toolchain is needed. For FPGAs, these toolchains cover several stages of the build process:

1. **Synthesis:** converts the HDL into a netlist of logical cells (such as LUTs, flip-flops, and BRAMs), nets to connect them, and top-level I/O ports.

2. **Place and Route (PnR):** assigns each logical cell to an actual physical element of the chosen FPGA board (*place*), and connects these elements together (*route*).

3. **Optimization:** identifies and eliminates redundant logic elements, introduces buffers to solve hold interval violations, places related logical elements closer, etc.

4. **Bitstream generation:** converts the implemented design into a bitstream file that configures the FPGA.

There are several different EDA toolchains available, usually maintained by the FPGA manufacturer, or by the open-source community. The two most common EDA toolchains are, not coincidentally, developed by the two leading FPGA manufacturers: Quartus[6], which is developed by Altera; and Vivado[7], which is developed by AMD (previously Xilinx).

Generally, a developer will choose the toolchain that is compatible with the FPGA being used for the project. In this case, since the NetFPGA SUME was used (see Subsection 5.1.1), this meant using the Vivado toolchain; specifically the 2025.1 release.

5.1.4 Programming languages

The range of tasks included in this project made necessary using several different programming languages.

As already mentioned in Subsection 5.1.2, Python was used for developing the overall SoC, since LiteX is built on it. Migen, a Python DSL for generating HDL code, was also used for designing simple hardware components to support the SoC's main functionalities. Moreover, as will be shown later, Python was also used for developing a test suite for the SoC, as well as an interactive dashboard.

[6] <https://www.altera.com/products/development-tools/quartus>
[7] <https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html>

When refactoring or developing more complex hardware cores — such as the Dilithium or the SHAKE accelerators — an RTL-level HDL was used. For this goal, Verilog/SystemVerilog was chosen. The reason is two-fold: firstly, as will be shown later, the existing cores that needed to be refactored were already written in Verilog; secondly, the tool used for simulating the SoC (i.e. Verilator) is made for Verilog/SystemVerilog, and has no support for VHDL.

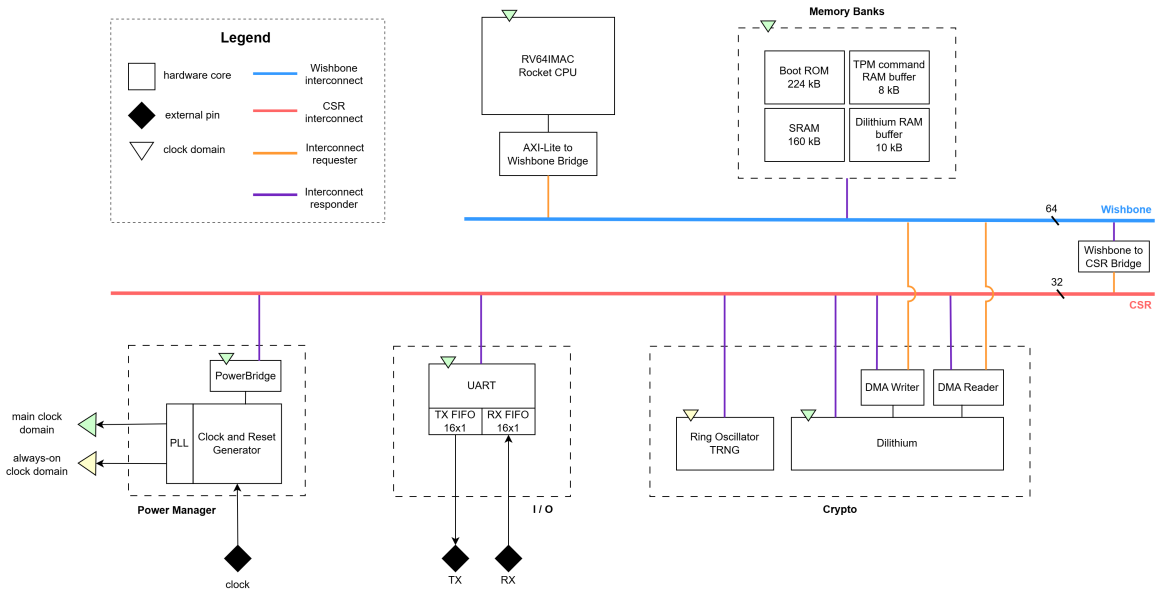
In addition, for developing the TPM firmware, the C language was chosen for several reasons. One is that the baremetal nature of the project already limits the choice to that between low level languages such as C, C++ and Rust. Between these, Litex already offers a collection of standard libraries for C (and some for C++). Finally, the reference TPM library is already implemented in C, meaning that using C made integration easier.

5.2 Petalite SoC

As mentioned in Section 4.2, the overall design strategy for this TPM is that of an SoC. In this section, an in-depth description of its architecture is given, along with its main components. During the development of this project, the SoC was named *Petalite*, and this is how it will be referred as throughout the remainder of this document.

Petalite is a single-core, dual-bus, 64-bit RISC-V SoC. A diagram of its architecture can be seen in Figure 8.

Figure 8 – Petalite SoC architecture diagram.



Source: created by the author

5.2.1 CPU and interconnects

The first component to be discussed in this section is the one that forms the backbone of any SoC: the system interconnect. For Petalite, a 64-bit Wishbone shared interconnect was chosen. The choice of Wishbone is based on it being both an open standard (OpenCores Organization, 2010), as well as the default and most well-documented interconnect used in LiteX. As a case in point, LiteX offers compatibility bridges between Wishbone and other common interconnects, such as AMBA AXI4 and Intel Avalon.

Wishbone is specified for different data widths, spanning from 8 bits to 64 bits. The reason for choosing a 64-bit version for Petalite is one of performance. To understand why, consider that many of the tasks of this TPM are communication-bound. For example: the system will have to transfer data from the UART buffer into a private command input buffer; the Dilithium accelerator will then have to read and write signing data into memory, which will also have to be transferred into a private command output buffer.

It is therefore beneficial to use an interconnect that supports a high bandwidth, thus reducing communication overhead. A similar argument also applies to the CPU: choosing a 64-bit version will not only reduce the overhead when performing CPU-driven memory operations, but also avoid the need for extra hardware in the form of converter bridges, which would be necessary to communicate a 32-bit CPU to a 64-bit interconnect.

Accordingly, the CPU used by Petalite is a single-core RV64IMAC Rocket, originally developed by UC Berkeley researchers in Asanović et al. (2016). Rocket was chosen since it is a well-documented CPU, commonly used by the LiteX community, and also demands less resources compared to other 64-bit offerings available in the framework, such as the RV64GC BlackParrot (PETRISKO et al., 2020).

Although LiteX does offer multi-core versions of Rocket, and having multiple cores could reduce the communication bottleneck on the system, these were not used since they would overcomplicate the firmware. That is, depending on the parallelization strategy used, it would be necessary to develop ad-hoc concurrency and synchronization mechanisms.

Finally, Petalite also has a secondary interconnect: LiteX's Control Status Register (CSR) Bus. The CSR Bus is designed to facilitate the integration of custom cores into the SoC. Instead of forcing cores to have a Wishbone (or, for that matter, any other interconnect) interface to interact with the CPU, the developer defines the core's I/O ports as CSRs using LiteX's API. These then become memory-mapped registers connected to the CSR Bus, which is itself connected to the system interconnect through a bridge.

Petalite — like any other LiteX SoC — uses both interconnects in tandem. For instance, as will be shown later, runtime configurations of the Dilithium core are set through CSRs, but the data itself is transferred via Wishbone to a memory buffer.

5.2.2 Communication protocol

As stated in Section 2.8, the TPM is a co-processor intended to provide services to a host platform. Hence, it must have an I/O interface through which it (the responder) can communicate with the host (the requester).

Interestingly, although the TPM base specification (Trusted Computing Group, 2025e) assumes the existence of abstract "command buffers" — in which input commands are received by the TPM, and output commands are sent to the host — it does not define how the data should be ingested into these structures. In other words, the specification defines a communication protocol at the application level (the TPM 2.0 commands) but it is otherwise agnostic towards the underlying communication layers.

Instead, other supporting specifications produced by the TCG, such as the one for designing TPMs for a PC host target (Trusted Computing Group, 2025a), define both the transport and the data-link layers for the TPM device, as summarized in Figure 9. In the case of the transport layer, TCG defines a legacy FIFO streaming interface (TPM Interface Specification, TIS) and a current memory-mapped interface (Command Response Buffer, CRB).

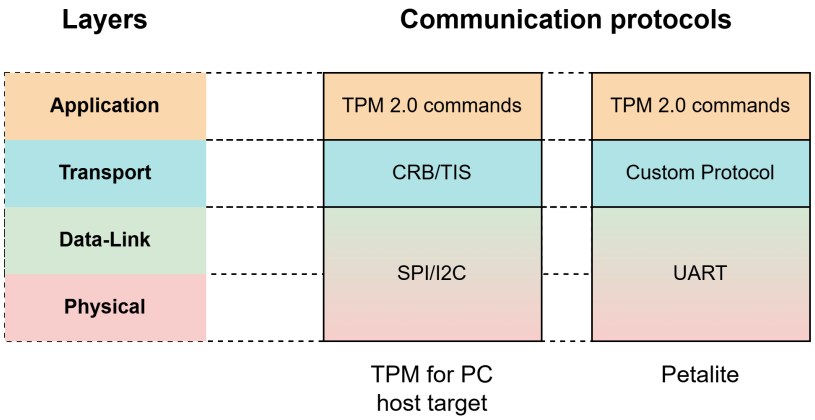
As regards the data-link layer, TCG defines SPI and I2C alternatives. The reason for these choices of protocols is so the TPM can be easily connected to the controller hub of modern PCs, and thus readily available during boot-up for providing services.

For Petalite, the communication protocols aimed at being simple and flexible. If the goal is to develop a generic TPM that does not limit itself to a specific host with some expected architecture or capabilities, adhering to the PC target specification is at odds with that objective. Furthermore, deciding not to conform to that specification has the secondary benefit — in terms of development process — of limiting time spent ascertaining that the TPM is in accordance not only with the base specification, but another supplementary set of rules and guidelines.

The SoC itself is also restricted in its options for I/O by the tooling used to develop it. Although LiteX does offers hardware cores and software libraries for the SPI and I2C data-link protocols, these are 'one-sided' implementations: they function solely as the requester (not the responder, as would be needed for the TPM), since their intended use case is for the SoC to interact with its own co-processors. Moreover, LiteX does not provide the infrastructure necessary for simulating these protocols on software, thus making the development cycle for the project more difficult.

Conversely, there are other protocols for which LiteX provides software simulation support via TCP sockets, namely Ethernet and UART. Between these two, Ethernet is not a protocol suitable for TPMs, since it is not generally available during the early stages of the boot-up procedure, as the NIC and Ethernet controller are initialized later.

Figure 9 – Diagram of the layered communication protocol used by a standard TPM targeting a PC host, and by Petalite.



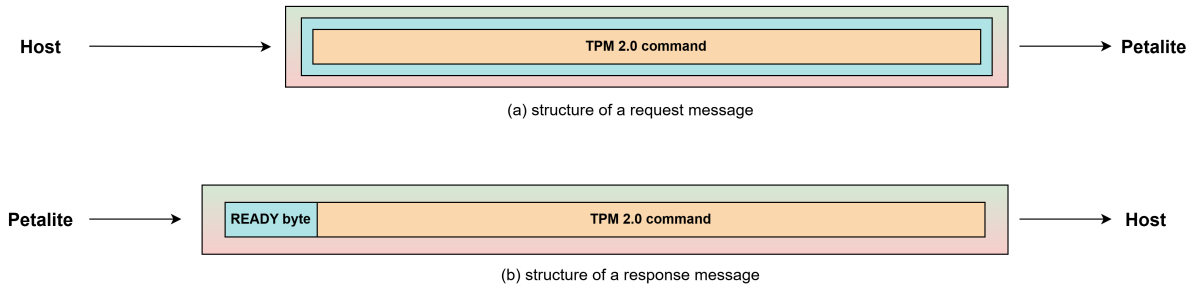
Source: created by the author

As such, the data-link protocol used by Petalite is UART at a baud rate of 115,200 bit/s. Although the throughput of UART, due to its serial nature, is much lower than that of other more complex protocols, it is nevertheless easily implemented by even the most resource-restricted platforms.

Concerning the custom transport protocol used by Petalite, it is a minimal wrapper on top of the TPM command: host request messages are sent without additional framing, while TPM response messages are prefaced by a single byte. This minimizes the communication overhead for the transport layer, while still maintaining the functionalities required by a TPM device. The overall message formats used can be seen in Figure 10.

Notably, the transport layer frame does not include the length of the message. However, that information is already included in the TPM 2.0 command header (Trusted Computing Group, 2025e), and can therefore be extracted and used by the transport layer parser — although not ideal from the perspective of a protocol design.

Figure 10 – Diagram of the request and response message formats for Petalite, using the same color scheme as in Figure 9.



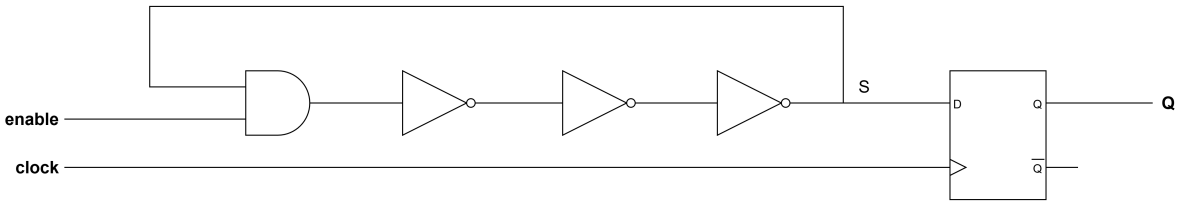
Source: created by the author

5.2.3 Random Number Generator

In order to provide true randomness for the implemented cryptographic schemes, Petalite must have a built-in TRNG. That means, as explained in Subsection 2.2.2, that a reliable source of entropy must also be determined. FPGA-based TRNGs usually implement at least one of two well-studied sources of entropy: jitter and metastability. The advantage of such sources is that, although they are analog noise signals, they can be sampled using purely digital designs that can be synthesized in FPGAs.

The core idea of sampling jitter is that of a Ring Oscillator (RO). A basic ring oscillator is an odd-numbered chain of buffered logical NOT gates such as in Figure 11.

Figure 11 – Schematic of a simple ring oscillator, with $N = 3$ inverters.



Source: created by the author

Each inverter introduces a delay of T_I, and therefore the total oscillation period T_{osc} for the RO is $T_{osc} = 2\,N\,T_I$, where N is the number of inverters. However, T_I is not constant, and is instead better modeled as a random variable:

$$T_I \;=\; t_{I,\,avg} \;+\; t_{PV} \;+\; T_{noise} \tag{5.1}$$

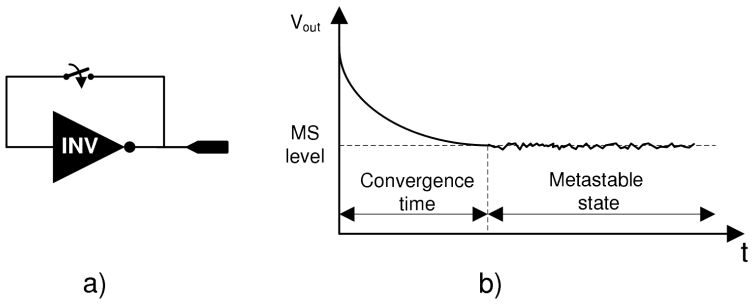
where $t_{I,\,avg}$ is the average delay for the inverter, t_{PV} is a delay component due to build process irregularities in the transistors, and T_{noise} is a random variable that depends on multiple factors, such as temperature, humidity, and electromagnetic interference (PARRILLA et al., 2023). The timing variations (i.e. jitter) of T_I and thus T_{osc} mean that sampling the output `S` of the inverter chain over a period $T_{sampling} >> T_{osc}$ provides a random binary stream. The sampling process itself is most easily done by registering `S` in a D-type flip-flop.

The main limitation of traditional ring oscillators like the one in Figure 11 is throughput: $T_{sampling}$ needs to be large enough so that sufficient jitter (and more generally, entropy) can be accumulated over several T_{osc} periods, ensuring a sufficiently random output. A high $T_{sampling}$ thus limits the maximum throughput of TRNGs based on ROs.

Common strategies for solving this problem involve increasing the amount of entropy that influences the RO. Some such improvements include XORing multiple inverter chains, or having the input `clock` signal itself be derived from a source with high jitter.

Another novel strategy, presented in Vasyltsov et al. (2008), decreases the time necessary for accumulating jitter by leveraging the second source of entropy mentioned: metastability. Metastability happens when a digital signal does not settle in its high or low logic values, and instead remains in an analog region near its threshold voltage. A simple example of a metastable digital circuit is that of a single unbuffered NOT gate with a feedback loop, as shown in Figure 12.

Figure 12 – Digital circuit employing metastability using an inverter (a), and its convergence process (b).



Source: Vasyltsov et al. (2008)

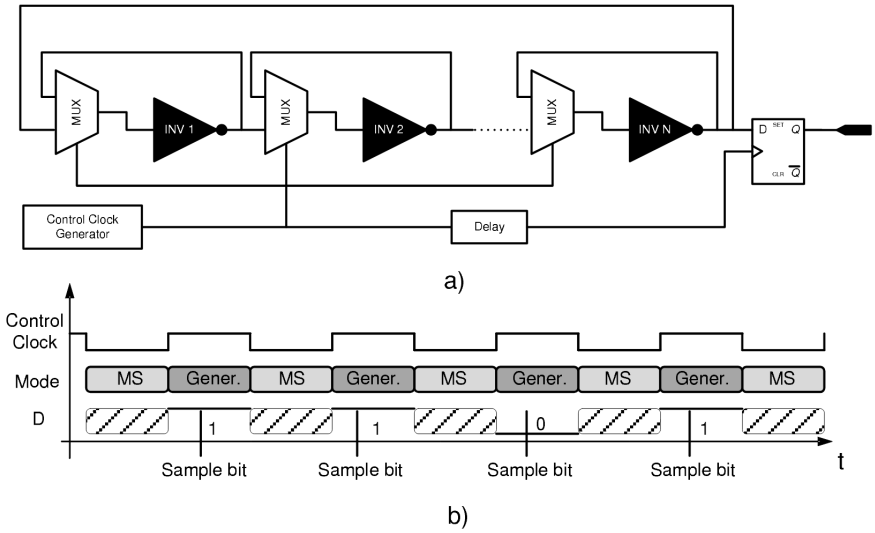
The behaviour of a circuit in metastability is stochastic and depends on multiple sources of noise, such as a mismatch of transistors, temperature imbalance within a chip, ionizing radiation, etc. Moreover, the final settled voltage of a circuit after it has exited metastability — for example, if the feedback loop is Figure 12 is interrupted — is also stochastic, and thus likewise a good source of entropy.

The design of Vasyltsov et al. (2008) is illustrated in Figure 13. Similar to a typical RO circuit, it has an odd number of NOT gates. However, instead of chaining inverters directly, a multiplexer is placed between every two inverters. Each multiplexer's output is connected to the input of a corresponding inverter, and they select between the current inverter's output, or the previous inverter's output. By doing so, they force the design to behave as either N parallel metastable circuits (Figure 12), or one RO chain (Figure 11).

In the beginning of the design's operation, its FSM sets the inverters to their metastable configuration, effectively accumulating independent entropy (MS mode). After a given period of time, the FSM then sets the circuit to behave as the single RO chain (Generation mode). This results in a faster accumulation of jitter, thus reducing $T_{sampling}$, and finally providing a higher overall throughput for the TRNG.

This architecture, called Meta-RO by the original authors, is the basis for the TRNG used in this project. The Petalite TRNG is connected to the SoC's CSR bus, and its register interface can be seen in Table 4. It is built using 33 inverters, and for every cycle — consecutive MS and Generation modes — the design shifts a new bit into a 32-bit output buffer, which can then be sampled by the SoC as a random value.

Figure 13 – Meta-RO circuit diagram (a) and timing diagram of the sampling process (b).



Source: Vasyltsov et al. (2008)

There are some small differences between the design used for Petalite and the one proposed by Vasyltsov et al. (2008); the Petalite revision includes programmable periods for each mode, and the option to XOR the output of each inverter into the final output buffer during the MS mode, accumulating further entropy.

Table 4 – Petalite TRNG CSR interface.

Register	Field	Bits	Access	Reset	Description
`trng_ctl`	`ena`	0	R/W	0	Enables or disables the TRNG core.
	`gang`	1	R/W	1	Enables XORing individual inverters during MS mode.
	`delay`	9:2	R/W	8	Clock cycles spent on Generate mode.
	`dwell`	29:10	R/W	100	Clock cycles spent on MS mode.
`trng_rand`	`rand`	31:0	R	0	Random output word of the TRNG.
`trng_status`	`fresh`	0	R	0	Set when a new word has been produced; cleared on read of `rand`.

Finally, one extra consideration is that, strictly speaking, a RO circuit is a combinatorial loop, and therefore a violation of the design practices enforced by most EDA toolchains. As such, to actually synthesize this core in an FPGA, constraints directives are used to force Vivado not to optimize or perform timing analysis on this specific circuit.

5.3 Dilithium hardware accelerator

As mentioned in Chapter 1, the goal of this project is not only to develop a TPM capable of performing Dilithium operations, but to also have these operations be accelerated on custom hardware. It follows that a RTL implementation is needed.

One possibility would be to develop a completely novel implementation of a Dilithium core. Although this could prove to be a valuable learning opportunity, this process is extremely time-consuming; previous designs have taken teams of experts several months to complete. Moreover, as mentioned in Section 1.4, there are already various implementations of Dilithium cores, targeting different computing platforms and different performance trade-offs.

These designs also commonly already have state-of-the-art architectural optimizations tailored for Dilithium, ranging from pipelines for different stages of an operation, to efficient modular reduction, and specific memory access patterns for faster NTT and NTT^{-1} computations. PUT REFERENCES HERE.

By focusing on adapting and validating a pre-existing design, instead of making a new one completely, the overall development process of this project is expedited, while still achieving top-tier performance for the hardware acceleration.

Therefore, the next step is to decide on a Dilithium core to validate, and then make any changes to its original design if needed.

5.3.1 Choice of design

According to the bibliographical research conducted at the beginning of this project, there were three permissively-licensed designs for hardware-accelerated Dilithium that were readily available online: a high-performance implementation presented in Beckwith, Nguyen and Gaj (2021)[8], a resource-efficient design from Land, Sasdrich and Güneysu (2021)[9], and a hybrid hardware-software approach of Wang et al. (2024)[10].

Between these three, the one in Wang et al. (2024) is the most different. It is itself a complete SoC, that performs some operations on hardware — such as NTT-based polynomial arithmetic — and offloads others — such as hashing and sampling polynomials — to four identical cores, and a fifth core that monitors and synchronizes the others.

This specific design has several characteristics which make it of limited value for the scope of this project. Firstly, it specifically targets a PolarFire SoC board, which combines a hard multi-core RISC-V processor and a FPGA fabric into the same device.

[8] <https://github.com/GMUCERG/Dilithium>
[9] <https://github.com/Chair-for-Security-Engineering/dilithium-artix7>
[10] This implementation, previously hosted in <https://github.com/Acccrypto/RISC-V-SoC>, could not be found by the end of this project. It is not reasonably characterized as “readily accessible” anymore.

The design therefore assumes many specific features and aspects of this board, ranging from the DSP widths used by its FPGA, the memory addresses for accessing the FPGA fabric, and notably the use of four RV64GC cores.

Most importantly, the authors of the original article note that, although this implementation is faster than pure software approaches, it still presents significant worse performance compared to complete hardware approaches. As such, the remaining two designs — which follow a hardware-centric strategy — were prioritized as options instead.

Both other designs are complete RTL cores, but they differ in their optimization targets, a fact which is mentioned in Beckwith, Nguyen and Gaj (2021) itself, since it also compares its own design to previous ones, including Land, Sasdrich and Güneysu (2021).

The design on Beckwith, Nguyen and Gaj (2021) (which will hereafter be called *HighPerf*) intends to minimize latency at the cost of higher resource usage, while Land, Sasdrich and Güneysu (2021) (hereafter called *LowRes*) aims to do the opposite trade-off. This results in meaningful architectural differences between the cores, namely:

a) **Modularity**

LowRes is actually composed of three base designs — one for each Dilithium operation — and one additional design that integrates the other three to provide a complete suite for the algorithm. Each design itself also comes in three variants, one for each security level specified for Dilithium.

HighPerf, on the other hand, is a single unified design capable of performing all three Dilithium operations, using any of the three security level specified.

b) **Functional units and interfaces**

HighPerf uses three instances of Keccak cores for hash operations, while LowRes uses a single one. HighPerf has a 64-bit streaming interface that implements a `READY-VALID` handshake, while LowRes has a (similar) 32-bit interface.

c) **Pipelining and data reusability**

HighPerf has a pipelined architecture that allows for simultaneously executing different steps of the same operation; LowRes uses significantly less pipelining.

On the other hand, while HighPerf is based on streaming all the data necessary for each Dilithium operation, LowRes performs intermediate loading and storing operations separately from the main processing steps. This thus allows for reusing data across different operations, reducing data transfer times.

For example, suppose several signatures had to be computed using the same secret key. HighPerf requires loading the key for every signature, while LowRes would only load the key when performing the first signature, and reuse that value for the subsequent ones.

A more thorough comparison between both designs was also produced and can be found in a public repository made available[11].

In order to validate the performance claims made by both articles, an effort was undertaken to replicate their results. With that aim, for each Dilithium operation — `KeyGen()`, `Sign()`, and `Verify()` — a unified testbench was developed, capable of interfacing with both HighPerf and LowRes. The testbench used the KAT vectors supplied in CRYSTALS's reference implementation of the algorithm.

When developing the testbenches, concerns related to each design were found, which will be later tackled in Subsection 5.3.2. However, it must be noted that LowRes had critical issues which led it to produce incorrect results. This can be partly attributed to a bug on the NTT module, which was discovered in Wang et al. (2022). Other (previously unaddressed) bugs were also found, such as an out-of-bounds memory access during `Verify()` operations using security level 3 parameters.

The latency metrics obtained, and a comparison with the values originally reported in each article, are summarized in Table 5 for HighPerf and in Table 6 for LowRes. In the case of the `Sign()` operation, both the best-case (i.e. the signature is accepted in the first try) and average latencies are reported.

Table 5 – Cycle lantecy metrics for HighPerf, across operations and security levels.

		Security Level		
Operation	**Source**	**2**	**3**	**5**
Keygen	**This Work**	4,868.3	8,289.3	14,030.8
	Reference Work	4,875.0	8,291.0	14,037.0
	Δ (%)	-0.13%	-0.02%	-0.04%
Sign	**This Work**	10,966.7/29,012.2	16,178.7/51,048.7	24,853.0/93,435.8
	Reference Work	10,945.0/29,876.0	16,178.0/49,437.0	24,358.0/55,070.0
	Δ (%)	+0.20%/-2.89%	+0%/-3.26%	-0.02%/-69.67%
Verify	**This Work**	6,581.0	9,723.0	14,636.8
	Reference Work	6,582.0	9,724.0	14,642.0
	Δ (%)	-0.02%	-0.01%	-0.04%

Source: created by the author.

[11] <https://github.com/franos-cm/dilithium-comparison>

Table 6 – Cycle latency metrics for LowRes, across operations and security levels.

Operation	Source	Security Level		
		2	**3**	**5**
Keygen	**This Work**	21,615.6	38,640.3	56,598.6
	Reference Work	18,761.0	33,102.0	50,982.0
	Δ (%)	+15.22%	+16.73%	+11.01%
Sign	**This Work**	32,133.7/62,447.0	49,832.4/90,796.5	76,739.5/110,117.4
	Reference Work	19,423.0/66,966.0	26,979.0/105,129.0	36,609.0/112,145.0
	Δ (%)	+65.44%/-6.75%	+84.71%/-13.63%	+109.62%/-1.81%
Verify	**This Work**	21,364.9	34,295.5	55,729.0
	Reference Work	19,687.0	32,050.0	52,712.0
	Δ (%)	+8.52%	+7.01%	+5.72%

Source: created by the author.

The latency results obtained for HighPerf are statistically consistent with the ones reported in the original article. In contrast, the latency reported for LowRes appears significantly underestimated. This discrepancy is likely due to the fact that the original article for LowRes discounts the time spent loading the input data before an operation and unloading the output data after completing it.

Moreover, similar comparisons regarding the usage of FPGA resources are also shown in Table 7 for HighPerf and in Table 8 for LowRes. All designs were implemented in Vivado targeting Xilinx's Artix-7 `XC7A100TCSG324-1` FPGA, since both original articles also use a `XC7A100T` device to present their metrics.

The most notable discrepancy in the resource usage reports is regarding the maximum frequency. In that sense, it is important to note that the maximum frequency reported in this work was obtained directly from Vivado's post-implementation timing analysis. HighPerf's original figures, on the other hand, rely on an external optimization tool called Minerva (FARAHMAND et al., 2017), and there are no details in LowRes's article about how their frequency figures were achieved.

Nonetheless, as will be shown in **??**, the maximum frequency for the complete SoC was significantly higher than the ones obtained for the Dilithium accelerators, and therefore the latter figures should be viewed critically. One possible reason for this gap is

Table 7 – Resource usage comparison for the HighPerf design.

Security Level	Source	LUT	FF	DSP	BRAM	Max Freq. (MHz)
all	**This Work**	57,984	29,054	16	29	67.0
	Reference Work	53,187	28,318	16	29	116
	Δ (%)	+9.02%	+2.60%	0.0%	0.0%	-43.24%

Source: created by the author.

Table 8 – Resource usage comparison for the LowRes design across security levels.

Security Level	Source	LUT	FF	DSP	BRAM	Max Freq. (MHz)
2	**This Work**	28,830	10,687	45	15	107.5
	Reference Work	27,433	10,681	45	15	163.0
	Δ (%)	+5.09%	+0.06%	0.0%	0.0%	-34.05%
3	**This Work**	36,869	11,503	45	21	98.9
	Reference Work	30,900	11,372	45	21	145.0
	Δ (%)	+19.32%	+1.15%	0.0%	0.0%	-31.79%
5	**This Work**	48,333	13,953	45	31	89.0
	Reference Work	44,653	13,814	45	31	140.0
	Δ (%)	+8.24%	+1.00%	0.0%	0.0%	-36.43%

Source: created by the author.

simply the different target FPGAs: the Petalite SoC targets a higher-performance Kintex-7 fabric with lower logic delays and more abundant routing channels. This may allow the same RTL to achieve a higher maximum frequency.

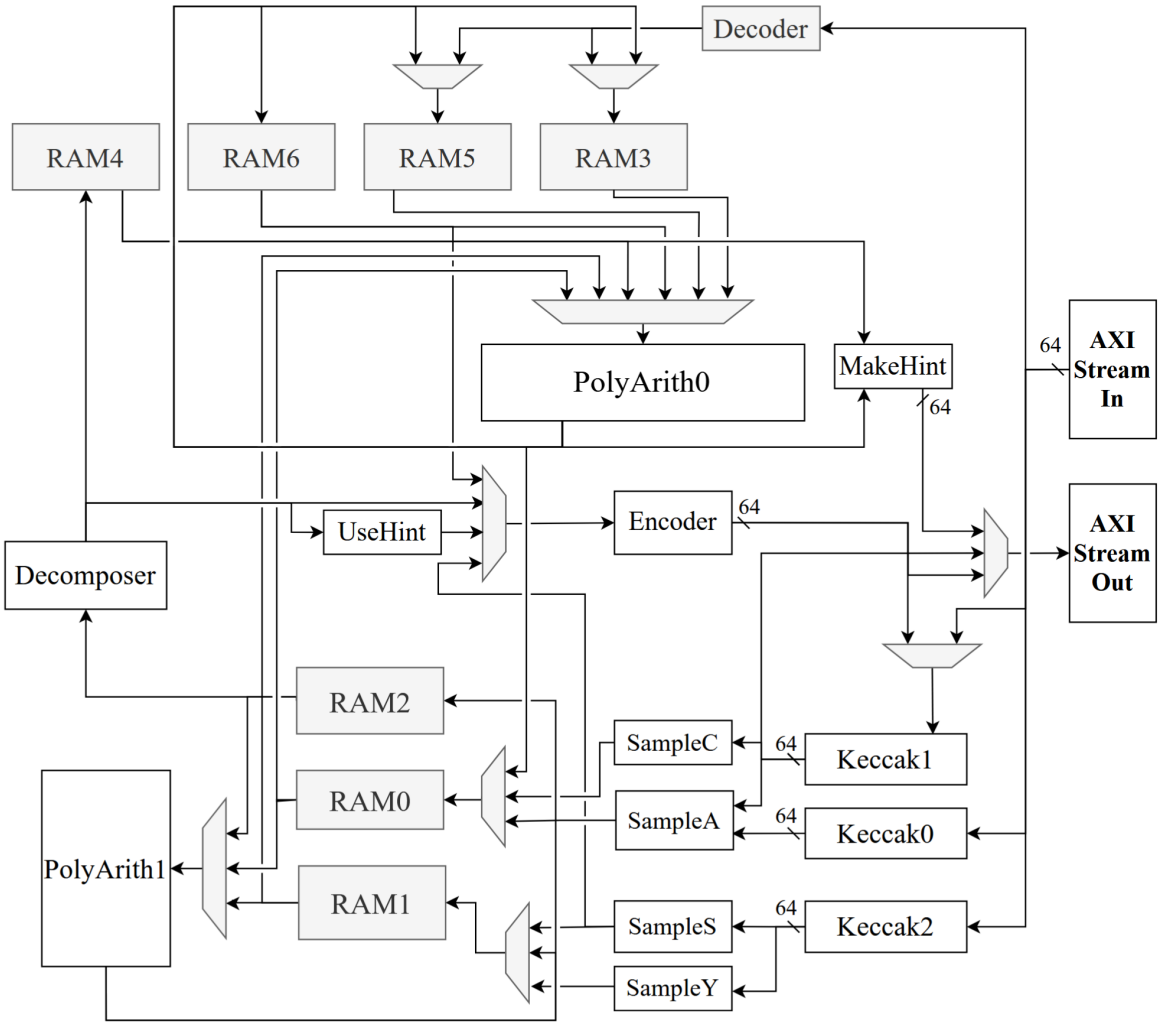
Based on these observations, HighPerf was the design chosen for Petalite's Dilithium accelerator. Firstly, in accordance with the non-functional requirement of *responsiveness* specified in Section 4.1, it presented a lower cycle-count latency than LowRes for all operations and security levels, even if at a cost of increased resource usage (which is not a critical requirement for this project).

Secondly, HighPerf is a more flexible implementation than LowRes. That is, it can perform operations for different security levels on runtime. This allows for Petalite to easily include all Dilithium security levels in its revision of the TPM 2.0 command set.

Finally, and most importantly, HighRes did not present any fatal implementation flaws. It correctly reproduced all KAT vectors in the testbench, and was thus considered a valid design without any major refactoring. The same could not be said for LowRes.

The overall architecture for HighPerf can be seen in Figure 14, while its CSR interface and its streaming interface can be seen in Table 9 and Table 10, respectively. To better understand the design, reading Beckwith, Nguyen and Gaj (2021) is highly recommended. All signals are active-high, and the **Direction** column in Table 10 is given with respect to the accelerator.

Figure 14 – Block diagram for the combined architecture of the HighPerf core. Bus widths are 96 bits unless shown otherwise.



Source: Beckwith, Nguyen and Gaj (2021)

Table 9 – Petalite Dilithium accelerator CSR interface.

Register	Bits	Access	Reset	Description
`mode`	1:0	R/W	0	Selects the Dilithium operation to perform.
`security_level`	2:0	R/W	0	Selects the security level used.
`start`	0	R/W	0	Start strobe that requests a new operation.
`reset`	0	R/W	0	Local synchronous reset; ORed with the global system reset.

Source: created by the author.

Table 10 – Petalite Dilithium accelerator streaming interface.

Interface	Signal	Direction	Width	Description
`sink`	`valid`	in	1	Asserted by the requester when an input word is available on `sink.data`.
	`ready`	out	1	Asserted by the responder when it is ready to accept a new input word.
	`data`	in	64	Input data stream to the responder.
`source`	`valid`	out	1	Asserted by the responder when an output word is available on `source.data`.
	`ready`	in	1	Asserted by the requester when it is ready to accept a new output word.
	`data`	out	64	Output data stream from responder.

Source: created by the author.

5.3.2 Revisions and improvements

Although HighPerf did produce the correct results when testing it in the new unified testbench, the same did not happen when integrating it into the Petalite SoC. In this section, the implementations flaws, and their solutions, are presented. Most of the problems are related to data flow-control mechanisms.

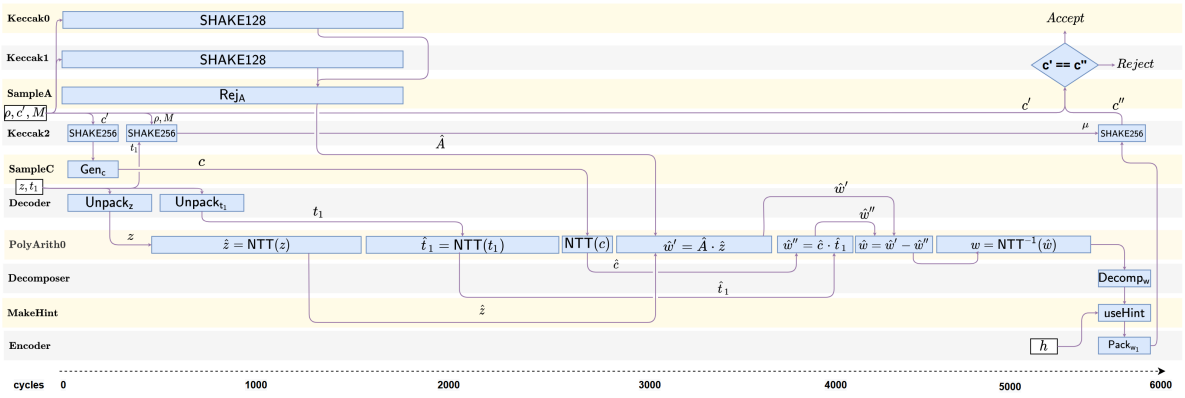
Firstly, although Figure 14 explicitly states that the core implements a standardized AXI-Stream interface, and Table 10 shows a similar interface, in practice the core did not strictly follow the `READY-VALID` handshake imposed by this protocol. EXPLAIN THIS KIND OF HANDSHAKE

Specifically, HighPerf would both ingest input data regardless of the requester asserting `sink.valid`, and continuously stream output data regardless of the requester asserting `source.ready`. In other words, the core would assume the requester is always ready to both send input data and receive output data.

While this assumption is correct in the case of the testbench, it certainly is not generally applicable for real systems, in which there may be interconnect contention, and the requester may be unavailable. In the case of HighPerf, this caused the core to read spurious input data from the interconnect, and to also overwrite its output without properly checking if the requester had read it beforehand.

Secondly, as already mentioned in Subsection 5.3.1, the original core was developed with a high degree of parallelism, as illustrated in Figure 15 for the `Verify` operation. In some cases, this meant the core was performing computations on one task while also ingesting data for another task.

Figure 15 – HighPerf's scheduling of tasks during a `Verify` operation for security level 2.



Source: Beckwith, Nguyen and Gaj (2021)

However, the original design assumed that such ingestion occurred in a fixed amount of time, since the request would (supposedly) always be available. As such, its FSMs only checked if the (supposedly) more time-consuming task was finished, rather than both. In a real scenario, where data ingestion may be the bottleneck instead, the core would incorrectly transition states before one of its tasks was concluded. This then resulted in incorrectly computed values.

An example of this flaw happens in the `Verify` operation. By the time the core finishes computing Unpack(t_1), it assumes that it has already done SHAKE256($tr \mid\mid M$). Crucially, the FSM would only check for Unpack(t_1) before changing states, and stop SHAKE256($tr \mid\mid M$) even if M had not been fully ingested.

Therefore, there were three issues that needed to be solved. The first one was making sure that the core would not overwrite its output data before the requester had

read it. The most straight-forward way of guaranteeing that was to create a buffer to capture the core's output when `source.valid` was asserted. This buffer than retransmits the output data to the requester, correctly following the `READY-VALID` handshake.

This buffer needs to be large enough to store the core's largest output possible. For HighPerf, this happens during the `Keygen` operation for security level 5, in which the keypair $(sk,\ pk)$ is outputted, spanning 7,456 bytes. Although this solution does increase the amount of resources used, this extra buffer represents less than 6% of the total memory originally used by the core's internal BRAMs (122,880 bytes) and ROMs (1,536 bytes).

The second issue was assuring that the core would correctly check that all current tasks were completed before transitioning to another state. This required a thorough revision on the core's scheduler and all its functional units. In the end, minor adjustments to the design's FSMs, coupled with a trivial increase of hardware resources to implement some latches, were able to solve this problem.

Finally, a third minor issue was that of undefined reset behaviour for some of the core's FSMs, due to likely misnamed signals. Although not critical, this caused the design to fail when performing functional simulations using specific tools, such as QuestaSim and Active-HDL's simulator.

The final, revised version of the design can be found online in a public repository[12].

5.3.3 Keccak core

Although the core presented by the end of Subsection 5.3.2 is already a completely functional and correct hardware implementation of Dilithium, another change was made to its design: the Keccak core. This functional unit is responsible for performing all hash operations necessary in Dilithium, specifically using the SHAKE128 and SHAKE256 XOFs.

The original design's Keccak core is available online[13], and was developed by the same team as HighPerf, but as an earlier stand-alone project. That is somewhat evidenced by the fact that it is the only functional unit of HighPerf that is described using VHDL instead of Verilog.

The choice of HDL for designing a core is mostly superfluous if they all describe the same behaviour. Nevertheless, as mentioned in Section 5.1, LiteX (and its built-in simulation tool, Verilator) only supports simulating Verilog/SystemVerilog designs.

Therefore, in order to simulate Petalite during the initial development cycle of the project, it was beneficial to have the complete SoC be written in Verilog/SystemVerilog. Since the Keccak core was the only exception to this convention, a revision of this design was developed using SystemVerilog.

[12] <https://github.com/franos-cm/dilithium-rtl>
[13] <https://github.com/GMUCERG/SHAKE>

This SystemVerilog version, also available online[14], closely matches the behaviour of the original design, with improvements to overall code organization and readability. The only significant functional difference is that, while the original core was also capable of performing SHA3 operations, the revision did away with this functionality, as it is irrelevant for implementing Dilithium.

The core is designed with a 64-bit streaming interface that uses a *modified* `READY-VALID` handshake. This handshake protocol is not strictly compatible with the one defined in AXI-Stream: the responder can only assert `READY` *after* the requester asserts `VALID` (`READY` is thus more intuitively interpreted as a `READ` signal).

The message and digest sizes, as well as which XOF to use — SHAKE128 or SHAKE256 — are all specified in the first word transmitted by the requester, as shown in Table 11. Moreover, if the input size is not an integer multiple of the block size (1,344 bits for SHAKE128 and 1,088 bits for SHAKE256), the core pads the message by applying the algorithm's standard rules.

Table 11 – Format for the Keccak core's input header.

din[63:60]	**din[59:32]**	**din[31:0]**
`0xC` → SHAKE128 `0xE` → SHAKE256	output data size (bits)	input data size (bits)

Source: created by the author.

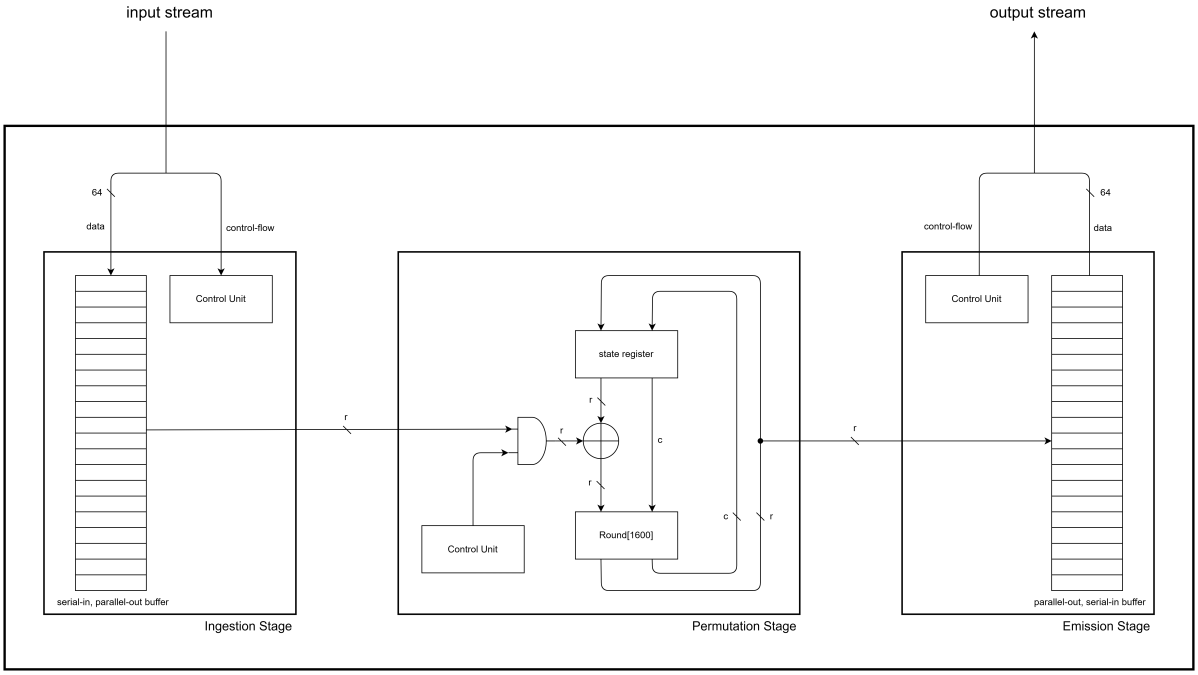
Internally, the core has a three-stage pipelined architecture: one stage for ingesting the message, one for calculating the Keccak-f ($f = 1600$) permutation, and one for emitting the digest. Figure 16 shows a high-level summary of this architecture. The Ingestion Stage receives the input serially in 64-bit words, and forwards it as an r-length block to the Permutation Stage. This second stage first *absorbs* the block into a Keccak-1600 permutation, and then later *squeezes* the r-length result in parallel to the Emission Stage. The Emission Stage then outputs this block serially in 64-bit words.

The illustration in Figure 16 simplifies the design by omitting several key details, such as the units necessary for decoding the header, defining r and c dynamically based on which version of SHAKE is being executed, communicating between stages, etc.

This separation allows the core to operate on different data blocks simultaneously, increasing the overall throughput. As evidence of that, the Keccak team's reference implementation of a high-performance hardware core (BERTONI et al., 2012), which has a two-stage pipeline, obtains an average throughput of 42.67 bits/cycle, while HighPerf's core achieves 53.74 bits/cycle. Both use a single instance of a Keccak Round[1600] functional unit, and as such execute one Keccak-1600 permutation every 24 cycles.

[14] <https://github.com/franos-cm/shake-sv>

Figure 16 – High-level architecture of the Keccak core.



Source: created by the author.

The original design has some limitations. Firstly, although the input and output sizes in Table 11 are specified in bits, the core does not correctly interpret messages or digests whose length are not a whole number of bytes. More relevantly, given the widths also in Table 11, the input and output sizes are implicitly limited to 256MiB and 4 GiB, respectively.

From a formal perspective, this second issue makes the core inconsistent with the XOFs it implements, as by definition, they are supposed to allow for an input of arbitrary length, and likewise an output of arbitrary length and *extendable*. That is, it should be possible to invoke the function multiple times to produce sequential chunks of output.

In practice, the limitations regarding the output size and its extendability are inconsequential for implementing Dilithium. The maximum size for the input, however, does determine the maximum size for messages that can be signed by the Dilithium core. That said, traditional use cases for digital signatures very rarely require signing messages with sizes on the order of 256MiB.

These limitations are feasibly solved by changing the way the core receives information about sizes. For instance, instead of relying on the input header specified in Table 11, the design could implement a `LAST` control-flow signal — akin to AXI-Stream — so both the requester can indicate when the input has been fully ingested, and the responder can indicate when the output has been fully emitted.

These hypothetical changes were not made in the SystemVerilog revision of the core, as they would imply in modifying its interface, and thus refactoring HighPerf to correctly interact with it. Since the original limitations were not considered prohibitive, and the necessary changes provided little other benefit, forgoing this redesign was seen as an acceptable trade-off for expediting the development process.

As mentioned, the SystemVerilog revision of the core was originally intended to be used for simulations. Nevertheless, since it provided a comparable performance to that of the original VHDL core, it was thus also used in the final implementation of Petalite.

5.4 Changes to the TPM 2.0 Library

Before discussing the firmware that implements the TPM 2.0 Library in the Petalite SoC, it is perhaps better to first consider the design rationale used to integrate the Dilithium algorithm into said specification.

The TPM 2.0 Library supports the inclusion of arbitrary new operations in the form of *vendor-specific* commands. The only additional requirement for such commands is that the `V` field (bit 29) of its `TPM_CC` command code is `SET`, which prevents conflicts with the existing `TPM_CC`s values.

On the other hand, the Library does not support adding new algorithms as alternatives for command parameters. All TPM 2.0 commands that reference a specific algorithm must do so by using identifiers — such as `TPM_ALG_SHA256`, `TPM_ALG_ECDSA`, `TPM_ALG_RSA` — that are defined in the TCG Algorithm Registry in Trusted Computing Group (2025b). This registry has no concept of 'vendor-specific' algorithms.

As will be explained throughout this section, integrating Dilithium into the TPM 2.0 Library required both adding four new `TPM_CC` command codes, and one new `TPM_ALG` identifier. This latter change is the only which is not strictly compliant with the specification, but it is significantly more straightforward and consistent with the overall Library design.

Furthermore, it is assumed that once the TCG officially adds PQC schemes to the TPM 2.0 Library, it will need to similarly append new `TPM_ALG` identifiers to the official registry. As such, this minor deviation from the specification is considered acceptable.

5.4.1 Key Generation

There are three commands for generating either symmetric keys or assymetric keypairs: `TPM2_Create`, `TPM2_CreateLoaded`, and `TPM2_CreatePrimary`. They each serve different purposes. `TPM2_CreatePrimary` creates a key as the root object of an object hierarchy in the TPM; `TPM2_Create` creates a child object in a hierarchy; `TPM2_CreateLoaded` also creates a child object and immediately loads the key into volatile memory.

All of these commands include a `TPMT_PUBLIC` data structure, shown in Table 12, as one of their parameters. This structure is responsible, among other things, for defining what algorithm is used for creating the object (`type` field), and additional data related to that algorithm (`parameters` field, whose structure is itself defined by `type`).

Table 12 – `TPMT_PUBLIC` structure

Fields		
Type	**Name**	**Description**
TPMI_ALG_PUBLIC	type	Algorithm associated with this object. Determines which subtype of `TPMU_PUBLIC_PARMS` and `TPMU_PUBLIC_ID` is used.
TPMI_ALG_HASH	nameAlg	Hash algorithm used to compute the Name of the object. `TPM_ALG_NULL` is permitted.
TPMA_OBJECT	objectAttributes	Attributes that, together with `type`, determine the allowed operations on this object.
TPM2B_DIGEST	authPolicy	Optional authorization policy for using the object. Empty buffer if no policy is present. Policy digest is computed using `nameAlg`.
TPMU_PUBLIC_PARMS	[type]parameters	Algorithm-specific parameters. This union is selected by `type`.
TPMU_PUBLIC_ID	[type]unique	Unique identifier for the structure.

Source: Trusted Computing Group (2025f), revised by the author.

Thus, in order to have Dilithium as an option for key generation, a `TPMI_ALG_PUBLIC` identifier value must first be attributed to this algorithm, so it can be referenced in the `type` field. The value for `TPM_ALG_DILITHIUM` was then defined as `0xB6`, given that it is one of the first available (i.e. unattributed and unreserved) values in the TCG Algorithm Registry (Trusted Computing Group, 2025b).

Secondly, a `TPMU_PUBLIC_PARMS` structure with supplementary information about Dilithium must also be defined, so it can be referenced in the `parameters` field. The `TPMU_PUBLIC_PARMS` type is a `union` of different structures that are selected based on the algorithm used, as shown in Table 13. For example, when using the RSA algorithm, the `TPMS_RSA_PARMS` structure shown in Table 14 is instanced, which includes both fields that are expected of any `*_PARMS` structure to function with the TPM 2.0 Library (`symmetric` and `scheme`), and those which are specific to the RSA algorithm (`keyBits` and `exponent`).

Table 13 – `TPMU_PUBLIC_PARMS` union

Member	Type	Selector
keyedHashDetail	TPMS_KEYEDHASH_PARMS	TPM_ALG_KEYEDHASH
symDetail	TPMS_SYMCIPHER_PARMS	TPM_ALG_SYMCIPHER
rsaDetail	TPMS_RSA_PARMS	TPM_ALG_RSA
eccDetail	TPMS_ECC_PARMS	TPM_ALG_ECC
asymDetail	TPMS_ASYM_PARMS	default selector

Source: Trusted Computing Group (2025f), revised by the author.

Table 14 – `TPMS_RSA_PARMS` structure (summary)

Parameter	Type	Description (summary)
symmetric	TPMT_SYM_DEF_OBJECT	Symmetric algorithm for restricted decryption keys; otherwise `TPM_ALG_NULL`.
scheme	TPMT_RSA_SCHEME	RSA signing/decryption scheme; `TPM_ALG_NULL` for unconstrained keys.
keyBits	TPMI_RSA_KEY_BITS	Length of the RSA modulus in bits.
exponent	UINT32	Public exponent (odd, > 2).

Source: Trusted Computing Group (2025f), revised by the author.

Likewise, the proposed new `TPMS_DILITHIUM_PARMS`, shown in Table 15, includes the fields inherent to the `*_PARMS` template, and a `securityLevel` field specifically for the Dilithium scheme. Hence, this new parameter, coupled with the Dilithium core's capability of having its security level defined at runtime (see Section 5.3) alllows the host to select between the three different security levels for performing Dilithium operations.

Table 15 – `TPMS_DILITHIUM_PARMS` structure (summary)

Parameter	Type	Description (summary)
symmetric	TPMT_SYM_DEF_OBJECT	Shall be `TPM_ALG_NULL` for Dilithium signing keys that are not parents.
scheme	TPMT_DILITHIUM_SCHEME	Must be `TPM_ALG_NULL`; kept for compatibility reasons.
securityLevel	UINT8	Indicates the Dilithium parameter set (must be either 2, 3, or 5).

Source: created by the author.

The new `TPMS_DILITHIUM_PARMS` structure is then included in the `union` definition of `TPMU_PUBLIC_PARMS`, so it can be used by TPM commands.

These changes are sufficient to integrate the new key generation operation into the overall TPM 2.0 Library architecure. In summary, the host sends a key generation command (e.g. `TPM2_Create`) that specifies its algorithm as the Dilithium signature scheme (`TPMT_PUBLIC.type = TPM_ALG_DILITHIUM`) along with its security level (`TPMT_PUBLIC.parameters.securityLevel = Union[2, 3, 5]`).

Internally, the Library interprets these parameters and must then invoke the routines that interact with the Dilithium accelerator to ultimately produce the keypair (see Subsection 5.5.2). The key objects also store their corresponding `TPMT_PUBLIC` instance, so it can be referenced later.

5.4.2 Sign

The TPM command used for performing signing operations is `TPM2_Sign`. However, as shown in Table 28 of Appendix B, this command has no parameter for the input message. Instead, it accepts a `digest` parameter, which is supposed to be the digest of the message after it has been through a CRHF, such as when using `TPM2_Hash`.

There are some reasons for this design. The first aspect to consider is that traditional asymmetric cryptographic schemes — such as signature schemes — are generally slower than its symmetric counterparts. Therefore, first hashing the message, and then performing the signature on the (generally smaller) digest instead, is a common practice for minimizing latency.

The second aspect to consider is that the maximum length of a TPM command is limited by the device's I/O buffer capacity, with common values on the order of 2 to 8 KiB. Therefore, the TPM Library must have a way of allowing for an arbitrarily large message (in particular, larger than the buffer capacity) to be ingested.

The solution adopted by the TPM 2.0 Library is to define `sequence` objects and commands. For example, for performing hash functions, one could use either the `TPM2_Hash` command, or a sequence composed of one `TPM2_HashSequenceStart` command, followed by N `TPM2_SequenceUpdate` commands, and a final `TPM2_SequenceComplete` command. The latter option creates a hash `sequence` object (`TPM2_HashSequenceStart`), that is updated over time as new chunks of the message are ingested (`TPM2_SequenceUpdate`), and finally returns the digest after the sequence is finished (`TPM2_SequenceComplete`).

Therefore, to avoid both the performance bottleneck of signing long messages, and the need to include additional `sequence` commands for signing operations, the preferred strategy of the TPM Library is to assume that `TPM2_Sign` acts on the digest of the message — although evidently, one could still provide the original message on the `digest` field, provided it is small enough.

Dilithium, however, works differently to other (traditional) signature schemes: it already performs a hash of the message M as a step of its `Sign` and `Verify` operations (see Appendix A). Thus, there is no real performance gain by hashing the message *before* invoking the signature scheme.

Consequently, in order to both strictly adhere to the Dilithium algorithm — i.e. its input is the message, and hashing happens internally — and avoid redundant hashing operations, the original approach of a TPM hash `sequence` followed by `TPM2_Sign` was not leveraged. As such, there are two alternatives for a new approach.

One alternative is to develop new `sequence` commands for sign operations, that perform both the hashing step and the signing step. These commands are then appropriate for more modern schemes, like Dilithium, that already include this joint two-step design.

The second alternative is to maintain this division between hashing and signing operations at the TPM command level, and to separate the execution of Dilithium (or any other similar scheme) in these two distinct steps. In this scenario, it would still be necessary to either develop a novel hash command, or modify the existing one — Dilithium actually does not hash M by itself; it hashes a concatenation that *includes* M, and this logic would need to be added to the hash command — but the existing `TPM2_Sign` would be kept unchanged.

Ultimately, the first alternative was preferred over the second one for performance reasons. To illustrate this, consider that the second alternative requires $N + 3$ commands for performing a signature: $N + 2$ commands for the hashing step, and 1 command for the signing step. The first alternative, on the other hand, by integrating both steps in the same command sequence, demands only $N + 2$ commands. That is, the communication overhead — and thus latency — is reduced in the first alternative.

A secondary, and less tangible, reason for this choice is that the first approach is less logically aligned with the overall structure of the TPM 2.0 Library. Modifying the TPM hash operations to allow for a concatenation between the input message and arbitrary internal data — in the case of Dilithium, the *tr* component of the secret key — seems to violate the simple scope of a hash command. Moreover, by doing so, internal data from the algorithm are exposed. While this by itself does not amount to a practical security risk in the case of Dilithium, there is also no advantage in doing so.

In light of these considerations, new TPM `sequence` commands were developed, that are responsible for performing both hashing and signing steps specified in Dilithium: `TPM2_HashSignStart` and `TPM2_HashSignFinish`. The formats for these commands were heavily inspired by the existing ones mentioned in this section — such as `TPM2_Sign`, `TPM2_HashSequenceStart`, and `TPM2_SequenceComplete` — which are all detailed in Appendix B.

The formats for the `TPM2_HashSignStart` command and response can be seen in Table 16 and Table 17, respectively.

Table 16 – `TPM2_HashSignStart` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_HashSignStart`.
Handles		
TPMI_DH_OBJECT	keyHandle	Handle of the private key used for signing.
Parameters		
UINT32	msgLen	Total message length in bytes (must be larger than zero).

Source: created by the author.

Table 17 – `TPM2_HashSignStart` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMI_DH_OBJECT	sequenceHandle	Handle for subsequent calls to `TPM2_SequenceUpdate` and `TPM2_HashSignFinish`.

Source: created by the author.

`TPM2_HashSignStart` has a `keyHandle` parameter, that references an existing private key loaded in the TPM. As mentioned, key objects already carry a `TPMT_PUBLIC` structure, so the corresponding signing algorithm and security level are easily determined.

The other input parameter is `msgLen`, which refers to the total size of the message to be signed. The need for this parameter does not arise from the TPM 2.0 Library, or the Dilithium algorithm itself, but due to the limitations of the hardware core as detailed in

Section 5.3. As such, if the core is later revised so that *a priori* knowledge of the message length is unnecessary, this parameter could be removed.

Following the TPM 2.0 Library's `sequence` paradigm, the `TPM2_HashSignStart` response returns a handle that can be used in later commands. This handle internally references an instance of a new `DLHS_STATE state` structure, shown in Table 18, that has information about the secret key (`keyHandle`), the remaining message bytes to be ingested (`remaining`), and a corresponding platform identifier for the operation (`ctx_id`).

This last `ctx_id` field, although not currently used, could be useful to segregate between different signing sessions. For example, if the host asks for a second message to be signed before finalizing the command sequence for the first message, these sequences would use different `ctx_id`s. The underlying platform would then need to have a mechanism for switching session contexts (i.e. storing data from one, and loading data from the other).

Table 18 – `DLHS_STATE` structure

Field	Type	Description
ctx_id	UINT32	Platform-specific context identifier for the signing sequence.
remaining	UINT32	Number of message bytes left in this sequence.
keyHandle	TPMI_DH_OBJECT	Private key handle associated with this sequence.

Source: created by the author.

After ingesting the message using a sequence of `TPM2_SequenceUpdate` commands, the host signals the sequence is finished using the new `TPM2_HashSignFinish` command. As shown in Table 19 and Table 20, this command has a `sequenceHandle` input parameter (the same handle as in the `TPM2_HashSignStart` response), and returns the signature.

Table 19 – `TPM2_HashSignFinish` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_HashSignFinish`
Handles		
TPMI_DH_OBJECT	sequenceHandle	Handle for the signing `sequence`.

Source: created by the author.

Table 20 – `TPM2_HashSignFinish` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMT_SIGNATURE	signature	Signature produced by the sequence.

Source: created by the author.

These changes are thus sufficient for integrating the Dilithium signature operation into the TPM 2.0 Library architecture.

5.4.3 Verify

Lastly, for incorporating Dilithium's verify operation into the TPM 2.0 Library, another two `sequence` commands were created, namely `TPM2_HashVerifyStart` and `TPM2_HashVerifyFinish`. The need for these commands mirrors the need for the ones in Subsection 5.4.2. In summary: the original `TPM2_VerifySignature` command, as seen in Table 34 of Appendix B, is built on the assumption of using the digest of the original message; Dilithium already internally hashes the message during `Verify` operation; changing the original hash `sequence` commands to accommodate for this specific use case is an overreach of its intended scope.

The only additional caveat to be considered for this operation is that the original `TPM2_VerifySignature` returns a `validation` ticket that can be used at a later time to prove that the TPM has previously verified this message. Notably, `validation` is a Hash-based Message Authentication Code (HMAC) that takes the digest of the message as one of its input. As such, the `HashVerify` sequence must still calculate this digest, in order to maintain consistency with existing TPM functionalities that use such tickets.

That is to say, the `HashVerify` sequence calculates two *different* digests — one for the `validation` ticket, one internally in Dilithium — and it could then be argued that this solution does not boast a performance benefit on par with the one in Subsection 5.4.2; it does not completely eliminate a (as in the case of `HashSign`, redundant) hash operation.

Nonetheless, this solution still has the remaining positives aspects detailed in Subsection 5.4.2: it strictly adheres to the Dilithium algorithm, it demands only $N + 2$ commands instead of $N + 3$, and it keeps the original hash `sequence` commands unchanged. The formats for the `TPM2_HashVerifyStart` command and response can be seen in Table 21 and Table 22, respectively.

Table 21 – `TPM2_HashVerifyStart` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_HashVerifyStart`
Handles		
TPMI_DH_OBJECT	keyHandle	Handle of the public key used for signature verification.
Parameters		
UINT32	msgLen	Total message length in bytes (must be larger than 0).
TPMT_SIGNATURE	signature	Signature to be verified.

Source: created by the author.

Table 22 – `TPM2_HashVerifyStart` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMI_DH_OBJECT	sequenceHandle	Handle for the verify `sequence`.

Source: created by the author.

`TPM2_HashVerifyStart`, similarly to `TPM2_HashSignStart`, also has a `msgLen` and a `keyHandle` parameter; the only difference being that `keyHandle` references a public key (instead of a private one) that will be used for validating the signature. The signature itself is also a parameter of this command. Note that `signature`, due to its bounded size, can be ingested in the `TPM2_HashVerifyStart` command, but the message will be ingested over the following `TPM2_SequenceUpdate` commands.

`TPM2_HashVerifyStart` responds with a `sequenceHandle` that will be used in subsequent commands of the sequence. This handle also internally references a `state` instance, which is implemented as the new `DLHV_STATE` structure defined in Table 23. This structure is very similar to the one used in the new signing sequences (`DLHS_STATE`), with two additional fields (`ticketHashAlg` and `ticketHash`) to account for the `verification` ticket previously mentioned.

Table 23 – `DLHV_STATE` structure

Field	Type	Description
ctx_id	UINT32	Platform-specific context identifier for the verify sequence.
remaining	UINT32	Number of message bytes left in this sequence.
keyHandle	TPMI_DH_OBJECT	Public key handle associated with this sequence.
ticketHashAlg	TPMI_ALG_HASH	Hash algorithm used for producing the `verification` ticket.
ticketHash	HASH_STATE	Rolling hash state over the message to generate the `verification` ticket.

Source: created by the author.

Finally, the formats for the `TPM2_HashVerifyFinish` command and response are shown in Table 24 and Table 25. At the end of a sequence of `TPM2_SequenceUpdate` commands, this new command is sent with a `sequenceHandle` input parameter (the same handle as in `TPM2_HashVerifyStart`), and returns the `validation` ticket as an output parameter, along with the result of the operation in its header. A `responseCode` of `TPM_RC_SUCCESS` indicates a successful validation, while `TPM_RC_SIGNATURE` indicates an incorrect one.

Table 24 – `TPM2_HashVerifyFinish` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_HashVerifyFinish`.
Handles		
TPMI_DH_OBJECT	sequenceHandle	Handle of the Dilithium verify sequence.

Source: created by the author.

These last changes — along with the necessary *marshaling* and *unmarshaling* routines for parsing all the new commands, parameters, and identifiers created — conclude all the modifications and inclusions made to the TPM 2.0 Library in order to perform the three Dilithium operations. In view of that, focus is now given to how the TPM 2.0 Library was actually implemented in the Petalite SoC.

Table 25 – `TPM2_HashVerifyFinish` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMT_TK_VERIFIED	validation	Verification ticket for the Dilithium signature (same semantics as `TPM2_VerifySignature` on success).

Source: created by the author.

5.5 Firmware

As discussed in Section 4.2, the TPM 2.0 Library is best implemented in the form of a bare-metal firmware that is executed in the SoC's CPU. There are essentially two alternatives for developing such a firmware: either fully building a novel implementation, or adapting an existing one, such as the ones mentioned in Section 2.8.

The implicit trade-off is between the time it takes to make an existing implementation compatible with the target platform, and the (probably much longer) time it takes to both completely develop and validate a novel implementation, but potentially have it optimized for the target platform.

In this project, the choice was made for adapting the reference software implementation of the TPM 2.0 Library[15] into a firmware. The reasons for that are:

a) developing and validating a TPM 2.0 Library implementation can take dedicated teams of engineers several months to complete, and is thus unfeasible for the scope and timeline of this project;

b) the target platform — that is, the Petalite SoC — is not critically restrained of resources to an extent that makes an optimized implementation of the TPM 2.0 Library crucial.

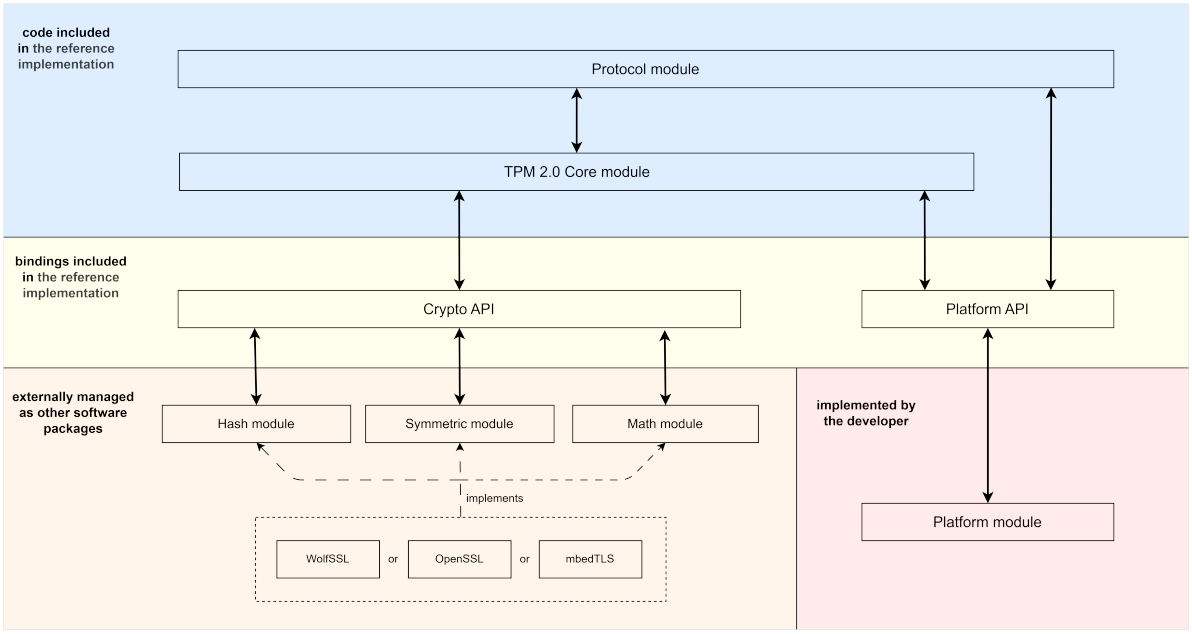
As regards choosing the implementation to adapt, the reference software implementation (hereafter named the *tcg-tpm*, due to its maintainer) had some notable advantages over its alternatives. First, differently from swtpm — which uses standard filesystem artifacts for storing objects and even the state of the TPM — tcg-tpm uses almost no functionalities of an (assumed to exist) underlying OS. Secondly, it is actively maintained

[15] <https://github.com/TrustedComputingGroup/TPM>

by a reliable source, the Trusted Computing Group itself, and thus receives frequent updates almost in sync with new versions of the specification.

The tcg-tpm is an implementation completely written in the C programming language, and as shown in Figure 17, it is divided into different modules that communicate via APIs. As a result, a module can be customized or even completely substituted and, as long as it honors its API, maintain the same overall functionality for the system. This greatly facilitates developing modifications.

Figure 17 – Architecture diagram for the reference TPM 2.0 Library implementation.



Source: created by the author.

The **Protocol module** is the layer responsible for interacting with the host, largely by receiving TPM commands and sending back responses. As such, it must access both the underlying device's I/O functionalities via the **Platform API**, and also the **TPM 2.0 Core module** for actually executing the commands.

The **TPM 2.0 Core module** is responsible for parsing parameters, loading objects, and invoking the necessary cryptographic functions. It does this last task by interfacing with separate cryptographic modules — the **Hash module**, the **Symmetric module**, and the **Math module** — via the **Crypto API**.

These cryptographic modules themselves are not a part of the implementation, and are instead meant to leverage existing cryptographic software libraries. In theory, the original repository supports build targets using either OpenSSL[16] or wolfSSL[17], but any

[16] <https://github.com/openssl/openssl>
[17] <https://github.com/wolfSSL/wolfssl>

other library with equivalent capabilities should suffice.

Finally, the **Platform module** is intended to provide a Hardware Abstraction Layer (HAL) that allows other modules to access functionalities such as non-volatile storage, I/O, power-down and reset signaling, among others.

Considering that the implementation was originally intended to be executed as a software process in a general-purpose OS, there are five principal tasks required for adapting tcg-tpm for this project:

a) Developing a new **Platform module** that correctly interfaces with the underlying SoC and provides all functions defined in the **Platform API**.

b) Substituting the current **Protocol module** (which uses TCP communication) for one that follows the UART protocol detailed in Subsection 5.2.2.

c) Verifying that no module depends on OS routines that would be unavailable in a baremetal scenario. Such routines, if they cannot be completely eliminated, should be substituted for calls to the **Platform module**.

d) Modifying the **TPM 2.0 Core module** as described in Section 5.4.

e) Including additional routines in the **Platform module** that allow for interfacing with the Dilithium hardware accelerator.

These changes were successfully incorporated, and some are detailed in the following sections. Other additions were also made to the firmware, such as a Log submodule that interacts with a second UART core to stream debug messages when simulating the SoC. The complete modified TPM 2.0 Library firmware is publicly available online[18].

5.5.1 Cryptographic library

As referenced earlier, the original repository has bindings for either OpenSSL or wolfSSL. For this project, OpenSSL is not a viable option: it is far too large for embedded firmware (the library size is on the order of several MiBs), and it assumes the existence of a general-purpose OS that can be invoked for functionalities like generating randomness, allocating dynamic memory, and creating threads.

Therefore, wolfSSL — a cryptographic library specifically optimized for embedded use cases — would seem like the obvious alternative choice. wolfSSL selectively includes only the requested cryptographic schemes in its final binary file, has options for avoiding dynamic memory allocation, and allows for deferring randomness to an arbitrary function call defined by the developer. In practice, however, wolfSSL bindings in tcg-tpm are "provided in a best-effort basis", and compilation fails when trying to use it.

[18] <https://github.com/franos-cm/pq-tpm>

Nevertheless, wolfSSL provides a compatibility layer to OpenSSL. It provides bindings for (most) OpenSSL functions, and can then be used as a 'back-end' for OpenSSL function calls. This feature would be sufficient for most projects, but tcg-tpm violates the OpenSSL API by interacting directly with some of its *opaque* objects; that is, structures whose internal layout the invoker should not depend on, since they are not guaranteed to be maintained throughout different versions of OpenSSL. As wolfSSL's internal representation of these objects does not match the ones in OpenSSL, the tcg-tpm cannot effectively use this compatibility feature.

Consequently, it was easier to build upon the original (nonfunctional) wolfSSL bindings, and modify them until the firmware could be successfully compiled. This was eventually completed, and it can thus be said that this version of the tcg-tpm fixes a considerable limitation of the original, allowing for the official reference implementation to be more easily tested in embedded platforms.

Finally, it is important to note that, as required by the TPM 2.0 Library, wolfSSL supports performing all of its cryptographic operations in constant time, in order to mitigate possible side-channel vulnerabilities related to latency.

5.5.2 Hardware abstraction layer

As already established, tcg-tpm's **Platform module** needed to be completely re-implemented, taking the underlying SoC platform into consideration. This module includes routines such as `GetEntropy()`, `Signal_PowerOn()`, and `NvMemoryWrite()`.

Fortunately, LiteX already provides a basic HAL. It automatically generates C functions for memory-mapped accesses to the CSR registers of all the SoC's cores (see Subsection 5.2.1). These functions can then be leveraged to build more complex functionalities. For example, given the TRNG detailed in Subsection 5.2.3, LiteX generates functions such as `trng_ctl_write()` or `trng_rand_read()`, which were then used to build higher-level abstractions such as `trng_enable()` or `trng_read_n_bytes()`.

Considering the scope of this project, it is especially important to note the HAL functions developed for interacting with the Dilithium accelerator. As illustrated in Figure 8 of Section 5.2, the accelerator has a CSR interface for runtime configuration, and a Wishbone interface for streaming data. This Wishbone interface, however, is driven by DMA engines, which are themselves configured through CSR registers.

Hence, higher-level HAL functions were developed for Dilithium that, briefly:

a) Get entropy from the TRNG, if necessary (as is the case for `KeyGen` operations);

b) Reset the core and configure its CSR registers according to the security level and operation requested, when beginning a new operation;

c) Configure the DMAs to read from the input buffers and write to the output buffers, when either are applicable;

d) Start the Dilithium core and the DMA engines;

e) Wait for the operation to finish.

An example of one of these HAL functions, specifically for the case of the `Keygen` operation, is shown in Appendix C and largely follows the sequence above.

After implementing all the HAL functions required by the TPM 2.0 Library — and also the novel ones developed for Dilithium — the firmware was successfully compiled and linked, producing the final binary data that must be loaded into the SoC's boot ROM. Thus, having completed the development of all the hardware and software necessary for the TPM, the next step was to validate it.

5.6 Test suite

The TPM 2.0 Library does not specify a method for validating the functionalities of a TPM device. That is, there is no test suite that simulates host behaviour in order to ascertain if the correct responses are returned to any command sent. The closest equivalent to that are the certification programs provided by the TCG, exclusively for its member organizations[19].

Nevertheless, the TPM 2.0 Library defines a self-test routine that the device must perform during its initial start-up (i.e. after receiving a `TPM2_Startup` command) or after having it specifically requested through a `TPM2_SelfTest` command. This routine executes basic cryptographic schemes in the TPM — CRHFs, HMACs, AES, RSA, etc. — using Known Answer Tests (KATs) to validate their functional behaviour. Failing these tests puts the TPM in a fatal error state. This self-test routine is present in the reference implementation of the Library, and is thus replicated in the revised version developed.

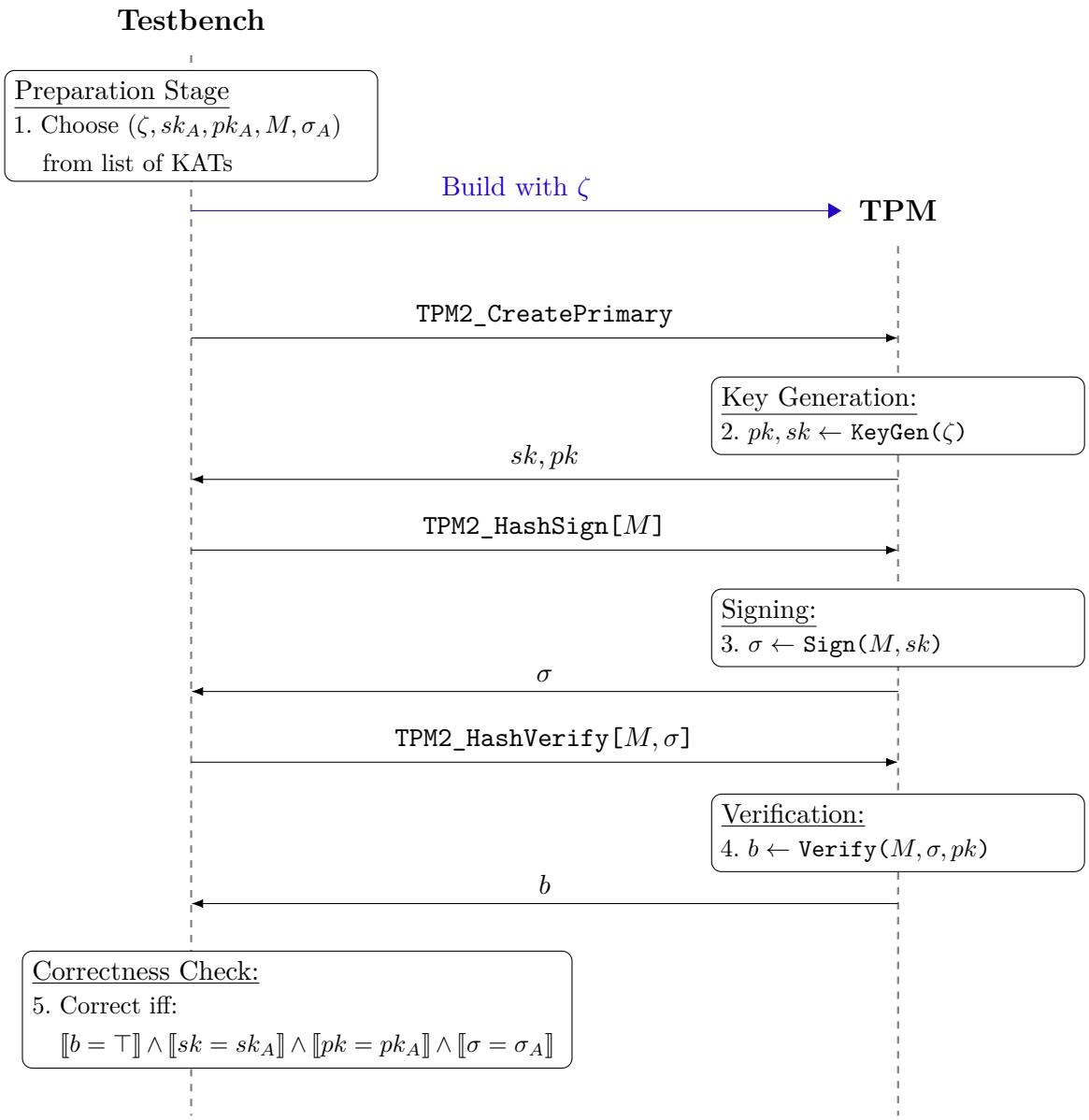
Conversely, in order to test the correctness of the new Dilithium operations, a simple automated test suite was developed that uses the commands described in Section 5.4. This test suite includes two alternative procedures, but both perform a complete `Keygen-Sign-Verify` series of commands: first, a keypair (sk, pk) is generated from a seed ζ; then, the secret key sk is used to sign a message M; finally, the public key pk is used to check the correspondence between the the message and the signature σ.

The first procedure, shown in Figure 18, uses the KAT vectors provided by NIST for the Dilithium algorithm (see Section 2.7). This allows for an in-depth functional validation of the implementation, by comparing the expected values for (sk, pk, σ) with the ones obtained by the TPM. This, however, demands building a modified firmware image that

[19] <https://trustedcomputinggroup.org/membership/certification/>

is only used in this procedure: ζ needs to be supplied at build-time so the TPM can use it instead of the TRNG, and the secret key needs to be leaked as plaintext to the host, which is not allowed by the TPM 2.0 Library.

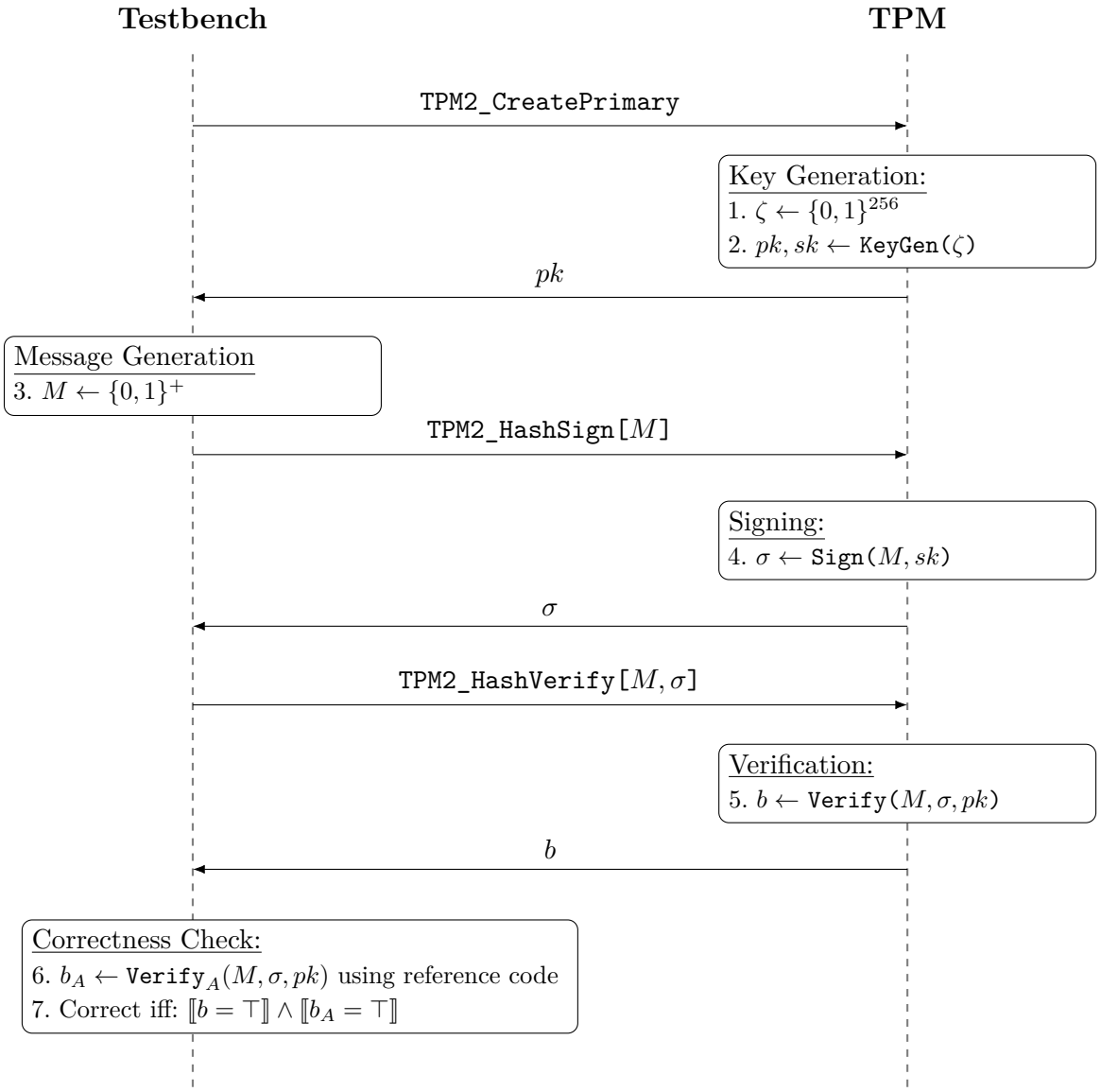
Figure 18 – Testing procedure for the Dilithium commands using KATs from NIST.



Source: created by the author.

The second procedure is a complementary validation using the final version of the firmware (instead of a modified test version, as in the first procedure). The procedure sends TPM commands to sign a random message M, and then verifies if the CRYSTALS reference implementation of the Dilithium algorithm also accepts the signature σ. This thus demonstrates that the signatures produced by the TPM for arbitrary messages are also valid signatures. This is illustrated in Figure 19.

Figure 19 – Testing procedure for the Dilithium commands using the Dilithium reference implementation.



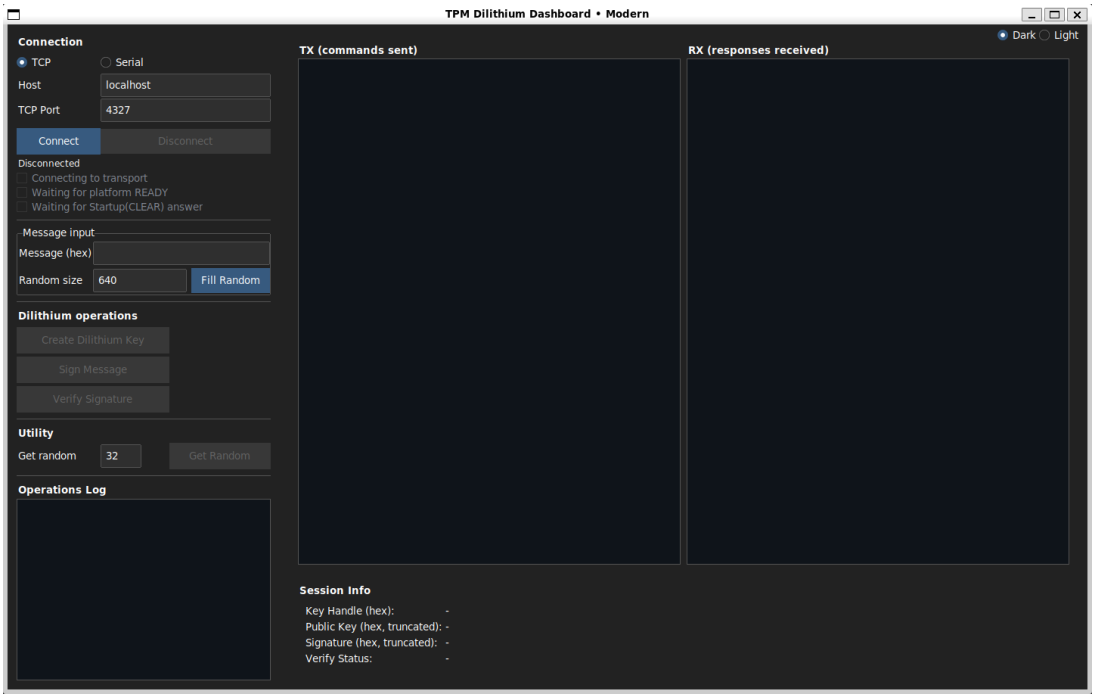
Source: created by the author.

This test suite was both simulated in software and later synthesized in the FPGA development board. When simulating the TPM, communication with the host happened via a simulated UART using TCP, as described in Subsection 5.2.2; when using the FPGA, the UART signals were transported over a USB interface to the external host platform.

In both cases, the TPM successfully performed the Dilithium algorithm in all tests conducted, using all three security levels. As a result, the implementation detailed was considered functionally valid.

Finally, in addition to the automated test suite, an interactive dashboard was also developed that allows for examining the commands and responses more visually. This dashboard can be seen in Figure 20.

Figure 20 – Dashboard for interactively testing the TPM.



Source: created by the author.

5.7 Performance metrics

Table 26 – Cycle latency for the new TPM Dilithium (security level 3) commands, implemented both in hardware and software.

	TPM 2.0 command sequence		
Implementation	**CreatePrimary**	**HashSign**	**HashVerify**
Reference software	14,738,219	107,385,854	11,891,749
Hardware accelerated	5,679,331	880,151	1,118,290
Speedup (%)	**259.5%**	**12,200.8%**	**1,063.4%**

Source: created by the author.

6 Conclusions

This final section details the conclusions drawn from the development of the project. This section has not yet been written, as it us out of the scope of this document.

Bibliography

AJTAI, M. Generating hard instances of lattice problems (extended abstract). In: *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1996. (STOC '96), p. 99–108. ISBN 0897917855. Disponível em: <https://doi.org/10.1145/237814.237838>. Citado na página 31.

AJTAI, M.; DWORK, C. A public-key cryptosystem with worst-case/average-case equivalence. In: *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*. New York, NY, USA: Association for Computing Machinery, 1997. (STOC '97), p. 284–293. ISBN 0897918886. Disponível em: <https://doi.org/10.1145/258533.258604>. Citado na página 31.

AMD. *7 Series Product Tables and Product Selection Guide (XMP101)*. [S.l.], 2021. Document ID: XMP101. Disponível em: <https://docs.amd.com/v/u/en-US/7-series-product-selection-guide>. Citado na página 44.

AMID, A. et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, v. 40, n. 4, p. 10–21, 2020. ISSN 1937-4143. Citado na página 45.

ASANOVIĆ, K. et al. *The Rocket Chip Generator*. [S.l.], 2016. Disponível em: <https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-17.pdf>. Citado na página 48.

BAI, S. et al. *CRYSTALS-Dilithium: Algorithm Specifications and Supporting Documentation (Version 3.1)*. 2021. <https://pq-crystals.org/dilithium/>. Submission to the NIST Post-Quantum Cryptography Standardization Project, Round 3. Citado 5 vezes nas páginas 33, 34, 41, 103, and 104.

BANERJEE, U.; UKYAB, T. S.; CHANDRAKASAN, A. P. *Sapphire: A Configurable Crypto-Processor for Post-Quantum Lattice-based Protocols (Extended Version)*. 2019. Cryptology ePrint Archive, Paper 2019/1140. Disponível em: <https://eprint.iacr.org/2019/1140>. Citado na página 20.

BECKWITH, L.; NGUYEN, D. T.; GAJ, K. High-performance hardware implementation of crystals-dilithium. In: *2021 International Conference on Field-Programmable Technology (ICFPT)*. [S.l.: s.n.], 2021. p. 1–10. Citado 4 vezes nas páginas 54, 55, 59, and 61.

BECKWITH, L.; NGUYEN, D. T.; GAJ, K. Hardware accelerators for digital signature algorithms dilithium and falcon. *IEEE Design Test*, v. 41, n. 5, p. 27–35, 2024. Citado na página 19.

BERTONI, G. et al. *The Keccak Reference – Version 3.0*. [S.l.], 2011. Version 3.0; specification and design rationale of Keccak sponge functions. Disponível em: <https://keccak.team/files/Keccak-reference-3.0.pdf>. Citado na página 23.

BERTONI, G. et al. *Keccak Implementation Overview — Version 3.2*. [S.l.], 2012. Overview of software and hardware implementations of Keccak (including side-channel

protection). Disponível em: <https://keccak.team/files/Keccak-implementation-3.2.pdf>. Citado na página 63.

BOS, J. W.; RENES, J.; SPRENKELS, A. *Dilithium for Memory Constrained Devices*. 2022. Cryptology ePrint Archive, Paper 2022/323. Disponível em: <https://eprint.iacr.org/2022/323>. Citado na página 19.

DIFFIE, W.; HELLMAN, M. New directions in cryptography. *IEEE Transactions on Information Theory*, v. 22, n. 6, p. 644–654, 1976. Citado na página 25.

FARAHMAND, F. et al. Minerva: Automated hardware optimization tool. In: *2017 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*. [S.l.: s.n.], 2017. p. 1–8. Citado na página 57.

FIAT, A.; SHAMIR, A. How to prove yourself: Practical solutions to identification and signature problems. In: ODLYZKO, A. M. (Ed.). *Advances in Cryptology — CRYPTO' 86*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1987. p. 186–194. ISBN 978-3-540-47721-1. Citado na página 28.

FIOLHAIS, L.; SOUSA, L. *QR TPM in Programmable Low-Power Devices*. 2023. Disponível em: <https://arxiv.org/abs/2309.17414>. Citado na página 20.

GALBRAITH, S. *Mathematics of Public Key Cryptography*. [S.l.]: Cambridge University Press, 2012. ISBN 9781107013926. Citado na página 30.

GOLDREICH, O. The foundations of modern cryptography. In: _____. *Modern Cryptography, Probabilistic Proofs and Pseudorandomness*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999. p. 1–37. ISBN 978-3-662-12521-2. Disponível em: <https://doi.org/10.1007/978-3-662-12521-2_1>. Citado 3 vezes nas páginas 21, 22, and 23.

GOLDREICH, O.; GOLDWASSER, S.; HALEVI, S. *Collision-Free Hashing from Lattice Problems*. 1996. Cryptology ePrint Archive, Paper 1996/009. Disponível em: <https://eprint.iacr.org/1996/009>. Citado na página 31.

GUPTA, N.; JATI, A.; CHATTOPADHYAY, A. Crystals-dilithium on risc-v processor: Lightweight secure boot using post-quantum digital signature. In: *2023 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. [S.l.: s.n.], 2023. p. 1–7. Citado na página 20.

GUPTA, N. et al. *Lightweight Hardware Accelerator for Post-Quantum Digital Signature CRYSTALS-Dilithium*. 2022. Cryptology ePrint Archive, Paper 2022/496. Disponível em: <https://eprint.iacr.org/2022/496>. Citado na página 19.

IMAM, R. et al. Systematic and critical review of rsa based public key cryptographic schemes: Past and present status. *IEEE Access*, v. 9, p. 155949–155976, 2021. Citado na página 17.

Infineon Technologies AG. *OPTIGA TPM SLB 9670: Trusted Platform Module (TPM) 2.0*. 2021. Accessed: 2025-11-11. Disponível em: <https://www.infineon.com/dgdl/Infineon-SLB_9670_2.0-FactSheet-v01_00-EN.pdf>. Citado na página 18.

KERMARREC, F. et al. *LiteX: an open-source SoC builder and library based on Migen Python DSL*. 2020. Disponível em: <https://arxiv.org/abs/2005.02506>. Citado na página 45.

LAND, G.; SASDRICH, P.; GüNEYSU, T. *A Hard Crystal - Implementing Dilithium on Reconfigurable Hardware*. 2021. Cryptology ePrint Archive, Paper 2021/355. Disponível em: <https://eprint.iacr.org/2021/355>. Citado 2 vezes nas páginas 54 and 55.

LENSTRA, A.; LENSTRA, H.; LáSZLó, L. Factoring polynomials with rational coefficients. *Mathematische Annalen*, v. 261, 12 1982. Citado na página 30.

LYUBASHEVSKY, V. Fiat-shamir with aborts: Applications to lattice and factoring-based signatures. In: *Proceedings of the 15th International Conference on the Theory and Application of Cryptology and Information Security: Advances in Cryptology*. Berlin, Heidelberg: Springer-Verlag, 2009. (ASIACRYPT '09), p. 598–616. ISBN 9783642103650. Disponível em: <https://doi.org/10.1007/978-3-642-10366-7_35>. Citado na página 33.

LYUBASHEVSKY, V. *Basic Lattice Cryptography: The concepts behind Kyber (ML-KEM) and Dilithium (ML-DSA)*. 2024. Cryptology ePrint Archive, Paper 2024/1287. Disponível em: <https://eprint.iacr.org/2024/1287>. Citado 2 vezes nas páginas 31 and 33.

LYUBASHEVSKY, V.; PEIKERT, C.; REGEV, O. *On Ideal Lattices and Learning with Errors Over Rings*. 2012. Cryptology ePrint Archive, Paper 2012/230. Disponível em: <https://eprint.iacr.org/2012/230>. Citado 2 vezes nas páginas 31 and 33.

MARTíN-LóPEZ, E. et al. Experimental realization of shor's quantum factoring algorithm using qubit recycling. *Nature Photonics*, Springer Science and Business Media LLC, v. 6, n. 11, p. 773–776, out. 2012. ISSN 1749-4893. Disponível em: <http://dx.doi.org/10.1038/nphoton.2012.259>. Citado na página 17.

MICCIANCIO, D.; WALTER, M. *Fast Lattice Point Enumeration with Minimal Overhead*. 2014. Cryptology ePrint Archive, Paper 2014/569. Disponível em: <https://eprint.iacr.org/2014/569>. Citado na página 30.

Microchip Technology Inc. *AT97SC3205T Trusted Platform Module (TPM)*. 2018. Accessed: 2025-11-11. Disponível em: <https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-9629-Crypto-AT97SC3205T-Datasheet.pdf>. Citado na página 18.

MISHRA, M.; KAR, J. A study on diffie-hellman key exchange protocols. *International Journal of Pure and Apllied Mathematics*, v. 114, 05 2017. Citado na página 17.

MONTANARO, A. Quantum algorithms: an overview. *npj Quantum Information*, v. 2, p. 15023, 2016. Disponível em: <https://www.nature.com/articles/npjqi201523>. Citado 2 vezes nas páginas 28 and 29.

National Institute of Standards and Technology. *FIPS 202: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. [S.l.], 2015. Publication date: August 2015; DOI: 10.6028/NIST.FIPS.202. Disponível em: <https://csrc.nist.gov/pubs/fips/202/final>. Citado na página 23.

National Institute of Standards and Technology. *Post-Quantum Cryptography*. 2016. Access date: 18 mar. 2025. Disponível em: <https://csrc.nist.gov/projects/post-quantum-cryptography>. Citado na página 17.

OpenCores Organization. *WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores, Revision B4*. [S.l.], 2010. Accessed: 2025-11-14. Disponível em: <https://cdn.opencores.org/downloads/wbspec_b4.pdf>. Citado na página 48.

PARRILLA, L. et al. Revisiting multiple ring oscillator-based true random generators to achieve compact implementations on fpgas for cryptographic applications. *Cryptography*, v. 7, n. 2, 2023. ISSN 2410-387X. Disponível em: <https://www.mdpi.com/2410-387X/7/2/26>. Citado na página 51.

PETRISKO, D. et al. Blackparrot: An agile open-source risc-v multicore for accelerator socs. *IEEE Micro*, v. 40, n. 4, p. 93–102, 2020. Citado na página 48.

PRADHAN, P. K.; RAKSHIT, S.; DATTA, S. Lattice based cryptography : Its applications, areas of interest future scope. In: *2019 3rd International Conference on Computing Methodologies and Communication (ICCMC)*. [S.l.: s.n.], 2019. p. 988–993. Citado na página 17.

REGEV, O. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, Association for Computing Machinery, New York, NY, USA, v. 56, n. 6, set. 2009. ISSN 0004-5411. Disponível em: <https://doi.org/10.1145/1568318.1568324>. Citado na página 31.

REGEV, O. The learning with errors problem (invited survey). In: *2010 IEEE 25th Annual Conference on Computational Complexity*. [S.l.: s.n.], 2010. p. 191–204. Citado na página 32.

RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, Association for Computing Machinery, New York, NY, USA, v. 21, n. 2, p. 120–126, fev. 1978. ISSN 0001-0782. Disponível em: <https://doi.org/10.1145/359340.359342>. Citado 2 vezes nas páginas 25 and 27.

SCHNORR, C. P. Efficient identification and signatures for smart cards. In: BRASSARD, G. (Ed.). *Advances in Cryptology — CRYPTO' 89 Proceedings*. New York, NY: Springer New York, 1990. p. 239–252. ISBN 978-0-387-34805-6. Citado na página 27.

SHOR, P. W. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM Journal on Computing*, Society for Industrial Applied Mathematics (SIAM), v. 26, n. 5, p. 1484–1509, out. 1997. ISSN 1095-7111. Disponível em: <http://dx.doi.org/10.1137/S0097539795293172>. Citado 2 vezes nas páginas 17 and 28.

STMicroelectronics. *ST33TPM Trusted Platform Module*. 2021. Accessed: 2025-11-11. Disponível em: <https://www.st.com/resource/en/datasheet/st33htph2x.pdf>. Citado na página 18.

Trusted Computing Group. *PC Client Platform TPM Profile Specification for TPM 2.0, Version 1.06*. [S.l.], 2025. Published April 30, 2025. © TCG 2025. Disponível em: <https://trustedcomputinggroup.org/wp-content/uploads/PC-Client-Specific-Platform-TPM-Profile-for-TPM-2p0-Version-1p06_pub.pdf>. Citado na página 49.

Trusted Computing Group. *TCG Algorithm Registry*. [S.l.], 2025. Disponível em: <https://trustedcomputinggroup.org/wp-content/uploads/TCG-Algorithm-Registry-Version-2.0_pub.pdf>. Citado 2 vezes nas páginas 65 and 66.

Trusted Computing Group. *TPM Library Specification – Parts 1 to 4, Specification Version 2.0, Version 184*. [S.l.], 2025. Disponível em: <https://trustedcomputinggroup.org/resource/tpm-library-specification/>. Citado 4 vezes nas páginas 18, 35, 41, and 93.

Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 0: Introduction*. [S.l.], 2025. Disponível em: <file:./Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf>. Citado 2 vezes nas páginas 34 and 37.

Trusted Computing Group. *Trusted Platform Module Library Part 1: Architecture*. [S.l.], 2025. Disponível em: <https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-1-Version-184_pub.pdf>. Citado 4 vezes nas páginas 35, 36, 49, and 50.

Trusted Computing Group. *Trusted Platform Module Library, Part 2: Structures*. [S.l.], 2025. Disponível em: <https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-2-Version-184_pub.pdf>. Citado 2 vezes nas páginas 66 and 67.

Trusted Computing Group. *Trusted Platform Module Library, Part 3: Commands*. [S.l.], 2025. Disponível em: <https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-3-Version-184_pub.pdf>. Citado 5 vezes nas páginas 37, 95, 96, 97, and 98.

Trusted Computing Group. *What is post quantum cryptography, and how is TCG implementing it?* 2025. Web page. Accessed: 2025-11-11. Disponível em: <https://trustedcomputinggroup.org/what-is-post-quantum-cryptography-and-how-is-tcg-implementing-it/>. Citado na página 18.

VANDERSYPEN, L. M. K. et al. Experimental realization of shor's quantum factoring algorithm using nuclear magnetic resonance. *Nature*, Springer Science and Business Media LLC, v. 414, n. 6866, p. 883–887, dez. 2001. ISSN 1476-4687. Disponível em: <http://dx.doi.org/10.1038/414883a>. Citado na página 17.

VASYLTSOV, I. et al. Fast digital trng based on metastable ring oscillator. In: OSWALD, E.; ROHATGI, P. (Ed.). *Cryptographic Hardware and Embedded Systems – CHES 2008*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008. p. 164–180. ISBN 978-3-540-85053-3. Citado 2 vezes nas páginas 52 and 53.

WANG, T. et al. Efficient implementation of dilithium signature scheme on fpga soc platform. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, v. 30, n. 9, p. 1158–1171, 2022. Citado 2 vezes nas páginas 20 and 56.

WANG, T. et al. Optimized hardware-software co-design for kyber and dilithium on risc-v soc fpga. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, v. 2024, n. 3, p. 99–135, Jul. 2024. Disponível em: <https://tches.iacr.org/index.php/TCHES/article/view/11671>. Citado 3 vezes nas páginas 19, 20, and 54.

ZHAO, C. et al. A compact and high-performance hardware architecture for crystals-dilithium. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, v. 2022, p. 270–295, 11 2021. Citado na página 19.

ZILBERMAN, N. et al. Netfpga sume: Toward 100 gbps as research commodity. *IEEE Micro*, v. 34, n. 5, p. 32–41, 2014. Citado 2 vezes nas páginas 43 and 44.

Appendix

APPENDIX A – TPM supported algorithms

Table 27 – Supported algorithms defined in the TPM2.0 Library Specification.

Primitive / Family	**Supported Algorithms (TPM 2.0 Library)**
Hash Functions	TPM_ALG_SHA1 TPM_ALG_SHA256 TPM_ALG_SHA384 TPM_ALG_SHA512 TPM_ALG_SM3_256 TPM_ALG_SHA3_256 TPM_ALG_SHA3_384 TPM_ALG_SHA3_512
Symmetric Ciphers	TPM_ALG_AES TPM_ALG_SM4 TPM_ALG_CAMELLIA
Symmetric Modes	TPM_ALG_CTR TPM_ALG_OFB TPM_ALG_CBC TPM_ALG_CFB TPM_ALG_ECB TPM_ALG_CCM TPM_ALG_GCM TPM_ALG_EAX TPM_ALG_KW TPM_ALG_KWP
MAC / KeyedHash	TPM_ALG_HMAC TPM_ALG_CMAC TPM_ALG_SMAC
RSA Family	TPM_ALG_RSA
RSA Signatures	TPM_ALG_RSASSA TPM_ALG_RSAPSS
RSA Encryption	TPM_ALG_RSAES TPM_ALG_OAEP
ECC Family	TPM_ALG_ECC
ECC Signatures	TPM_ALG_ECDSA TPM_ALG_ECDAA TPM_ALG_ECSCHNORR TPM_ALG_SM2 TPM_ALG_EDDSA TPM_ALG_EDDSA_PH
ECC Key Exchange	TPM_ALG_ECDH TPM_ALG_ECMQV
KeyedHash Object	TPM_ALG_KEYEDHASH TPM_ALG_XOR
KDF / MGF	TPM_ALG_MGF1 TPM_ALG_KDF1_SP800_56A TPM_ALG_KDF2 TPM_ALG_KDF1_SP800_108
Hash-Based Signatures	TPM_ALG_LMS TPM_ALG_XMSS
Structural	TPM_ALG_NULL TPM_ALG_XOR

Source: Trusted Computing Group (2025c), compiled by the author.

APPENDIX B – Format of relevant TPM 2.0 commands and responses

Table 28 – `TPM2_Sign` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_Sign`.
Handles		
TPMI_DH_OBJECT	keyHandle	Handle of key used for signing.
Parameters		
TPM2B_DIGEST	digest	Digest to be signed.
TPMT_SIG_SCHEME	inScheme	Signing scheme to use if the key's scheme is `TPM_ALG_NULL`.
TPMT_TK_HASHCHECK	validation	Ticket proving the digest came from the TPM; may be a NULL ticket for non-restricted keys.

Source: Trusted Computing Group (2025g), revised by the author.

Table 29 – `TPM2_Sign` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag (see Part 3, Clause 6).
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMT_SIGNATURE	signature	The resulting signature.

Source: Trusted Computing Group (2025g), revised by the author.

Table 30 – `TPM2_HashSequenceStart` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_HashSequenceStart`.
Parameters		
TPM2B_AUTH	auth	Authorization value for later use of this sequence.
TPMI_ALG_HASH	hashAlg	Hash algorithm to use; `TPM_ALG_NULL` starts an Event Sequence.

Source: Trusted Computing Group (2025g), revised by the author.

Table 31 – `TPM2_HashSequenceStart` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Handles		
TPMI_DH_OBJECT	sequenceHandle	Handle used to reference the newly created sequence.

Source: Trusted Computing Group (2025g), revised by the author.

Table 32 – `TPM2_SequenceUpdate` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_SequenceUpdate`.
Handles		
TPMI_DH_OBJECT	sequenceHandle	Handle for the sequence object.
Parameters		
TPM2B_MAX_BUFFER	buffer	Data to be added to the running hash sequence.

Source: Trusted Computing Group (2025g), revised by the author.

Table 33 – `TPM2_SequenceUpdate` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.

Source: Trusted Computing Group (2025g), revised by the author.

Table 34 – `TPM2_VerifySignature` command format

Header		
Type	**Name**	**Description**
TPMI_ST_COMMAND_TAG	tag	`TPM_ST_SESSIONS` if a session is present; otherwise `TPM_ST_NO_SESSIONS`.
UINT32	commandSize	Size of the command (bytes).
TPM_CC	commandCode	`TPM_CC_VerifySignature`.
Handles		
TPMI_DH_OBJECT	keyHandle	Handle of the public key used for signature validation.
Parameters		
TPM2B_DIGEST	digest	Digest of the signed message.
TPMT_SIGNATURE	signature	Signature to be tested.

Source: Trusted Computing Group (2025g), revised by the author.

Table 35 – `TPM2_VerifySignature` response format

Header		
Type	**Name**	**Description**
TPM_ST	tag	Response tag.
UINT32	responseSize	Size of the response (bytes).
TPM_RC	responseCode	TPM return code.
Parameters		
TPMT_TK_VERIFIED	validation	Ticket indicating the signature was successfully verified.

Source: Trusted Computing Group (2025g), revised by the author.

APPENDIX C – Example of a HAL high-level firmware function

```
uint32_t _plat__Dilithium_KeyGen(uint8_t sec_level,
                                 uint8_t *pk, uint16_t *pk_size,
                                 uint8_t *sk, uint16_t *sk_size)
{
    // Validate input arguments
    if (!(pk_size && pk && sk_size && sk)) return -1;

    // Get seed for generating keys
    if (_plat__GetEntropy(seed, DILITHIUM_SEED_SIZE) != (int)DILITHIUM_SEED_SIZE)
        return -2;

    // Generate keypair
    uint32_t rc = dilithium_keygen(sec_level, seed, pk, pk_size, sk, sk_size);

    // wipe seed information
    for (size_t i = 0; i < sizeof(seed); i++) ((volatile uint8_t *)seed)[i] = 0;
    return rc;
}

uint32_t dilithium_keygen(uint8_t sec_level, const uint8_t *seed,
                          uint8_t *pk, uint16_t *pk_size,
                          uint8_t *sk, uint16_t *sk_size)
{
    LOGD("Starting Keygen op...");
    uint8_t *keypair_addr = _dilithium_buffer_start;
    uint32_t keypair_size = get_pk_len(sec_level) + get_sk_len(sec_level) - DILITHIUM_RHO_SIZE;

    // Set Dilithium
    dilithium_setup(DILITHIUM_CMD_KEYGEN, sec_level);
    dilithium_reset();

    // Setup Dilithium DMA
    dilithium_write_setup(keypair_addr, (uint32_t)keypair_size);
    dilithium_write_start();
    dilithium_read_setup(seed, (uint32_t)DILITHIUM_SEED_SIZE);
    dilithium_read_start();

    // Start dilithium core and wait for it to finish responding
    dilithium_start();
    dilithium_read_wait();
    dilithium_write_wait();

    LOGD("Keygen done!");
    return pack_keypair(sec_level, keypair_addr, pk, pk_size, sk, sk_size);
}
```

Source: created by the author, available in <https://github.com/franos-cm/pq-tpm>

Annex

ANNEX A – Pseudo-code for Dilithium

Algorithm 3 – The complete pseudo-code for CRYSTALS-Dilithium

KeyGen()

$\zeta \leftarrow \{0,1\}^{256}$
$(\rho, \rho', K) \in \{0,1\}^{256} \times \{0,1\}^{512} \times \{0,1\}^{256} := H(\zeta)$ ▷ H is SHAKE256
$A \in \mathcal{R}_q^{k \times \ell} := \text{ExpandA}(\rho)$ ▷ Generated and stored in NTT representation as $\hat{A}$
$(s_1, s_2) \in S_\eta^\ell \times S_\eta^k := \text{ExpandS}(\rho')$
$t := As_1 + s_2$ ▷ Compute As_1 as $\text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(s_1))$
$(t_1, t_0) := \text{Power2Round}_q(t, d)$
$\text{tr} \in \{0,1\}^{256} := H(\rho \| t_1)$
return $(\text{pk} = (\rho, t_1),\ \text{sk} = (\rho, K, \text{tr}, s_1, s_2, t_0))$

Sign(sk, M)

$A \in \mathcal{R}_q^{k \times \ell} := \text{ExpandA}(\rho)$ ▷ Generated and stored in NTT representation as $\hat{A}$
$\mu \in \{0,1\}^{512} := H(\text{tr} \| M)$
$\kappa := 0, \quad (z, h) :=\perp$
$\rho' \in \{0,1\}^{512} := H(K \| \mu)$ ▷ or $\rho' \leftarrow \{0,1\}^{512}$ for randomized signing
while $(z, h) =\perp$ **do** ▷ Precompute $\hat{s}_1 := \text{NTT}(s_1)$, $\hat{s}_2 := \text{NTT}(s_2)$, $\hat{t}_0 := \text{NTT}(t_0)$
  $y \in \tilde{S}_{\gamma_1}^\ell := \text{ExpandMask}(\rho', \kappa)$
  $w := Ay$ ▷ $w := \text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(y))$
  $w_1 := \text{HighBits}_q(w,\ 2\gamma_2)$
  $\tilde{c} \in \{0,1\}^{256} := H(\mu \| w_1)$
  $c \in \mathcal{B}_\tau := \text{SampleInBall}(\tilde{c})$ ▷ Store c in NTT representation as $\hat{c} := \text{NTT}(c)$
  $z := y + cs_1$ ▷ Compute $cs_1 := \text{NTT}^{-1}(\hat{c} \cdot \hat{s}_1)$
  $r_0 := \text{LowBits}_q(w - cs_2,\ 2\gamma_2)$ ▷ Compute $cs_2 := \text{NTT}^{-1}(\hat{c} \cdot \hat{s}_2)$
  if $\|z\|_\infty \geq \gamma_1 - \beta$ **or** $\|r_0\|_\infty \geq \gamma_2 - \beta$ **then**
    $(z, h) :=\perp$
  else
    $h := \text{MakeHint}_q(-ct_0,\ w - cs_2 + ct_0,\ 2\gamma_2)$ ▷ Compute $ct_0 := \text{NTT}^{-1}(\hat{c} \cdot \hat{t}_0)$
    if $\|ct_0\|_\infty \geq \gamma_2$ **or** # of 1's in $h > \omega$ **then**
      $(z, h) :=\perp$
    end if
  end if
  $\kappa := \kappa + 1$
end while
return $\sigma = (\tilde{c}, z, h)$

Verify(pk, $M, \sigma = (\tilde{c}, z, h)$)

$A \in R_q^{k \times \ell} := \text{ExpandA}(\rho)$ ▷ Generated and stored in NTT representation as $\hat{A}$
$\mu \in \{0,1\}^{512} := H(H(\rho \| t_1) \| M)$
$c := \text{SampleInBall}(\tilde{c})$
$w_1' := \text{UseHint}_q(h,\ Az - ct_1 \cdot 2^d,\ 2\gamma_2)$ ▷ $\text{NTT}^{-1}(\hat{A} \cdot \text{NTT}(z) - \text{NTT}(c) \cdot \text{NTT}(t_1 \cdot 2^d))$
return $[\![\|z\|_\infty < \gamma_1 - \beta]\!] \wedge [\![\tilde{c} = H(\mu \| w_1')]\!] \wedge [\![\#1s(h) \leq \omega]\!]$

Source: Bai et al. (2021)

Algorithm 4 – Supporting functions used in Algorithm 3

Power2Round$_q(r, d)$

$r := r \bmod^+ q$
$r_0 := r \bmod^\pm 2^d$
return $\left(\frac{r-r_0}{2^d},\ r_0\right)$

Decompose$_q(r, \alpha)$

$r := r \bmod^+ q$
$r_0 := r \bmod^\pm \alpha$
if $r - r_0 = q - 1$ **then**
    $r_1 := 0$
    $r_0 := r_0 - 1$
else
    $r_1 := \frac{r-r_0}{\alpha}$
end if
return (r_1, r_0)

SampleInBall(ρ)

Initialize $c = c_0 c_1 \dots c_{255} = 00 \dots 0$
for $i := 256 - \tau$ **to** 255 **do**
    $j \leftarrow \{0, 1, \dots, i\}$
    $s \leftarrow \{0, 1\}$
    $c_i := c_j$
    $c_j := (-1)^s$
end for
return c

MakeHint$_q(z, r, \alpha)$

$r_1 := \text{HighBits}_q(r, \alpha)$
$v_1 := \text{HighBits}_q(r + z, \alpha)$
return $[\![r_1 \neq v_1]\!]$

UseHint$_q(h, r, \alpha)$

$m := \frac{q-1}{\alpha}$
$(r_1, r_0) := \text{Decompose}_q(r, \alpha)$
if $h = 1$ **then**
    if $r_0 > 0$ **then**
        return $(r_1 + 1) \bmod^+ m$
    else
        return $(r_1 - 1) \bmod^+ m$
    end if
else
    return r_1
end if

HighBits$_q(r, \alpha)$

$(r_1, \_) := \text{Decompose}_q(r, \alpha)$
return r_1

LowBits$_q(r, \alpha)$

$(\_, r_0) := \text{Decompose}_q(r, \alpha)$
return r_0

Source: Bai et al. (2021)