POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Post-quantum algorithms support in Trusted Execution Environment

Supervisor
prof. Antonio Lioy

Candidate
Giuseppe Caruso

Academic year 2023-2024

Alla mia famiglia

† Ai miei nonni Peppe ed Ida

Summary

The scope of the thesis' work is to extend the algorithms' support in Trusted Execution Environments, to include a PQ algorithm, in such a way that the algorithm can be deployed to provide a quantum-safe system that can be relied upon to perform sensitive tasks. Particular attention is paid to device with limited capabilities, since such solution aims to the adoption of the quantum-safe alternative within devices in which several contraints further complicate the deployment of a PQ solution.

In the first chapter presented are discussed the security properties that a Trusted Execution Environment offers, along with its components, features, and security requirements. Moreover, the Keystone framework, core of the thesis' work, is described. The second chapter is used to present the characteristics of the Device Identifier Composition Engine specification, adopted to enable the definition of a strong device identity by means of software techniques and minimal silicon requirements in devices in which hardware used to provide such feature is not available, with particular reference to the heavy usage of digital signature in the layered system characterising DICE. The work proceeds then with a deep analysis on the quantum impact that quantum computers pose on the current cryptographic technologies, evaluating the impact on the public-key cryptosystems, symmetric-key cryptosystems, and hash functions.

The thesis work is then described in the subsequent chapters, starting from the fifth chapter in which the solutions' design is addressed, evaluating the Post-Quantum submissions selected for standardization in relation with the ARM Cortex-M4 platform, and analysing Falcon, the selected algorithm among the submissions. Then, the identified quantum issues in the DICE-based Keystone version are presented, underlining the issues that affect the framework from a quantum-safe perspective. Finally, in the sixth chapter are presented the changes and the features introduced to provide a quantum-safe alternative to be deployed, which is further analysed under the functional and performance points of view in the seventh chapter, used to showcase various tests performed on the software.

Acknowledgements

I would like to thank my family for all the support given to me in this academic journey, my father Rocco for his patience, advice and help, my mother Lucia for the comfort, love and availability, my sister Ida for the laughter, the events we experienced, the advice and the support given to me during these years of studying in Turin.

I would also like to thank all my friends, both those I met during my university career and previous ones, with whom I have shared wonderful moments of leisure and fun.

List of Figures

List of Tables

Listings

Contents

Chapter 1

Introduction

Nowadays, significant attention is being given to the topic of quantum computing, alternative information processing approach based on quantum computers' ability to generate and manipulate quantum bits (in short qubits, subatomic particles) in order to perform operations [7] [8]. Two fundamental properties characterize qubits: *Superposition* and *Entanglement*. Superposition refers to the ability of a qubit to exist in a combination of several possible configurations, allowing groups of qubits in superposition to create complex computational spaces. Entanglement occurs when two qubits are paired in a single quantum state, and any change in the state of one qubit causes an instantaneous and predictable change in the state of the other qubit. By leveraging these properties, a quantum computer can perform operations on various combinations of values simultaneously, enabling the solution of problems that exceed the capabilities of classical and supercomputers due to their complexity. Despite the numerous benefits and advancements introduced by quantum computing, it has also a detrimental effect on cybersecurity, posing potential threats in the cryptographic field [9]. Current cryptosystems heavily rely on public-key cryptographic algorithms, such as RSA and Elliptic Curve Cryptography (ECC), and on the computational complexity in factorising large numbers or computing discrete logarithms, that are thought to be computationally infeasible problems for current computers. However, quantum computers can exploit Shor's algorithm to efficiently solve these problems, rendering the underlined algorithms vulnerable [10]. As proof-of-concept, by using 20 million qubits, it would be possible to crack an RSA 2048-bit key in 8 hours [11]. Despite the widespread deployment of quantum computers is still in its early stages, it is critical to implement suitable countermeasures prior to the widespread adoption of practical systems. In this scenario, the development of quantum-resistant solutions, known as *Post-Quantum Cryptography* (PQC) [12], becomes essential. The National Institute of Standards and Technologies (NIST) in 2017 started a process to standardize quantum-resistant cryptographic algorithms, narrowing down the proposals to a few selected algorithms in 2022 [13].

Alongside quantum computing, the concept of confidential computing [14] has also made significant advancements. It is the protection of data in use in a hardware-based *Trusted Execution Environment* (TEE). A TEE is an area of the system secured by a processor, hence secrets, sensitive data, and operations involving such elements can be safeguarded within a TEE [15]. Typically, a TEE is commonly associated with a *Rich Execution Environment* (REE), denoting the Operating System (OS) that the device is running (e.g. Android, iOS), the hardware resources, and the regular running applications. In this particular scenario, the OS is referred to as *Rich OS*. According to this approach, an application in which sensitive operations are performed can be divided into sections. Sections that do not need protection are executed into the REE, meanwhile the sensitive ones are executed within the TEE. Applications running in the TEE are known as *Trusted Applications* (TA). There are two extents of isolation: isolation with respect to the REE and isolation with respect to the TAs. Hence, the REE is unable to directly gain access to the TEE but only via designated interfaces, while TAs are incapable of interfering with other TAs. It is important to underline that a TEE must guarantee to TAs exclusive and trusted access to peripherals, ensuring that there are no threats between the trusted application and the I/O. This principle is referred to as *Trusted Path*.

In order to attain the intended level of security, it is of utmost importance that the TEE becomes an integral component of the device's secure boot chain. This needs a *Root of Trust* (RoT), a trusted hardware component (which cannot be compromised by software) assumed to be untainted and which the platform owner relies upon to ensure security. In recent years, the *Trusted Computing Group* (TCG) [16] and the *Global Platform* (GP) alliance [17] have collaborated to define the concept of a TEE based on the *Trusted Platform Module* (TPM) as a RoT [18]. Developed by the TCG, the TPM is a chip which enables the reporting and verification of a device's state by securely storing artefacts and measurements, that can be used to authenticate and attest the trustworthiness of the system [19]. Stored within the TPM's *Platform Configuration Registers* (PCRs), the measurements consist of hash values related to the software executed and the system state, reported to a verifier by digitally signing an attestation report with a unique key. This schema enables the accomplishment of *Remote Attestation* (RA), which entails the capability to monitor all activities within the operating system and validate the presence and execution of solely authorized software. It is important to note that the usage of a unique secret enables the provision of *Device Identity* (DevID), an identifier that can be leveraged to authenticate the device.

To facilitate the establishment of device identity and attestation, a framework known as *Device Identifier Composition Engine* (DICE) [20] has been developed. This framework incorporates a variety of hardware and software techniques. The major goal of this specification is to augment security and privacy in systems equipped with a TPM while also offering security and privacy foundations for systems lacking a TPM, introducing new solutions with minimal hardware requirements. One of the key requirements involves generating a *Compound Device Identifier* (CDI) [3]. The CDI is created by applying a one-way function using the CDI from preceding layers of the device boot sequence. The initial CDI is formed through the combination of the *Unique Device Secret* (UDS), which is a distinct secret value specific to each device and incorporated during the manufacturing or provisioning, and certain measurements.

However, despite the robustness of this architecture against current threats, RA and DICE employ distinct keys for the purpose of signing authentication challenges, attestation evidence, and issuing certificates for every layer within the device's boot sequence. This heavy reliance on digital signatures makes the attestation process and the security properties provided vulnerable to quantum computers.

Hence, this thesis focuses on enabling protections in such an environment, expanding the functionalities of a TEE to support post-quantum cryptographic algorithms. It also takes into consideration the limitations posed by embedded systems' constrained resources and computational capabilities. To achieve this goal, the *Keystone Enclave* [21] open-source framework is leveraged. Developed at the University of California, Berkeley, Keystone offers software-defined blocks to build a customized TEE for RISC-V processors [2]. On the other hand, the DICE specifications deployment guarantees DevID and RA for devices with limited resources.

This document starts by providing a comprehensive description of the TEE functionalities and architecture, including a description of the Keystone framework. Subsequently, it covers the utilization of RA within the TEE and contextualizes it with the DICE specification. After an analysis conducted on the state of the art, the ongoing PQC standardization process, and the migration process needed to ensure a correct migration towards a quantum-safe world, the selected algorithms from the third round of the NIST call for proposal are presented. Afterwards, the methods employed for implementing the TEE utilizing the mentioned frameworks are thoroughly examined, along with the necessary expansion to encompass the selected PQC algorithms. Lastly, the accomplished results and goals are showcased and directions for further developments on the topic are presented.

Chapter 2

Trusted Execution Environment

2.1 TEE definition

Devices offer an extensible and versatile operating environment called *Rich Execution Environment* (REE). This REE, also called *Rich Operating System*, brings flexibility and features to the device, but leaving it vulnerable to several security threats. In light of such concerns, the GlobalPlatform (GP) introduced the concept of *Trusted Execution Environment* (TEE). A TEE is designed to be paired with the REE and other execution environments to provide a safe context in which protect assets and execute trusted code [1]

For example, in an Android device, the operating system is run as rich OS. If a safe execution environment is needed, is possible to run a TEE to perform sensitive operations in a safe environment. The applications running in the REE can call specific modules inside the trusted area by means of the TEE Client API.

Anything that runs in each execution environment cannot provide harm to the trusted part. Additionally, any trusted application cannot damage or be damaged from any other trusted application.

At the highest level, according to the GP, a TEE is an environment where:

- the code executing within the TEE is authenticated;
- the assets' integrity and confidentiality of a TEE is ensured through isolation, cryptography, or other mechanisms;
- the TEE capabilities such as isolation can be used to provide confidentiality of the TA code asset;
- the TEE resists known software attacks, and a set of external hardware attacks;
- both code and assets are protected from unauthorized debug tracing and control operations being performed though the device's bug and test features.

The isolation feature ensures that at each time a resource is controlled either by the REE or the TEE. To transfer the resource's control from the TEE to the execution environments, an explicit authorization is needed. Assets that cannot be shared by a TEE are considered to be trusted resources. These resources can be accessed only by trusted resources, thereby creating a closed system that is isolated and protected from other environments.

Unlike the previous case, another Execution Environment may permit some of its resources to be accessible by the TEE without specific permission. In such cases, the TEE should respect the security and access control policies of the other execution environment.

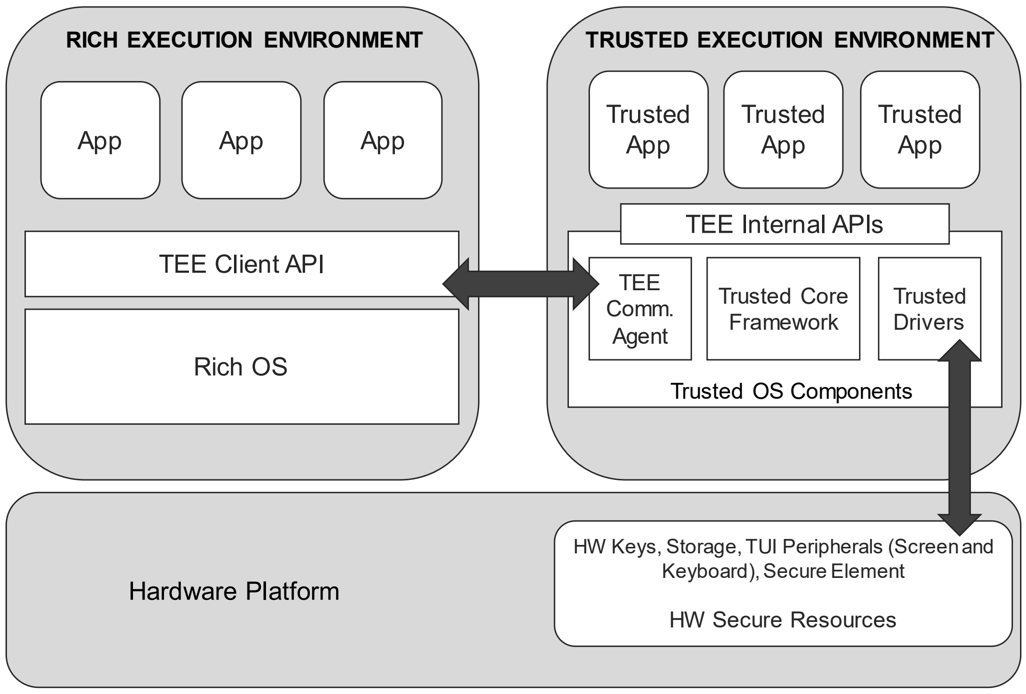


Figure 2.1. TEE and REE (source: [1]).

2.2 TEE security requirements

On a high-level security perspective, the TEE requirements that must be satisfied can be expressed as follows.

First of all, a **Secure Boot** process is needed, where the runtime environment is instantiated from a RoT, using assets bound to the TEE and isolated from the REE. The integrity and authenticity achieved through secure boot is extended and retained throughout the entire lifetime and state transitions of a TEE.

Isolation, as underlined, is a compulsory feature. There are two extents of isolation:

- the TEE is isolated from the REE and other environments;
- the Trusted Applications running in the TEE are isolated from other Trusted Applications.

This is achieved by means of hardware mechanisms that other environments cannot control, providing protection against them. In fact, the primary purpose of a TEE is to protect its assets from the REE. Software outside the TEE cannot call directly the functionalities offered by the TEE. Such requests must undergo through protocols that allow the trusted OS or TA to verify the acceptability of the operations requested.

There is the concept of **Trusted Path**. The computation is protected, but concerns can be raised as regards the I/O. In fact, a TEE must guarantee that a TA may require to have an exclusive and trusted path to a specific peripheral (e.g. entering a pin to unlock a key). To achieve this protection the TEE cut out from the bus temporarily the rich OS, ensuring that between the input and the TA there are no malicious actors. Hence, an infected rich OS is not able to affect the TEE. This protects the TEE from attacks such as key logging or screen capture.

Finally, a TEE must provide **Trusted Storage** of data and keys. The storage is bound to a particular TEE on a particular device, so that no unauthorized entity can access, copy, or modify the data contained within it. The usage of Trusted Storage ensures protection against rollback attacks.

2.2.1 Root of Trust (RoT)

To have secure boot, there is the need to rely on an element that is trusted and must always behave as expected (its misbehaviour cannot be detected). The RoT are the building blocks to establish trust in a platform.

In the context of a TEE, there are three RoT that must be provided:

- *Root of Trust for Measurement* (RTM): element that compute measurements and sends them to the RTS ;
- *Root of Trust for Storage* (RTS): the measures computed must be protected in a secure hardware storage, otherwise an attacker could change the values.
- *Root of Trust for Reporting* (RTR): entity which reports securely (e.g. digital signature) the content of the RTS.

2.3 TEE building blocks

2.3.1 REE interfaces to the TEE

Within the REE, the components that constitute the interface towards the TEE are an optional protocol specification layer, an API, and a supporting communication agent.

The **REE Communication Agent**, shared among all Client Applications, provides REE support for exchanging messages between the Client Application and the related Trusted Application.

The **TEE Client API** is a communication interface designed to allow a Client Application running in the Rich OS to access and exchange data with a Trusted Application running inside a TEE.

The **TEE Protocol Specifications Layer** in the REE offers Client Applications a set of higher level APIs to access some TEE services. TA developers can develop additional proprietary TEE APIs at the TEE Protocol Specifications layer.

2.3.2 Trusted OS components

Within the TEE, two classes of software are defined: the *Trusted OS Components*, and the *Trusted Applications*.

Trusted OS Components consist of:

- the **Trusted Core Framework** which provides OS functionalities to Trusted Applications and is part of the TEE Internal Core API;
- the **Trusted Device Drivers** which provide a communication interface to trusted peripherals.
- the **TEE Communication Agent** is a special case of a Trusted OS component that works with its peer, the **REE Communication Agent**, to safely transfer messages between CA and TA.

2.3.3 Trusted applications (TAs)

The *Trusted Applications* are the applications running securely in the TEE.

The TAs communicate with the other components of the system via APIs exposed by Trusted OS components:

- the **TEE Internal APIs** define the fundamental software capabilities of a TEE;
- Other Internal APIs that can be defined to support interfaces to further functionalities.

In particular, when a Client Application wants to initiate a session with a TA, connects to an instance of that TA. Each instance of TA has physical memory address space separated from the ones of other TA instances and the session is used to create a logical connection to the multiple commands issued in a TA. Thereby, each session has its own state which contains the session context and the context of the task executing the session.

2.3.4 Relationship between TEE APIs

The TEE Client API architecture

The **TEE Client API** provides interfaces to enable communications between a Client Application and a Trusted Application. On top of the functionalities exposed by the TEE Client API, is possible to build high level standards and protocol layers, for example to support common tasks such as trusted storage, cryptographic services, and so on.

Applications typically use the TEE Client API to perform the following tasks:

- establish communication with a TEE;
- establish a session with a TA;
- if there is the need to exchange data with the TA, the TEE Client API can be used to set up a shared memory;
- issue specific commands to the TA, leveraging the service provided by that TA;
- close the communications.

The TEE Internal API architecture

Global Platform specifies a series of APIs to provide a common implementation for functionalities typically required by many Trusted Applications.

The **TEE Internal Core API** defines the various interfaces to enable to a Trusted Application usage of the standard TEE capabilities. If further low-level functionalities are needed, optional TEE Internal APIs such as the TEE Secure Element API, TEE Sockets API, and TEE TA Debug API can be deployed.

As the TEE Client API, higher level standards and protocol layers can be built on top of the features provided by the TEE Internal APIs, for example to support tasks such as creating a trusted password entry screen for the user.

The TEE Internal Core API provides a number of functionalities to the Trusted Application. Examples of such functionalities are as follows:

- **Trusted Core Framework API** which provides integration, communication, memory management and system information retrieval interfaces;
- **Trusted Storage API** for data and keys which provides Trusted Storage;
- **Cryptographic Operations API** which provides cryptographic capabilities;
- **peripheral API** which enables interactions among Trusted Application and peripherals via the Trusted OS.

The **TEE Sockets API** provides a modular interface for the TA to communicate to other network nodes as a client. It consists in the general API for accessing and handling sockets.

The **TEE TA Debug API** provides services that to support TA development and testing of the TEE Internal APIs. Two services are relevant in this context that are the Post Mortem Reporting (PMR) service and the Debug Log Message (DLM) service

The first provides a method for a TEE to report to clients the termination status of TAs that enter the Panic state. Without this capability, there would not be possible to certify correct functionality of the TEE Internal APIs, as the Panic state is used to report various error conditions that need to be tested. The second one provides a method for a TA to report simple debug information on authorized systems.

The **TEE Secure Element API** is a layer that enables communication to Secure Elements (SEs) connected to the device in which the TEE is implemented. A Secure Element is a chip that is by design protected from unauthorized access and used to run a limited set of applications, as well as store confidential and cryptographic data. SEs can be connected in a shared way via the REE or exclusively to the TEE. A SE connected exclusively to the TEE is accessible by a TA without using any resources from the REE since the communication is considered trusted. A SE connected to the REE is accessible by a TA using resources lying in the REE. In this last case, a secure channel must be used to protect the communication between the TA and the SE, providing protection against threats in the REE.

The **TEE Untrusted User Interface API** permits to provide a user interface while achieving three objectives:

- secure display: information displayed to the user cannot be accessed, modified, or obscured by any unauthorized component.

- secure input: information entered by the user cannot be derived or modified by any unauthorized component;
- security indicator: the user can be confident that the screen display is not a spoofed screen.

2.4 Keystone Enclave

Keystone [2] is an open-source framework for building customized TEEs on RISC-V architectures. Before diving in the Keystone architecture, it is critical to provide an overview of the main RISC-V features leveraged in building secure enclaves.

2.4.1 RISC-V overview

RISC-V is an open-source instruction set architecture (ISA) [22], that has the properties of modularization and extensibility. It consists of basic instruction sets and modular extensions to be suitable in different scenarios, allowing to customize the ISA to perform specific computing tasks. Therefore, users can freely combine RISC-V instruction set extensions and customize private instruction sets according to the requirements needed.

For example an important field in which the RISC-V is employed is that of embedded microprocessor applications. Moreover, Post-Quantum RISC-V ISA extension has been proposed to fulfil the demands of high throughput and low latency in modern applications by Post-Quantum algorithms.

Particularly important are two features of the architecture:

- the *Privileged ISA*, which defines four levels of privileges;
- the *Physical Memory Protection*, that ensures isolation among the TEE, the REE, and the TAs.

RISC-V privileged ISA

The RISC-V provides privileged architecture for running operating systems, supporting virtualization, and running secure Enclaves [23]. Hence a RISC-V hardware thread (called *hart*) is running at some privilege according to the encoding provided within one or more control and status registers (CSRs). The RISC-V privilege levels are defined in table 2.1.

Table 2.1. RISC-V privilege levels

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Hypervisor	H
3	11	Machine	M

Privilege levels are used to provide protection between different components of the software stack. Whenever a component attempts to perform operations not permitted by the current privilege mode an exception is raised. The exceptions normally propagate as traps to the underlying execution environment which then can perform operations according to a policy.

The highest privileges are provided by the **Machine level** (M-mode). It is the only mandatory privilege level for a RISC-V hardware platform. In fact, the simplest implementation may provide only M-mode, but this will provide no protection against malicious and incorrect code. The code running in M-mode is usually inherently trusted since it has unfettered access to the whole machine and it can be used to manage secure execution environments on RISC-V.

Right after the M-mode, the **Hypervisor-mode** (H-mode) is intended to support virtual machine monitors. In particular, the H-mode aims to provide isolation between a virtual machine monitor and an environment running in machine mode.

Given the issues that arise with a single privilege level, many RISC-V implementations support the S-mode and the U-mode. The **Supervisor-mode** (S-mode) can be added to provide isolation among the operating system running in S-mode and the secure execution environment. The **User-mode** (U-mode) is the level attributed to classic applications.

According to the privilege level, a core set of privileged ISA extensions with optional extensions and variants is available. For instance, M-mode supports several optional extensions to perform address translation and memory protection.

It is possible to include in the implementations a **Debug-mode** (D-mode). The D-mode can be considered as an additional privilege level with more access than the M-mode, and is intended to support off-chip debugging and manufacturing testing. It works by reserving some CSR addresses accessible only with this mode and may also reserve some physical memory space on a device.

RISC-V PMP

As already mentioned, the RISC-V architecture standard employs a mechanism defined *Physical Memory Protection* (PMP). It aims at segregating the system memory into regions and each region has a set of policy and permissions to be accessed. This prevents unprivileged software from accessing regions without the required privileges. Such feature is critical in the context of secure processing, since it is necessary to limit the physical addresses accessible by an unprivileged context running on a hart.

To achieve this goal, the PMP works with per-thread M-mode control registers that allow to specify the operations (i.e. read, write, execute) possible on a certain memory region. Whenever a thread is running in any of the aforementioned modes, the PMP checks are applied and the violations are trapped at the processor.

The granularity and encoding of the PMP access control settings are platform-specific and there might be different granularities and encodings of permissions for different physical memory regions on a single platform. This means that some PMP designs might just employ few CSRs to protect a small number of physical memory segments, while others might employ techniques to protect large physical memory spaces.

It is critical to underline that each RISC-V core has its set of PMP entries. Whenever an enclave is created, the PMP changes must be propagated to all the cores and the SM, executing on each of the cores, removes the access of other cores to the enclave. Such communications are done only at creation and destruction. Modifications to the PMP entries during execution are local to the core executing the enclave.

2.4.2 Customizable TEEs

As already stated, Keystone is a framework that allows to build customized TEEs.

One reason for the development of a framework for customizable TEEs is that depending on the specific platform, use case, or application, the threat model differs. Moreover, even on the same platform, different applications may operate under different threat models. This differs from the existing commercial TEEs which lacks of flexibility.

Unlike existing commercial TEEs, in Keystone is possible to specify a per-enclave configuration of security features. For instance, TEEs deployed in data centre or home appliance may not need to operate with physical adversary protection since a physical attacker may not be a realistic concern and Keystone can be configured to operate only with the required protections.

There are some requirements that must be satisfied in a hardware platform supporting Keystone:

- a trusted boot process;
- a device-specific secret key visible only to the trusted boot process;
- a hardware source of randomness.

In a system compliant with Keystone several components are present as the ones depicted in the figure 2.2.

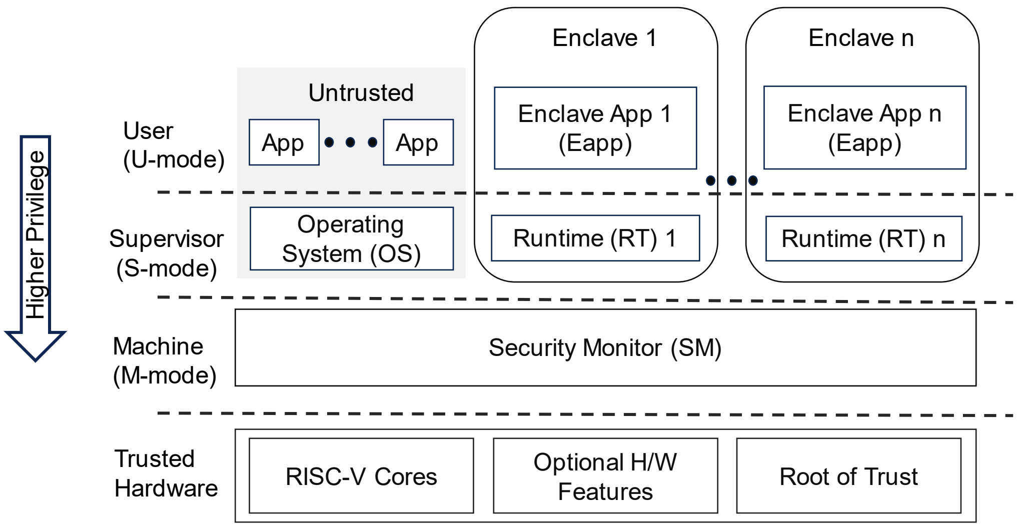


Figure 2.2. Keystone system's example (source: [2]).

- **trusted hardware**: comprehensive of RISC-V cores, optional hardware features, and RoT;
- **Security Monitor**: software component that runs in M-mode used to manage the enclaves' lifecycle;
- **enclave**: secure execution environment isolated from other enclaves and from the REE;

- **enclave application** (eapp): the application running in a certain enclave;
- **Runtime** (RT): per-enclave component used to provide OS functionalities to the enclave (i.e. system calls, traps, etc.).

2.4.3 Entities in TEE lifecycle

Five logical entities can be identified in the TEE's lifecycle:

- **hardware manufacturer**: designs and fabricates RISC-V hardware including relevant components needed by the trusted boot process;
- **Keystone platform provider**: purchases hardware, operates it, and configures the SM;
- **Keystone programmer**: develops Keystone software components including SM, RT, and eapps;
- **Keystone user**: chooses a Keystone configuration of RT and an eapp, and instantiate an enclave executing on hardware provisioned by the Keystone platform provider;
- **eapp user**: interacts with the eapp executing in an enclave on the TEE instantiated using Keystone.

The provisioning and deployment of the elements that compose Keystone are shown in the figure 2.3:

1. the platform provider configures the SM;
2. Keystone compiles and generates the SM boot image;
3. the platform provider deploys the SM;
4. the developer writes an eapp and configures the enclave;
5. Keystone builds the binaries and compute measurements;
6. untrusted host binary is deployed to the machine;
7. host deploys the RT, the eapp, and initiates the enclave creation;
8. the remote verifier can attest based on known platform specifications, keys, and SM measurements.

2.4.4 Security Monitor

The most important component of a Keystone TEE is the *Security Monitor* (SM). Since it uses only standard RISC-V features, it is easily portable to other RISC-V platforms. In addition, Keystone provides an easy way of configuring and compiling the SM depending on the underlying platform.

It is used to provide isolation and security features, as well as ensuring that the security features required are respected.

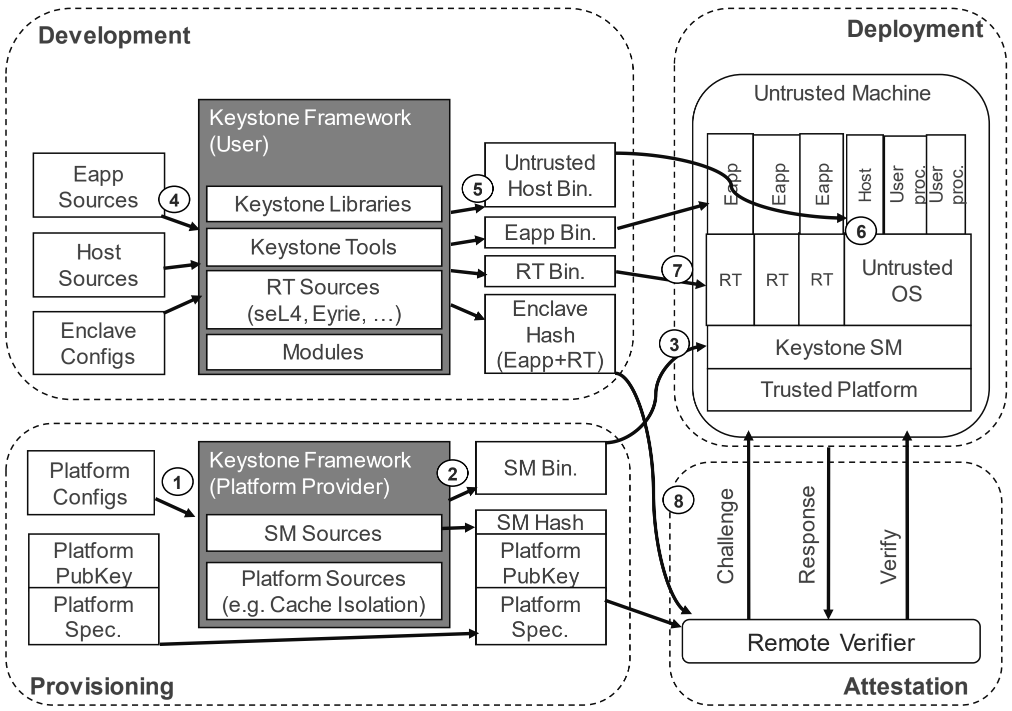


Figure 2.3. Entities in TEE lifecycle (source: [2]).

Memory isolation

As regards memory isolation, Keystone uses the PMP feature provided by RISC-V.

Through the definitions of PMP entries, Keystone restricts the accesses of components operating in S-mode and U-mode to certain physical memory regions, since each PMP entry controls the permissions of a physical memory region. More in depth, the PMP address registers (e.g. pmpaddr in the figure 2.4) contain an encoding of the address of a contiguous memory region, and the PMP configuration register (e.g. pmpcfg in the figure 2.4) contains the bits specifying the read, write, execute permissions for the U-mode and the S-mode. If any of two modes attempts to access a physical address and it does not match any PMP address range, the hardware does not grant any access permissions.

PMP makes Keystone memory isolation flexible in three ways:

- multiple discontiguous enclave memory regions can coexist instead of having one memory region shared by all enclaves;
- PMP entries can cover regions of different size, allowing for arbitrarily sized enclaves;
- PMP entries can be reconfigured during execution to dynamically create new regions or release a region's memory.

During the boot phase of the SM, Keystone configures the first PMP entry (that has the highest priority) to cover the SM memory region. This prevents any access

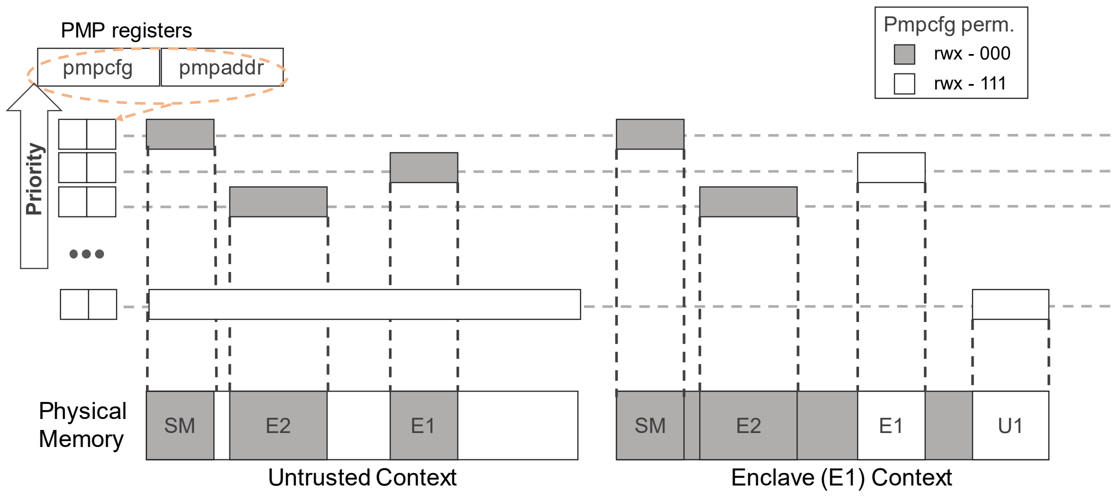


Figure 2.4. How Keystone uses RISC-V PMP (source: [2]).

from unprivileged modes to the SM physical memory region. Then, the SM configures the last PMP entry (that has the lowest priority) to cover the whole memory with all the permissions enabled and associates it to the OS. Without such configuration, the OS would not be able to access the memory since would not be covered by any PMP entry. When an enclave must be created, the host application requests the OS to finds an appropriate contiguous physical region and calls the SM which validates the request and protects the enclave memory by adding a PMP entry (with priority higher than the OS PMP entry) with no permissions. To be validated, the request requires that the region that will be allocated to the enclave does not overlap with other enclaves' regions or the SM region. When transferring control to an enclave, the SM:

- sets PMP permission bits of the enclave memory region;
- removes all OS PMP entry permissions to protect the memory from the enclave.

This allows the enclave to access its own memory and no other regions. When a CPU context-switch would swap to a non-enclave context, the SM disables all permissions for the enclave region and re-enables the OS PMP entry to allow default access from the OS.

Each enclave requires a PMP entry for each isolated memory region used and such entries are freed with the enclave destruction.

Given that N is the number of PMP entries available, N-2 simultaneous enclaves are supported in Keystone. It is possible to further expand this limit virtualizing the PMP entries.

Post-creation in-enclave page management

Keystone has a different memory management design from most TEEs.

Once the OS generates the page tables, the virtual to physical memory mapping is delegated entirely to the enclave during its execution. Thanks to the per-hardware M-mode thread views of the physical memory and the PMP feature provided by the RISC-V architecture, it is possible to have concurrent and multi-threaded enclaves accessing the separated memory partitions. Once created the isolated enclave operating in S-mode, Keystone can execute its virtual memory management manipulating the enclave page tables located inside the isolated memory space of the enclave. Delegating the memory management to the enclave bring two advantages:

- more flexible virtual memory management with several Runtime modules;
- no controlled side-channel attacks are possible since the OS cannot modify or observe the enclave memory mapping.

Interrupts and exceptions

During the enclave execution, while the interrupts trap directly to the SM, exceptions may be propagated to the RT via a RISC-V feature called exception delegation register. At this point, the RT handles exceptions as needed by means of a standard kernel abstraction implementation and may forward traps to the untrusted OS via the SM.

The problem here, is that the enclave could hold a core, hence causing a Denial of Service (DoS) in the host. To avoid this issue, the SM sets a machine timer before it enters the enclave. When the timer interrupt triggers, the SM regains control and it may return control to the OS or request the enclave exit.

Enclave lifecycle

Enclaves' lifecycle is made up by three distinct states.

During **creation**, Keystone computes measurement of the enclave memory to ensure that the enclave binary is correctly loaded to the physical memory. The initial virtual memory layout is used for the measurement since the physical layout can vary across different executions. In particular, the OS initialize the enclave page tables and allocate the memory for the enclave. The SM waits the OS and then analyses the OS-provided page table, seeking for invalid mappings and ensuring a unique one. Here, the SM hashes the page contents, the virtual addresses, and the configuration data and at the execution, the SM sets PMP entries and transfers control to the enclave.

At **execution**, the SM sets the appropriate PMP entries and transfers the control to the enclave.

The enclave **destruction** is initiated by the OS. The SM frees the enclave memory region and returns such memory to the OS, frees all the enclave resources, PMP entries, and enclave metadata.

2.4.5 TEE primitives

The standard TEE primitives supported in Keystone are listed in the following:

- **Secure Boot**: at each CPU reset, the RoT measures the SM, creates a fresh attestation key thanks to a secure source of randomness, stores it into the SM memory, and signs the measurements and the public key;
- **secure source of randomness**: Keystone provides a secure Supervisory Binary Interface call, *random*, which returns a 64-bit random value; the SBI is privileged firmware that initialise hardware and allows unprivileged components to call low-level functionalities such as managing cores;

- **Remote Attestation**: the SM computes the measurement and the attestation based on the provisioned secret;
- **other primitives**: Keystone can support other primitives if required by the TEE, such as allowing enclaves to access the read-only timer registers.

2.4.6 Modular runtime

As the SM isolates each of the enclaves, we can safely run private S-mode code into the enclave. The private S-mode code running into the enclave is called *Runtime* (RT).

The RT can be thought as a per-enclave kernel, without requiring all the kernel functionalities. In Keystone the modular RT, namely *Eyrie*, allows to include only the intended functionalities, thus reducing the TCB and the attack surface.

The usage of S-mode, allows to implement the selected functionality without the need to modify the user applications. Furthermore, introducing an additional privilege layer gives the chance to allow only the RT to access the shared memory buffer, enforcing the security. Moreover, this enables the portability of microkernel within an enclave.

Enclave memory management modules

An enclave can run code in S-mode and U-mode. Given the in-enclave memory management capability, the code running in the enclave does not need to cross the isolation boundary. The issue here is that enclaves' memory regions occupy a fixed contiguous physical section, allocated by the OS with a statical mapping among virtual and physical memory at load time. This limits the memory usage of most legacy applications.

To enable a more flexible enclave memory management, some modules can be employed:

- **free memory module**: Eyrie RT performs page table management hence there is no need to have pre-defined page mappings;
- **in-enclave self paging module**: generic in-enclave page swapping module for Eyrie, which handles the enclave page faults and uses a generic page backing-store that manages the page storage and retrieval, relaxing the memory restrictions in an enclave due to a limited memory size;
- **module for protection of the page content leaving the enclave**: module that allows to achieve confidentiality and integrity of the page contents when an enclave handles a page fault, copying the content of the page out of the secure memory.

Finally, as examples, two modules provided in Eyrie are the Edge Call Interface and the multi-threading.

As regards **multi-threading**, Keystone supports multi-threaded eapps by delegating the thread management to the runtime. Parallel multi-core enclave execution can be implemented by allowing the SM to invoke the execution of an enclave multiple times on different cores.

The **Edge Call Interface** comes in place whenever there is the need to read or write data outside of the enclave. In particular, eapps cannot access memory that is outside the enclave's memory, hence the edge calls are performed by the Eyrie RT on behalf of the eapps. An edge call consists of an index pointing to a function implemented in the untrusted host, as well as the parameters needed by it. The call is safely propagated to the untrusted host, the function is executed and the return values are copied into the enclave and then sent to the eapp. To enable this mechanism is required access by Eyrie to a buffer shared with the host. The shared buffer is allocated by the OS in the host memory space and its address is passed to the SM at enclave creation. Here, the SM passes the address to the enclave and a PMP entry is configured to enable OS access to the shared buffer. This is needed to allow the RT and the OS to access the shared buffer.

Such module can be used to add support for syscalls, enclave-enclave communication and so on.

2.4.7 Security analysis

Protection of the enclave

The attestation mechanism employed within Keystone ensures that any unauthorized modification of the SM, the RT, or the eapps is detected at enclave creation. At enclave execution, the PMP ensures that any malicious access to the enclave memory is denied. Moreover, the data contained within the enclave, can be modified only by the enclave itself or the SM, both isolated.

The usage of a dedicated page management and in-enclave page tables provide protections against **controlled side channel attacks**, rendering them not possible in Keystone. **Cache side channel attacks** as well are not possible in Keystone, since the enclaves do not share any state with the host OS or the user application and when performing context-switching, the SM flushes the enclave state. Moreover, only the SM can see enclave's events which are not visible to the host OS (e.g. interrupts, faults).

As regards the **mapping attacks**, the RT ensures that the virtual-to-physical memory mappings are valid, creates valid ones, and is trusted by the eapp. At enclave creation the RT initializes the page tables or load a static mapping validated by the SM. During execution the RT ensures that the mapping updates do not corrupt the memory layout. New empty pages provided by the enclave are controlled by the RT, that determines if the page can be safely used before mapping them to the enclave. This holds even when any page is removed from an enclave, whose content is freed before returning the space to the OS. Such mechanisms ensure that Keystone is not susceptible to mapping attacks.

The RT modules can be leveraged in Keystone to achieve protection against **syscall tampering attacks**. In fact, whenever there is the need to invoke untrusted functions implemented in the host process or execute OS syscall, there is the culprit for Iago attacks and system call tampering attacks. RT modules are available and can be used to protect the enclaves against such threats.

Protection of the host OS

The host OS is not susceptible to attacks coming from the enclave. This is due to the fact that the RTs execute at the same privilege level of the host OS. This means that the enclave cannot reference any memory outside the one allocated due to the PMP isolation feature and the host state cannot be corrupted at context-switch time since the SM performs it correctly and completely. Moreover, the page tables belonging to the host application or to the host OS cannot be modified. Finally, an enclave cannot hold a core for too much causing a DoS, since the enclave will be interrupted by the timer set by the SM, which in turn returns the control to the OS.

Protection of the SM

The security of the SM is ensured thanks to the PMP isolation and to the trust model employed. In particular, the SM does not trust the lower privilege software components (e.g. eapps, RTs, host OS, etc.) and its memory region is inaccessible to both enclaves or host OS.

Concerns could arise considering the SM SBI, which constitutes another component, hence another attack path. In Keystone the SBI code presents in the SM does not require a complex resource management and has been formally verified due to its small size, ensuring its security.

The SM is not subject to DoS since it does not require a scheduled execution time.

Other attacks such as side channel attacks are not a problem since the SM is protected against such attacks by means of known techniques.

Protection against physical attackers

One last security consideration is related to the protection against physical threats that could control the memory. In Keystone such protection is achieved via platform features and a modification of the bootloader.

The enclave itself is physically protected by using an on-chip memory and the RT's paging module, with encryption and integrity protection on the pages leaving the on-chip memory. The page backing-store is PMP protected, now containing only the encrypted pages. This fully guarantees the confidentiality and integrity of the enclave code and data from an attacker with control of DRAM.

The SM should be executed entirely from the on-chip memory, which may require a modification of the trusted bootloader but it should be noted that the SM code is relatively small, and statically sized.

With these techniques in place, content outside of the chip is either untrusted or encrypted and integrity protected.

Chapter 3

Device Identifier Composition Engine and Remote Attestation

Cyber-attacks are becoming more and more sophisticated meanwhile new market segments like the Internet of Things (IoT) and embedded systems are built and designed with innovative architectures, based on solutions in which resource and power constraints lead to difficulties in implementing security solutions with a certain extent of security. Leveraging on Trusted Platform Module allows to achieve an optimal and flexible solution but not all the systems and devices are equipped with such silicon-based capabilities.

The Device Identifier Composition Engine specification proposed by the Trusted Computing Group (TCG) [3], aims at providing a strong device identity in this context, in which no special hardware for attestation is typically available. The main idea behind the DICE specification is to combine software techniques with minimal and simple silicon capabilities to establish a cryptographic strong device identity, attest software and security policy, and assist in safely deploying and verifying software updates.

3.1 Device Identifier Composition Engine (DICE)

To achieve the definition of a strong device identity, the idea behind DICE is to rely on a *Unique Device Secret* (UDS) which is a statistically unique, uncorrelated value, provisioned at manufacturing time that identifies a specific platform. This is used to derive all the other values needed in the process.

There are some requirements and considerations that must be underlined around the UDS. First of all, the UDS must be securely stored and not be rewritable, with a length of at least 256 bits. In fact:

- changes in the UDS would cause a change in the identity of the device, preventing proper attestation and access to secrets previously stored;
- a sufficient bit-length improves the likelihood of the implementation's durability.

To protect the UDS from unauthorized access, the access to the UDS must be disabled until the reset of the platform and before the execution of the untrusted code. To achieve such protection is possible to place to UDS into read-once memory and to enable access only within the range of DICE instructions addresses, or enable access from a secure coprocessor on which DICE is executed. Moreover, to prevent computation of the UDS starting from values that could be used to reach this goal, the DICE engine must erase such values before the execution of the first mutable code. The **first mutable code** is defined as untrusted replaceable code executed after the DICE engine.

At each boot of the platform, the DICE uses the UDS (having exclusive access) to generate a value called *Compound Device Identifier* (CDI). The CDI generation is described in the figure 3.1.

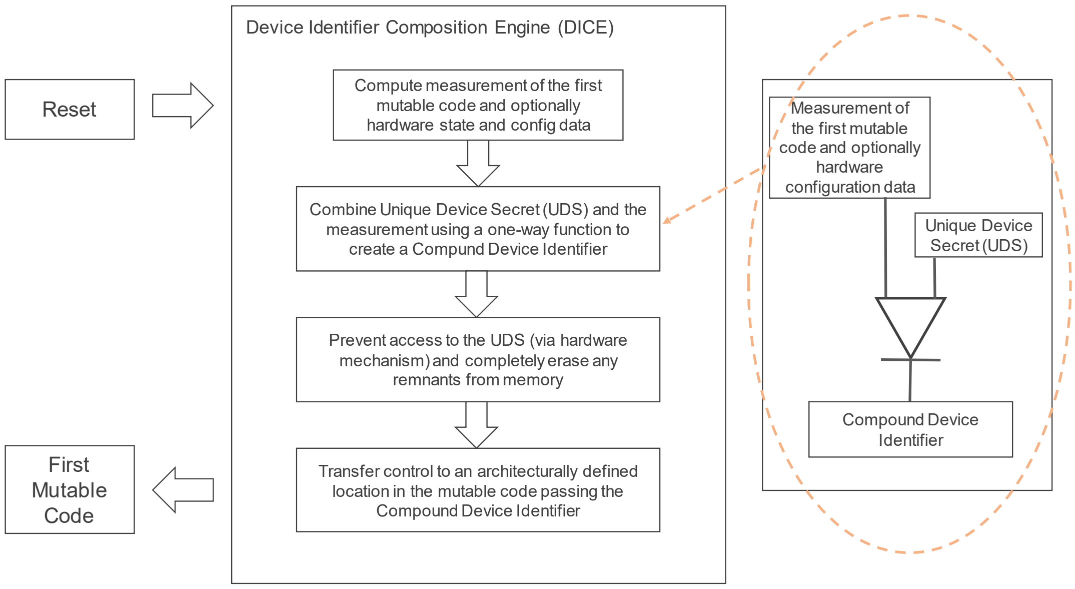

Device Identifier Composition Engine (DICE)
Reset
Compute measurement of the first mutable code and optionally hardware state and config data
Combine Unique Device Secret (UDS) and the measurement using a one-way function to create a Compund Device Identifier
Prevent access to the UDS (via hardware mechanism) and completely erase any remnants from memory
Transfer control to an architecturally defined location in the mutable code passing the Compound Device Identifier
First Mutable Code
Measurement of the first mutable code and optionally hardware configuration data
Unique Device Secret (UDS)
Compound Device Identifier

Figure 3.1. CDI generation process (source: [3]).

More in depth, the CDI is derived using both the UDS and the measurement of the first mutable code, and can optionally include measurements of data that influences the execution of the first mutable code. A **measurement** is defined as a cryptographic condensed representation of code.

The CDI computation must be carried out using techniques that make infeasible the computation of the UDS with knowledge of both the CDI and the measurement. This may be accomplished using a **secure hash algorithm** [24]:

$$H(UDS \| H(FirstMutableCode));$$

or by means of a **secure HMAC** function [25]:

$$CDI = HMAC(UDS, H(FirstMutableCode))$$

Once computed, the CDI is passed to the first mutable code which takes control of the platform.

This process enables the benefit of obtaining a different CDI if the first mutable code is modified in any way, preventing any access to precedent CDI to malicious code. Any CDI leakage would cause the device to be vulnerable to replay or impersonation attacks. To achieve such protection, two solutions are possible:

- dedicated hardware to protect the access to the CDI;

- protect the CDI by means of the first mutable code which results responsible of the CDI protection.

An example of the second approach may be erasing the CDI from the memory as soon as possible after its usage. In this case, the first mutable code must have exclusive access as long as needed to the location in which the DICE write the CDI, erasing the CDI and all the values that could be used to determine the CDI.

Finally, the specification defines two categories of immutability for DICE:

- in simple devices the engine and its dependencies may be invariant and must be such by the end of the manufacturing process;
- in complex systems the engine may be subject to updates which must take place only through a secure process controlled by the manufacturer.

Any update must not influence the CDI.

3.2 Layering architecture

The DICE specification can be extended beyond a single-layer system to a multi-layers system, where each layer is be made up by one or more components.

Defined as *DICE Layering Architecture* [4], it provides a way to know on a system which components are being executed and if behaving as expected. It describes how to extend the identifier composition beyond a single layer to a multi-layered TCB, in such a way that the trusted functionality can be provided to higher TCB layers.

Starting from the DICE hardware RoT (assumed to be trustworthy) the approach over which the DICE Layering Architecture is based, considers the execution states entered progressively. Each transition between layers involves creating a CDI value and securely passing it to the next layer, hence only when a layer is considered in a trusted state is possible to move from one layer to the next one.

By convention the layers are enumerated following the underlined workflow, therefore the layer above DICE implementation is considered the layer 0. This allows to build a chain of trust, enabling the attestation of multiple components within the same device.

3.2.1 Layer capabilities

In oder to build the chain of trust, each TCB layer needs to have access to a set of functionalities that must be protected when executing. The set of capabilities needed and generated at each step:

- **TCB Component Identifier** (TCI): value that allows to describe a specific TCB component as measurement of the code that will be executed and may include other information.
- **Compound Device Identifier** (CDI): computed by means of a one-way function using as input the CDI of the previous layer and measurement (TCI) of the current one;
- **One-way Function**: pseudo-random function compliant with [26].

The way in which such values are computed in a multi-layered system is depicted in the figure 3.2, where each layer securely passes the CDI generated by means of the OWF to the next layer. which in turn uses as input to generate a CDI for the current layer and passes it to the next one and so on.

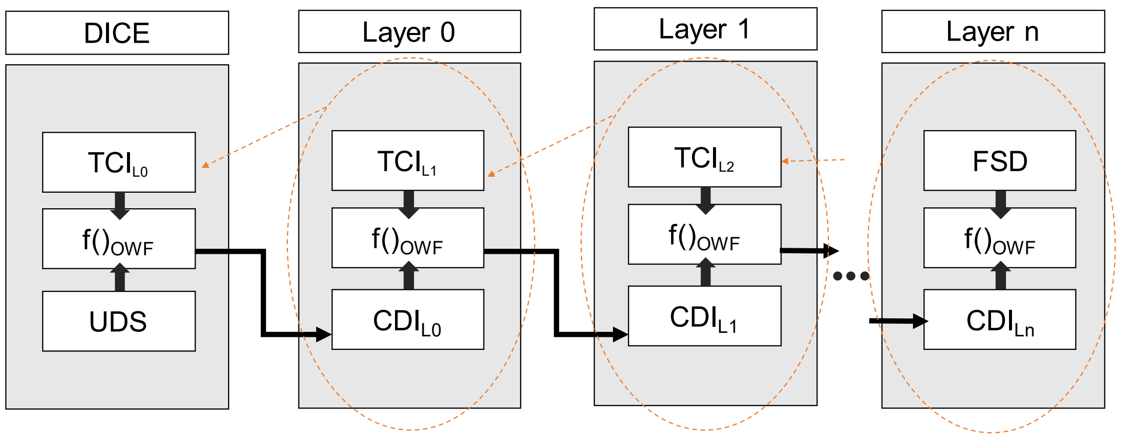


Figure 3.2. CDI generation process in a multi-layered system (source: [4]).

It should be noticed that to achieve the creation of a strong DICE layered identity a precise chain of TCB components is needed since the CDIs computed are accumulation of the measurements of preceding TCB components. Therefore, the identity is determined not only by the identities of the TCB components but also by the order in which each component executes and is usually referred as **DICE Layered Identity**.

To achieve protection in terms of secrecy of the private keys, it is of utmost importance to protect the layers. In particular, is important to access and use secrets preventing other entities from accessing or intercepting such values, in the same way in which interference with the transition from DICE to the layer 0 must be infeasible.

Certification, authentication, and attestation

A DICE layer TCB can be configured to include additional capabilities beyond the ones already specified. The specification [4] identifies certification, attestation and authentication capabilities.

The **Certification** capability of a DICE layered component involves the issuance of a certificate or token that links the next layer TCB to the current layer TCB. In this context, certification can be granted by generating and certifying keys in the current layer before passing the control to the next layer.

Based on the key purpose, certifications differ. In particular, in case of asymmetric key generation, the certification is done by means of an **X.509 certificate**. On the other hand, if a symmetric key is generated, a **token** is constructed containing the next layer's TCI, the next layer's CDI, and a symmetric key derived from the CDI.

This certification takes two possible approaches:

- the TCB layer generates the key pair, issues the certificate for the keys, and finally passes keys and certificates to the next TCB layer;

- the TCB generates its own key pair and then sends a certificate request to the TCB layer below, that in turn creates the certificate and sends it to the requester layer.

Once generated, the certified keys can be used for attestation and authentication.

In particular, **attestation** of a DICE layered component refers to the use of a symmetric or asymmetric key that is approved by a Certification Authority. Such attestation asserts trustworthiness properties about a TCB layer or component which may be explicit (enumerated with an encoding available at a verifier) or implicit (properties are determined based conditions that otherwise would not be possible).

Authentication of a DICE layered component refers to the use of a symmetric or asymmetric key authorized to authenticate the device.

3.2.2 Keys and credentials

In the DICE layered architecture several types of keys and credentials may be managed by each TCB layer. In the following sections a comprehensive description of the types, uses, generation and derivation requirements and modalities are presented, along with security considerations.

Asymmetric keys

Asymmetric keys can be used in the attestation process to provide attestation of the trustworthiness properties of a TCB layer. As regards the generation, the asymmetric keys generation is seeded using the CDI. This means that the layer semantics are implicitly represented in the resulting asymmetric keypair since the CDI is computed by means of measurements of the current layer and CDI of preceding layers. As shown in the figure 3.3, the key-derivation function is seeded using the CDI of the current layer, generating a keypair for the current layer.

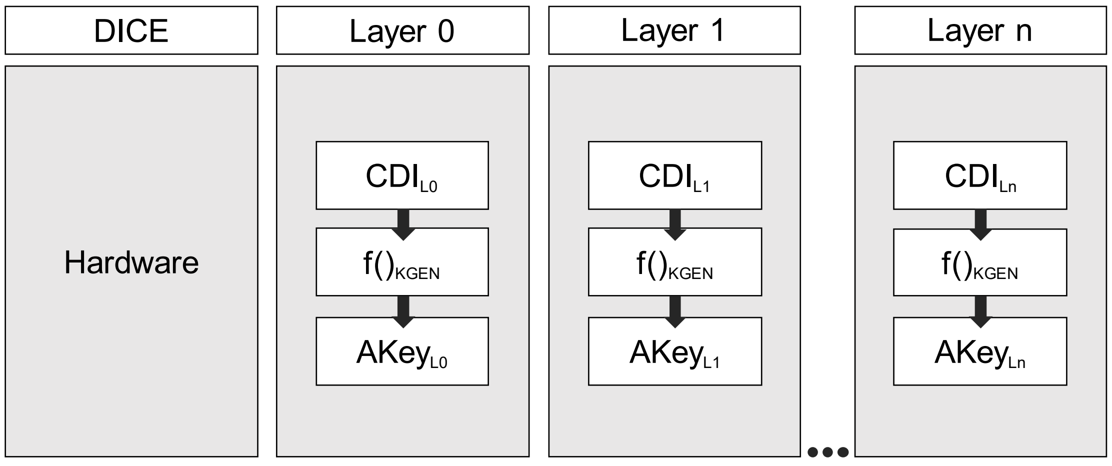


Figure 3.3. Asymmetric keys generation example (source: [4]).

Typical algorithms used for keys generation are the ECDSA and RSA algorithms. When deploying **ECDSA**, the TCB context must influence the generation of the key pair. In particular, a value derived from the CDI can be used to seed the key-derivation function such as a random number generating derived from the CDI, while using the **RSA** algorithm, the influence of the TCB context can be provided using the CDI-derived value to seed the random number generator that produces the p and q primes. In constrained environment, the ECDSA algorithm is preferred over RSA due to the reduced keys size and improved sign operation performance. The first case is preferred for constrained environments due to smaller key size and improved sign operation performance.

Whenever a certificate is needed for keys that are generated for the current or a higher layer, an *Embedded Certificate Authority Key* (ECA) can be used by a TCB component to sign the issued certificate. It is critical to sign only data that is known to the TCB layer, hence external structures cannot be signed with an ECA key.

A key used to sign attestation evidence is called *Attestation Key*. As the previous case, data structures external and unknown to the TCB layer cannot be signed with an Attestation key.

A Key used to sign authentication challenges is called *Identity Key*.

The *DeviceID Key* is the asymmetric key derived from the CDI value that is computed by the DICE, hence is dependent on the device's UDS and the measurement of Layer 0. Certified during the manufacturing process, the key is a long-lived identifier used to sign certificates for key generated for a TCB layer or that may contain attestation evidence, therefore it can be classified as an ECA Key as well as an Attestation Key, and an Identity Key since it is dependent on the CDI.

Unlike the DeviceID Key, the key pair generated using the last CDI value in the chain of TCB layers is defined *Alias Key*. In the DICE derivation chain, certificate related to an Alias Key contains attestation evidence related to the top-level device firmware. Therefore, the Alias Key can be classified as an Identity Key and an Attestation Key.

Symmetric keys

As regards the **symmetric key** generation, a **Key Derivation Function** (KDF) can be used by seeding the KDF with the UDS or a CDI value derived from the UDS. When performing this operation, is critical to use a CDI with adequate length to avoid cryptographic overlap among the seed value and the symmetric key generated. To achieve this goal, the CDI can be increased with additional data. An example of the symmetric key derivation is shown in the figure 3.4, where the CDI and the TCI of the current layer are used as input to the KDF, generating a symmetric key for the current TCB layer.

Analogously to the Asymmetric Key case, a *Symmetric Alias Key* is a key type generated based on the CDI that can be used for Symmetric Key Attestation and Layered Identity.

A *Wrapping Key* is a symmetric key used to protect asymmetric key pairs. This solution is valuable whenever the regeneration of Asymmetric Key Pairs on each boot is undesirable, providing persistence to the generated keypair.

Credential types

The *IDevID* is a credential type containing a DeviceID key that is issued by a device manufacturing process. The TCB layer that generates an IDevID should be immutable or under the control of the device manufacturer.

The *LDevID* is a credential type containing a DeviceID-derived key issued at deployment time in the owner's network.

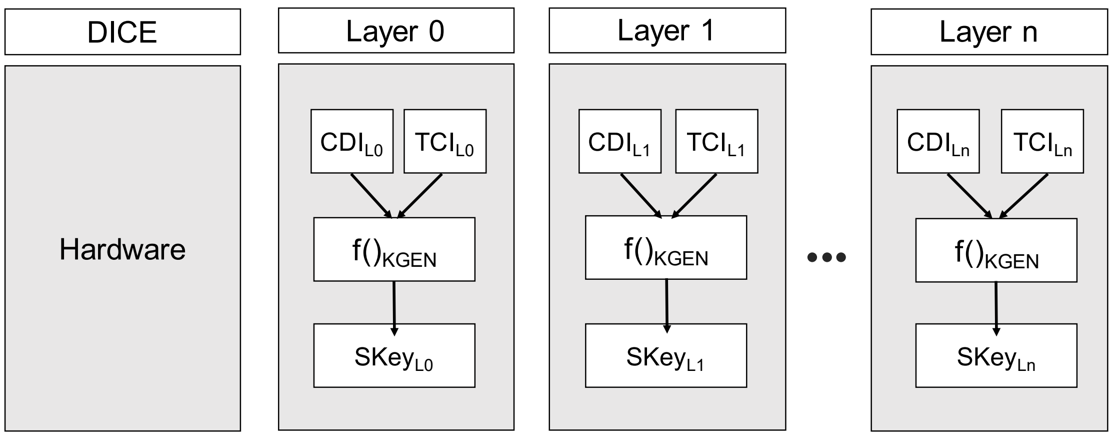


Figure 3.4. Symmetric key derivation example (source: [4]).

Security considerations

There are some considerations that are needed for a correct key management.

The TCB context values (i.e. TCI, UDS, CDI, etc.) can be exploited by an attacker to perform impersonation by generating the keys for a given layer. Therefore, the current TCB layer must protect such values, avoiding the exposure of the values above the layer trusted to protect the secrets such as the private portion of a key. Moreover, before transferring control to the next layer, the current one needs to erase any value used to generate keys to avoid inappropriate duplication or unauthorized usage.

Another critical point regards the **key persistence**. If private keys are generated without dependency on the CDI value and are made persistent, then a change to the layer TCB would not be reflected in the persistent key. This means that the resulting persistent key would implicitly attest the trustworthiness properties of a configuration no more used, misrepresenting the state of the TCB layer. Due to this problem, if the layer TCB code is updated, it is necessary to revoke the current set of asymmetric keys and generate new asymmetric keys from the updated layer.

3.2.3 Layered certification

In a DICE layered architecture is possible to implement a CA hierarchy consisting of a manufacturing root CA that issues device identity and attestation certificates, starting from the Device manufacturer certificate of the DICE hardware RoT (HRoT).

A **Device identity** certificate is a data structure certifying that a cryptographic identity has been embedded during the manufacturing process, allowing to verify the expected device provenance.

A **Device attestation** certificate asserts that the device's manufacturer has embedded a cryptographic key in a device, allowing a verifier to determine which manufacturer created the device, and to authenticate evidence of trustworthiness properties of the system.

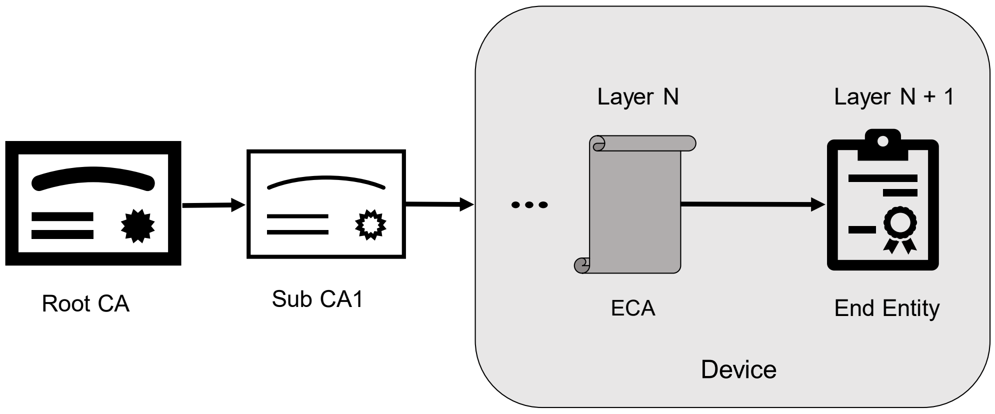


Figure 3.5. Certificate hierarchy with Embedded CA (source: [4]).

Certification

Within a DICE layer TCB is possible to implement an **Embedded Certificate Authority** (ECA) rooted in the DICE HRoT, that allows to issue certificates for a higher layer's keys. This enables to higher layers and external entities the ability to verify the trustworthiness properties for a given DICE TCB layer.

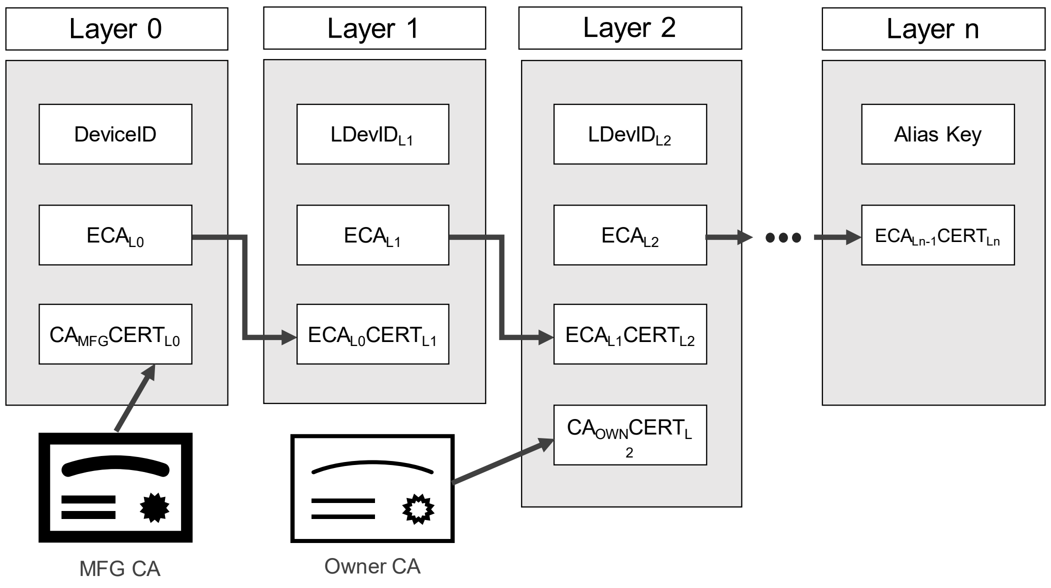


Figure 3.6. Layered certification example (source: [4]).

For example in the figure 3.6 is depicted the scenario in which layered certification is provided. In particular, each TCB layer has generated its specific **identity key**, while the layer 0 has obtained a manufacturing certificate used as **IDevID**. It should be noted that each TCB layer implements an ECA that issues certificates for a higher layer.

The specification [4] defines two models for intra-layer certification:

- an ECA-embedded **policy** describes how and when to issue a certificate for a higher layer;

- a **Certificate Signing Request** (CSR) is created by the higher layer and is sent to the ECA accepting certificate enrolment requests.

Another solution to obtain device identity is to interact with an external CA (i.e. not an ECA). In this case, the identities can be provisioned when deploying the device in the network or at manufacturing time.

During manufacturing two general approaches are adopted:

- device keys and identity credentials are provisioned when the device is still not operating;
- device keys and identity credentials are generated after the device has requested the provisioning.

3.3 Attestation architecture

Whenever changes are made to a trustworthy environment, is necessary to determine if the new state of the system is trustworthy as well. In this context, an **attestation architecture** provides a framework to determine if a trust change occurred and to enable appropriate responses.

The attestation architecture specification [5] defines a set of roles (i.e. Attester, Endorser, Verifier, Relying Party, and Owner) characterising the attestation architecuture. Each **role** consists in an entity exchanging messages with other roles according to the deployment model adopted. Even if different deployment models can be adopted, the selected model does not change the roles and their responsibilities. Furthermore, the framework includes the definition of certificate extensions that can be leveraged to construct attestation evidence

The basic functionalities of the architecture are the generation, transportation, and appraisal of the evidence.

3.3.1 Attestation roles

There are five roles defined within the attestation architecture as shown in figure 3.7, where each role produces or consumes attestation information and interacts with the other roles based on the deployment model adopted.

Attester role

The main responsibility of the **Attester** role is to provide attestation evidence to a Verifier, leveraging an identity used to achieve authentication of the evidence. The underlying identity is often provisioned during the manufacturing process, where the manufacturer embeds the credentials in the entity implementing the Attester role.

As shown in the figure 3.8, the Attester is made up by two components:

- the **Target Environment**;
- the **Attesting Environment** which collects the assertions, called **Claims**, about the trustworthiness properties of the Target Environment.

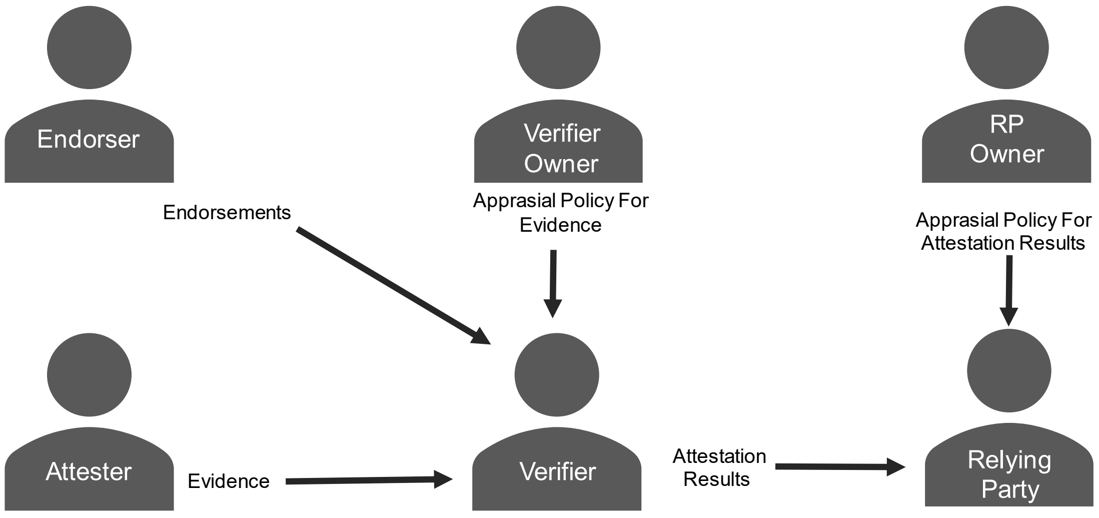


Figure 3.7. Attestation Roles and message flow (source: [5]).

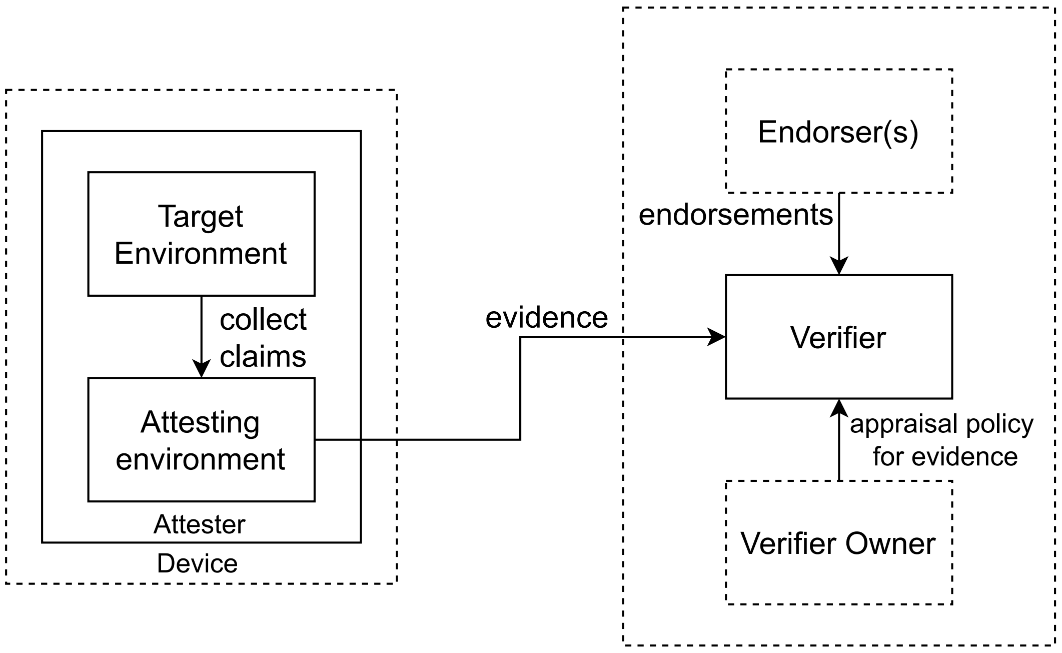


Figure 3.8. Device with Attesting Environment and Target Environment (source: [5]).

In light of the definitions provided, any TCB layer can be an Attesting Environment that generates **Evidence** that is a Claim authenticated and integrity protected. For instance, prior the execution of a layer L_N, the preceding layer (e.g. layer L_{N-1}) performs the role of Attesting Environment, while the former is the Target Environment. When the layer L_N takes control, it becomes in turn the Attesting Environment for the layer L_{N+1}, and so on.

Endorser role

The **Endorser** is the entity that creates reference values or measurements known to be correct, called **Endorsements**, containing assertions about the device's intrinsic trustworthiness properties that can be used as reference values by a Verifier seeking to compare Evidence to values that correspond to a manufacturer's result.

Such Endorsements are typically authorized using digital signatures such as with certificates.

For a given DICE layer, there may be multiple Endorsers. In this scenario, if the layer $\mathrm{L_N}$ or an higher one implements the Attester role, the previous layer $\mathrm{L_{N\text{-}1}}$ implements the Endorser role.

Evidence and Endorsement values transmitted over a public network can be subject to privacy concerns. Due to this, is responsibility of the protocol used to perform this task to confidentiality protect the payloads of Evidence and Endorsement values.

Verifier role

The Evidence and Endorsements are sent to the **Verifier**. The entity implementing the Verifier role, accepts Endorsements and Evidence, applies Appraisal Policies for Evidence, and then sends the **Attestation Results** to one or more Relying Party. For example, in a Remote Attestation scenario, this role is implemented by a service provider.

The policies are obtained from an entity with whom the Verifier has a trust relationship. Once obtained, the policies are authenticated and the Verifier is trusted to correctly apply the supplied policies.

Verifier owner role

The policies obtained from the Verifier are provided by the **Verifier Owner** role. In particular, the Verifier Owner generates **Appraisal Policy for Evidence** and sends it to the Verifier, which uses it to determine the acceptability of the Evidence and Endorsements.

A policy is used for various goals:

- to determine which Evidence and Endorsements are acceptable, based on a database of acceptable Endorsements;
- to determine the trustworthiness state of the Attester;
- to best represent the state of the Attester as Attestation Results.

Relying party role

The Attestation Results produced by the Verifier are conveyed to the **Relying Party**. Once received, the Attestation Results are evaluated against the Appraisal Policies for Attestation Results that are received from a trusted entity and according to the evaluation, the Relying Party takes subsequent actions. For example, possible actions include granting or denying access, triggering transactions, and so on.

The Verifier is trusted from the Relying Party to correctly fulfil its role (i.e. evaluate Attestation Evidence, apply Appraisal Policies for Evidence, and produce Attestation Results).

Relying party owner role

The trusted entity that provisions the Appraisal Policies for Attestation Results to the Relying Party is the **Relying Party Owner**. The Relying Party Owner role is implemented by an entity that determines if the policies supplied are applied correctly and if the results generated based on the policies are the intended ones.

The **Appraisal Policies for Attestation Results** defines whether Attestation Results provided by the Verifier about an Attester can be considered acceptable.

It should be noted that, as with some other Attestation Roles, the Relying Party Owner Role and the Relying Party Role may be implemented by the same actor.

3.3.2 Role messages

Role messages consist of Claims flowing among the roles.

Actors evaluate the role messages based on the reputation of the entity asserting the role message. As part of the attestation trust model, such messages must be authenticated and integrity protected, so that the receiving Actor is able to determine both if the sender is the expected one and if the message has been subject to unauthorized manipulations.

Critical in this context if the freshness of Role messages. In fact, the efficacy of the trustworthiness properties can decrease over time or after the collection and reporting of Evidence.

Message freshness can be achieved by including as part of Evidence:

- a requester supplied nonce;
- a timestamp;
- a validity period.

Evidence

Evidence is a role message containing Claims from the Attester, describing the state of the device or entity.

Normally this role message is generated as a result of a request (i.e. challenge). For example, when the attesting environment at layer L_N collects claims about the target environment at layer L_{N-1}, it can supply Evidence about the layer L_N in an identity credential issued by the attesting environment.

Finally, Evidence can describe previous states of the device such as the state of the Attester during the initial boot and also states that may change among different requests and should contain freshness Claims that enhance the Evidence.

Appraisal policy for evidence

An **Appraisal Policy for Evidence** is a role message, used by the Verifier, that contains policies that reconcile trustworthiness Claims in Evidence with expected operational conditions involving the Attester.

Endorsements

Endorsements are reference values used by the Owners to form Appraisal Policies, signed by the Endorser.

In a DICE layered architecture, if the layer L_{N-1} creates the Endorsement values, then the layer can supply Endorsements about a layer L_N. For example, if the lower layer applies memory layout randomization to the upper layer when loading an executable in memory, the former one collects the measurements by accessing the upper layer and creating the Endorsement values.

Attestation results

Attestation Results are role messages containing the results of attestation Evidence appraisals that are conveyed to the Relying Party and expected to comply with the Verifier Owner policies. Subsequently, the Relying Party enables actions based on the results provided.

The Verifier must provide authentication, integrity, and confidentiality, protecting the message.

Appraisal policy for attestation results

An **Appraisal Policy for Attestation Results** is an input to a Relying Party that contains policies that reconcile trustworthiness claims in the Attestation Results with expected operational conditions involving the Attester.

3.3.3 Layered device attestation

In the DICE layered architecture, the trust in a given layer depends on the trustworthiness of the previous ones. With **layered attestation** the specification [5] refers to the process of attesting the layer L_N based on Evidence signed by the layer L_{N-1}which in turn is attested by the layer L_{N-2} and so on up to the DICE HRoT that is inherently trusted. As a result, a Verifier of layered attestation must evaluate the attestation evidence of previous layers before trusting the current layer. The overall process is shown in the figure 3.9.

Attestation Verifiers require attestation evidence that can be conveyed using several techniques. The specification describes the following approaches:

- X.509 identity certificates and certificate revocation lists (CRLs) with extensions that contain Evidence;
- X.509 attribute certificates containing Evidence;
- manifest containing Evidence.

Evidence as X.509 certificate extensions

X.509 extensions encode reference Endorsements about a DICE TCB environment. Such extensions are used by certificate issuer to assert the trustworthiness claims that apply to

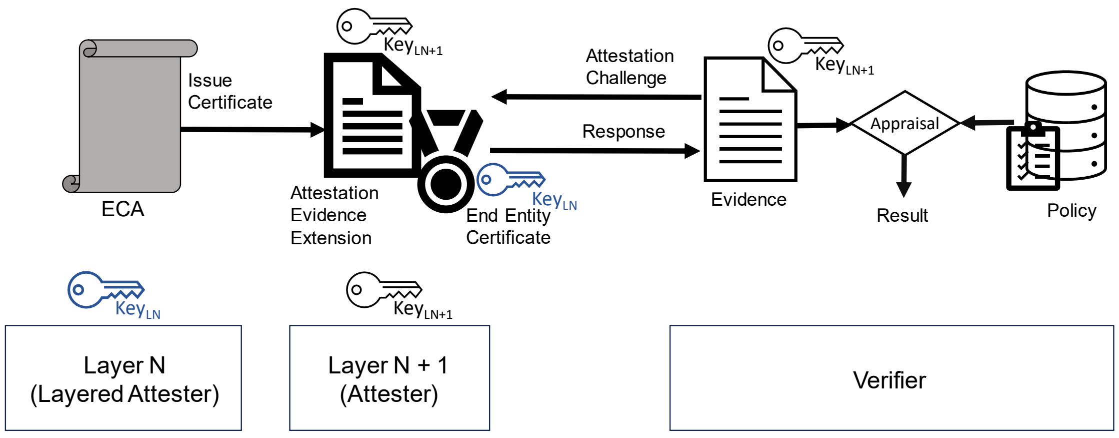


Figure 3.9. Layered Attestation (source: [5]).

the DICE TCB that protects the subject private key identified by the certificate subject public key. This field is used to carry the public key and identify the algorithm with which the key is used (e.g. RSA, DSA, or DH) [27]. In this way, when the certificate is presented to a Verifier, the reference Endorsements are available for trustworthiness evaluation.

A CRL issuer uses an extension to assert that these trustworthiness claims apply to the DICE TCB environment that protects the subject private key that is identified by the certificate serial number, since it identifies the certificate that identifies the subject public key. A Verifier having the CRL, can use Endorsements contained in the extension to evaluate the TCB properties associated with the revocation request. Endorsements in a CRL describe claims that are no longer trustworthy.

The **TCB Info Evidence Extension** defines attestation Evidence about the DICE layer that is associated with the Subject key. The Subject and SubjectPublicKey fields identify the entity to which the DiceTcbInfo extension applies. When this extension is used, the measurements in Evidence usually describe the software/firmware that will execute within the TCB. If this extension is supplied, the AuthorityKeyIdentifier must be supplied since it allows the Verifier to locate the signer's certificate by means of the keyIdentifer that identifies the Issuer public key. The issuer of the certificate must ensure that any non-constant field within the extension contributes to the CDI that generated the subject key. For non-constant field (e.g. measurement and values that would affect the trustworthiness properties of a device if changed), the issuer must ensure that it derives the value by measuring the subject layer L_{N+1}. The Verifier queries a database containing Endorsements that correspond to this Evidence using fields contained in the DiceTcbInfo extension such as querying by digest, vendor, model, and version values.

When the initial state of a DICE TCB is represented by multiple measurement, the **Multiple DiceTcbInfo Structures Extension** can be used. This certificate extension defines a sequence of DiceTcbInfo structures, one for each measurement.

The **UEID Evidence Extension** contains a UEID that is an identifier which identifies an individual manufactured entity, hence must be universally and globally unique across manufacturers and countries. Therefore, this extension identifies the device containing the private key and is identified by the certificate's subjectPublicKey. The issuer must ensure that the content of this extension contributes to the CDI which generated the subject key, such that a change in the field value will cause a change in the CDI.

Evidence as X.509 attribute certificate

Evidence might be created using an X.509 attribute certificate that is signed by an attestation key. Evidence is collected by the Attesting Environment that controls the attestation signing key about a Target environment.

Endorsements

To encode Endorsements three approaches are defined within the specification.

Endorsement values may be conveyed using certificate extensions containing a manifest structure that is signed by an Endorser. A **manifest** is defined as a set of references. In particular, it contains reference values about the target TCB and can be used by a Verifier to appraise Evidence contained in a DiceTcbInfo extension. It is possible to include extension containing a **Universal Resource Identifier** (URI) that locates a reference manifest.

Finally, two other possible approaches are deploying **attribute certificates** which may contain attribute values that endorse an Attester or **Endorsement manifest**, that is a signed structure containing Endorsement claim. In the last case the manifest signing key may have an associated certificate or other credential that contains Endorsement claims.

Chapter 4

Quantum impact on the current technologies

In this section, an overview of the quantum advent's effects on the current cryptographic technologies is given. The analysis starts by describing the state of the art of today's cryptosystems and the quantum impact on such technologies, followed by considerations on the migration process towards a quantum-safe state.

Finally, the *Post-Quantum Cryptography* is introduced, along with the cryptographic primitives over which quantum-safe solutions are built upon. Crucial in this context is the National Institute of Standards and Technology (NIST) standardization process [13]. Started in 2016, the goal of this Call for Proposal is the standardization of quantum-resistant cryptographic algorithms. Therefore, the selected algorithms are presented and briefly analysed.

Before diving in the discussion, it is essential to provide an overview of the algorithms that a quantum computer would rely upon to efficiently carry out attacks on the highlighted cryptographic systems. Two algorithms, Shor's and Grover's algorithms, stand out as the most significant ones in this context.

4.1 Shor's algorithm

Shor's algorithm is known for its ability to endanger public-key cryptography since it can efficiently factor large numbers and solve the Discrete Logarithm Problem, foundation of security for many public-key algorithms [10].

The algorithm is made up by two components:

- a reduction phase in which the Factoring problem is reduced to the Order-Finding problem, lowering the complexity;
- the quantum algorithm to solve the Order-Finding problem.

The *Order-Finding problem* consists in determining the period of a modular exponential function.

Formally, given the exponential function a^x the modular exponential function is defined as the remainder of the division among the exponential function and an integer N:

$$F_N(x) = a^x \bmod N$$

The order of the modular exponential is the smallest positive integer x such that:

$$a^x \bmod N = 1$$

The strategy over which the Order-Finding solution is based on consists in computing the function $F_N(x)$ for many values of x in parallel, aiming to detect the period in the function sequence's values. It is known that using randomization, factorisation can be reduced to finding the order of an element. In the following is described on a high level the algorithm's workflow:

1. pick a random value $x \bmod N$;
2. compute the order r of x;
3. compute the $\gcd(x^{r/2} - 1, N)$.

The quantum algorithm to compute the order can be found in [10]. The value computed in the last step fails to be a non-trivial factor of N only under the two conditions

$$r \bmod 2 = 0$$

$$x^{r/2} = -1 \bmod N$$

Using this criterion, it can be shown that this procedure, when applied to a random $x \bmod N$, yields a factor of N with probability at least

$$p = 1 - 1/2^{k-1}$$

where k is the number of distinct odd prime factors of N.

The time complexity of the algorithm is $O((\log N)^3) = O(n^3)$, respectively the number to factor (e.g. N) and the number of bits required to represent N.

Hence, a quantum computer equipped with adequate resources and computational power can effectively utilize Shor's algorithm to efficiently solve the factorisation problem as well as the DLP, thereby jeopardizing systems reliant on this mathematical property.

It is crucial to emphasize that while there are currently no implementations capable of factoring large numbers, successful factorizations of small numbers have been achieved [28]. Therefore, it can be confidently stated that the algorithm is effective and implementable.

However, it is uncertain when a large and practical machine will be constructed to handle large numbers.

4.2 Grover's algorithm

Grover's algorithm, in contrast, endangers symmetric algorithms since it is able to reduce the amount of operations needed to break a symmetric key, or more in general provides a speed up for searching an unstructured database with respect to the classical algorithms.

The task that the algorithm aims to solve can be expressed as follows:

given an abstract function *f(x)* that accepts search items x, a item x_0 is a solution to the search task if

$$f(x_0) = 1$$

otherwise

$$f(x_0) = 0$$

The *Search Problem* consists in finding any item such that $f(x_0) = 1$.

Based on this problem, the purpose of the algorithm is essentially inverting a given function $y = f(x)$ such that on a quantum computer, Grover's algorithm allows to solve the search problem. Hence the algorithm enables the search for specific solutions across all possible input combinations.

Formally, as the ETSI states [29], if a problem has the following four properties:

- the only way to solve the problem is by repeatedly answering and checking the answers;
- the number of possible answers to check corresponds to the number of inputs;
- it takes the same amount of time to check every possible answer;
- there are no indications of which answers may be better, thus randomly generating candidates is equivalent to follow a precise order.

The time required for a quantum computer to solve problems with these four properties is proportional to the square root of the number of inputs.

Classically, in the worst case, $f(x)$ has to be evaluated a total of $N - 1$ times, where N is the total number of items (e.g. in the symmetric algorithms and hash functions contexts, this can be seen as all the possibilities that must be evaluated to respectively brute force the secret key or a pre-image). On a quantum computer, the algorithm can find a specific entry in an unsorted database of N entries in $\sqrt{N}$ searches [30]. Stems that, roughly, the algorithm's complexity is $O(\sqrt{N})$.

4.3 Cryptographic schemes affected

4.3.1 Asymmetric schemes

Any cryptosystem, security protocol, and product relying on algorithms that exploit the mathematical complexities of Integer Factoring and Discrete Logarithm will no longer be deemed secure. This includes RSA, Diffie-Hellman (DH), and the Elliptic Curve Cryptography (ECC) [31].

RSA is an asymmetric cipher based on the complexity of Integer Factorisation used to encrypt and sign data.

When encrypted traffic is intercepted, the public key could be determined by examining the destination of the traffic and using Shor's algorithm, the private key could be derived from the public key. Recall that for a public key (N, e), the private key is the value d such that

$$e \cdot d = 1 \bmod \phi(N)$$

where

$$\phi(N) = (p - 1)(q - 1)$$

$$N = p \cdot q$$

Once N is factored into primes, it is straightforward to compute the private key. Furthermore, publishing the public key would be equivalent to posting the private key as well.

Not only would all data encrypted with this method be vulnerable, but no message could be guaranteed to be secure, effectively destroying the purposes of the encryption and digital signatures.

Unlike RSA, DH and ECC variants base their security on the Discrete Logarithm Problem. *Diffie-Hellman* is an asymmetric cipher widely deployed that uses the aforementioned properties to transmit keys securely over a public network. Rather than exploiting the classic DLP, algorithms that employ *Elliptic Curve Cryptography* (e.g. ECDH) rely on the hardness of computing the Elliptic Curve Discrete Logarithm for their security.

The difficulty of breaking these cryptosystems is based on the difficulty in determining the integer r such that:

$$g^r = x \bmod p$$

which can be expressed as:

$$r = \log_g x \bmod p$$

The integer r is called the DLP of x to the base g.

Unfortunately, both the technologies are under the same threat that RSA is: a modification of the Shor's algorithm could solve the DLP behind ECC and DH [32]. In fact, smaller keys are needed for an ECC system than for an equivalent RSA system, therefore smaller quantum computer could break ECC before they could break RSA.

The underlined algorithms are widely used in various applications:

- Public Key Infrastructure (PKI);
- Secure Software Distribution;
- Key Exchange over a Public Channel;
- Virtual Private Networks;
- Secure Web Browsing (e.g. SSL/TLS);
- Secure Boot.

4.3.2 Symmetric schemes

For symmetric cryptography quantum computing is considered a minor threat [31]. This is because the foundational components that these algorithms depend on are not based on computational assumptions vulnerable to the Shor's algorithm. The only known threat is the Grover's algorithm.

In the case of symmetric ciphers (e.g. block and stream ciphers), the primary target of a quantum attack is the brute force search of the secret symmetric key. The Grover's algorithm demonstrates that a quantum computer can enhance the search process, resulting in a **quadratic speed up** compared to classical search methods.

Consequently, a quantum computer would not significantly speed up the brute force search to find symmetric keys much faster than classical computers, ensuring that the level of security can be retained by doubling the key length, assuming that the symmetric key algorithm does not rely on a structure that can be exploited by quantum computers. This means that for a *n-bit* cipher the quantum computer operates in $\sqrt{2^n} = 2^{n/2}$ attempts.

In practice, a symmetric cipher with a key length of 128-bits would provide a security level of 64-bits key. For example, the *Advanced Encryption Standard* (AES) is considered to be one of the resilient cryptographic algorithms in a quantum era, but only when is used with key sizes of 192 or 256 bits that results in a security equivalent to 96-bits and 128-bits key sizes. Moreover, the Grover's algorithm does **not parallelise** well. As described in [33], even running several instances of quantum computers in parallel, the improvements offered to the brute force search would be minimal.

In the table 4.1, are shown the classic and quantum key strength provided by different cryptographic schemes in light of the aforementioned quantum algorithms.

Table 4.1. Comparison of classical and quantum security levels for the most used cryptographic schemes

Crypto Scheme	Key size	Classic strength (bits)	Quantum strength (bits)
RSA-1024	1024	80	0
RSA-2048	2048	112	0
ECC-256	256	128	0
ECC-384	384	256	0
AES-128	128	128	64
AES-256	256	256	128

4.3.3 Hash functions

The family of hash functions suffer from a similar problem as symmetric ciphers.

In the context of hash algorithms, attackers aim to achieve either a **collision**, which refers to seeking for two different messages that produce the same hash value, or a **pre-image**, which means looking for the message corresponding to a specific hash output. According to previous considerations on the Grover's algorithm, it can be exploited to find a pre-image of a specific hash value in square root steps of its length.

To ensure pre-image resistance, it is necessary to increase the size of the output by doubling it to keep the same level of security.

On the other hand, it has been proved that is possible to combine Grover's algorithm with the Birthday Paradox to enhance the collision search process [34].

The *Birthday Paradox* [35] refers to the phenomenon whereby the likelihood exceeds 50% that within a group of at least 23 randomly selected individuals, there will be at least 2 individuals sharing the same birthday. Exploiting this paradox, it is possible to conduct an attack known as *Birthday Attack* [36]. Birthday attack is a collision-find attack to detect two messages that collide to the same hash value. If x is the given hash-length in bits, then with probability 50% in roughly $2^{x/2}$ time with $2^{x/2}$ random messages as input is possible to find a collision.

By creating a table of size $\sqrt[3]{N}$ (e.g. this is the quantum resource requirement that must be satisfied by a quantum computer) and utilizing Grover's algorithm to find a collision, an attack is said to work effectively. This means that to provide a b-bit collision resistance, a hash function must provide at least a 3b-bit output. As a result, many hash algorithms are disqualified in the quantum era, except algorithms such as *SHA-2* and *SHA-3* with longer outputs.

In the table 4.2 is summarized the impact on the different most-widely used cryptographic algorithms.

Table 4.2. Summary of the quantum impact on different cryptographic algorithms

Crypto Algorithm	Type	Purpose	Quantum impact
AES	Symmetric key	Encryption	Larger keys needed
SHA-2, SHA-3	—	Hash functions	Larger output needed
RSA	Public key	Signatures, key establish.	No longer secure
ECDSA, ECDH	Public key	Signatures, key exchange	No longer secure
DSA	Public key	Signatures, key exchange	No longer secure

4.4 Migration to the Quantum safe state

4.4.1 Migration challenges

According to the National Institute of Standards and Technology (NIST), the PQC migration cannot be achieved as a simple drop-in replacement [37]. In fact, the multitude of contexts in which the transition will occur, encompassing IoT devices, embedded systems, applications, and protocols causes the migration to be difficult for several reasons.

For instance, PQC algorithms present significant differences in key sizes, ciphertext sizes, and signature sizes, as well as communication and computational requirements that may not be compatible with the current infrastructures.

Also, many algorithms introduce new requirements (e.g. state management) that will demand modification to existing frameworks, and these cryptographic schemes may be suitable in certain scenarios but not aligned in others.

Furthermore, the introduction and adoption of innovative schemes necessitates the NIST submissions to undergo additional cryptanalysis. Thus is crucial to emphasize that there will not be a single replacement or solution, but rather multiple options according to the needs and environment considered.

Such features must be carefully considered under several considerations such as performance, security, and implementation.

Performance considerations

Considering that PQC algorithms demand greater computation, memory, storage and communication requirements, an important research field regards the performance's assessment in various deployment scenarios. In fact, before PQC can be adopted in real world contexts and systems, research is needed to develop practical and suitable implementations.

For example, consider networking. Larger key sizes, digital signatures and keys, introduces latency and demands an higher bandwidth within secure communication protocols (e.g. TLS). As a consequence, the performance is directly affected and research is needed to develop efficient and scalable solutions.

The modifications in turn impact the set of network devices optimized for the current cryptographic protocols in favour of performance and scalability.

Included in this research challenge is the need to understand performance in IoT devices where compute, memory and battery constraints are first order considerations. This can affect the practical adoption of PQC schemes within those devices until a practical implementation is made available.

Security considerations

The implementation's impact of new public key algorithms affect the performance as well the security.

First, unlike RSA and ECC algorithms, PQC candidates have a different set of trade-off in configurable parameters such as key sizes, ciphertext size, and computation time. Hence a key challenge is understanding the trade-off between security and algorithm requirements for a wide variety of usage domains since a wrong configuration could extend the attack surface decreasing the security offered.

Furthermore, the PQC candidates are currently under further cryptanalysis with respect to the fully-understood RSA and ECC algorithms (e.g. Multivariate Cryptographic Schemes). In particular, a critical research field is the analysis of the algorithms across protocols and contexts. Here researchers analyse systems to discover weaknesses under various models which rely on assumptions on the adversary behaviour.

One important category of weaknesses in this area are the **Side Channel Vulnerabilities**. A Side Channel Attack consists in exploiting an information leak within a certain implementation which allows to gather information to negatively affect a system. This is possible since, in general, individual PQC algorithms will introduce new patterns of memory usage, timing, communication, failure modes which could be exploited by an attacker to compromise a system. Hence, how can implementations help to guard against side channel attacks is an important research field as well.

Implementation considerations

The implementation of cryptographic algorithms itself is critical for the security of the systems. In fact, a wrong implementation may introduce vulnerabilities.

This includes not only the implementation of the schemes within a system, but also the translation of the algorithm from a mathematical formulation to platform specific architectures. This is due to the complexity of mathematical algorithms, and the difficulty in implementing and translating them. Factors such as data representation and layout, and its interactions with the system memory and buffering mechanisms, can introduce vulnerabilities which may go unnoticed.

The implementation complexities and issues become even more complicated when dealing with embedded systems and IoT devices. The variety of devices in memory size, computing resources and power availability causes the implementation to be more difficult, especially considering that IoT devices are exposed to tampering.

Finally, it should be noted that the existing reference code associated to the NIST submissions cannot be practically adopted, hence it is important to rely on robustly implemented libraries (e.g. Liboqs by Open Quantum Safe [38]) to support experimentation on specific platforms.

4.4.2 Cryptographic Agility

Definitions

With the new advancements in cryptanalysis, the effort required to break a cryptographic algorithm will decrease, rendering systems vulnerable. Furthermore, new algorithms may be necessary in response to known vulnerabilities or as a better alternative to existing solutions. With such premises, it is needed to assume in advance that as the technology evolves and the cryptanalytic techniques improve, any algorithm will be at a certain point in time obsolete.

In light of this, is critical to design and project systems with a certain degree of flexibility, in such a way that allows to apply modifications and replacements without the need to re-design the whole infrastructure. This property is defined as *Cryptographic Agility*.

As defined in [39], crypto-agility describes the feasibility of replacing and adapting cryptographic schemes in software, hardware and infrastructure, and should enable such procedures without interrupting the flow of a running system.

To ensure cryptographic agility there are some principles that must be followed when designing systems:

- design in light of crypto-agility from the earlier design stages;
- assume that the cryptographic algorithms will be broken at a certain point in the future;
- implement the crypto algorithms in a crypto-agile manner;
- allow efficient change of algorithms.

Besides the just defined *Algorithmic Agility*, defined as the capability to migrate from a cryptographic suite to another with as little as possible human intervention, several forms of crypto-agility or **modalities** are described in [39]. The goal of crypto-agile modalities is to enable modifications and provide flexibility in broader contexts.

Modalities

Whereas algorithmic agility is restricted to considerations on algorithms and cryptographic schemes, *Implementation Agility* is one modality that answer the question of how the implementation process can be enhanced so that applications can be considered crypto-agile.

Besides the agility in terms of implementation, is important to provide a convenient workflow that enables combining cryptographic building blocks in a secure way, such as with hybrid schemes. This modality is defined as *Composability Agility*.

In [39], particular importance is given to the *Security Strength Agility*, defined as the dynamic scaling of an algorithm's security level based on the provided configuration. This concept applies especially to new vulnerabilities or broken algorithms, enabling a fast and secure adaption.

A cryptographic algorithm need to run across different platforms and systems. The *Platform Agility* property, is defined as enabling the independent and seamless usage of the algorithms on different platforms, independently from the underlying software and hardware components. This is critical in the context of IoT and embedded systems, due to the variety of both the worlds.

A similar property is the *Context Agility*, which indicates the flexibility of algorithms and security levels so that the algorithm can be dynamically configured according to the system's properties. With respect to the Security Strength Agility, this modality refers to the limitations posed by a system's properties rather than achieving a certain extent of security by means of a provided configuration.

A platform should be designed in such a way that enables migrating from one scheme to another when necessary. Supporting multiple schemes leads to a more flexible system which can be adapted based on real-time needs. This is called *Migration Agility*. An example of this crypto-agility modality can be found within the cipher-suite negotiation within the TLS protocol.

Due to the aforementioned issues, since a cryptographic algorithm may become obsolete, it is necessary to enable a seamless retirement of vulnerable cryptographic schemes. Defined as *Retirement Agility*, this property enables the retirement of obsolete or broken algorithms from a system.

There is no one single security standard that must be followed. Moreover, different security standards may be required and the infrastructure should be designed to allow modifications to be compliant with frameworks and regulations. The *Compliance Agility* is defined as the capability to reconfigure the infrastructure so that is compliant with regulations and frameworks. This includes the local laws and the modalities in which service providers ensure the availability of a product in different regions.

4.4.3 Hybrid schemes

The IETF in [40] addresses the need to employ hybrid schemes in which traditional and PQ algorithms are combined according to specific patterns, since there may be the desire or requirement for protocols to use both the algorithm types. This approach enables protections against quantum attackers in addition to the security properties provided by traditional algorithms. Furthermore an hybrid setting allows to implement post-quantum algorithms alongside traditional ones for ease of migration. Schemes that combine post-quantum and traditional algorithms for key establishment or digital signatures are referred to as **hybrids**.

More in general, a *Post-Quantum Traditional Hybrid* (PQ/T) scheme is defined as a multi-algorithm scheme where at least one component is quantum-resistant and at least one is traditional. A *Multi-Algorithm Scheme* is a cryptographic scheme in which are incorporated more algorithms with the same purpose.

The security of hybrid schemes depend on the security of its components. If the post-quantum component is broken, the scheme will be vulnerable to a quantum-attacker but secured against a classical attacker.

The hybrid schemes takes more importance in the context of *Retrospective Decryption*. Aslo defined as "Harvest Now, Decrypt Later" (HNDL) attack, it refers to an attacker storing encrypted information with the scope of decrypting the stored data when a practical quantum computer will be available. The deployment of hybrid schemes may help to guard against HNDL.

Fo example, how post quantum certificates should be managed in a migration scenario, enabling a hybrid configuration, is addressed in [40]. In the cited document, the IETF states that hybrid certificates need to contain public keys for two or more component algorithms where at least one is a traditional algorithm and at least one is a post-quantum algorithm.

The public key could be included as a composite public key or as individual public keys. A *Composite Cryptographic Element* is defined as a cryptographic element that incorporates multiple Component Cryptographic Elements of the same type to compose a Multi-Algorithm Scheme. A *Component Cryptographic Element* is a cryptographic component in a Multi-Algorithm Scheme.

Altough separate certificates could be used for individual component algorithms, an hybrid certificate could be used to facilitate a hybrid authentication protocol.

Is important to clarify that, as IETF underlines, the use of PQ/T hybrid certificates does not necessarily achieve hybrid authentication of the identity of the sender. For example, an end-entity certificate that contains a composite public key, but which is signed using a single-algorithm digital signature scheme could be used to provide hybrid authentication of the source of a message, but would not achieve hybrid authentication of the identity of the sender.

4.5 Post-Quantum Cryptography

4.5.1 Post-Quantum families

Post-Quantum Cryptography is the cryptographic branch that can provide security against both quantum and traditional computers, since PQC algorithms are based on mathematical problems that cannot be compromised by the aforementioned algorithms.

There are five families of primitives that these algorithms fall into, all based on **trapdoor functions**.

A trapdoor function consists in a function that is computationally feasible to compute in one direction but infeasible in the opposite direction without specific information.

Lattice-based cryptography

Lattice-based Cryptography is a form of public-key cryptography that avoids the weaknesses of RSA, having the foundation of its security on *lattices.*

A *lattice* is defined as a set of points in n-dimensional space with a periodic structure. Therefore, a lattice is defined as a set of vectors which can be represented as a matrix. Rather than multiplying primes, lattice-based cryptography involves computations among matrices.

The security of these constructions is based on the presumed hardness of lattice problems, with the **Shortest Vector Problem** (SVP) being the most fundamental among them. Here, is given as input a lattice represented by an arbitrary basis (e.g. combination of vectors), and the goal is to output the shortest non-zero vector in it. This problem is NP-hard.

More in depth, lattice-based cryptographic schemes take advantage of the *worst-case to average-case reduction* principle. This means that breaking such schemes is at least as hard as solving several lattice problems in the worst case. In other words, breaking the cryptographic construction implies an efficient algorithm for solving any instance of the underlying lattice problem, which in most cases consists in approximating lattice problems (e.g. SVP) to within polynomial factors, that is conjectured to be a hard problem [41]. This implies that all the keys are strong and hard to break in the easiest case as in the worst case.

Lattice-based algorithms are very fast and considered quantum safe. Moreover the flexibility offered by the configurable parameters allows to adapt the scheme to several cases. The drawback is that a wrong configuration would lower the degree of security, expanding the attack surface.

Currently there are no known quantum algorithms for solving lattice problems that perform significantly better than the best known classical algorithms.

Code-based cryptography

Code-based cryptography uses the theory of error-correcting codes [42].

An **error-correcting code** is an encoding scheme that allows to transmit messages as binary numbers, in such a way that the message can be recovered even if some errors occur during the transmission causing changes to one or more bits.

The most widely known efficient error correcting codes are **Goppa codes**. To build a secure coding scheme around Goppa codes, it is necessary to keep the encoding and decoding function a secret and to publicly disclose a disguised encoding function. The purpose of this last encoding function is to allow a mapping of a message to a set of code words. To remove the secret mapping and recover the plaintext is necessary to possess the secret decoding function.

In code-based cryptographic algorithms, the ciphertext consists of a codeword which have been combined with errors that can be removed only with the knowledge of private information. Such algorithms provides further redundancy to the communication so that the receiver can in real time correct errors, which may occur during transmission.

This type of scheme is based on a mathematical problem called **Syndrome Decoding Problem**. This is known to be an NP-complete problem without the knowledge of the code, hence is computational hard to reverse using either a conventional or quantum computer.

Generally, Code-based schemes offer short signatures but very large key sizes, which is the biggest drawback of this cryptosystem that prevents its practical adoption.

Despite the advantages offered, code-based algorithms perform weakest among the quantum safe primitives.

Multivariate polynomial cryptography

Multivariate cryptography is based on the hardness of finding a solution to a system of multivariate quadratic equations over finite fields [43].

The trapdoor function leveraged in this primitive takes the form of a multivariate quadratic polynomial map over a finite field. This mapping transforms linear quadratic equations into non-linear quadratic polynomials.

More efficient schemes leverage systems of equations with some hidden structure that is known only to the person that built the system, while the systems of equations are perceived as random to outsiders.

Not so efficient, multivariate encryption schemes present large public keys and long decryption times. This issue stems from the inefficiency that lies behind the decryption process since it includes some guessing that is a required part of the algorithm, ensuring its security.

Furthermore such schemes offer a short signature, rendering it suitable for several deployment scenarios.

Hash-based signatures

Hash-based cryptography offers signature schemes based on hash functions [44] and due to their ubiquity, the security of practical hash functions is well understood.

Such schemes are interesting in this context since the non-number theoretic assumptions allow to avoid the quantum issues aforementioned. More importantly, it is known that even quantum computers cannot significantly improve the complexity of generic attacks against cryptographic hash functions.

Despite the advantages of this kind of signatures, there are some drawbacks.

The main problem is that they usually arise in the context of **one-time signatures** (OTS). One-time signature scheme can only be used once to sign one single message with a specific key. Reusing one-time signature keys would allow an attacker to gather enough information to forge a valid signature.

To overcome the OTS limitation, such schemes are combined with binary tree structures so that instead of using a signing key for a OTS, a key may be used for a number of signatures limited by and bounded to the size of the binary tree.

The main concept behind using a binary tree with hash signature schemes is that each position on the tree is calculated to be the hash of the concatenation of their children nodes. Nodes are thus computed successively, with the root of the tree being the public key of the global signature scheme. The leaves of the tree are built from one-time signature verification keys.

To perform a signature, a OTS unused key pair is picked from the tree. To sign a *i-th* message, the signature is then computed taking into account the *i-th* leaf in the tree, the OTS public key picked, the digest of the leaf, and its authentication path. The authentication path is defined as the path from the root (e.g. the public key) to the leaf selected to compute the signature.

One important feature regards the design of hash-based signature schemes. The designs fall into **stateless** and **stateful** scheme. The former work as normal signatures, while for the latter the signer needs to keep track of some information (e.g. the number of signatures generated using a given key). State-fullness is an important drawback, in that the signer must keep track of which one-time signature keys have already been used, which can be cumbersome in a large scale environment.

A significant strength of hash-based signature schemes is their flexibility as they can be used with any secure hash function, and so if a flaw is discovered in a secure hash function, a hash-based signature scheme needs only to switch to a new and secure hash function to remain effective.

4.5.2 NIST selected algorithms

In this section a general background on the objects considered in the NIST competition is given.

The PQC algorithms address three fundamental applications: encryption, key encapsulation mechanisms (KEMs), and digital signatures.

A *Public Key Encryption* (PKE) scheme consists of three algorithms.

- the key generation algorithm which generates a key pair consisting of a private and public key;
- the encryption algorithm takes a message and a public key to compute a ciphertext;
- the decryption algorithm take a ciphertext and a private key to compute the plaintext.

It is required to obtain the same message in decryption phase if the correct private key is used.

A *Key Encapsulation Mechanism* (KEM) is a cryptographic technique to securely exchange a symmetric key over an insecure channel. The KEM encapsulation results in a fixed-length symmetric key that can be used to encrypt data or to derive an encryption key. It consists of three algorithms:

- the key generation algorithm generates a key pair consisting of a public and private key;
- the encapsulation algorithm takes a public key and a session key to compute a ciphertext;
- the decapsulation algorithm takes a private key and a ciphertext to compute the session key.

If the correct private key is used, then the decapsulation of the ciphertext returns the same key as the encapsulation that generated it.

A *Digital Signature Scheme* (DSS) consists of three algorithms.

- the key generation algorithm generates a key pair consisting of a public and private key;
- the signature algorithm takes a message and a private key and returns a signature;
- the verification algorithm takes a signature, a public key, and a message and returns true or false.

For correctness, a correctly generated signature verifies under the right public key.

After three rounds of evaluation and analysis, the NIST has selected the first algorithms it will standardize as a result of the PQC standardization process.

- PKE/KEM: CRYSTALS-Kyber;
- DSS: CRYSTALS-Dilithium, Falcon, SPHINCS$^+$.

Besides the selected algorithms, another algorithm worth mentioning is the *Extended Merkle Signature Scheme* (XMSS) which is gaining more and more importance in the quantum safe context given the capabilities and features offered by this hash-based signature scheme.

In addition to the selected group of algorithms, which is considered to be the most promising to fit most use cases and ready to be standardized at the end of the third round, the algorithms regarded as alternate candidates instead, are also considered promising algorithms based on the high confidence in their security.

The main issue regards the performance or the algorithms have shown acceptable performance but necessitate further assessment of their security. As a result, the alternate candidates have been advanced to the fourth round of evaluation.

Acceptance criteria

The NIST required the submissions to respect some security properties.

In particular, for PKE/KEM schemes, the Call for Proposal required **Indistinguishability under adaptive Chosen-Ciphertext Attack** (*IND-CCA2*). This security notion for encryption scheme ensures the confidentiality of the plaintext, resistance against chosen-ciphertext attacks, and prevents an attacker from forging new ciphertexts.

A *chosen ciphertext attack* is a cryptanalysis technique in which the attacker can select specific plaintexts and obtain the corresponding ciphertexts. The goal consists in gathering information to compromise the security of the system.

On the other hand, digital signatures schemes were required to provide **Existentially Unforgeable Signatures with respect to an adaptive Chosen Message Attack** (*EUF-CMA*).

A *chosen message attack* is a cryptanalysis technique in which the attacker can select and modify messages to be signed in order to gather information to compromise the security of the cryptographic scheme.

This security property guarantees that an adversary cannot forge a valid signature for an arbitrary message even when the attacker has access to a signing oracle. The EUF-CMA property provides protection against forgery attacks, ensuring the integrity and authenticity of signatures by preventing unauthorized modifications or fraudulent signatures.

Evaluation criteria

To quantify and evaluate the security of the submissions, NIST defined five security strength levels.

More in depth, the levels are based on the computational resources required to perform certain brute-force attacks against the existing standards for AES and SHA3 under different models of the cost of computation, both classical and quantum.

It is important to note that the strength of the categories are defined in a way that leaves open the relative cost of the computational resources. This means that the security levels are valid under any plausible assumption regarding the relative cost of the various resources in a real-world scenario. Hence valid opinions may arise regarding what is considered to be a plausible assumption regarding the cost of computational resources.

Furthermore, there is still uncertainty related to the effective cost and resources that an actual attack may require.

The security categories are described in the table 4.3 and are categorized based on the security, expressed as a function of resources required to break the specified AES and SHA3 algorithms.

Table 4.3. PQC Security levels

PQC Security Level	AES/SHA hardness	PQC Algorithms
1	exhaustive key recovery in AES-128	Kyber512, Falcon512
2	collision search in SHA3-256	Dilithium2
3	exhaustive key recovery in AES-192	Kyber768, Dilithium3
4	collision search in SHA3-256	No algorithm tested at this level
5	exhaustive key recovery in AES-256	Kyber1024, Falcon1024, Dilithium5

The second most important evaluation criteria is the cost. As the NIST specified in the Call for Proposal, cost includes the computational costs associated to the key generation, public and private key operations, the transmission costs for public keys and signatures or ciphertexts, and the implementation costs in terms of resources needed (e.g. RAM).

Additionally, when comparing the overall performance of the algorithms, both computational cost and data transfer cost were considered. For general-purpose algorithms, the evaluation takes in account the cost of transferring public key and signature or ciphertext during each transaction.

For KEMs the key generation cost is evaluated since many applications in which forward secrecy has been achieved, make use of a new KEM key pair for each transaction. On the other hand, in the digital signature schemes the key generation cost is considered less important.

According to the evaluation criteria presented, the algorithms selected result to be the most promising under all the criteria considered.

CRYSTALS-Kyber

Kyber is a IND-CCA secure PKE/KEM, whose security has been thoroughly analysed and is based on results in lattice-based cryptography.

On the performance level it has achieved excellent overall results in software, hardware, and hybrid settings. In particular it has shown fast key generation, encapsulation and decapsulation.

Like the other structured lattice KEMs, Kyber's public key and ciphertext sizes are on the order of a thousand bytes.

In the table 4.4 are summarized the sizes (in bytes) of the keys and ciphertext with respect to traditional cryptographic solutions.

The submission lists three different configurations with three different security levels:

- Kyber-512 with a configuration that provide a security level that matches AES-128;
- Kyber-768 with a configuration that provide a security level that matches AES-192;

Table 4.4. Comparison among Kyber and traditional KEMs/KEXs public key, private key and ciphertext sizes (in bytes)

Security Level	Algorithm	Pub. key size	Priv. key size	Ciphertext size
Traditional	P256_HKDF_SHA256	65	32	65
Traditional	P521_HKDF_SHA512	133	66	133
Traditional	X25519_HKDF_SHA256	32	32	32
1	Kyber512	800	1632	768
3	Kyber768	1184	2400	1088
5	Kyber1024	1568	3168	1588

- Kyber-1024 with a configuration that provide a security level that matches AES-256.

According to the consideration on the quantum algorithms (e.g. Grover) and the symmetric key algorithms, among the sets the usage of Kyber-768 is recommended since it achieves a security level known to be secured against both classic and quantum attacks.

Digital Signature Schemes (DSS)

As previously stated, the DSS selected are Dilithium, Falcon, and SPHINCS$^+$. All the schemes provide EUF-CMA security, as requested in the Call for Proposal.

Moreover, the schemes are based on the **Hash-and-sign** paradigm. This means that the signature is computed only after that the original message is hashed. In this way the scheme can benefit of the collision-resistance of the underlying hash function. Furthermore, the paradigm leads to improvements on the performance level, reducing the size of the signed messages since a fixed digest is signed, ensuring compatibility with a wide range of protocols and systems.

Dilithium is a digital signature scheme that is strongly secure under chosen message attacks based on the hardness of lattice problems. This security notion means that an adversary having access to a signing oracle cannot produce a signature of a message whose signature he hasn't seen yet, nor produce a different signature of a message that he already saw signed.

Internally, this signature scheme applies a hash function and uses the result for the signature generation process.

Dilithium offers a number of options for varying parameters in order to increase security at the cost of either increased sizes and/or slower performance.

It is, along with Falcon, one of the two most efficient signature protocols in Round 3, suitable for a broad range of cryptographic applications.

Falcon is a lattice-based cryptographic signature scheme.

It generally has shorter keys and signatures than Dilithium, although Dilithium has the benefit of not requiring floating-point arithmetic, which leads to difficulties in secure implementations (e.g. to achieve constant-time signing) against side-channel attacks.

Due to its low bandwidth (e.g. public key size plus signature size) and fast verification, Falcon may be a superior choice in some constrained protocol scenarios. However, signing and keys generation are slower than Dilithium and the complex data structures used make the algorithm's implementation challenging with respect to other lattice signature schemes.

SPHINCS$^+$ is a stateless hash-based signature scheme.

The scheme combines the use of one-time signatures, few-times signatures, and trees to construct a digital signature scheme that is suitable for general use.

Is a stateless scheme, hence it does not require the user to keep track of any state between signatures. In contrast, there are also stateful hash-based signature schemes which are faster and produce smaller signatures but require the user to keep state across signatures with negative consequences if the state is mismanaged.

The security of this scheme is based on the underlying hash function. Because of the way SPHINCS$^+$signatures are formed, keys generation and verification are much faster than signing. In particular, performs internally randomized message compression using a keyed hash function that can process arbitrary length messages.

Two designs have been proposed:

- a faster signature computation at the cost of larger signatures;
- smaller signatures at the cost of a slower signature computation.

It is possible to select more extreme trade-offs which may be convenient in some cases. Furthermore, SPHINCS$^+$public keys are very short, but signatures are quite long.

The biggest drawback is related to the complexity of the scheme since a complex scheme's implementation is error-prone. This renders also the security evaluation of the whole scheme harder.

Finally, the design of the algorithm imposes limitations on the number of signatures: for any number of signatures using a public key, there is a negligible likelihood that the signatures will reveal enough of the private key to allow an attacker to forge a signature. To keep this likelihood low, it is necessary to limit the number of signatures performed (e.g. NIST required the ability to securely perform 2^64 signatures).

In the table 4.5 are shown the differences in the key and signature sizes among the described PQC solutions and traditional algorithms (e.g. RSA with a 2048-bit key length and the ECC signature algorithm P256). The SPHINCS$^+$entries are considered in the design of the faster construction at the cost of larger signatures.

Table 4.5. Comparison among PQC DSS and traditional signature algorithms

Security Level	Algorithm	Pub. key size	Priv. key size	Ciphertext size
Traditional	RSA2048	256	256	256
Traditional	P256	64	32	64
1	Falcon512	897	1281	666
1	Sphincs SHA256-128f Simple	32	64	17088
2	Dilithium2	1312	2528	768
3	Dilithium3	1952	4000	3293
3	Sphincs SHA256-192f Simple	48	96	35664
5	Falcon1024	1793	2305	1280
5	Sphincs SHA256-256f Simple	64	128	49856

Extended Merkle Signature Scheme (XMSS)

Stateful hash-based signature scheme, XMSS allows to sign a potentially large but fixed number of messages without relying on mathematical problems [45].

Instead, it is proven that it only relies on the properties of cryptographic hash functions. In particular, an advantage of this scheme is that it does not require that the underlying cryptographic hash function is collision-resistant for its security.

XMSS is suitable for compact implementations, is relatively simple to implement, and naturally resists side-channel attacks but the statefulness of the scheme requires that the secret key used to perform computations changes over time, otherwise no security guarantees remain since it would become feasible to forge a signature over a new message.

It has the smallest signatures among hash-based schemes and used with a multi-tree variant is possible to solve the slow key generation problem.

Unlike most other signature systems, hash-based signatures can so far withstand known attacks using quantum computers.

In the table 4.6 are summarized the families of PQC algorithms, along with examples and notable attributes.

Table 4.6. PQC algorithms families, examples, and notable attributes

PQC Family	Use	Examples	Notable Attributes
Hash-based Crypto	Digital Signatures	XMSS, SPHINCS$^+$	Large signature
Lattice-based Crypto	KEM, Digital Signatures	Falcon, Kyber, Dilithium	Short parameter set
Code-based Crypto	KEM	BIKE, Classic McEliece	Large public keys
Multivariate Crypto	Digital Signatures	EMSS, LUOV, Rainbow	Large keys

Chapter 5

Post-Quantum support in Keystone: design

The purpose of the designed solution is to provide PQ algorithms support in TEE, migrating the system from a quantum-unsafe to quantum-safe state. In the following sections are outlined and described the primary evaluations conducted to design the quantum-safe solution. More in depth, the analysis conducted on the NIST submissions is discussed, along with considerations and motivations that led to select Falcon as the best algorithm in the given context, followed by a qualitative analysis of the algorithm, with features, advantages, and limitations. Then the assessment of the quantum-vulnerable operations performed in DICE and Keystone is presented, which constitutes the starting point in the solution's implementation, discussed in the next chapter.

5.1 PQC submissions evaluation

In order to select the PQ submission to be implemented, the algorithms have been evaluated taking into account the following criteria:

- NIST **Security Levels**;
- **Keypair size**;
- **Signature size**;
- Performance in **keypair generation**, **signature computation**, and **signature verification**.

Given the need to provide a solution suitable for constrained devices, an initial performance assessment was conducted using the **pqm4** [6] benchmark. Such benchmark addresses the evaluation of the algorithms' performance on the ARM Cortex-M4 platform. The considerations derived from this first assessment are the starting point into determining the most promising algorithm

It is critical to underline that the evaluations reported in the following sections regard the algorithms' **clean** implementation, which means that the implementation does not include any platform specific improvement.

5.1.1 SPHINCS$^+$

As regards this hash-based DSS submission, three different signature schemes have been proposed, according to the underlying hash function:

- SPHINCS$^+$-SHAKE256;
- SPHINCS$^+$-SHA-256;
- SPHINCS$^+$-Haraka.

For each version, two variants have been submitted:

- slow version: slower computation, smaller signature size;
- fast version: faster computation, bigger signature size;

Among the three versions, only the **SHA-256** version with the **slow** variant in **simple** form (i.e. it does not use a bit mask when hashing) is evaluated below due to the following:

- the biggest issue in SPHINCS$^+$is the excessive signature size hence reducing the signature size is necessary given the limited storage available in embedded systems, and the SHA-256 variant shows the smallest signature;
- the SHA-256 version is the fastest;
- the memory footprint is smaller.

The main aspects taken into account in the SPHINCS$^+$evaluation is the signature size. As shown in the table 5.1, the algorithm shows private and public keys with sizes comparable to the actual ECC algorithms, but excessive signatures.

Considering the certificates management and the signatures computation usage in the DICE-based Keystone version, the algorithm's deployment would be critical due to memory constraints.

Table 5.1. SPHINCS$^+$parameters size.

scheme	Public key size	Private key size	Signature size
sphincs-sha256-128s-simple	32	64	7856
sphincs-sha256-192s-simple	48	96	16224
sphincs-sha256-256s-simple	64	128	29792

Note. — sizes are expressed in *bytes*

As regards the performance, according to the aforementioned benchmark SPHINCS$^+$shows worse general performance than the other DSS submissions. Such performance are depicted in the table 5.2 considering 3 executions of the algorithm.

Hence, due to the excessive memory needed to store the signatures and the worse performance with regards to the other submissions, SPHINCS$^+$has been disqualified.

Table 5.2. SPHINCS$^+$performance (source: [6]).

scheme	keypair generation	sign	verify
sphincs-sha256-128s-simple	AVG: 985,367,046	AVG: 7,495,603,716	AVG: 7,165,875
sphincs-sha256-192s-simple	AVG: 1,450,073,477	AVG: 13,764,196,955	AVG: 11,763,703
sphincs-sha256-256s-simple	AVG: 952,799,879	AVG: 12,304,132,668	AVG: 16,715,346

Note. — values expressed in CPU *cycles*

5.1.2 CRYSTALS-Dilithium

Dilithium is considered the best algorithm among the submissions due to the following reasons:

- fastest key generation;
- fastest signature computation;
- fast signature verification, comparable to Falcon.

Considering that the algorithm belongs to the lattice-based schemes' family, the keys and signature sizes (shown in the table 5.3) are compact and smaller with respect to many of the PQC submissions.

Table 5.3. Dilithium parameters size.

scheme	Public key size	Private key size	Signature size
Dilithium2	1312	2528	2420
Dilithium3	1952	4000	3293
Dilithium5	2592	4864	4595

Note. — sizes are expressed in *bytes*

The algorithm performance are depicted in the table 5.4 considering 10000 executions of the algorithm.

Evaluating both the performance and the parameters set, the algorithm looks a more promising choice than SPHINCS$^+$and in many scenarios may be the best solution.

The only drawback of the algorithm, besides the parameters size is the memory footprint. In fact the algorithm is RAM demanding, as shown in the table 5.5.

Hence the Dilithium submission has been disqualified due to the memory consumption and the parameters size in favour of Falcon, which has lower RAM footprint, smaller keys and signature, and offers a better trade-off in terms of security level offered and memory required in both execution and storage.

Table 5.4. Dilithium performance (source: [6]).

scheme	keypair generation	sign	verify
Dilithium2	AVG: 1,944,539	AVG: 7,144,383	AVG: 2,064,129
Dilithium3	AVG: 3,365,142	AVG: 11,634,591	AVG: 3,430,286
Dilithium5	AVG: 4,826,422	AVG: 8,779,067	AVG: 4,705,693

Note. — sizes are expressed in CPU *cycles*

Table 5.5. Dilithium memory evaluation (source: [6]).

scheme	keypair generation	sign	verify
Dilithium2	38,308	51,932	36,220
Dilithium3	60,844	79,588	57,732
Dilithium5	97,692	115,932	92,788

Note. — values expressed in *bytes*

5.1.3 Falcon

Lattice-based signature scheme, has been selected among the alternatives due to the following reasons:

- smallest parameter set size;
- lower RAM footprint than Dilithium;
- best tradeoff among security level and keys/signature sizes.

In fact, in the table 5.6 are shown the parameters size, and it should be noted that the most secure version of Falcon, with a security level equals to 5, has $|pk| + |sk| + |sig|$ smaller than the Dilithium version with security level 2, offering a better trade-off.

On the other hand, Dilithium results the best algorithm is terms of performance, as shown in the table 5.7, where 100 executions of Falcon are considered.

Nevertheless, the constraints imposed by the Keystone project disqualifies Dilithium due to the higher sizes and the higher memory required. In fact, Falcon has a considerable smaller RAM footprint than Dilithium, as shown in the table 5.8, with a memory consumption corresponding to the security level 5 version comparable with the security level 2 version of Dilithium.

5.1.4 Further considerations

It is critical to underline that the memory evaluations have been proved by implementing and testing the two algorithms within the framework.

As a result, the Dilithium keypair generation caused the boot hart to hang, meanwhile the Falcon-512 keypair generation has been completed successfully. Moreover, the

Table 5.6. Falcon parameters size.

scheme	Public key size	Private key size	Signature size
Falcon-512	897	1281	666
Falcon-1024	1793	2305	1280

Note. — values expressed in *bytes*

Table 5.7. Falcon performance (source: [6]).

scheme	keypair generation	sign	verify
Falcon-512	AVG: 217,740,948	AVG: 61,102,456	AVG: 774,876
Falcon-1024	AVG: 589,059,133	AVG: 133,598,726	AVG: 1,547,336

Note. — values expressed in CPU *cycles*

Dilithium keypair generation in the SM layer caused overflows into adjacent buffer used within the PMP configuration, rendering the PMP management impossible.

Nevertheless, Falcon-1024 works as expected in all the layers, but performing many operations at enclave's layer caused the same issue presented with Dilithium in all the layers.

5.2 FALCON

Falcon is the algorithm selected as the best algorithm and suitable for the implementation in Keystone. It stands for the acronym **Fast Fourier lattice-based compact signatures over NTRU** [46].

At a high-level the algorithm is made up by three components:

- **GPV framework** for constructing lattice-based signature schemes.
- **NTRU**, a lattice-based open-source public key cryptosystem.
- **Fast Fourier Transform** (FFT)-based trapdoor sampling, an optimized algorithm to compute the Discrete Fourier Transform, which is used in the lattice vectors computation for signature and verification.

5.2.1 The FALCON framework

The algorithm is designed following the principle to minimize the quantity $|pk| + |sk| + |sig|$, respectively public key, secret key, and signature sizes. This is due to the fact that transitioning from a pre-quantum signature algorithm to a post-quantum signature algorithm poses challenges due to the increased size of keys, signatures, or both, and relying on a lattice-based signature scheme allows to achieve this goal.

The algorithm offers the following features:

Table 5.8. Falcon memory evaluation (source: [6]).

scheme	keypair generation	sign	verify
Falcon-512	18,416	42,508	4,724
Falcon-1024	36,296	82,532	8,820

Note. — values expressed in *bytes*

- **Security**: a true Gaussian sampler is used internally to generate independent key generation coefficients, which guarantees negligible leakage of information on the secret key up to more than 2^{64} signatures.
- **Compactness**: thanks to the usage of NTRU lattices, signatures are shorter than in any lattice-based signature scheme with the same security guarantees, while the public keys are around the same size
- **Fast signature generation and verification**: the signature generation and verification procedures are very fast, especially for the verification algorithm.
- **Scalability**: operations have cost $O(nlogn)$, allowing the use of long-term security parameters at moderate cost.
- **RAM economy**: the enhanced Falcon's key generation algorithm uses less than 30 KB of RAM, making it compatible with memory-constrained devices.

5.2.2 Floating point operations

The biggest drawback of Falcon is the need to perform FP operations. In fact, FP operations are complex and may lead to non-constant time operations, which translates to the possibility of building a system that is vulnerable to side-channel attacks. Moreover, devices with limited capabilities may lack of FP Unit (FPU) to perform such computation.

In light of such concerns, in the last reference implementations some improvements and new features have been introduced:

- FPA is emulated by means of integers operations using the $uint32\_t$ and $uint64\_t$ C types, hence devices do not have the necessity to be equipped with a FPU, which can still be used to achieve a faster and more efficient computation
- FP operations are performed in constant-time, avoiding related side-channel vulnerabilities

5.2.3 Hash-and-sign

As for any signature scheme, the first step when computing or verifying a signature involves the hash computation of the message.

During this procedure, as specified in the FIPS 202 [47], an approved *Extendable-Output Hash Function* (XOF) must be used, and it must have a security level at least equal to the security level targeted by the signature scheme.

A XOF is a function in which the output can be extended to any desired length. Meanwhile the suffixes used with the hash functions indicate the digest length, with XOF the suffixes indicate the security strength that the function can support. When an application requires a cryptographic hash function with a non-standard digest length, an XOF is a natural alternative to constructions that involve multiple invocations of hash function and/or truncation of the output bits.

The XOF used within Falcon is *SHAKE-256*. It is used in two places:

- As a **PRNG**: pseudorandom data is needed by the keypair generation and signature computation functions
- As the **hashing mechanism** involved in the signature computation

The code is written in pure ANSI C and avoids the usage of dynamic memory, which is not supported in Keystone.

5.3 Quantum issues in Keystone and DICE

In this section are outlined the quantum-vulnerable operations performed in the Keystone project.

The starting point of the solution design and implementation is the DICE-based Keystone [48] version developed by the TORSEC group of Turin's Politecnico which integrates the DICE specification into the Keystone framework. Thanks to the efforts put by the TORSEC group the following features have been added:

- secure boot support;
- DICE certificates support.

The developed solution aims at migrating the DICE-based Keystone version to a quantum-safe state.

Before describing the identified issues, it is necessary to give a brief overview of the cryptographic elements that are included within the system, followed by a description of the DICE-based Keystone project, enlightening the sections that require modifications, presented in the next chapter.

5.3.1 Cryptographic functions

When designing the solution, the most sensitive task has been identifying all the quantum-vulnerable operations performed within the DICE-based Keystone project. This goes along with the analysis of the cryptographic capabilities over which this system relies upon to perform cryptographic operations.

During the analysis of the system, the usage of the following cryptosystems has been identified:

- the **SHA3** hash functions family
- the **ECC ed25519** cryptosystem

SHA3

According to the developed solution, the usage of the SHA3 family answers to several issues since it is based on the KECCAK cryptographic function [49]. This family of hash function is based on what is called a *sponge construction*: after the pre-processing (which divides the message into blocks and provides padding) the sponge construction consists of two phases:

- **Absorbing (or input) phase**: the message blocks x_i are passed to the algorithm and processed;
- **Squeezing (or output) phase**: an output of configurable length is computed.

In light of the quadratic speed up in collision and pre-image searches deriving from the Grover's algorithm, this hash function allows to retain the level of security by roughly doubling the output size, hence it remains a valid choice as a post-quantum solution. Furthermore, the SHA3-512 hash function is used in both the developed solution and the original DICE-based Keystone version to compute the measurement and the CDI values used in the process, providing a level of security equal to $n/2 = 512/2 = 256$ bits, which is considered quantum-safe.

ed25519

As regards the ed25519 implementation, it comes for free with the reference DICE-based Keystone version. Following are listed the functions composing the API exposing the cryptographic functionalities:

- `void ed25519_create_keypair(unsigned char *public_key, unsigned char* private_key, const unsigned char *seed)`: function used to perform the key-pair generation, starting from the seed parameter `const unsigned char *seed`;
- `void ed25519_sign(unsigned char *signature, const unsigned char * message, size_t message_len, const unsigned char *public_key, const unsigned char *private_key)`: function used to compute the signature over the `const unsigned char *message` parameter with the private key `const unsigned char *private_key`;
- `int ed25519_verify(const unsigned char *signature, const unsigned char *message, size_t message_len, const unsigned char *public_key)`: function used to verify the provided signature `const unsigned char *signature` using the public key `const unsigned char *public_key`.

Given the usage of a quantum-vulnerable algorithm to perform public-key operations and due to the concerns raised by the existence of the Shor's algorithm, following are listed on a high-level the actions taken in the developed solution to migrate the system from the quantum-unsafe to the quantum-safe state:

- replace the pre-quantum keys with PQ keys;
- perform public-key operations with a quantum-safe algorithm (e.g. Falcon);
- replace the certificates used in the system with PQ certificates in which PQ keys and signatures are used.

Hence the solution implementation starts by implementing the Falcon algorithm in all the layers composing the system, removing the vulnerable operations, and replacing them with the quantum-safe selected alternative.

5.3.2 X.509 certificate management

The original Keystone project does not provide any support for the creation and parsing of X.509 certificates.

Thanks to the effort of the TORSEC group, the functionalities to manage certificates are provided within the `X509custom` library, built starting from the `Mbed_TLS` library, embedding all the functionalities necessary for the correct management of the X.509 certificates. Such library includes functions that operate on data in *Distinguished Encoding Rules* (DER) format, encoding supported by the X.509 standard.

It includes the following files:

- `oid_custom.h`: header file containing the OIDs definitions;
- `x509custom.c`: source file containing the functions' implementation;
- `x509custom.h`: header file containing structure declarations and the functions' declaration;

In the proposed design, modifications to this library are needed due to the following:

- support for further PQ keys context structure;
- correct parsing of PQ keys;
- signature computation over the certificate with PQ algorithm;
- correct parsing of certificate in DER format containing PQ keys;
- new OIDs definition.

all the modifications are discussed in the implementation chapter.

5.3.3 Root of Trust

At manufacturing time, several elements are provisioned by the manufacturer. In particular according to the DICE specification the RoT that is produced by the manufacturer comes provided with:

- **UDS**: Unique Device Secret as described in the DICE specification;
- **$\mathrm{TCI}_{\mathrm{SM}}$ signature**: signature computed over the SM's reference measure with the manufacturer private key;
- **Manufacturer certificate**: X.509 certificate in DER format used in the $\mathrm{TCI}_{\mathrm{SM}}$ signature verification;
- **DRK certificate**: X.509 certificate in DER format of the RoT keypair, issued by the manufacturer.

During the system boot, the first step is the verification of the SM signature using the public key contained in the manufacturer certificate:

1. the RoT computes the $\mathbf{TCI_{SM}}$;
2. the reference signature computed over the $\mathbf{TCI_{SM}}$ provided by the manufacturer is verified using the public key contained in the manufacturer certificate;
3. if the verification is correct the boot process continues, otherwise the process is stopped.

The process is summarized in the figure 5.1.

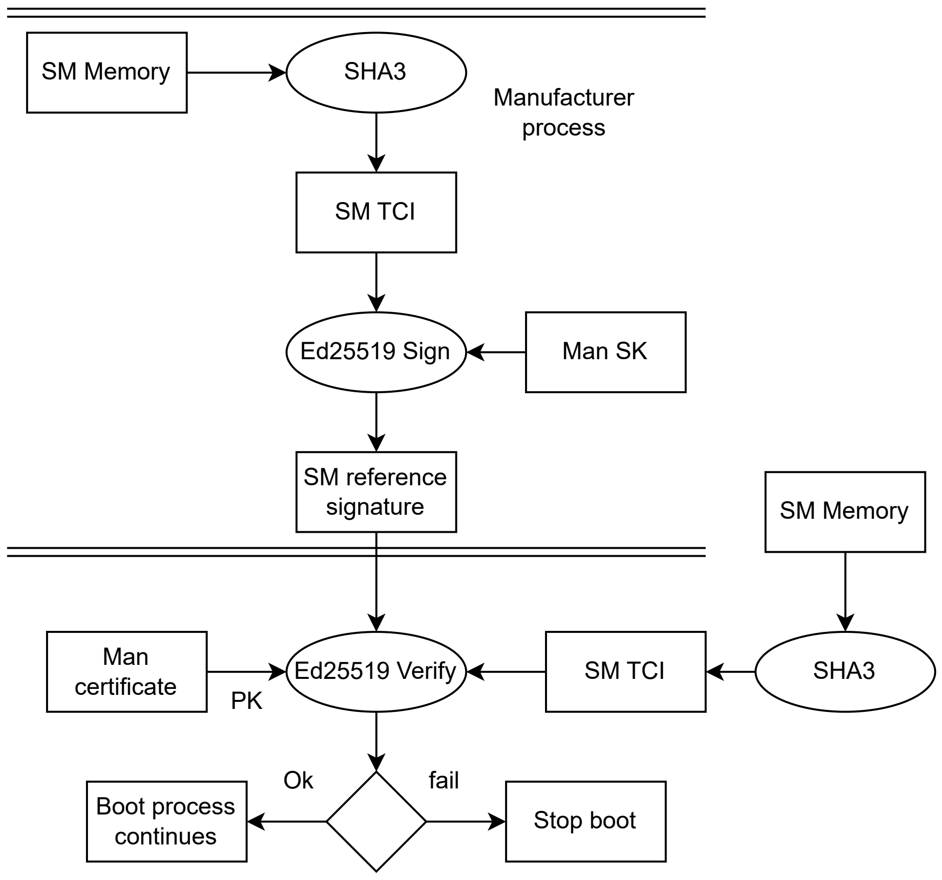


Figure 5.1. Secure boot process using quantum vulnerable algorithm.

If the signature verification process is successful, the keypairs needed in the process are generated as shown in the figure 5.2:

1. the $\mathbf{CDI_{L0}}$ is computed hashing the $\mathbf{TCI_{SM}}$ and the **UDS**
2. the **DRK keypair** is generated using the $\mathbf{CDI_{L0}}$ as seed;
3. the seed for the ECA keypair generation is computed by hashing the $\mathbf{CDI_{L0}}$ and the $\mathbf{TCI_{SM}}$;
4. the **ECA keypair** is computed starting from its seed;

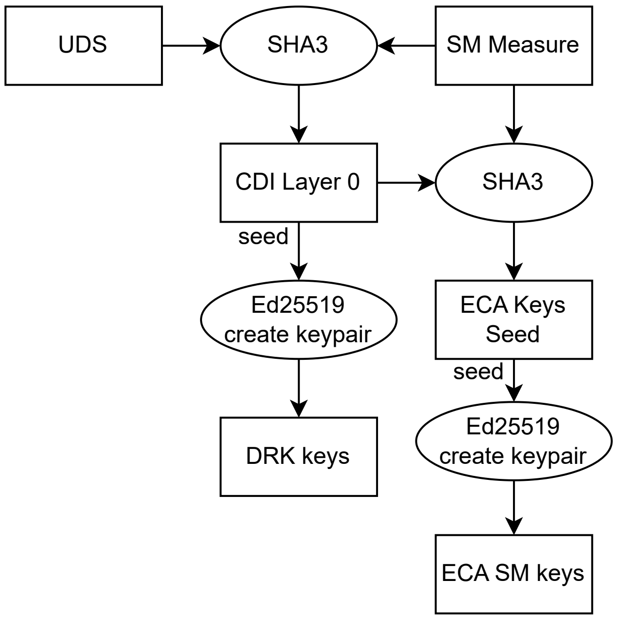


Figure 5.2. Keypairs creation with quantum vulnerable algorithm.

The final steps are shown in the figure 5.3, in which after the keypairs generation, the respective certificates are issued. For developing purposes, in the TORSEC Keystone project, the DRK certificate is not stored in memory and is recomputed on the fly by the RoT at each boot. This is due to the fact that the **DRK keypair** depends on the $\mathbf{CDI_{L0}}$, hence in this scenario in which the SM is subject to changes, the resulting keypair reflects such changes. Therefore, a precomputed certificate would corresponds to a wrong keypair after a SM modification.

In the following are listed the steps performed by the RoT as regards:

- the certificates issuance;
- the SM keypair generation;
- the SM signature computation.

1. the **DRK certificate** is generated and signed with the $\mathbf{Man_{SK}}$;
2. the **SM ECA certificate** is generated and signed with the $\mathbf{DRK_{SK}}$;
3. the **SM keypair** is generated using as seed the hash computed over the $\mathbf{DRK_{SK}}$ and the $\mathbf{TCI_{SM}}$;
4. the quantity $\mathbf{TCI_{SM}}||\mathbf{SM_{PK}}$ is computed;
5. the signature over the previous quantity is computed with the $\mathbf{DRK_{SK}}$.

5.3.4 Security Monitor

At this layer, the SM, first, receives securely the following parameters from the BootROM:

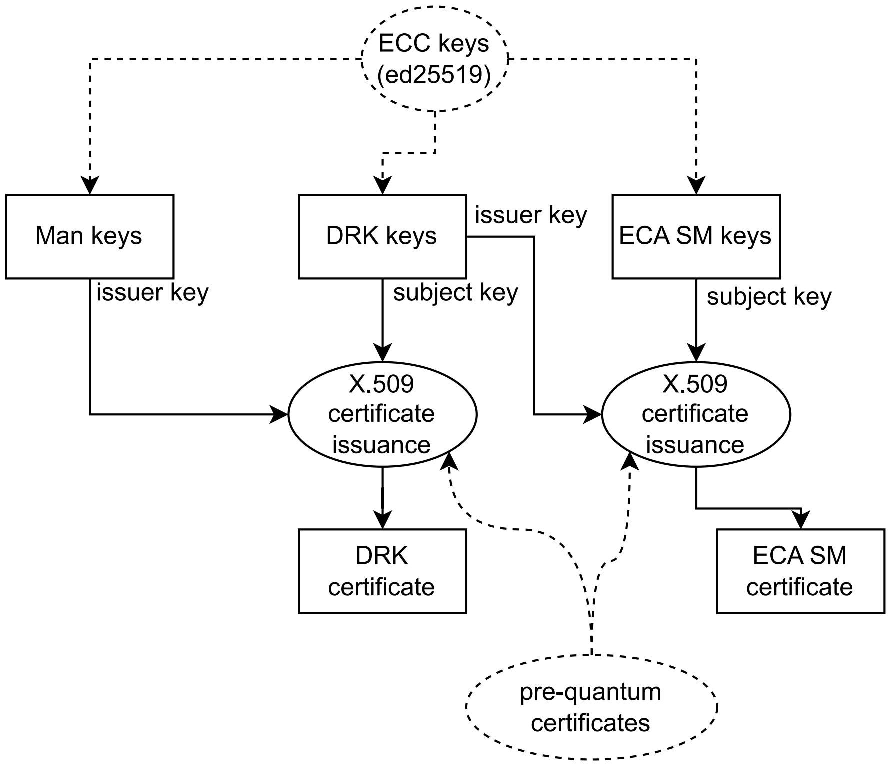


Figure 5.3. Certificates generation with quantum vulnerable algorithm.

- $\mathbf{TCI_{SM}}$: SM measure;
- $\mathbf{CDI_{L0}}$: SM CDI computed in the previous layer;
- **SM keypair**: keypair used in the attestation process;
- **ECA certificate**: used in this layer to issue certificates for upper layers;
- **Manufacturer certificate**;
- **DRK certificate**;

Once received the parameters from the RoT, the SM proceeds as follows:

1. the **ECA certificate** is parsed from DER format to `mbedtls_x509_crt` struct;
2. the **DRK certificate** is parsed from DER format to `mbedtls_x509_crt` struct;
3. the **manufacturer certificate** is parsed from DER format to `mbedtls_x509_crt` struct;
4. the certificates are validated;
5. the **ECA** certificate **signature** is verified using the $\mathbf{DRK_{PK}}$;
6. the **DRK** certificate **signature** is verified using the $\mathbf{Man_{PK}}$;
7. the $\mathbf{TCI_{SM}}$ contained in the `diceTcbInfo` extension is verified with respect to the computed $\mathbf{TCI_{SM}}$;

8. the **ECA keypair** is recomputed.

if any of the steps listed above fails, then the boot process is stopped, otherwise the SM is initialized and the system can be deployed. The steps are summarized in the figures 5.4 and 5.5.

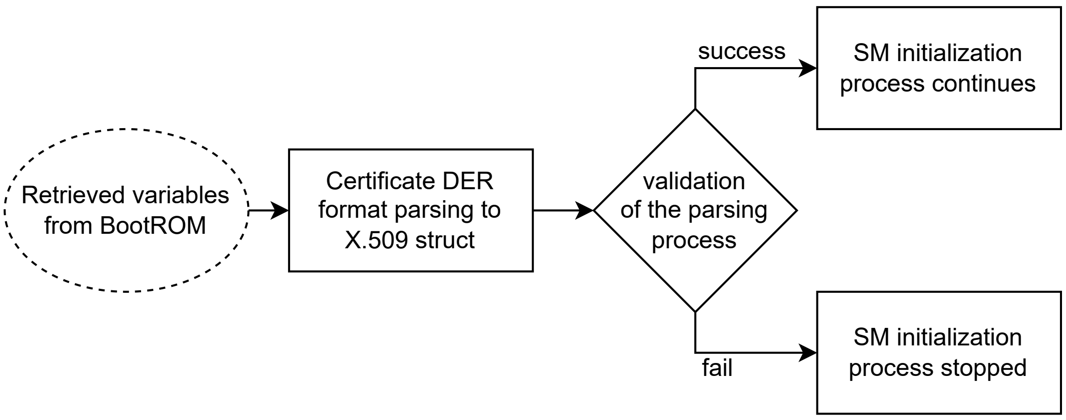

Retrieved variables from BootROM
Certificate DER format parsing to X.509 struct
validation of the parsing process
success
SM initialization process continues
fail
SM initialization process stopped

Figure 5.4. Certificates verification with quantum vulnerable algorithm.

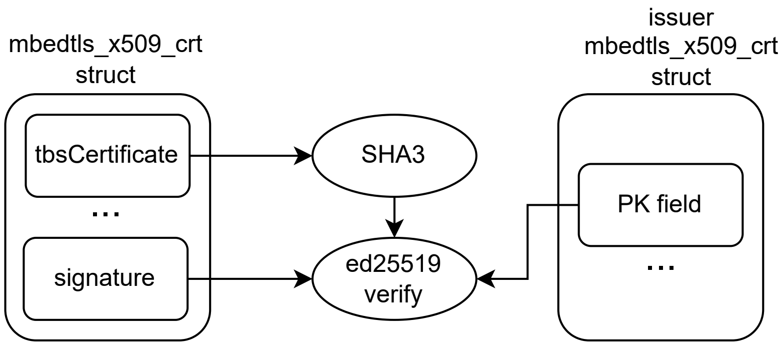


Figure 5.5. Signature verification with quantum vulnerable algorithm.

At this point, enclaves can be created and executed. When creating an enclave, the SM is responsible for the generation of the following entities:

- **Local Attestation Keys** (LAK): keypair created when a new enclave has to be created;
- **Local Attestation Keys certificate**: X.509 certificate in DER format released for the public part of the Local Attestation keypair.

the LAK are generated starting from the enclave's CDI, computed as follows:

$$CDI_{enclave} = \mathbf{SHA3}(CDI_{L0}||TCI_{enclave})$$

The process is summarized in the figure 5.6.

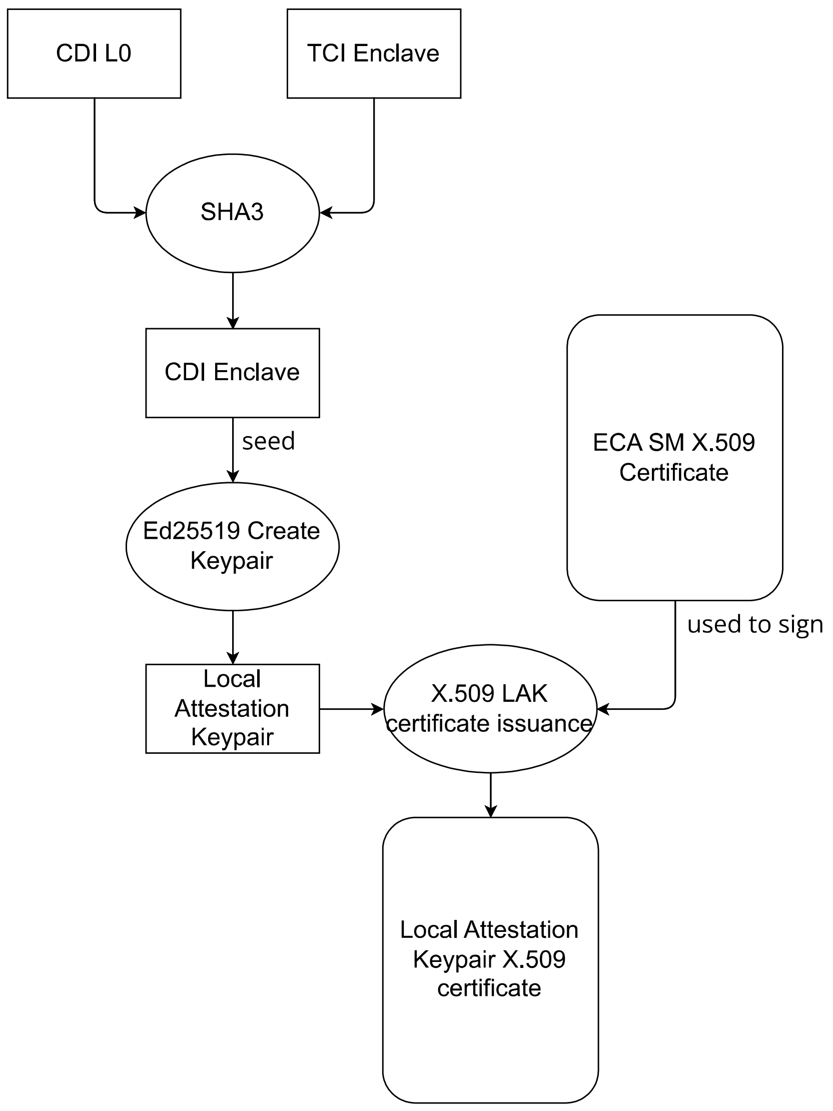

CDI L0
TCI Enclave
SHA3
CDI Enclave
seed
ECA SM X.509 Certificate
Ed25519 Create Keypair
used to sign
Local Attestation Keypair
X.509 LAK certificate issuance
Local Attestation Keypair X.509 certificate

Figure 5.6. Local Attestation Keys and X.509 certificate generation with quantum vulnerable algorithm.

5.3.5 Enclave

When it comes to the Enclave execution, critical tasks from a quantum security perspective are the following possible operations:

- the **Remote Attestation**;
- the **Sealing Key Derivation**.

Both are provided in the original Keystone framework and make heavy usage of the quantum vulnerable algorithm `ed25519`.

Remote Attestation

Once an enclave has started, it may request the SM to provide a **signed enclave report** and **signed SM report**.

The **SM report** contains:

- the $\mathbf{TCI_{SM}}$;
- the $\mathbf{SM_{PK}}$;
- the signature computed over the previous elements with the $\mathbf{DRK_{SK}}$;

The **enclave report** contains:

- the $\mathbf{TCI_{Enclave}}$;
- a data block from the enclave of up to 1 KB;
- the signature computed over the previous elements with the $\mathbf{SM_{SK}}$.

The attestation evidence generation is shown in the figure 5.7

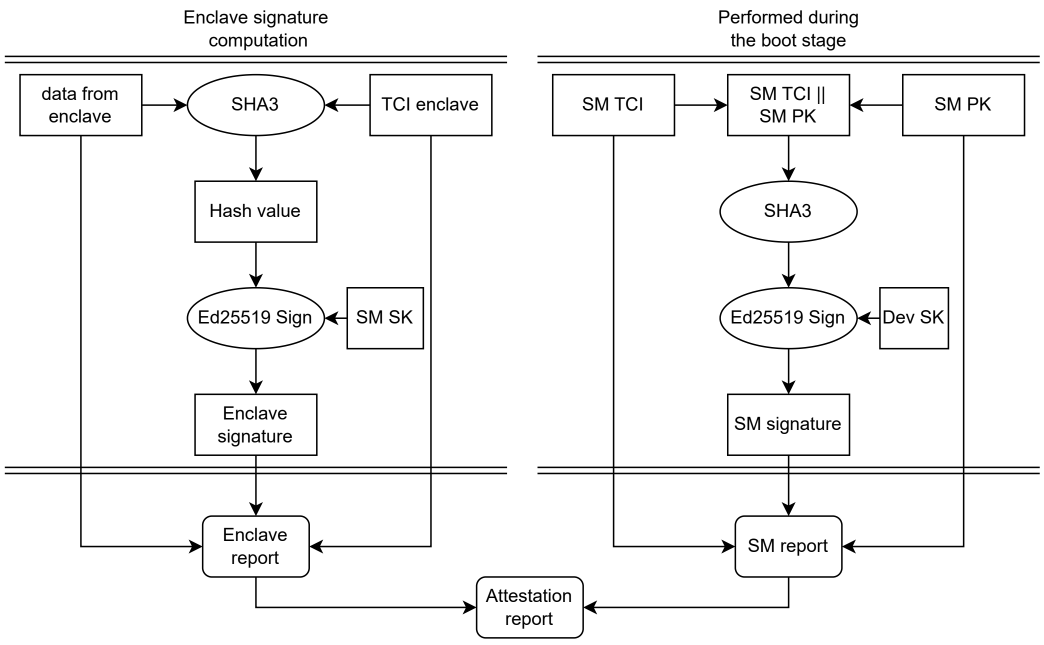


Figure 5.7. Attestation evidence generation with quantum vulnerable algorithm.

The verifier, when provided with the $\mathbf{DRK_{PK}}$, $\mathbf{TCI_{SM}}$, and $\mathbf{TCI_{Enclave}}$, will verify the signatures these reports. In particular:

1. the Verifier process is executed, generating a nonce;
2. the untrusted host process is executed, receives the nonce, and requests the enclave creation;
3. the enclave is created and executed, receives the nonce, and requests the SM to provide the report through an `SBI_CALL`;
4. the SM creates the report and copies it to the enclave memory;
5. the enclave sends the report back to the host which returns it to the Verifier;
6. the Verifier validates the reports and verifies the signatures.

The process is depicted in the figure 5.8

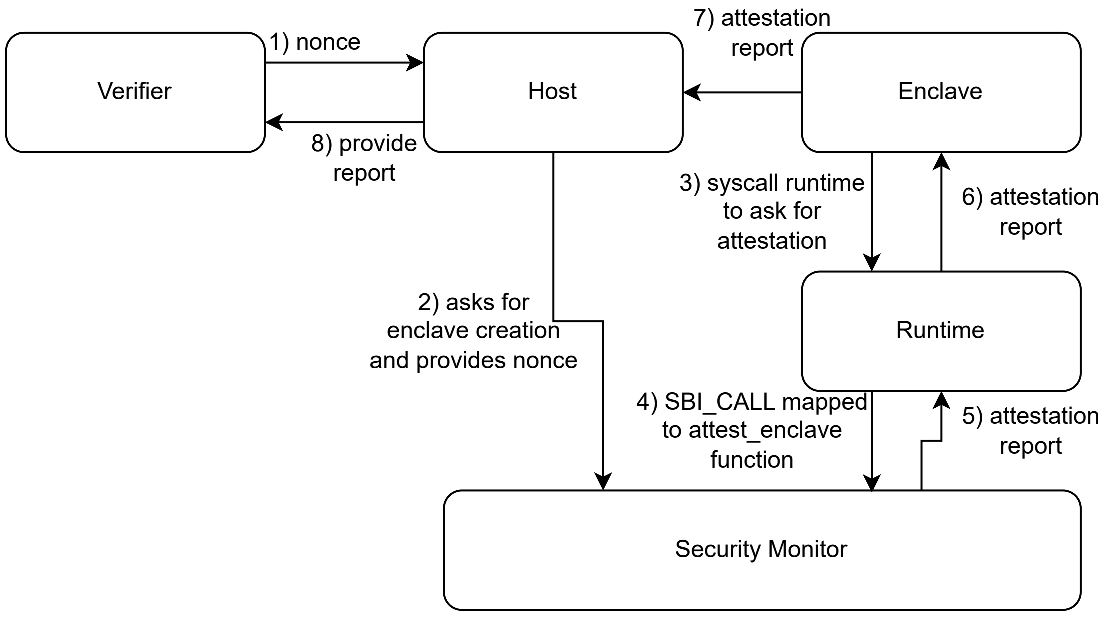

Verifier
1) nonce
Host
7) attestation report
Enclave
8) provide report
3) syscall runtime to ask for attestation
6) attestation report
2) asks for enclave creation and provides nonce
Runtime
4) SBI_CALL mapped to attest_enclave function
5) attestation report
Security Monitor

Figure 5.8. Remote attestation flow.

Sealing-Key derivation

The *data-sealing* feature allows an enclave to derive a key for data encryption, to be able to sava data in untrusted, non-volatile memory outside the enclave. This key is bound to the identity of the processor, the SM and the enclave. Hence only the same enclave running on the same SM and same processor is able to derive the same key.

After an enclave restart, the enclave can derive the same key again, fetch the encrypted data from the untrusted storage and decrypt them using the derived key.

The root of the key hierarchy is the asymmetric processor key pair $(\mathbf{Dev_{SK}}, \mathbf{Dev_{PK}})$. The asymmetric security monitor keypair $(\mathbf{SM_{SK}}, \mathbf{SM_{PK}})$ is derived from the measurement of the SM and the private processor key $\mathbf{Dev_{SK}}$. Therefore, the resulting SM keypair is bound to the processor and to the identity of the SM itself.

Starting from this key hierarchy, the process deriving the sealing-key in Keystone is depicted in the figure 5.9.

The key is derived using three main inputs:

- the $\mathbf{SM_{SK}}$;
- the $\mathbf{TCI_{Enclave}}$;
- a **key identifier**.

Following are listed the properties that stem from the sealing-key derivation process:

- the $\mathbf{SM_{SK}}$ ensures that the resulting sealing-key is bound to the identity of the processor and the identity of the SM;
- whenever one of the two components change, the resulting sealing-key is different;
- the $\mathbf{TCI_{Enclave}}$ ensures that the sealing-key is bound to the enclave's identity;

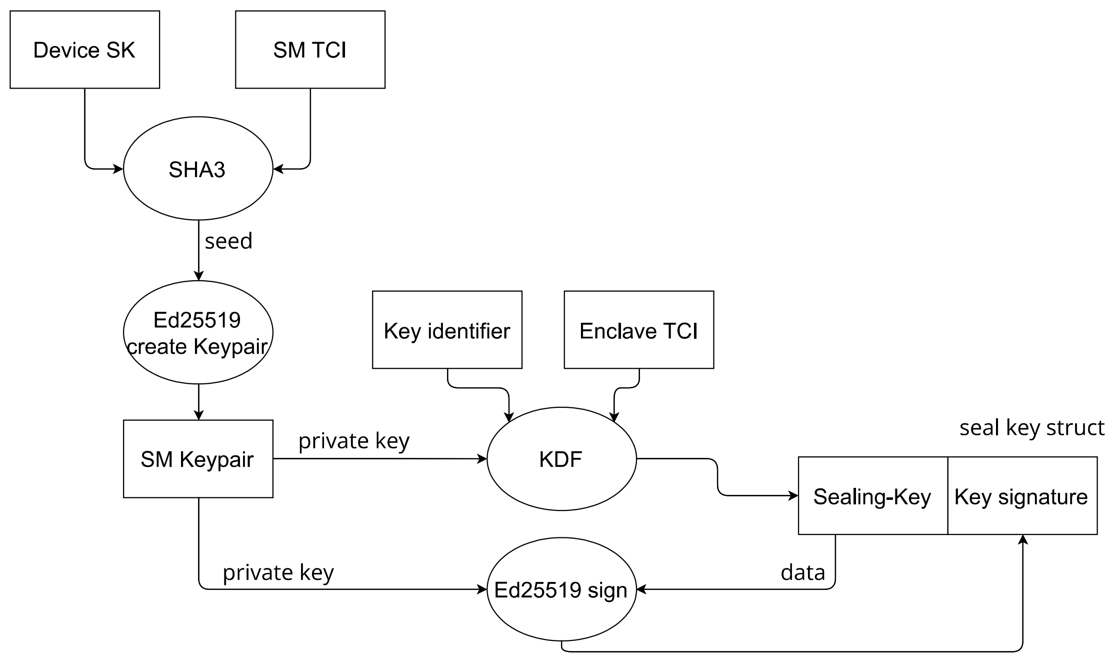


Figure 5.9. Keystone Sealing-Key Derivation with quantum vulnerable algorithm (source: [2]).

- the **key identifier** is an additional input to the key derivation function which can be chosen by the enclave, allowing the enclave to derive multiple keys by choosing different values for the key identifier.

Chapter 6

Post-Quantum support in Keystone: implementation

In this chapter are described the implementation steps and the modifications applied to integrate the designed solution into the Keystone TEE.

More in depth, in this chapter are discussed:

- the **Falcon reference implementation**: configuration file, API;
- `x509custom` modifications presented in the design chapter;
- the changes applied into the Keystone project, involving the following components: `sm`, `bootloader`, `SDK`, `runtime`, `opensbi`;

6.1 Falcon reference implementation and modifications

The Falcon's code has been retrieved from the original project website, in which several resources can be consulted [46].

The algorithm is implemented in pure ANSI C, which makes it portable, built without standard dependencies. This allows to reduce the amount of code included into the executable, leading to the following advantages:

- the attack surface is reduced;
- the applications can be used in contrained context (e.g. embedded systems, IoT devices);

The Falcon reference implementation includes the following files, grouped according to the purpose:

- `falcon.h`, `falcon.c`: files containing the external Falcon **API**, relative implementation, and several macros useful in the implementation and deployment of the algorithm;
- `config.h`: configuration file containing compile-time options which can be used to enable compilation options and platform-specific improvements;

- `fpr.h`, `fpr.c`: files containing the aforementioned FP emulations, useful in context in which a FPU may not be available or FP operations are complex;
- `fft.c`, `fft.h`: FFT implementation used in the signature computation and verification;
- `keygen.c`: file containing the functions' implementation to perform keypair generation;
- `vrfy.c`: file containing the functions' implementation to perform signature verification;
- `sign.c`: file containing the functions' implementation to perform signature computation;
- `rng.c`: file containing the PRNG logic based on the `SHAKE256` XOF, the `ChaCha20`-based PRNG, and the interface to the enable the OS-provided RNG if available.
- `shake.c`: file containing the `SHAKE256` implementation in pure ANSI C;
- `codec.c`: file containing the encoding and decoding functions' implementation and used internally in the public-key operations;
- `common.c`: file containing support functions to perform the signature;
- `inner.h`: file containing internal functions for Falcon, providing all the primitives on which wrappers are built to provide the external APIs.

Due to the fact that the system is built without standard dependencies, to correctly integrate the algorithm in the Keystone project, the library included with the header file `string.h` has been removed and a custom library `my_string.h` has been included, replacing all the calls to the `mem*` family of functions. Following are listed the functions included within the new custom library for which the replacement was necessary:

- `void* my_memcpy(void* dest, const void* src, size_t len)`: to copy `len` bytes from the source (`src`) memory address to the destination memory address (`dest`);
- `void* my_memset(void* dest, int byte, size_t len)`: to set `len` bytes to the `byte` value starting at the address `dest`;
- `void* my_memmove(void *dest, void const *src, size_t count)`: to move `count` bytes from the source (`src`) memory address to the destination memory address (`dest`);
- `int my_memcmp (const void *str1, const void *str2, size_t count)`: to compare `count` bytes between the string `str1` memory address and the string `str2` memory address;
- `unsigned int my_strlen(const char *s)`: to compute the length of the string `s`;
- `int my_strncmp( const char * s1, const char * s2, size_t n )`: to compare `n` bytes among the string `s1` memory address and the string `s2` memory address;
- `char* my_strncpy(char* destination, const char* source, size_t num)`: to copy `num` bytes from the source string (`source`) memory address to the destination string (`destination`) memory address;.

6.1.1 Configuration file

In the reference implementation is included the configuration file `config.h` which includes several macros to configure the cryptosystem. Each macro is an option which can be enabled by setting the macro value to 1, otherwise 0. In the implemented solution, two options have been enabled:

- `FALCON_FPEMU`: macro that enables the usage the emulated floating-point implementation, necessary since in the RISC-V FP operations are complex and not constant-time and in general a FPU is not available in embedded devices;
- `FALCON_KG_CHACHA20`: macro that enables the usage of a PRNG based on `ChaCha20` and seeded with `SHAKE256` for keypair generation purposes, which provides a speed up the keypair generation. It has been enabled to provide a small speed up to the slow keypair generation issue that affects Falcon.

The emulation uses only integer operations with `uint32_t` and `uint64_t` C types and all the operations are performed in constant-time, provided that the underlying platform offers constant-time opcodes for the following operations:

- multiplication of two 32-bit unsigned integers into a 64-bit result;
- left-shift or right-shift of a 32-bit unsigned integer by a shift count in the range (0, 31).

6.1.2 External FALCON API

All the functions needed to perform public-key operations and to manage the `SHAKE256` context are contained within the header file `falcon.h`.

SHAKE256 implementation

The `SHAKE256` context is defined as follows:

```
typedef struct {
        uint64_t opaque_contents[26];
  } shake256_context;
```

The advantage of this declaration is the usage of the `uint64_t opaque_contents[26]` field, that is pure data with no pointer. This means that there is no need to release the context explicitly and the data is allocated on the stack, avoiding the dynamic memory allocation which is critical in embedded devices. Furthermore Keystone does not support at the time of writing a dynamic memory management in the lower layers composing the system.

The functions needed to properly manage the `SHAKE256 context` are listed in the following:

- `void shake256_init(shake256_context *sc)`: to perform the initialization of the `shake256_context`;
- `void shake256_inject(shake256_context *sc, const void *data, size_t len)`: to inject `len` data bytes into the `shake256_context`;

- `void shake256_flip(shake256_context *sc)`: flip the `shake256_context` state to output mode, enabling the call to the `shake256_extract` function;
- `void shake256_extract(shake256_context *sc, void *out, size_t len)`: extract bytes from the `shake256_context` after flipping the context to output mode;
- `int shake256_init_prng_from_system(shake256_context *sc)`: initialize a `shake256_context` as a PRNG, using an initial seed from the OS-provided RNG;
- `void shake256_init_prng_from_seed(shake256_context *sc, const void *seed, size_t seed_len)`: to initialize the RNG from a provided seed rather than a OS-provided source of randomness.

Dute to the fact that Keystone does not have an adequate entropy source, and there is no OS-provided RNG at bootloader and SM level, in the implemented solution the `shake256_context` is always initialised from a provided seed.

Falcon public-key operations

The Falcon algorithm is parametrised by a degree $logn$, expressed as a power of two 2^{logn}. Formally only two values are possible:

- 512: corresponding to the `Falcon-512` version;
- 1024: corresponding to the `Falcon-1024` version.

In practice, the reference implementation supports lower degree as well (e.g. 2 to 256) but such versions do not offer an adequate level of security, hence should be used only for research purposes. In all the functions and macros used in the algorithm, the degree is provided logarithmically as the `logn` parameter, ranging from the value 1 to 10.

In the implemented solution, support for both the versions has been provided, by introducing in the file `falcon.h` the macro:

```
/*set 9 for the 512 version, 10 for the 1024 version*/
#define LOGN_PARAM 9
```

given that the `falcon.h` file is included by all the components that need visibility of the `LOGN_PARAM` value, it is sufficient to modify the value set to enable the usage of `Falcon-1024`. This has been done to provide a certain degree of flexibility in light of the cryptographic agility principle.

The main functions used within the solution developed to perform the public key operations are listed in the following:

- `int falcon_keygen_make( shake256_context *rng, unsigned logn, void * privkey, size_t privkey_len, void *pubkey, size_t pubkey_len, void *tmp, size_t tmp_len)`: function that generates the PQ keypair returning by reference in the `privkey` field the generated private key and in the `pubkey` field the generated public key. The function takes as input the `shake256_context` as RNG, which must be previously initialized by the caller;

- `int falcon_sign_dyn(shake256_context *rng, void *sig, size_t *sig_len, int sig_type, const void *privkey, size_t privkey_len, const void *data, size_t data_len, void *tmp, size_t tmp_len)`: method used to sign the data provided in the `data` parameter, using the specified private key `privkey`. The function takes as input the `shake256_context` as RNG, which must be previously initialized by the caller;
- `int falcon_verify(const void *sig, size_t sig_len, int sig_type, const void *pubkey, size_t pubkey_len, const void *data, size_t data_len, void *tmp, size_t tmp_len)`: function used to perform the verification of the signature `sig` with the provided public key `pubkey`, computed over the data parameter `data`.

All the functions require a **temporary buffer** (received by reference as the parameter `void *tmp`) to store intermediate values during the public-key operations processing and according to the operation to be performed, the buffer's size varies. The reference implementation comes with the definition of several macros to determine the size needed by the buffers. Such macros receive the `LOGN_PARAM` parameter and expand into the buffer size. Moreover, macros to compute the lengths of public key, private key, and signature are available and use the same logic expressed for the temporary buffers. This allows to achieve two goals:

- there is no need for dynamic memory allocation;
- the `LOGN_PARAM` parameter allows to to simulate a 'dynamic' allocation, therefore providing a big degree of flexibility in the declaration of the buffers and the algorithm's version to use.

All the macros are defined in the `falcon.h` header file.

Following are listed the macros used in the implemented solution:

- `FALCON_TMPSIZE_KEYGEN`: temporary buffer size for key pair generation
- `FALCON_TMPSIZE_SIGNDYN`: temporary buffer size for signature computation
- `FALCON_TMPSIZE_VERIFY`: temporary buffer size for signature verification
- `FALCON_PUBKEY_SIZE`: public key length
- `FALCON_PRIVKEY_SIZE`: private key length
- `FALCON_SIG_CT_SIZE`: signature size (constant format)

Finally, it is critical to underline that the `falcon_sign_dyn` function, receives the parameter `sig_type`, which allows to specify the signature type. Three are the signature types that can be set:

- `Compressed`: default format which yields to the shortest signatures on average, but the size is variable;
- `Padded`: compressed format but with extra padding bytes to obtain a fixed size at compile-time;
- `Constant`: fixed-size format which allows constant-time preprocessing with regard to the signature value and the message data.

in the implemented solution, the constant format is used due to the conservative choice to opt for to the constant-time preprocessing feature, but other formats are possible and deployable.

6.2 X.509 library modifications

To enable the usage of PQ keys and signature within the X.509 certificate management, in the implemented solution some changes have been introduced. The files interested by the modifications are the files included under the `x509custom/` folder:

- `oid_custom.h`: with the definitions of the OIDs for the two Falcon version;
- `x509custom.c`: with the modifications of the functions used to embed the PQ keys in the certificates and to compute the signature over the tbsCertificate field;
- `x509custom.h`: with the definition of the structures needed by functions defined in the `x509custom.c` file.

OIDs definition

The major modification applied to the `oid_custom.h` file is related to the introduction of two new **OIDs**:

- `MBEDTLS_OID_FALCON512_SHAKE256`;
- `MBEDTLS_OID_FALCON1024_SHAKE256`.

Since there are no official OIDs defined for the two versions at the time of writing, the OIDs defined have been retrieved from the OID mapping described in the IETF-Hackathon 116 [50].

x509custom.h modifications

Following are reported the major changes introduced in the `x509custom.h` header file to support the Falcon usage within Keystone:

- the Falcon implementation has been made available by including the `falcon.h` header file, rendering available the `LOGN_PARAM` macro to the library, along with the functions to perform the signature computation and to manage the `SHAKE256` context;
- the `enum mbedtls_pk_type_t` type definition has been modified to include the `MBEDTLS_PK_FALCON512` and `MBEDTLS_PK_FALCON1024` values;
- the `mbedtls_falcon_context` to embed the PQ keys has been defined;
- the generic `mbedtls_pk_context` has been modified to include the `mbedtls_falcon_context` struct as a field;

x509custom.c modifications

Here are reported the major changes introduced in the `x509custom.c` source file to support the Falcon keys and signature:

- `size_t falcon_get_bitlen`: new function that returns the Falcon public key size in bits, based on the `LOGN_PARAM`;

- `mbedtls_pk_write_pubkey_der`: this function has been modified to correctly write Falcon PQ public key in the `subjectPublicKey` field and to insert the respective OID (according to the Falcon version used) into the `algorithm` field of the `SubjectPublicKeyInfo` sequence of the certificate;
- `mbedtls_x509write_crt_der`: this function has been modified to compute the signature with Falcon over the `tbsCertificate` field of the certificate;
- `mbedtls_x509write_crt_der_tmp`: new function based on the `mbedtls_x509write_crt_der` function and modified to accept as further parameters the `unsigned char *tmp` temporary buffer, the `shake256_context *rng` RNG, and the `unsigned char* sig` buffer, all used in the signature computation. The advantage is that in this case the new parameters are provided by the caller, aiming at reducing the stack usage by reusing variables and buffers already allocated by the caller.

6.3 BootROM

The major changes implemented at this layer regards the file `bootrom.c` in which keys and certificates are created and the signature check for the secure boot is performed.

The DICE-based Keystone project comes with the file `test_dev_key.h`, which contains:

- an **ECC keypair** provided by the manufacturer;
- the **manufacturer certificate** in DER format.

both provided to the booting stage simulating a secure storage.

Therefore:

- A PQ keypair has been generated with the `falcon_keygen_make` function and the ECC keys have been replaced;
- the **manufacturer certificate** has been created using a script, in DER format, containing PQ keys and self-signed using the `falcon_sign_dyn` function.

both included in the `test_dev_key.h` file.

According to the DICE specification, the RoT should be provided by the manufacturer with the **DRK certificate**. Due to the fact that that the DRK keypair is generated starting from the CDI which depends on the SM measure, in this implementation the DRK certificate and the DRK keypair are computed on the fly, since any modification to the SM would cause the generation of a different DRK keypair, hence the provided certificate would be associated to the wrong keypair.

As previously described, the first operation performed by the RoT at boot stage is the SM signature verification, needed for the secure boot. For developing purposes, since the SM code may change in this context, the SM signature is computed during the boot stage, simulating a manufacturer-provided signature and verified.

The steps performed in the implemented solution are listed in the following (e.g. figure 6.1:

1. the $\mathbf{TCI_{SM}}$ is computed;
2. the `shake256_context` RNG is initialized with a random seed;
3. the reference **SM signature** is computed using the `falcon_sign_dyn` function, which receives the RNG and the PQ $\mathbf{Man_{SK}}$;
4. the $\mathbf{TCI_{SM}}$ is recomputed by the RoT;
5. the reference signature is verified with the `falcon_verify` function, which receives as parameter the PQ $\mathbf{Man_{PK}}$ contained in the certificate, the reference signature, and the $\mathbf{TCI_{SM}}$;
6. if the verification process is successful, the boot process continues otherwise is halted.

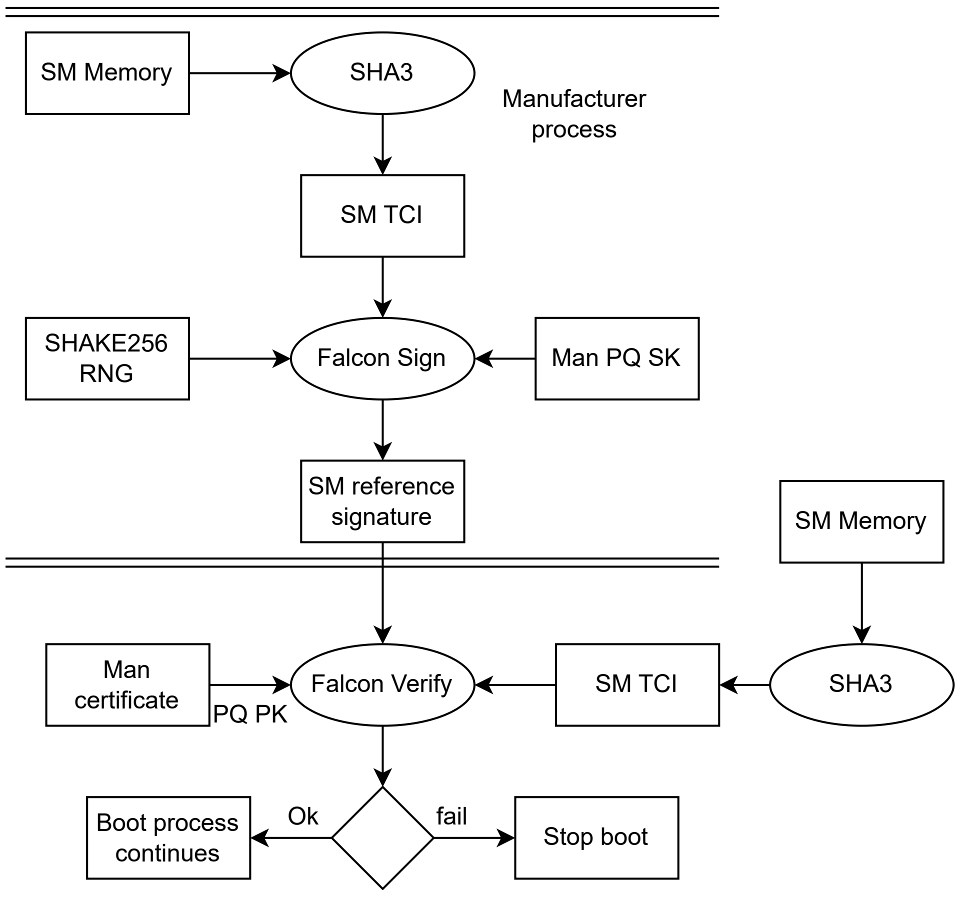


Figure 6.1. Implemented PQ secure boot process.

Once completed the signature verification successfully, the DRK and the ECA keypairs generation has been modified so that in the implemented solution is performed using the Falcon API. In particular:

1. the $\mathbf{CDI_{L0}}$ is computed;
2. the `shake256_context` RNG is initialized using the $\mathbf{CDI_{L0}}$ as seed;

3. the RNG is passed to the `falcon_keygen_make` function which is used to generate the **DRK keypair**;
4. the `ECA_keys_seed` variable is filled with the hash computed over the $\mathbf{SM_{measure}}$ and the $\mathbf{CDI_{L0}}$;
5. the `shake256_context` RNG is initialized using the `ECA_keys_seed`;
6. the RNG is passed to the `falcon_keygen_make` function which is used to generate the **ECA keypair**;

More in depth, the $\mathbf{CDI_{L0}}$ is computed in the following way:

$$CDI_{L0} = SHA3(UDS||SM_{Measure})$$

and the `ECA_keys_seed` is computed as follows:

$$seed = SHA3(CDI_{L0}||SM_{Measure})$$

The process is depicted in the figure 6.2

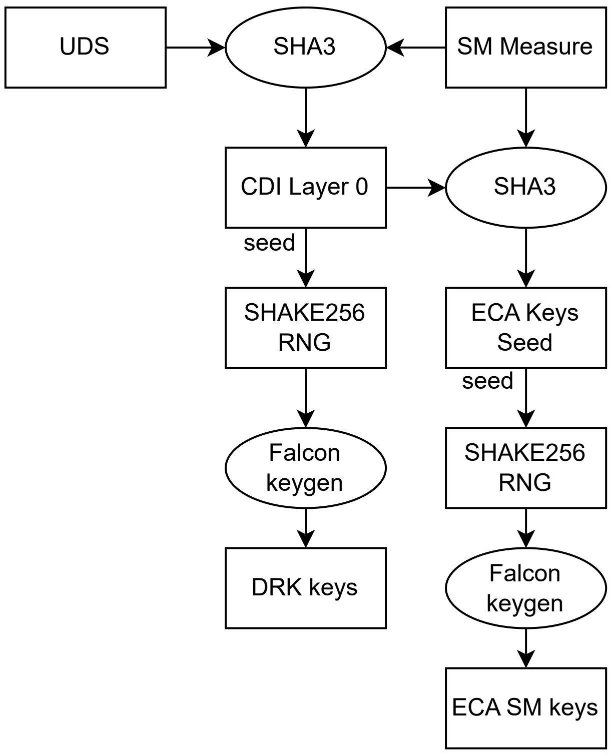


Figure 6.2. Implemented PQ keypairs generation.

After the keypairs generation, the **DRK** and the **ECA certificates** are created using the modified version of the `x509custom` library, so that the functions can handle the larger PQ keys and signatures(e.g. figure 6.3).

The last change implemented in the `bootloader.c` file is related to the **SM keypair** generation and the **SM signature** computation. More in depth, in the implemented solution (shown in the figure 6.4):

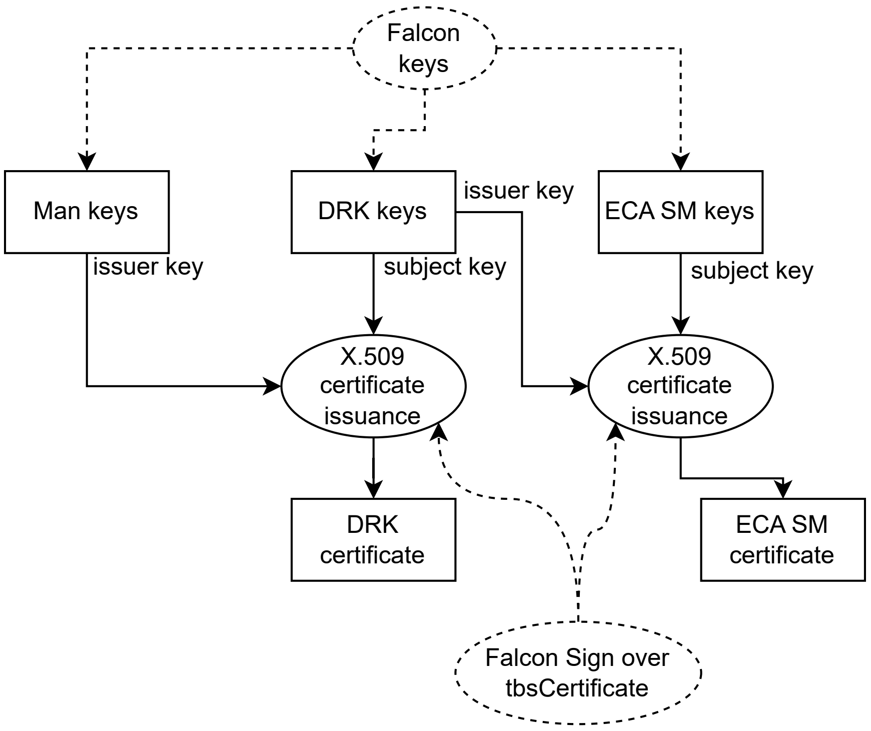


Figure 6.3. Implemented PQ certificate issuance.

1. the seed for the keypair generation is created hashing the $\mathbf{Device_{SK}}$ provided by the manufacturer and the $\mathbf{SM_{Measure}}$;
2. the `shake256_context` RNG is initialized from the computed seed;
3. a call to the `falcon_keygen_make` function, which receives the initialized RNG, is used to create the **SM keypair**;
4. the value $\mathbf{SM_{measure}}||\mathbf{SM_{PK}}$ is computed;
5. the **SM signature** is computed over the value computed in the previous step with the `falcon_sign_dyn` function, which uses the previously initialized RNG, and the $\mathbf{Device_{SK}}$.

The last operation performed is the cleanup of the memory from any secret used in the process.

6.3.1 Linker script file

The various layers in Keystone are built independently. Hence, to share parameters among the components, is necessary to rely on a **.lds** file (Linker Script file extension), which allows to specify to the linker where the variables have to be stored in memory, and the size of such variables. By setting the same linking option in both the `bootloader` and the `sm` components, variables can be shared among layers.

In particular, in the implemented solution some variables' size have been redefined to accommodate the bigger PQ keys, signatures and certificates within the `bootrom/sanctum_params.lds` file.

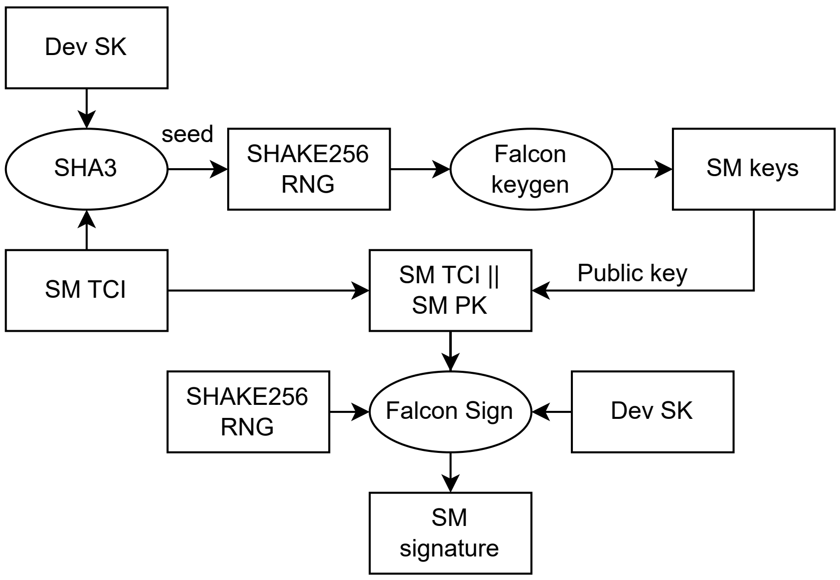


Figure 6.4. Implemented SM PQ keypair generation and PQ signature computation.

To reflect the same changes into the `sm` component, a patch has been created and located at `overlays/keysone/patches/opensbi/params.patch`. The modifications applied are described in the figure 6.5 and are related to the size of the parameters needed in the process using the `Falcon-512` version. To enable the `Falcon-1024` usage, besides the `LOGN_PARAM` value, the **.ldS** file must be modified accordingly. This would enable the flexible usage of both the versions, without the need to apply further changes to the file.

6.3.2 Compilation option

Finally, the last modification in this layer regards the compilation's optimization flag used.

In the Keystone project, the bootloader was compiled with the $-O2$ flag (in the `Makefile` contained in the `bootrom/` folder), which caused an infinite loop within the Falcon keypair generation. In particular, the custom `memcpy` function already present into the original project was trapped in an infinite recursive loop, causing the system to hang. Reducing the optimization flag from $-O2$ to $-O1$ solved the issue.

6.4 Security Monitor

Once the control is passed to the SM, the first operation performed is the parsing of the certificates obtained from the previous layer from DER format to `mbedtls_x509_crt` structure to verify that the certificates are formally correct. At this point, the first major change implemented is the verification of the certificates' signature, as shown in the figure 6.6, performed with the `falcon_verify` function. If the verification is successful then the public keys are extracted from the respective certificates and printed in **armored base64** (DER format encoding).

Once completed the certificates verification, the SM is initialised and it is possible to deploy the system to create and execute enclaves.

```
. = 0x801fd000; /* the last page before the payload */

/*CDI, hash are not shown*/
...

  /* 1281 Bytes : ECASM private key */
  PROVIDE( sanctum_ECASM_priv = . );
  . += 0x501;

  /* 891 Bytes : security monitor public key */
  PROVIDE( sanctum_sm_public_key = . );
  . += 0x381;

  /* 1281 Bytes : security monitor secret key */
  PROVIDE( sanctum_sm_secret_key = . );
  . += 0x501;

  /* 809 Bytes : security monitor's signature*/
  PROVIDE( sanctum_sm_signature = . );
  . += 0x329;

  /* 2065 Bytes : security monitor's certificate */
  PROVIDE( sanctum_cert_sm = . );
  . += 0x811;

  /* 1883 Bytes : root certificate */
  PROVIDE( sanctum_cert_root = . );
  . += 0x75b;

  /* 1903 Bytes : manufacturer certificate */
  PROVIDE( sanctum_cert_man = . );
  . += 0x76f;
```

Figure 6.5. Linker Script file modifications.

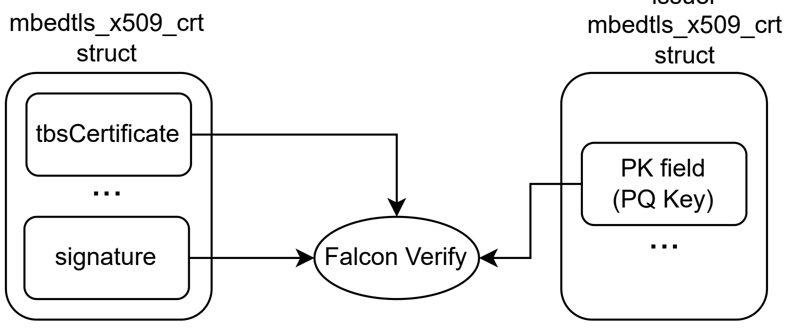


Figure 6.6. PQ Signature verification.

The Keystone project has been modified at this layer to perform the creation of a **PQ Local Attestation keypair** and the related certificate issuance. The function modified is the `create_enclave` function, implemented in the file `enclave.c`. More in depth, the implemented workflow is displayed in the figure 6.7:

1. the $\mathbf{CDI_{Enclave}}$ is created hashing the $\mathbf{CDI_{L0}}$ and the $\mathbf{TCI_{Enclave}}$;
2. the $\mathbf{CDI_{Enclave}}$ is used as seed to initialize the `shake256_context` used by the `falcon_keygen_make` function;
3. a call to `falcon_keygen_make` allows to create the **Local Attestation Keys** of the enclave;
4. the **X.509 certificate** for the Local Attestation Keys is created and signed with the $\mathbf{ECA_{SK}}$.

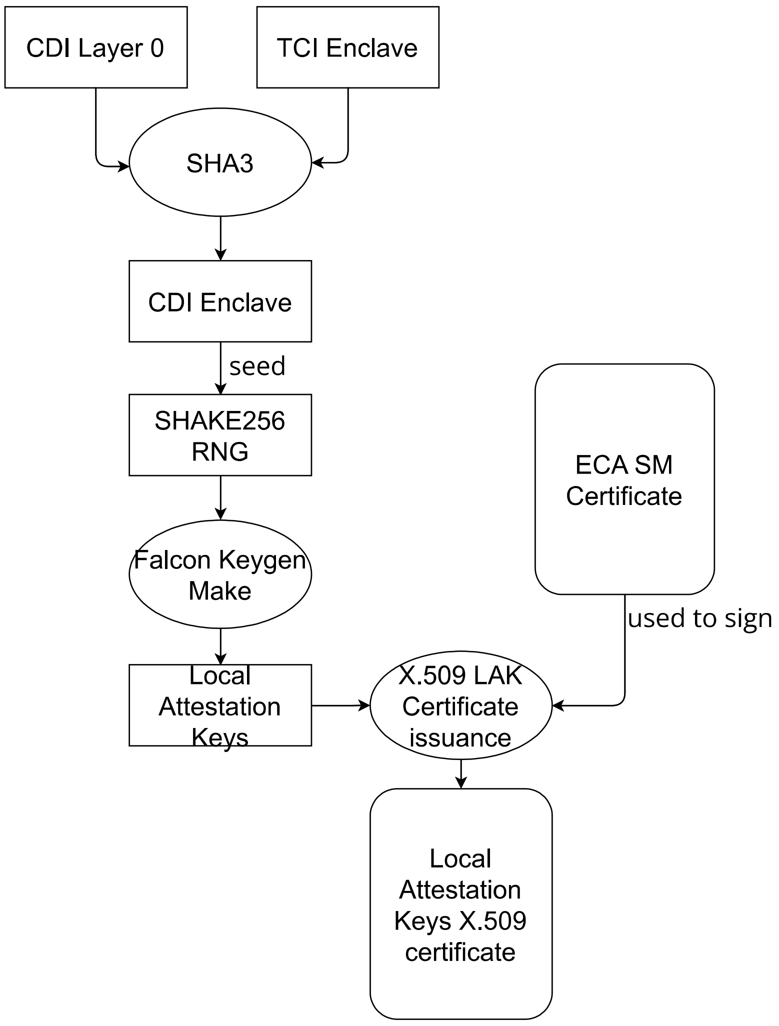


Figure 6.7. Implemented PQ Local Attestation Keys generation and relative X.509 certificate creation.

To correctly perform the Local Attestation Keys generation, other modifications were necessary. Such modifications have been applied to the `sm/enclave.h` and the `sm/enclave.c` files. In particular, due to the memory needed by the keypair generation and the certificate issuance, the stack was subject to overflow into adjacent memory locations. This in turn affected the functions used for the **PMP** management, which were using the modified memory locations, resulting in errors in the PMP entries' configuration. The issue has been solved applying the following changes, aimed at reducing the stack memory used:

- the `mbedtls_pk_context` structures used to embed the issuer and subject keys into the certificate have been removed from the `create_enclave` function and moved into the `enclave` structure;
- the `cert_der` array used in the DER format parsing of the certificate has been included in the `struct enclave`;

For the same reason above the method used to perform the parsing of certificate in DER format has been modified:

```
int mbedtls_x509write_crt_der_tmp(mbedtls_x509write_cert *ctx,
    unsigned char *buf, size_t size,
                             int (*f_rng)(void *, unsigned char *,
                                 size_t), unsigned char *tmp,
                                 shake256_context *rng, unsigned char*
                                 sig);
```

the function now includes three new parameters:

- `unsigned char *tmp`: parameter used to pass by reference the temporary buffer used in the signature computation over the `tbsCertificate` field. This allows to allocate the buffer in the caller, avoiding to allocate such buffer in the stack and reducing at the minimum the memory footprint posed by the operations performed by the SM. In the implemented solution, the buffer is allocated during the SM initialization and reused to perform various tasks.
- `shake256_context *rng`: the RNG is now provided by the caller and not allocated on the stack.
- `unsigned char *sig`: the buffer in which the signature is stored is provided by the caller and not allocated in the stack.

6.5 Enclave

The original Keystone project supports a simple **attestation** scheme, as well the **data-sealing** feature. In the implemented solution, PQ support for both the functionalities has been integrated.

6.5.1 Remote attestation

As regards the remote attestation, three entities are involved:

- the **Verifier**
- the **untrusted host**

- the **enclave**

Following are described the changes implemented within the various components involved in the attestation process, as well as the modifications applied to the `SDK` which provides the tools to validate the attestation report, and to the `runtime` that is involved in propagating the enclave's request to the SM and in copying the attestation buffer from the enclave to the host.

Verifier

The first step in implementing the PQ support in the attestation procedure was to include the Falcon algorithm in the `SDK`, since it contains the functions used by the Verifier to validate the reports.

After making the Falcon implementation available at the Verifier, the major modifications are:

- the `enclave_report_t` structure, defined in the `sdk/include/verifier/Report.hpp` file, now supports the computation of the PQ signature over the $\mathbf{TCI_{enclave}}$ and the `data` copied from the enclave;
- the `sm_report_t` structure now supports the insertion of the PQ $\mathbf{SM_{PK}}$;
- the `sm_report_t` structure, defined in the `sdk/include/verifier/Report.hpp` file, now supports the computation of the PQ signature over the $\mathbf{TCI_{SM}}$ and the $\mathbf{SM_{PK}}$.
- the `report_t` structure now supports the insertion of the PQ $\mathbf{Device_{PK}}$.
- the `checkSignaturesOnly` function (defined in `sdk/src/verifier/Report.cpp`) used to perform the signatures verification of the `report_t` structure has been modified by removing the calls to the `ed25519_verify` function, replaced with the `falcon_verify` method.

The structures' changes has been reflected in the corresponding structures defined in the `sm/enclave.h` file;

Enclave and runtime

Once the enclave has requested the SM to provide the attestation report, the SM is delegated to compute all the values needed in the report. To cross the enclave's boundary, the enclave must communicate via the edge call

```
int attest_enclave(void* report, void* data, size_t size)
```

Following are described the steps characterising the attestation flow:

1. the Verifier process is executed, generating a nonce;
2. the untrusted host process is executed, receiving the nonce;
3. the enclave process is executed, receives the nonce from the untrusted host via a shared buffer through the `copy_from_shared(nonce, retdata.offset, retdata.size)` edge call, and calls the `attest_enclave` edge call, passing as parameters the pointer to the `report` buffer (allocated in the enclave memory) which is filled with the attestation report computed by the SM, the `data` block from the enclave which contains the nonce, and the `size` of the nonce

4. the runtime exits the enclave with an `SBI_CALL`;
5. the SM resumes its execution, performs the attestation by executing the `attest_enclave` function that is mapped to the `SBI_CALL` called by the runtime, fills the buffer and returns;
6. on return the runtime copies the attestation report in the buffer allocated by the enclave;
7. the enclave copies the buffer into the untrusted host via the shared memory region with the `ocall(OCALL_COPY_REPORT, buffer, size, 0, 0)` edge call;
8. the untrusted host returns the attestation report to the Verifier which validates the report.

The Runtime is responsible to copy data from the enclave to the SM memory, from the enclave to the user space host process, and vice versa. In the original Keystone implementation the attestation buffer size copied into the host process memory is statically set to 2048 Bytes, hence the size of the buffer has been manually increased to correctly process the report. The modification have been applied to the file `runtime/call/syscall.c`:

```
copy_to_user((void*)arg0, (void*)rt_copy_buffer_1, 5120);
```

Attestation and Security Monitor

The major modifications related to the attestation are related to the Security Monitor.

The first update, as already stated is related to the reports structure, reflecting the changes to the `enclave_report`, `sm_report`, and `report` structs.

In the implemented scenario, the modifications applied to the attestation report creation process (e.g. figure 6.8) are:

- the PQ **Device$_{\mathbf{PK}}$** is copied into the `report` struct;
- the PQ **SM$_{\mathbf{PK}}$** and the **SM signature**, both computed in the boot stage, are copied into the `sm_report`;
- the PQ **Enclave signature** is computed using the `falcon_sign_dyn` function.

More in depth, the signature over the enclave is computed with the following method:

```
void sm_sign(void* signature, const void* data, size_t len)
{
  sign(signature, data, len, sm_private_key, tmp, &rng);
}
```

The call to the Falcon API has been implemented in the the function `sign` defined in the file `sm.c` which now receives the `shake256_context` parameter, initialized during the SM initialization. Internally, it performs the following call:

```
void sign(void* sign, const void* data, size_t len,const unsigned
    char* private_key, unsigned char* tmp, shake256_context *rng)
{
  size_t sig_len;
```

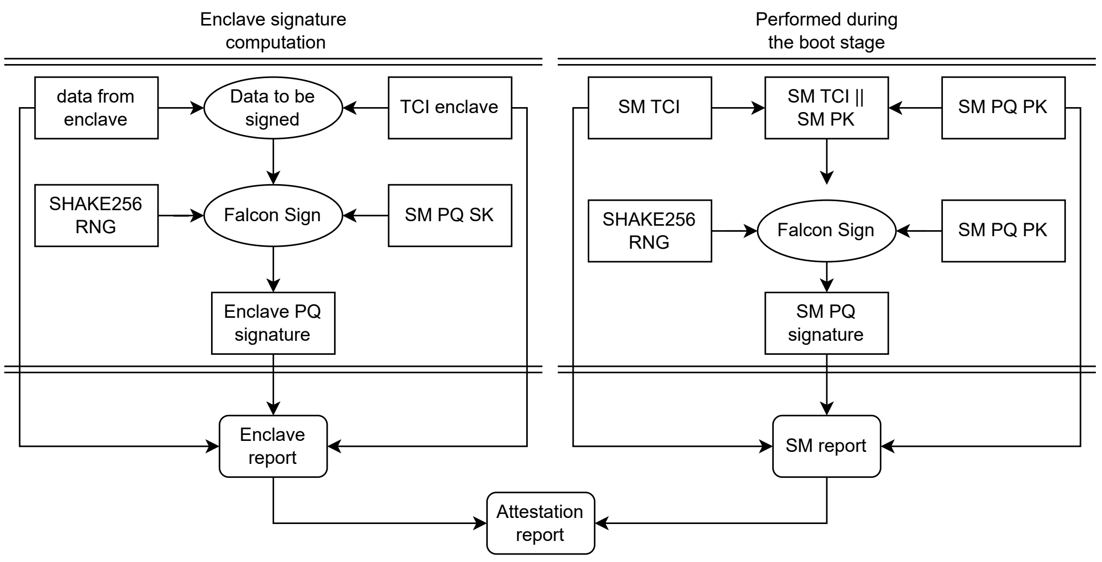


Figure 6.8. Implemented PQ attestation evidence generation.

```
  falcon_sign_dyn(rng, sign, &sig_len, FALCON_SIG_CT, private_key,
     FALCON_SK_SIZE, data, len, tmp,
     FALCON_TMPSIZE_SIGNDYN(LOGN_PARAM));
}
```

6.5.2 Sealing-key derivation

Finally, the last changes have been applied to the sealing-key derivation mechanism.

In particular the `sealing struct` has been modified (both in the `sdk/` folder and in the `sm/enclave.h` file) to include the PQ signature computed over the `sealing key`. The new implemented process is described in the figure 6.9:

1. the **SM PQ keypair** is generated as previously described in the `bootrom`;
2. the `KDF` receives as input the PQ $\mathbf{SM_{SK}}$, the $\mathbf{TCI_{Enclave}}$, a **Key identifier**, and returns the generated **Sealing-Key**;
3. the signature over the generated **Sealing key** is computed with the `falcon_sign_dyn` function using the PQ $\mathbf{SM_{SK}}$;
4. the **PQ signature** and the **Sealing key** are inserted in the `seal key struct`.

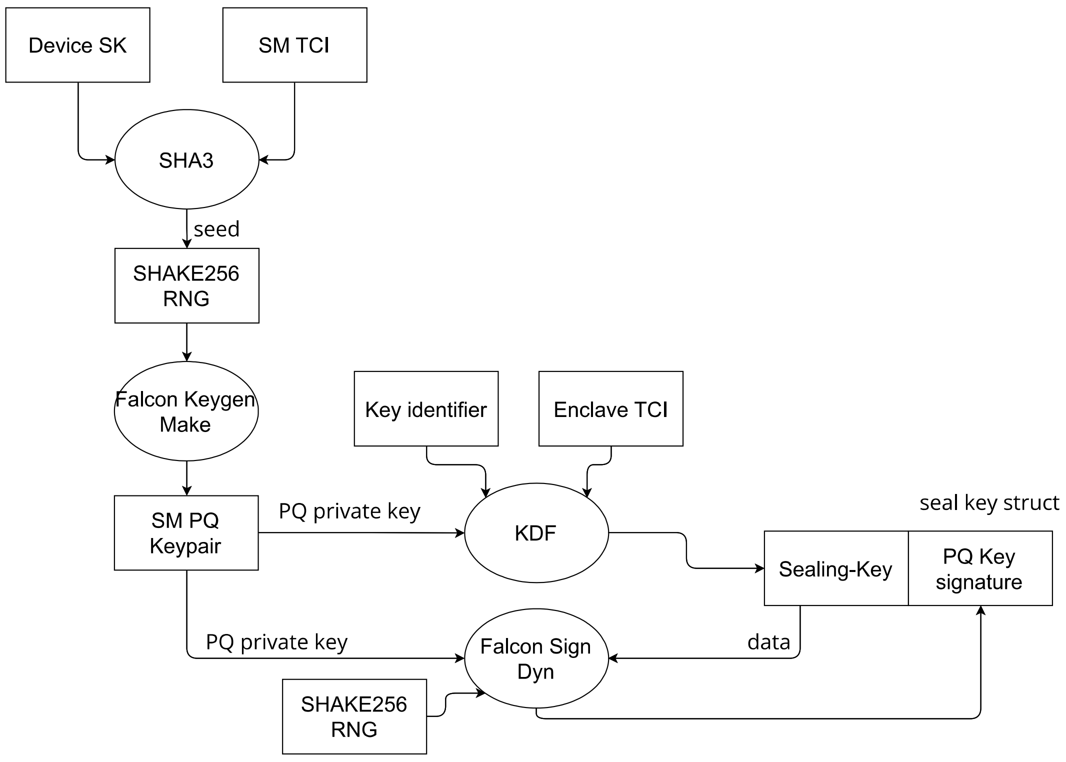


Figure 6.9. Implemented PQ Sealing-Key derivation.

Chapter 7

Test

This chapter contains the description of several tests conducted on the implemented solution.

The first part addresses the functionalities offered by the developed project, followed by performance test in which the execution time is analysed. Then, the Falcon algorithm is compared with the Dilithium algorithm, running simple RISC-V executables on the RISC-V emulated QEMU platform, showing the differences in sizes and performances in the keypair generation, signature computation, and signature verification.

7.1 Testbed

The tests have been conducted on a single machine by means of the RISC-V QEMU emulated platform provided by the original Keystone project. The testbed consists in a **Acer Aspire 7** `(A715-71G-76HB)` machine with the following specifications:

- **CPU**: Intel Core i7-7700HQ @ 2.8GHz;
- **GPU**: NVIDIA GeForce GTX 1050 2GB GDDR5 Dedicated VRAM;
- **RAM**: 8GB DDR4;
- **Storage**: 128 GB SSD, 1TB HDD;
- **OS**: Ubuntu 22.04 LTS, 64-bit.

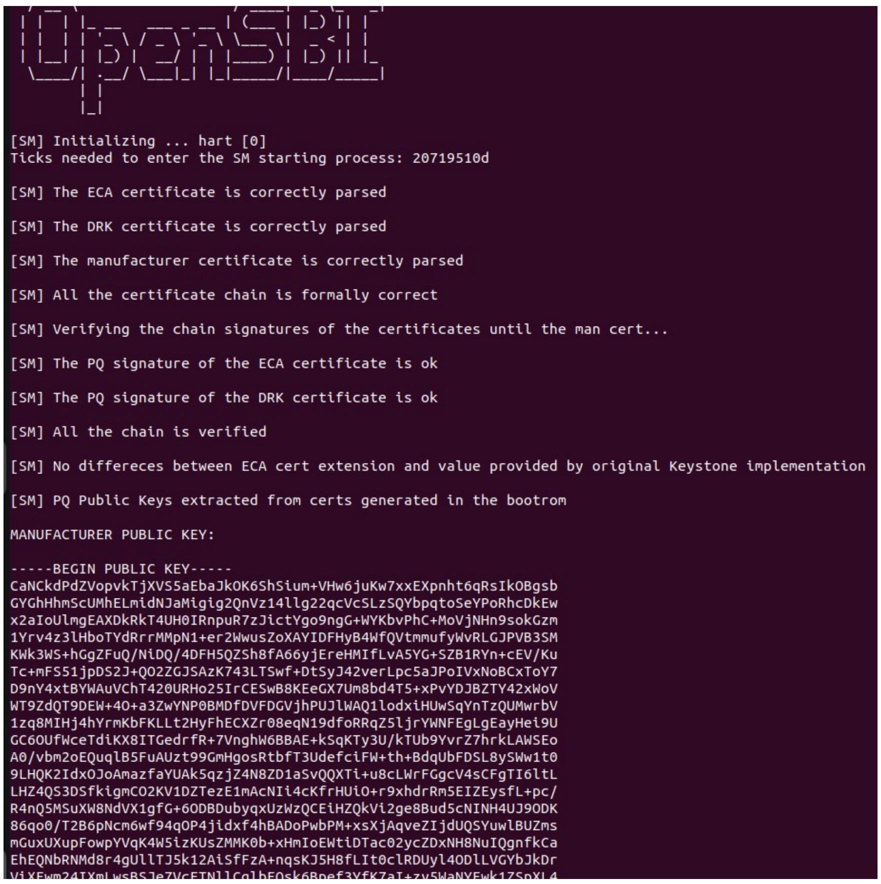
7.2 Functional tests

Different tests have been executed to check the correctness of the features introduced.

The first test presented shows the correct behaviour of the secure boot process with the PQ support, in which PQ keys, PQ signatures, and PQ certificates are generated/used. In the figure 7.1 is shown the screen output of the correct secure boot execution, along with the PQ public key extracted from the manufacturer certificate during the SM initialization. Due to space issues the public keys extracted from the ECA and the DRK certificates are not shown.

This example shows that the following steps are executed correctly:

- certificates parsing from DER format to `mbed_tls_x509` structure, in which PQ keys and PQ signature are parsed correctly;
- verifications of the PQ signatures computed over the certificates performed correctly with the `falcon_verify` function;
- verifications performed with the PQ keys extracted from the certificates' issuers;
- extraction of the PQ PK from the respective certificates.



```
[SM] Initializing ... hart [0]
Ticks needed to enter the SM starting process: 20719510d

[SM] The ECA certificate is correctly parsed

[SM] The DRK certificate is correctly parsed

[SM] The manufacturer certificate is correctly parsed

[SM] All the certificate chain is formally correct

[SM] Verifying the chain signatures of the certificates until the man cert...

[SM] The PQ signature of the ECA certificate is ok

[SM] The PQ signature of the DRK certificate is ok

[SM] All the chain is verified

[SM] No differeces between ECA cert extension and value provided by original Keystone implementation

[SM] PQ Public Keys extracted from certs generated in the bootrom

MANUFACTURER PUBLIC KEY:

-----BEGIN PUBLIC KEY-----
CaNCkdPdZVopvkTjXVS5aEbaJkOK6ShSium+VHw6juKw7xxEXpnht6qRsIkOBgsb
GYGhHhmScUMhELmidNJaMigig2QnVz14llg22qcVcSLzSQYbpqtoSeYPoRhcDkEw
x2aIoUlmgEAXDkRkT4UH0IRnpuR7zJictYgo9ngG+WYKbvPhC+MoVjNHn9sokGzm
1Yrv4z3lHboTYdRrrMMpN1+er2WwusZoXAYIDFHyB4WfQVtmmufyWvRLGJPVB3SM
KWk3WS+hGgZFuQ/NiDQ/4DFH5QZSh8fA66yjEreHMIfLvA5YG+SZB1RYn+cEV/Ku
Tc+mFS51jpDS2J+QO2ZGJSAzK743LTSwf+DtSyJ42verLpc5aJPoIVxNoBCxToY7
D9nY4xtBYWAuVChT420URHo25IrCESwB8KEeGX7Um8bd4T5+xPvYDJBZTY42xWoV
WT9ZdQT9DEW+4O+a3ZwYNP0BMDfDVFDGVjhPUJlWAQ1lodxiHUwSqYnTzQUMwrbV
1zq8MIHj4hYrmKbFKLLt2HyFhECXZr08eqN19dfoRRqZ5ljrYWNFEgLgEayHei9U
GC6OUfWceTdiKX8ITGedrfR+7VnghW6BBAE+kSqKTy3U/kTUb9YvrZ7hrkLAWSEo
A0/vbm2oEQuqlB5FuAUzt99GmHgosRtbfT3UdefciFW+th+BdqUbFDSL8ySWw1t0
9LHQK2IdxOJoAmazfaYUAk5qzjZ4N8ZD1aSvQQXTi+u8cLWrFGgcV4sCFgTI6ltL
LHZ4QS3DSfkigmCO2KV1DZTezE1mAcNIi4cKfrHUiO+r9xhdrRm5EIZEysfL+pc/
R4nQ5MSuXW8NdVX1gfG+6ODBDubyqxUzWzQCEiHZQkVi2ge8Bud5cNINH4UJ9ODK
86qo0/T2B6pNcm6wf94qOP4jidxf4hBADoPwbPM+xsXjAqveZIjdUQSYuwlBUZms
mGuxUXupFowpYVqK4W5izKUsZMMK0b+xHmIoEWtiDTac02ycZDxNH8NuIQgnfkCa
EhEQNbRNMd8r4gUllTJ5k12AiSfFzA+nqsKJ5H8fLIt0clRDUyl4ODlLVGYbJkDr
```

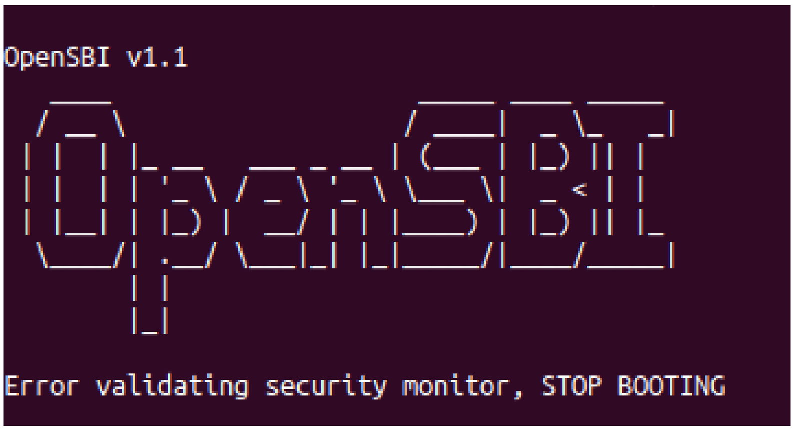
Figure 7.1. Correct Secure Boot in the PQ implementation.

The second example shows the effect of a failed SM signature verification in the `bootrom`. The secure boot feature permits the RoT to successfully halts the boot process due to the failure. In particular, in the figure 7.2 is shown the screen output when the verification fails.

The third test has been conducted to check the output whenever the verification of any of the signature contained in any PQ certificates provided by the RoT fails. More in depth, in the figure 7.3 is shown an example of verification failure of the ECA certificate's signature.

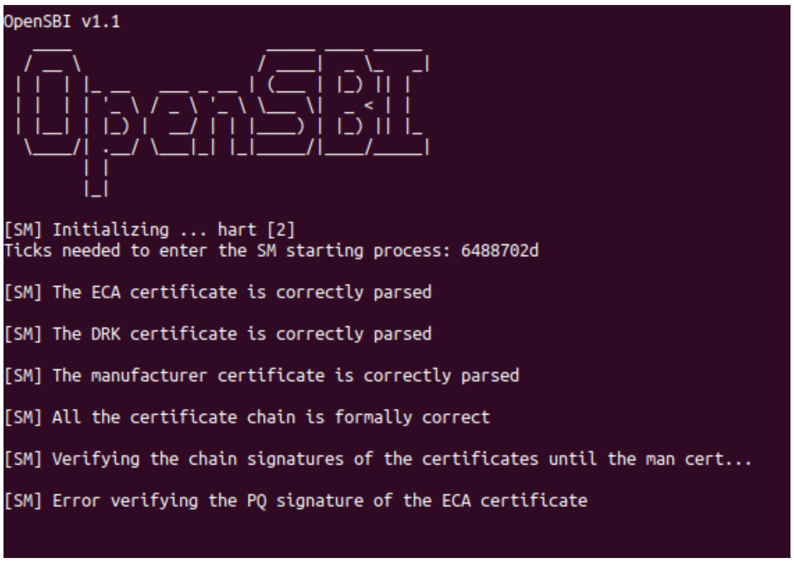
Due to space issues, in the following tests, certificates and public keys have been printed on screen only partially.

The fourth test aims to present the output in case of a correct remote attestation procedure. By running the modified version of the `attestor.ke` executable located at `/usr/share/keystone/example/attestor.ke`, the Verifier process, the host process,



```
OpenSBI v1.1
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

Error validating security monitor, STOP BOOTING
```

Figure 7.2. Wrong SM PQ signature verification.



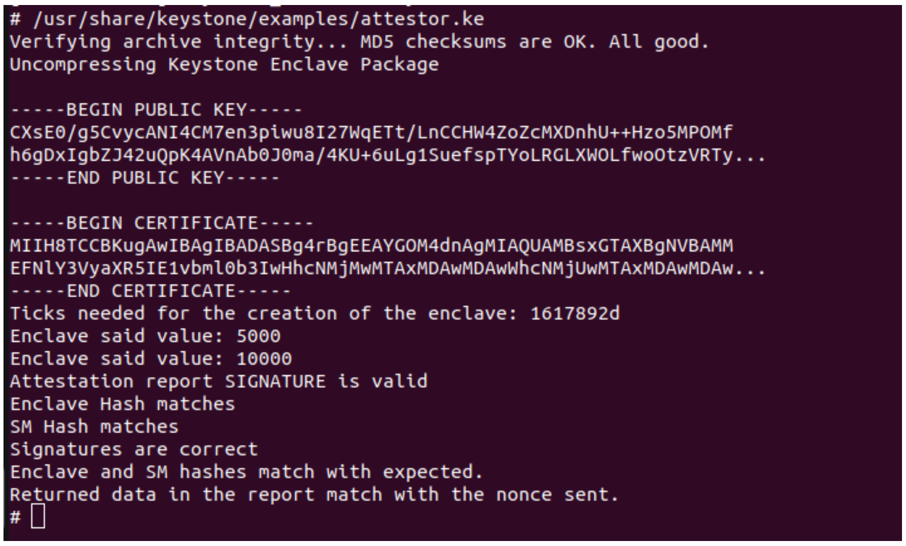
```
OpenSBI v1.1
   ____                    _____ ____ _____
  / __ \                  / ____|  _ \_   _|
 | |  | |_ __   ___ _ __ | (___ | |_) || |
 | |  | | '_ \ / _ \ '_ \ \___ \|  _ < | |
 | |__| | |_) |  __/ | | |____) | |_) || |_
  \____/| .__/ \___|_| |_|_____/|____/_____|
        | |
        |_|

[SM] Initializing ... hart [2]
Ticks needed to enter the SM starting process: 6488702d

[SM] The ECA certificate is correctly parsed

[SM] The DRK certificate is correctly parsed

[SM] The manufacturer certificate is correctly parsed

[SM] All the certificate chain is formally correct

[SM] Verifying the chain signatures of the certificates until the man cert...

[SM] Error verifying the PQ signature of the ECA certificate
```

Figure 7.3. Wrong certificate PQ signature verification.

and the enclave process are executed and the attestation is performed. The example (e.g. figure 7.4) shows the output of the correct procedure in which:

- the **PQ Local Attestation Keys** and **PQ certificate** are generated correctly at enclave creation;
- the **PQ signature** of the `report` structure is correct and verified;
- the $\mathbf{TCI_{SM}}$ and the $\mathbf{TCI_{enclave}}$ are correct and match with expected;
- the **PQ signature** of the `enclave_report` structure is verified correctly;
- the **nonce** provided by the Verifier matches with the one expected.

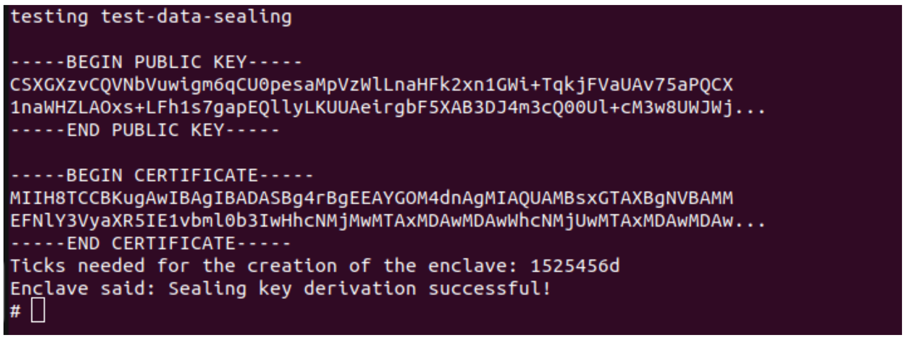
Finally, is shown the example executing the provided file `tests.ke` that allows to run several enclaves consecutively. More in depth, the figure 7.5 shows the output of **sealing-key derivation** process, which is the most relevant among the enclaves executed due to the changes introduced in the previous chapters. In particular, besides the **Local**



```
# /usr/share/keystone/examples/attestor.ke
Verifying archive integrity... MD5 checksums are OK. All good.
Uncompressing Keystone Enclave Package

-----BEGIN PUBLIC KEY-----
CXsE0/g5CvycANI4CM7en3piwu8I27WqETt/LnCCHW4ZoZcMXDnhU++Hzo5MPOMf
h6gDxIgbZJ42uQpK4AVnAb0J0ma/4KU+6uLg1SuefspTYoLRGLXWOLfwoOtzVRTy...
-----END PUBLIC KEY-----

-----BEGIN CERTIFICATE-----
MIIH8TCCBKugAwIBAgIBADASBg4rBgEEAYGOM4dnAgMIAQUAMBsxGTAXBgNVBAMM
EFNlY3VyaXR5IE1vbml0b3IwHhcNMjMwMTAxMDAwMDAwWhcNMjUwMTAxMDAwMDAw...
-----END CERTIFICATE-----
Ticks needed for the creation of the enclave: 1617892d
Enclave said value: 5000
Enclave said value: 10000
Attestation report SIGNATURE is valid
Enclave Hash matches
SM Hash matches
Signatures are correct
Enclave and SM hashes match with expected.
Returned data in the report match with the nonce sent.
# 
```

Figure 7.4. Remote attestation example with PQ support.

Attestation Keys creation and the **PQ certificate** issuance, the figure shows the output in case of correct sealing-key derivation.



```
testing test-data-sealing

-----BEGIN PUBLIC KEY-----
CSXGXzvCQVNbVuwigm6qCU0pesaMpVzWlLnaHFk2xn1GWi+TqkjFVaUAv75aPQCX
1naWHZLAOxs+LFh1s7gapEQllyLKUUAeirgbF5XAB3DJ4m3cQ00Ul+cM3w8UWJWj...
-----END PUBLIC KEY-----

-----BEGIN CERTIFICATE-----
MIIH8TCCBKugAwIBAgIBADASBg4rBgEEAYGOM4dnAgMIAQUAMBsxGTAXBgNVBAMM
EFNlY3VyaXR5IE1vbml0b3IwHhcNMjMwMTAxMDAwMDAwWhcNMjUwMTAxMDAwMDAw...
-----END CERTIFICATE-----
Ticks needed for the creation of the enclave: 1525456d
Enclave said: Sealing key derivation successful!
# 
```

Figure 7.5. Modified Sealing-key derivation output example.

7.3 Performance test

In this section are shown the performance of the developed solution with respect to the DICE-based reference Keystone version. In particular, in the tests are reported the ticks needed to perform various tasks, compared with the ticks needed to perform the same tasks in the TORSEC developed Keystone.

The ticks computation has been possible thanks to the usage of the `sbi_time.h` library, available with the original Keystone project, which allows to determine the time window in which the task is executed by means of calls to the `sbi_timer_value` function:

- `init_value = sbi_timer_value()`: the initial reference value is obtained;
- `final_value = sbi_timer_value()`: the final reference value is obtained;
- the ticks are computed as `final_value - initial_value` and printed on screen.

Such values are used in the following graphs to show the performance downgrade with respect to the following operations:

- booting stage;
- SM initialization;
- Local Attestation Keys and certificate generation at enclave creation.

In the figure 7.6 are shown the ticks reported in different execution of the two versions of the project. In particular, due to the overhead introduced by the Falcon algorithm with respect to the ed25519 algorithm, the ticks are increased on average around 30%. This is due to the complex operations performed by the PQ algorithms in performing the key-pair generation, signature computation, signature verification, and certificate generation performed during the boot stage.

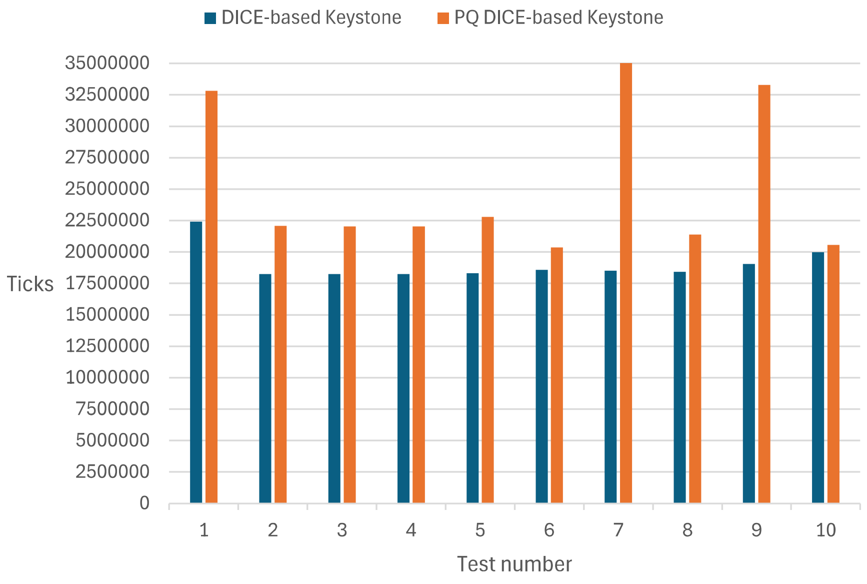


Figure 7.6. Ticks needed to complete the secure boot process.

As regards the SM initialization process (e.g. figure 7.7), in this case the performance downgrade is significantly smaller than the previous example, resulting around a 3% downgrade of performance on average. This is due to the operations performed at this stage, that are certificates parsing and signatures verification, in which Falcon shows the best performance among the PQ candidates with small overhead.

Finally the last performance test, depicted in the figure 7.8, shows the downgrade introduced due to the PQ Local Attestation Keys and related certificate issuance at enclave creation. More in depth, the keypair generation and certificate issuance are compared, excluding the entire enclave process. This is due to modifications introduced in the original Keystone project, over which this version is built upon, in which now the enclave measure is computed in a more efficient way. This does not allow to compare correctly entirely the enclave creation in the two versions. The performance downgrade here is clear, since the PQ process is more complex and critical under this point of view. More in depth, the PQ process' ticks are two orders of magnitude greater than the ed25519-based process.

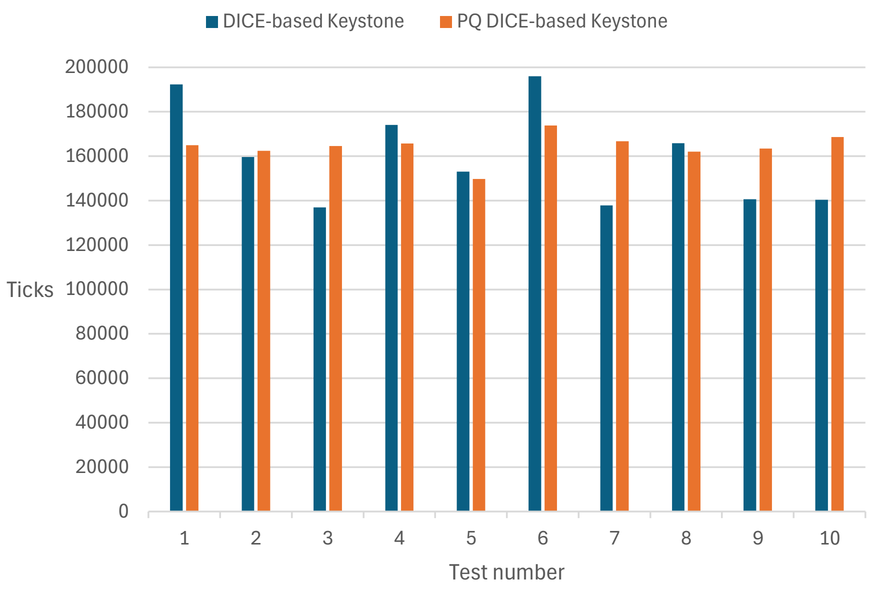


Figure 7.7. Ticks needed to complete the SM initialization.

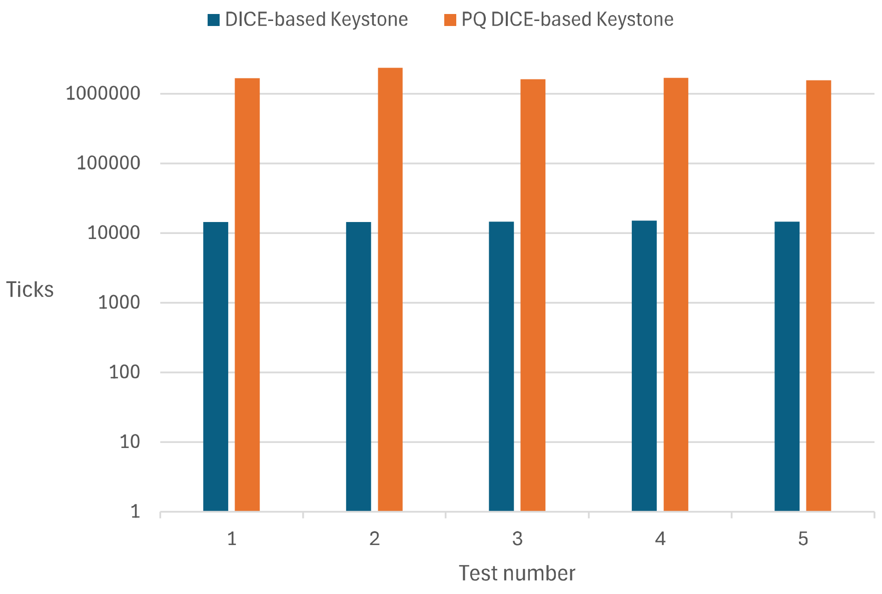


Figure 7.8. Ticks needed to complete the Local Attestation Keys and certificate generation at enclave creation.

7.4 PQ algorithm evaluation on RISC-V

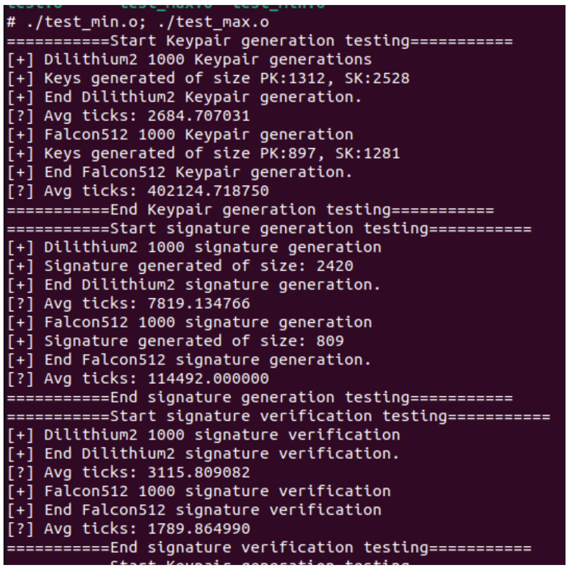
In this section are evaluated and compared the public-key operations performed with the following algorithms on the RISC-V platform provided by the original Keystone project:

- **Dilithium-2** vs **Falcon-512**;
- **Dilithium-5** vs **Falcon-1024**.

In particular are compared the average ticks necessary to perform:

- 1000 keypair generations;
- 1000 signature computation;
- 1000 signature verification.

To perform the abovementioned tests, two RISC-V executables have been generated with the `riscv64-unknown-elf-gcc` compiler (namely `test_min.o` to compare the **Dilithium-2** and **Falcon-512** versions, and `test_max.o` to compare the **Dilithium-5** and **Falcon-1024** versions). Both the executables then have been copied under the `build-generic64/overlay/root/` folder to provide the executables in the RISC-V emulated environment. The output is shown in the figure 7.9 and figure 7.11.



```
# ./test_min.o; ./test_max.o
===========Start Keypair generation testing===========
[+] Dilithium2 1000 Keypair generations
[+] Keys generated of size PK:1312, SK:2528
[+] End Dilithium2 Keypair generation.
[?] Avg ticks: 2684.707031
[+] Falcon512 1000 Keypair generation
[+] Keys generated of size PK:897, SK:1281
[+] End Falcon512 Keypair generation.
[?] Avg ticks: 402124.718750
===========End Keypair generation testing===========
===========Start signature generation testing===========
[+] Dilithium2 1000 signature generation
[+] Signature generated of size: 2420
[+] End Dilithium2 signature generation.
[?] Avg ticks: 7819.134766
[+] Falcon512 1000 signature generation
[+] Signature generated of size: 809
[+] End Falcon512 signature generation.
[?] Avg ticks: 114492.000000
===========End signature generation testing===========
===========Start signature verification testing===========
[+] Dilithium2 1000 signature verification
[+] End Dilithium2 signature verification.
[?] Avg ticks: 3115.809082
[+] Falcon512 1000 signature verification
[+] End Falcon512 signature verification
[?] Avg ticks: 1789.864990
===========End signature verification testing===========
```

Figure 7.9. Dilithium2 vs Falcon512.

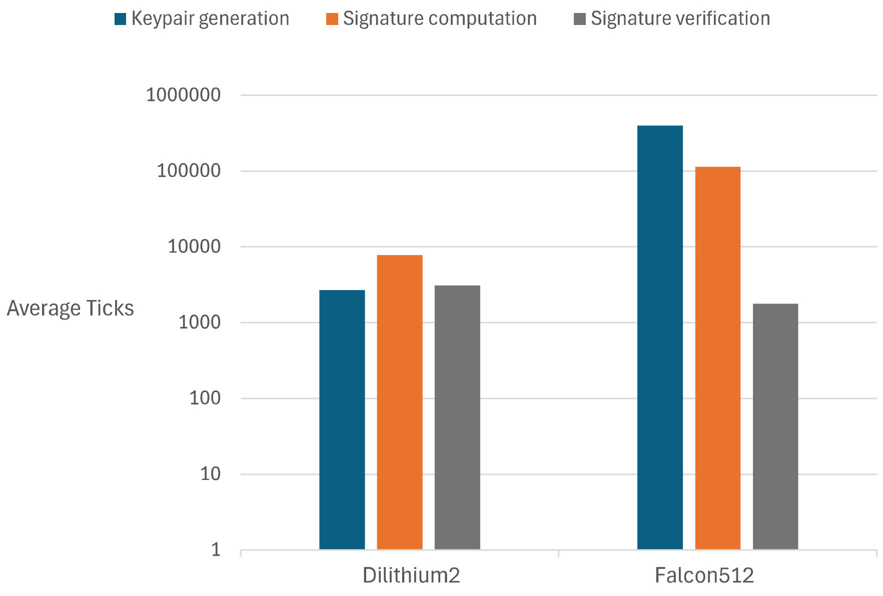
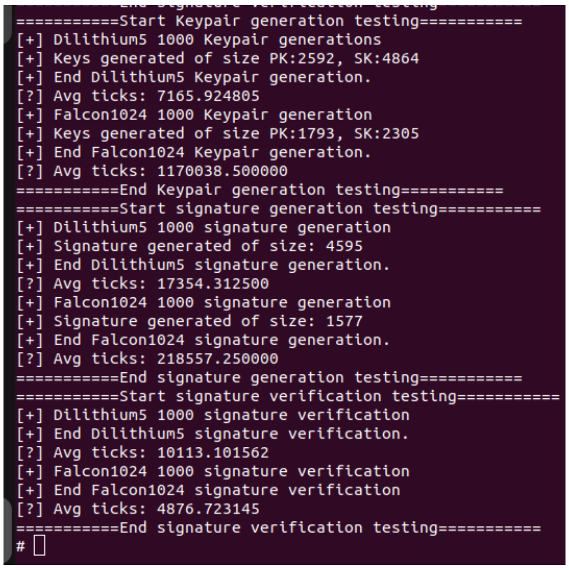


Figure 7.10. Falcon512 vs Dilithium2.

As previously stated, Dilithium remains the best choice in terms of performance with respect to Falcon (e.g. figure 7.10 and 7.12) which may be a viable choice in scenarios in which signature verifications are performed more frequently than the signature computation.



```
===========Start Keypair generation testing==========
[+] Dilithium5 1000 Keypair generations
[+] Keys generated of size PK:2592, SK:4864
[+] End Dilithium5 Keypair generation.
[?] Avg ticks: 7165.924805
[+] Falcon1024 1000 Keypair generation
[+] Keys generated of size PK:1793, SK:2305
[+] End Falcon1024 Keypair generation.
[?] Avg ticks: 1170038.500000
===========End Keypair generation testing==========
===========Start signature generation testing==========
[+] Dilithium5 1000 signature generation
[+] Signature generated of size: 4595
[+] End Dilithium5 signature generation.
[?] Avg ticks: 17354.312500
[+] Falcon1024 1000 signature generation
[+] Signature generated of size: 1577
[+] End Falcon1024 signature generation.
[?] Avg ticks: 218557.250000
===========End signature generation testing==========
===========Start signature verification testing==========
[+] Dilithium5 1000 signature verification
[+] End Dilithium5 signature verification.
[?] Avg ticks: 10113.101562
[+] Falcon1024 1000 signature verification
[+] End Falcon1024 signature verification
[?] Avg ticks: 4876.723145
===========End signature verification testing==========
#
```

Figure 7.11. Dilithium5 vs Falcon1024.

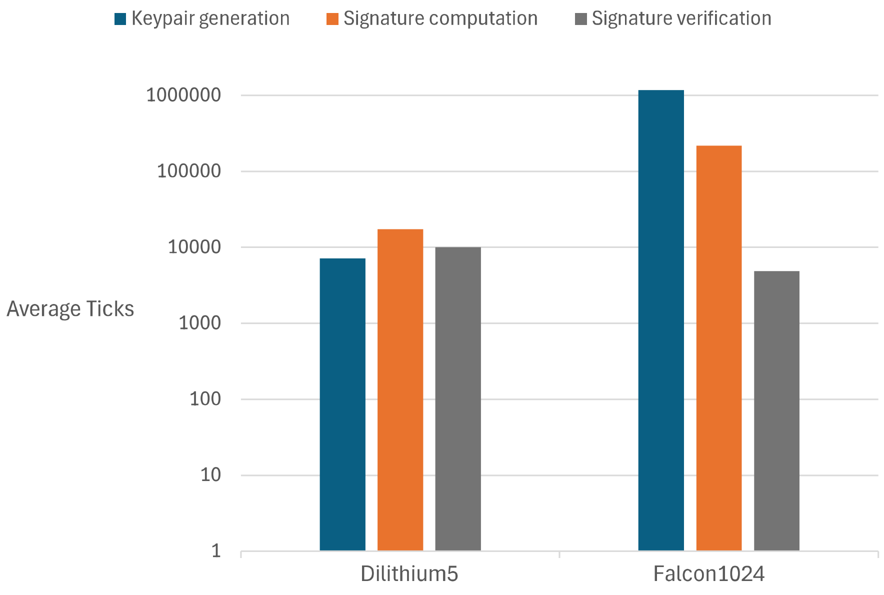


Figure 7.12. Falcon1024 vs Dilithium5.

Nevertheless, Falcon has been selected due to the reduced keys and signature sizes and due to the memory issues caused by the Dilithium implementation and deployment. In fact, considering the mechanism used in the Keystone project to share the certificates, keys, and useful values among the components composing the system, Falcon results the best choice in constrained environments in which such aspects are critical and must be considered.

Chapter 8

Conclusions

In this document, the extension of the algorithm's support in the Keystone TEE to include PQ algorithms has been described, designed and implemented.

Starting from analysis on the state of the art as regards the issues that the IT security is facing with respect to the quantum computing impact on the current technologies and an analysis conducted on the actual TEEs technologies, an evaluation of the PQ candidates led to select Falcon as the most promising algorithm, employed in modifying the DICE-based Keystone framework. Such modifications have been conducted in accordance to the best practices established by organizations that are putting effort in providing guidelines to migrate the actual technologies from a quantum-vulnerable state to a quantum-safe state, designing a system that is secured and not endangered by the quantum advent, posing attention to the requirements set by constrained devices in which several limitations further complicate the deployment of PQ algorithm.

The RISC-V platform over which Keystone is deployed, is perfectly suitable in this context due to its flexibility and the characteristic configuration extent, features that allow to deploy the platform as a basis over which IoT and small embedded devices can be designed and built upon.

The adoption of the DICE specification within the described solution, enables the definition of a strong device identity in such devices, and the consequent usage of the identity in performing operations that are crucial from a security perspective and hardened by the deployment of the PQ Falcon algorithm, which remains a suitable solution in embedded systems and IoT devices.

This work can be referenced as a starting point for further developments on the topic, to provide improvement in the performance issues that PQ algorithms may present in this context or to analyse and implement other suitable algorithms. Example of projects can be: implementation of the Dilithium algorithm due to the greater performance in the RISC-V platform but solving the issues mentioned in this document, extension of the algorithm support within Keystone to allow flexible swap among PQ and the ECC algorithms, hardware accelerator to increase the PQ algorithms' performance, or support for hybrid certificates.

Bibliography

[1] GlobalPlatform, “TEE System Architecture, version 1.3”, GPD SPE 009, May 2022, https://globalplatform.org/specs-library/tee-system-architecture/

[2] D. Lee, D. Kohlbrenner, S. Shinde, L. Asanovic, and D. Song, “Keystone: An Open Framework for Architecting Trusted Execution Environments”, EuroSys ’20: European Conf. on Computer Systems, Heraklion (Greece), April 27-30, 2020, pp. 1–16, DOI 10.1145/3342195.3387532

[3] TCG, “Hardware Requirements for a Device Identifier Composition Engine”, 2018, https://trustedcomputinggroup.org/wp-content/uploads/Hardware-Requirements-for-Device-Identifier-Composition-Engine-r78_For-Publication.pdf

[4] TCG, “DICE Layering Architecture, version 1.0”, July 2020, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf

[5] TCG, “DICE Attestation Architecture, version 1.0”, March 2021, https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-r23-final.pdf

[6] M. J. Kannwischer, R. Petri, J. Rijneveld, P. Schwabe, and K. Stoffelen, “PQM4: Post-quantum crypto library for the ARM Cortex-M4”, https://github.com/mupq/pqm4

[7] M. Giles, “Explainer: What is a quantum computer?”, MIT Technology Review, January 2019. https://www.technologyreview.com/2019/01/29/66141/what-is-quantum-computing/

[8] IBM, “What is quantum computing?”, https://www.ibm.com/topics/quantum-computing

[9] S. E. Yunakovsky, M. Kot, N. Pozhar, D. Nabokov, M. Kudinov, A. Guglya, E. O. Kiktenko, E. Kolycheva, A. Borisov, and A. K. Fedorov, “Towards security recommendations for public-key infrastructures for production environments in the post-quantum era”, EPJ Quantum Technology, vol. 8, May 2021, pp. 3–4, DOI 10.1140/epjqt/s40507-021-00104-z

[10] P. W. Shor, “Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer”, SIAM Journal on Computing, vol. 26, October 1997, pp. 1848–1509, DOI 10.1137/S0097539795293172

[11] C. Gidney and M. Ekerå, “How to factor 2048 bit RSA integers in 8 hours using 20 million noisy qubits”, Quantum, vol. 5, April 2021, p. 433, DOI 10.22331/q-2021-04-15-433

[12] M. Giles, “Explainer: What is post-quantum cryptography”, MIT Technology Review, July 2019. https://www.technologyreview.com/2019/07/12/134211/explainer-what-is-post-quantum-cryptography

[13] G. Alagic, D. Apon, D. Cooper, Q. Dang, T. Dang, J. Kelsey, J. Lichtinger, Y.-K. Liu, C. Miller, D. Moody, R. Peralta, R. Perlner, A. Robinson, and D. Smith-Tone,

"Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process", NIST IR 8413-upd1, July 2022, DOI 10.6028/NIST.IR.8413-upd1

[14] Confidential Computing Consortium, https://confidentialcomputing.io/

[15] M. Souppaya, M. Bartock, K. Scarfone, R. Savino, T. Knoll, U. Shetty, M. Cherfaoui, R. Yeluri, D. Banks, A. Malhotra, M. Jordan, D. Pendarakis, and P. Romness, "Hardware-Enabled Security: Enabling a Layered Approach to Platform Security for Cloud and Edge Computing Use Cases", NIST IR 8320, May 2022, DOI 10.6028/NIST.IR.8320

[16] The Trusted Computing Group, https://trustedcomputinggroup.org/

[17] Global Platform, https://globalplatform.org/

[18] Global Platform, "TEE & TPM - working together in harmony", https://globalplatform.org/tpm-tee-working-together-in-harmony/

[19] TCG, "Trusted Platform Module Library Part 1: Architecture", ISO/IEC 11889-1, November 2016, https://www.iso.org/standard/66510.html

[20] TCG, "Device Identifier Composition Engine", https://trustedcomputinggroup.org/work-groups/dice-architectures/

[21] The Keystone Enclave project, https://keystone-enclave.org/

[22] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey", IEEE Access, vol. 11, 2023, pp. 24696–24711, DOI 10.1109/ACCESS.2023.3246491

[23] A. Waterman, Y. Lee, R. Avizienis, D. A. Patterson, and K. Asanovi¸ "The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.9.1", Tech. Rep. UCB/EECS-2016-161, EECS Department, University of California, Berkeley, November 2016. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-161.html

[24] Q. Dang, "Secure Hash Standard", NIST FIPS.180-4, August 2015, DOI 10.6028/NIST.FIPS.180-4

[25] National Institute of Standards and Technology, "The Keyed-Hash Message Authentication Code", NIST FIPS.198-1, July 2008, DOI 10.6028/NIST.FIPS.198-1

[26] E. Barker, L. Chen, and R. Davis, "Recommendation for Key-Derivation Methods in Key-Establishment Schemes", NIST SP 800-56C, August 2020, DOI 10.6028/NIST.SP.800-56Cr2

[27] S. Boeyen, S. Santesson, T. Polk, R. Housley, S. Farrell, and D. Cooper, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC RFC-5280, May 2008, DOI 10.17487/RFC5280

[28] L. M. K. Vandersypen, M. Steffen, G. Breyta, C. S. Yannoni, M. H. Sherwood, and I. L. Chuang, "Experimental realization of Shor's quantum factoring algorithm using nuclear magnetic resonance", Nature, vol. 414, December 2001, pp. 883–887, DOI 10.1038/414883a

[29] European Telecommunications Standards Institute, "Quantum Computing Impact on security of ICT Systems; Recommendations on Business Continuity and Algorithm Selections", ETSI EG 203 310, June 2016, https://www.etsi.org/deliver/etsi_eg/203300_203399/203310/01.01.01_60/eg_203310v010101p.pdf

[30] L. K. Grover, "A Fast Quantum Mechanical Algorithm for Database Search", STOC '96: 28th Annual ACM Symposium on Theory of Computing, Philadelphia (PA, USA), May 22-24, 1996, pp. 212–219, DOI 10.1145/237814.237866

[31] European Telecommunications Standards Institute, " Quantum-Safe threat assessment", ETSI GR QSC 004, March 2017, https://www.etsi.org/deliver/etsi_gr/qsc/001_099/004/01.01.01_60/gr_qsc004v010101p.pdf

[32] J. Proos and C. Zalka, “Shor’s discrete logarithm quantum algorithm for elliptic curves”, arXiv:quant-ph/0301141, January 2004, DOI 10.48550/arxiv.quant-ph/0301141

[33] A. Banerjee, T. K. Reddy, D. Schoinianakis, and T. Hollebeek, “Post-Quantum Cryptography for Engineers”, Internet-Draft, IETF Secretariat, August 2023. `https://datatracker.ietf.org/doc/draft-ar-pquip-pqc-engineers/03/`

[34] G. Brassard, P. Høyer, and A. Tapp, “Quantum cryptanalysis of hash and claw-free functions”, ACM SIGACT News, vol. 28, June 1997, pp. 14–19, DOI 10.1145/261342.261346

[35] A. K. Lenstra, “Birthday paradox”, Encyclopedia of Cryptography and Security (H. C. A. van Tilborg and S. Jajodia, eds.), pp. 147–148, Springer, 2011, DOI 10.1007/978-1-4419-5906-5_440

[36] D. Coppersmith, “Another birthday attack”, CRYPTO ’85: Advances in Cryptology, Santa Barbara (CA, USA), August 18-22, 1985, pp. 14–17, DOI 10.1007/3-540-39799-X_2

[37] D. Ott, C. Peikert, and et al., “Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility”, arXiv:1909.07353, September 2019, DOI 10.48550/arXiv.1909.07353

[38] The Liboqs Library, `https://openquantumsafe.org/liboqs/`

[39] N. Alnahawi, N. Schmitt, A. Wiesmaier, A. Heinemann, and T. Grasmeyer, “On the state of crypto-agility”, Cryptology ePrint Archive, Paper 2023/487, 2023, `https://eprint.iacr.org/2023/487`

[40] D. Florence, “Terminology for Post-Quantum Traditional Hybrid Schemes”, Internet-Draft, IETF Secretariat, October 2023. `https://datatracker.ietf.org/doc/draft-ietf-pquip-pqt-hybrid-terminology/01/`

[41] D. Micciancio and O. Regev, “Lattice-based Cryptography”, Post-Quantum Cryptography (D. J. Bernstein, J. Buchmann, and E. Dahmen, eds.), pp. 147–191, Springer, 2009, DOI 10.1007/978-3-540-88702-7_5

[42] R. Overbeck and N. Sendrier, “Code-based Cryptography”, Post-Quantum Cryptography (D. J. Bernstein, J. Buchmann, and E. Dahmen, eds.), pp. 95–145, Springer, 2009, DOI 10.1007/978-3-540-88702-7_4

[43] J. Ding and B.-Y. Yang, “Multivariate Public Key Cryptography”, Post-Quantum Cryptography (D. J. Bernstein, J. Buchmann, and E. Dahmen, eds.), pp. 193–241, Springer, 2009, DOI 10.1007/978-3-540-88702-7_6

[44] C. Dods, N. P. Smart, and M. Stam, “Hash based digital signature schemes”, IMA ’05: 10th International Conf. on Cryptography and Coding, Cirencester (United Kingdom), December 19-21, 2005, pp. 96–115, DOI 10.1007/11586821_8

[45] A. Huelsing, D. Butin, S. Gazdag, J. Rijneveld, and A. Mohaisen, “XMSS: eXtended Merkle Signature Scheme”, RFC-5246, May 2018, DOI 10.17487/rfc8391

[46] P.-A. Fouque, J. Hoffstein, P. Kirchner, V. Lyubashevsky, T. Pornin, T. Prest, T. Ricosset, G. Seiler, W. Whyte, and Z. Zhang, “Falcon: Fast-Fourier Lattice-based Compact Signatures over NTRU”, October, 2020, `https://falcon-sign.info/falcon.pdf`

[47] National Institute of Standards and Technology, “SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions”, NIST FIPS.202, August 2015, DOI 10.6028/NIST.FIPS.202

[48] V. Donnini, “Integration of the DICE specification into the Keystone framework”, 2023, `https://webthesis.biblio.polito.it/secure/27657/1/tesi.pdf`

[49] C. Paar and J. Pelzl, “Understanding Cryptography: A Textbook for Students and Practitioners”, Springer Berlin Heidelberg, 2010, ISBN: 9783642041013, DOI 10.1007/978-3-642-04101-3

[50] IETF-Hackathon, “Object Identifiers for PQC and Composite”, https://github.com/IETF-Hackathon/pqc-certificates/blob/master/docs/oid_mapping.md

Appendix A

User manual

A.1 System setup

Below are listed all the operations needed to correctly setup and configure the system.

Dependencies installation

The following commands are used to install on the Ubuntu distribution the dependencies needed by the project:

```
sudo apt update;
sudo apt install autoconf automake autotools-dev bc bison
    build-essential curl expat jq libexpat1-dev flex gawk gcc git
    gperf libgmp-dev libmpc-dev libmpfr-dev libtool texinfo tmux
    patchutils zlib1g-dev wget bzip2 patch vim-common lbzip2 python3
    pkg-config libglib2.0-dev libpixman-1-dev libssl-dev screen
    device-tree-compiler expect makeself unzip cpio rsync cmake
    ninja-build p7zip-full;
```

Setup repository

Clone the github repository by means of the following command:

```
git clone --recurse-submodules
    https://github.com/CarusoGiuseppe/pq-keystone.git;
```

It allows to clone the repository containing the developed solution as well as the submodules, since Keystone uses Make and Buildroot to build the system's components and cloning the submodules allows to manage dependencies to the Buildroot system.

Build system's components

Due to the fact that Make is used as build system with Buildroot, in the root directory is located the Makefile responsible for the system's building. It provides an easy and flexible way to build the whole system, collecting the options needed by the Buildroot system and initiates the building which takes place in Buildroot. Hence, to build the system, the following command must be used in the root directory:

```
make -j$(nproc)
```

Options that can be passed to the make command aforementioned are listed below, with the related usage:

- `KEYSTONE_PLATFORM=<platform>`: configures the platform where Keystone is deployed. At the time of writing only the `generic` platform is supported, related to the QEMU virtual environment;
- `KEYSTONE_BITS=<bits_num>`: configures the `bits_num`-bit system. Allowed parameters are `64` and `32`;
- `BUILDROOT_CONFIGFILE=qemu_riscv$(KEYSTONE_BITS)_virt_defconfig`: Buildroot configuration file to be used;
- `BUILDROOT_TARGET=<target>-dirclean`: whenever a component is modified, changed components are detected by the Buildroot system, which keeps synchronization among the various components. This option allows to remove the stale source directory detected, which can be followed by the classic build command `make -j$(nproc)`.

Configure Buildroot and Linux

It is possible to configure Buildroot and Linux using the following commands:

```
make buildroot-configure

make linux-configure
```

This allows to configure the two components by means of a menu-based interface.

A.2 System deployment

A.2.1 Functional tests

Once completed the build of all the components, the system can be deployed. To run the QEMU emulation the following command must be used in the root directory:

```
make run
```

This will start the QEMU virtualization starting from the RoT which will run the SM that in turn boots Linux.

Following are listed the credentials to login into the system:

- username: `root`;
- password: `sifive`.

To run the modified enclaves, the Keystone driver must be inserted with the following:

```
modprobe keystone-driver
```

The modified enclaves are located under the directory `/usr/share/keystone/examples/`. The following executables are available:

- `attestor.ke`: performs the modified remote attestation procedure executing the verifier, the host, and the enclave processes;
- `hello.ke`: executables provided by the original Keystone project where an enclave is executed and asks the host the print the `Hello, world!` string on `stdout`;
- `hello-native.ke`: executable analogous in functionality to the previous one but running without relying on the `libc` library;
- `tests.ke`: executable running multiple enclaves consecutively, performing various functionalities, among which the modified sealing-key derivation.

A.2.2 Performance test

To perform the performance test and evaluate the performances related to the two versions of Keystone compared, is necessary to perform the following actions. One cloned the repository containing the implemented project described in this document, build, and run the the `PQ DICE-based Keystone` version.

Then run the following commands to clone and build the repository containing the TORSEC developed `DICE-based Keystone` version:

```
git clone https://github.com/valerio1805/keystone.git;
cd keystone;
./fast-setup.sh;
source source.sh;
mkdir build;
cd build;
cmake ..;
make buildroot KBUILD\_MODPOST\_WARN=1;
make qemu;
make linux KBUILD\_MODPOST\_WARN=1;
make sm;
make bootrom;
make driver;
make image;
make hello-package;
cp ./examples/hello/hello.ke ./overlay/root;
make image;
./scripts/run-qemu.sh;
```

after this process, the DICE-based Keystone version will be successfully built and will be running. Once completed the system boot, login with the credentials username:`root` password:`sifive` and run the following commands to execute the executable `hello.ke`;

```
insmod keystone-driver.ko
./hello.ke
```

In another terminal, run the `make run` command within the `pq-keystone/` directory and follow the previous instructions to run the modified version. The programs will be run and will print the number of ticks necessary to perform the various operations described in the performance test section 7.3, which can be compared with the version designed in this document.

A.2.3 Post-Quantum algorithm performance evaluation

To evaluate the performances of the PQ algorithms as described in the section 7.4, the following operations allow to make available the executable in the Linux environment:

```
cp ./test_min.o ./build-generic<KEYSTONE_BITS>/overlay/root;
cp ./test_max.o ./build-generic<KEYSTONE_BITS>/overlay/root;
make -j$(nproc);
make run;
```

There is no necessity to insert the Keystone driver to perform the following tests. At this point the executables are available to be run with the following command:

```
chmod 777 ./test_min.o;
chmod 777 ./test_max.o;
./test_min.o; ./test_max.o
```

Appendix B

Developer manual

B.1 Falcon usage

Below are listed information useful to configure and to apply correctly modifications to the Falcon implementation.

B.1.1 Configuration file

The reference configuration file is `config.h`, which contains all the configuration options defined as macros. Each option can be enabled by setting the macro's value to 1. Commenting or setting an option value to 0 disables the option.

- `FALCON_FPNATIVE`: Use the `double` C type for floating-point computations, using the CPU hardware and/or compiler-provided function; the first may be constant-time while the second is typically not constant-time. If neither `FALCON_FPNATIVE`, nor `FALCON_FPEMU` are enabled, the default behaviour is enabling the `FALCON_FPNATIVE` macro.
- `FALCON_FPEMU`: use floating-point emulation provided in the reference implementation.
- `FALCON_ASM_CORTEXM4`: Enable use of assembly for ARM Cortex-M4 CPU; Emulated FP operations with ARM assembly are constant-time and will be used unless other options override this choice.
- `FALCON_AVX2`: use AVX2 for better performance if available as ISA.
- `FALCON_FMA`: ISA used only if `FALCON_AVX2` is enabled;
- `FALCON_LE`: assert that the platform uses little-endian encoding, providing slightly better performance; if not enabled autodetection is applied.
- `FALCON_UNALIGNED`: assert that the platform tolerates accesses to unaligned multi byte values; if enabled, then some operations are slightly faster, otherwise autodetection is applied.
- `FALCON_KG_CHACHA20`: uses `ChaCha20` seeded with `SHAKE256` for the keypair generation; this provides, according to the platform, a speed-up in the computation.

- FALCON_RAND_GETENTROPY: targets the platform where the getentropy() system call is available to use as explicit OS-provided RNG;
- FALCON_RAND_URANDOM: targets the platform where the /dev/urandom special file is available to use as explicit OS-provided RNG;
- FALCON_RAND_WIN32: targets the platform where the CryptGenRandom() function call is available to use as explicit OS-provided RNG.

Among the OS-provided RNG, in small embedded-systems none will be available and the entropy source needs to be provided somehow. In the falcon.h file is defined the LOGN_PARAM used to set the Falcon degree used within the system.

B.1.2 Falcon usage in bootrom

The code listed in this section has been implemented in the bootrom.c file, removing the quantum-vulnerable operations, and replaced with the quantum safe solution.

In particular are shown

- the SM PQ signature verification for the secure boot functionality, shown in Lst. B.1;

Listing B.1. SM PQ signature verification

```
...
// Measure the SM to simulate manufacturer provided signature
  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, (void*)DRAM_BASE, sanctum_sm_size);
  sha3_final(sanctum_sm_hash, &hash_ctx);
  falcon_sign_dyn(&rng, sanctum_sm_sign, &sig_len, FALCON_SIG_CT,
      _sanctum_dev_secret_key, FALCON_SK_SIZE, sanctum_sm_hash,
      64, tmp_sig, falcon_tmpsign_size_test);

  //verify the signature
  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, (void *)DRAM_BASE, sanctum_sm_size);
  sha3_final(sanctum_sm_hash, &hash_ctx);
  if((falcon_verify(sanctum_sm_sign, sig_len, FALCON_SIG_CT,
      _sanctum_dev_public_key, FALCON_PK_SIZE, sanctum_sm_hash,
      64, tmp_vrfy, falcon_tmpvrfy_size_test)) != 0)
  {
    // The return value of the bootloader function is used to
        check if the secure boot is gone well or not
    return 0;
  }
...
```

- the ECA and the DRK keypairs generation in the Lst. B.2.

Listing B.2. ECA and the DRK PQ keypairs generation

```
// Combine the SM hash and the UDS to obtain the CDI
sha3_init(&hash_ctx, 64);
sha3_update(&hash_ctx, sanctum_uds, sizeof(*sanctum_uds));
sha3_update(&hash_ctx, sanctum_sm_hash,
   sizeof(*sanctum_sm_hash));
sha3_final(sanctum_CDI, &hash_ctx);

// The DRK are created from the CDI
shake256_init_prng_from_seed(&rng, sanctum_CDI,
   sizeof(*sanctum_CDI));

//generate device root key from CDI
falcon_keygen_make(&rng, LOGN_PARAM,
   sanctum_device_root_key_priv, FALCON_SK_SIZE,
   sanctum_device_root_key_pub,
   FALCON_PK_SIZE,tmp,falcon_tmpkeygen_size_test);

// The ECA keys seed is generated hashing the CDI and the SM
   measure
unsigned char seed_for_ECA_keys[64];

sha3_init(&hash_ctx, 64);
sha3_update(&hash_ctx, sanctum_CDI, 64);
sha3_update(&hash_ctx, sanctum_sm_hash, 64);
sha3_final(seed_for_ECA_keys, &hash_ctx);

//rng for the ECA keys
shake256_init_prng_from_seed(&rng, seed_for_ECA_keys, 64);

falcon_keygen_make(&rng, LOGN_PARAM, ECASM_priv,
   FALCON_SK_SIZE, ECASM_pk, FALCON_PK_SIZE, tmp,
   falcon_tmpkeygen_size_test);
```

- the SM PQ keypair and SM PQ signature generation in the Lst. B.3.

Listing B.3. SM PQ keypair and SM PQ signature generation

```
byte scratch[64 + FALCON_PK_SIZE];
sha3_init(&hash_ctx, 64);
sha3_update(&hash_ctx, _sanctum_dev_secret_key, FALCON_PK_SIZE);
sha3_update(&hash_ctx, sanctum_sm_hash,
   sizeof(*sanctum_sm_hash));
sha3_final(seed_for_SM, &hash_ctx);

shake256_init_prng_from_seed(&rng, seed_for_SM, 64);
falcon_keygen_make(&rng, LOGN_PARAM, sanctum_sm_secret_key,
   FALCON_SK_SIZE, sanctum_sm_public_key, FALCON_PK_SIZE, tmp,
   falcon_tmpkeygen_size_test);
```

```
  // Endorse the SM
  memcpy(scratch, sanctum_sm_hash, 64);
  memcpy(scratch + 64, sanctum_sm_public_key, FALCON_PK_SIZE);

  // Sign (H_SM, PK_SM) with SK_D
  falcon_sign_dyn(&rng, sanctum_sm_signature, &sig_len,
     FALCON_SIG_CT, _sanctum_dev_secret_key, FALCON_SK_SIZE,
     scratch, 64 + FALCON_PK_SIZE, tmp_sig,
     falcon_tmpsign_size_test);
```

B.1.3 Falcon usage in sm

In the code listed in this section are shown the major modifications applied to the SM in the `sm.c` and `crypto.c` files. In particular are shown:

- the certificates' signature verification performed during the SM initialization (e.g. Lst. B.4);

Listing B.4. Certificates' signatures verification during the SM initialization

```
...
if((falcon_verify(uff_cert_sm.sig.p, uff_cert_sm.sig.len,
    FALCON_SIG_CT, uff_cert_root.pk.pk_ctx.pub_key,
    FALCON_PUBKEY_SIZE(LOGN_PARAM), hash_for_verification, 64,
    tmp, falcon_tmpvrfy_size_test)) != 0){
    sbi_printf("[SM] Error verifying the PQ signature of the ECA
       certificate\n\n");
  }
...
    if(falcon_verify(uff_cert_root.sig.p, uff_cert_root.sig.len,
       FALCON_SIG_CT, uff_cert_man.pk.pk_ctx.pub_key,
       FALCON_PUBKEY_SIZE(LOGN_PARAM), hash_for_verification, 64,
       tmp, falcon_tmpvrfy_size_test) != 0){
      sbi_printf("[SM] Error verifying the PQ signature of the
         DRK certificate\n\n");
      sbi_hart_hang();
    }
...
```

- the modifications applied to the `sm_sign` function (e.g. Lst B.5);

Listing B.5. SM signature computation function's modifications applied.

```
void sm_sign(void* signature, const void* data, size_t len)
{
  shake256_init_prng_from_seed(&rng, seed, sizeof(*seed));
  sign(signature, data, len, sm_private_key, tmp, &rng);
}
...
void sign(void* sign, const void* data, size_t len,const unsigned
    char* private_key, unsigned char* tmp, shake256_context *rng)
{
  size_t sig_len;
  falcon_sign_dyn(rng, sign, &sig_len, FALCON_SIG_CT,
      private_key, FALCON_PRIVKEY_SIZE(LOGN_PARAM), data, len,
      tmp, FALCON_TMPSIZE_SIGNDYN(LOGN_PARAM));
}
...
```

B.1.4 Falcon usage in enclave

Here are listed the major code changes to correctly use Falcon in the following cases, affecting the files `enclave.c`, `enclave.h`, and `Report.cpp`:

- Local Attestation Keys creation and `enclave` struct changes (e.g. Lst. B.6);

Listing B.6. Modifications applied to the enclave struct and to the Local Attestation Keys generation

```
struct enclave
{
  byte sign[FALCON_SIG_SIZE];
  byte local_att_pub[FALCON_PK_SIZE];
  byte local_att_priv[FALCON_SK_SIZE];
...
};
...
    sha3_init(&hash_ctx_to_use, 64);
    sha3_update(&hash_ctx_to_use, CDI, 64);
    sha3_update(&hash_ctx_to_use, enclaves[eid].hash, 64);
    sha3_final(enclaves[eid].CDI, &hash_ctx_to_use);

    shake256_init_prng_from_seed(&rng, enclaves[eid].CDI, 64);

    if(falcon_keygen_make(&rng, LOGN_PARAM,
        enclaves[eid].local_att_priv,
        FALCON_PRIVKEY_SIZE(LOGN_PARAM),
        enclaves[eid].local_att_pub,
        FALCON_PUBKEY_SIZE(LOGN_PARAM), tmp,
        FALCON_TMPSIZE_KEYGEN(LOGN_PARAM)) != 0)
      {
        sbi_printf("\n[SM] Error during PQ keypair generation\n");
        goto unlock;
      }
...
```

- SDK modifications (e.g. Lst. B.7) to correctly provide the Falcon utilities to check the Falcon signatures computed and used in the attestation evidence.

Listing B.7. SDK modifications applied to correctly implement the remote attestation with PQ parameters.

```
...
struct enclave_report
{
  byte hash[MDSIZE];
  uint64_t data_len;
  byte data[ATTEST_DATA_MAXLEN];
  byte signature[FALCON_SIG_SIZE];
};
struct sm_report
{
  byte hash[MDSIZE];
  byte public_key[FALCON_PK_SIZE];
  byte signature[FALCON_SIG_SIZE];
};
struct report
{
  struct enclave_report enclave;
  struct sm_report sm;
  byte dev_public_key[FALCON_PK_SIZE];
};
...
 //modifications applied into the sdk/src/verifier/Report.cpp file
  /* verify SM report */
  sm_valid = falcon_verify(
      report.sm.signature, FALCON_512_SIG_SIZE, FALCON_SIG_CT,
          dev_public_key, FALCON_512_PK_SIZE,
          reinterpret_cast<byte*>(&report.sm), MDSIZE +
          FALCON_512_PK_SIZE,
      tmp, FALCON_TMPSIZE_VERIFY(LOGN_PARAM));
  /* verify Enclave report */
  enclave_valid = falcon_verify(
      report.enclave.signature, FALCON_512_SIG_SIZE,
          FALCON_SIG_CT, report.sm.public_key, FALCON_512_PK_SIZE,
          reinterpret_cast<byte*>(&report.enclave), MDSIZE +
          sizeof(uint64_t) + report.enclave.data_len,
      tmp, FALCON_TMPSIZE_VERIFY(LOGN_PARAM));
...
```

B.2 Apply patches

If there is the necessity to modify source files in the `opensbi` component such as the Linker Script file (e.g. with `.ldS` extension) used to share variables among the components of the system, such files cannot be modified directly. It is compulsory to create and apply patches. Following are listed the operations needed to apply such patches:

- identify the files to modify which will be located at `<path_to_file>/file`;
- from the parent directory run the following commands to create the related patch:

```
cp file file_mod;
//apply necessary modifications to the newly created file...
cd ..;
diff -Naur <path_to_file>/file <path_to_file>/file_mod >
   file.patch;
```

- modify the file so that the first lines result as follows:

```
--- a/<path_to_file>/file
+++ b/ <path_to_file>/file
```

- run the following commands to apply the patch:

```
cp file.patch <KEYSTONE_HOME_DIR>/overlays/patches/opensbi
BUILDROOT_TARGET=opensbi-dirclean make;
make -j$(nproc);
```

At the end of this process the patch is applied.

Index