POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

Self-Sovereign-Identity as a Service

Trusted computation offloading for IoT constrained devices

Supervisor
prof. Antonio Lioy

Candidate
Luca Giorgino

Tutors
LINKS FOUNDATION
Andrea Vesco, Ph.D.
Alberto Carelli, Ph.D.

Academic year 2021-2022

Summary

Digital identity employed on IoT devices makes them unique and distinguishable from each other. Self-Sovereign Identity (SSI) is a paradigm that aims to provide a digital identity that is both verified and verifiable to IoT objects while building a digital ecosystem for secure interactions between heterogeneous devices. However, in many real-world use cases, IoT devices cannot run natively a full self-sovereign identity stack implementation, due to hardware and software constraints. For this reason, an edge device has been designed with the capability of securely aiding constrained devices to create and manage their own identity according to the SSI paradigm. The software has been developed using Keystone, an open-source framework for building Trusted Execution Environments, for establishing a trusted communication channel between the IoT device and the edge device that handles offloaded operations. By defining a new paradigm called Self-Sovereign Identity as a Service, constrained devices can exploit the full SSI stack on demand. Such a solution has the advantage to increase the number of devices that can interact in a secure digital ecosystem of this kind by shifting the computational operations onto more powerful edge devices.

Acknowledgements

I would like to thank Professor Antonio Lioy for his supervision. I sincerely thank Ing. Andrea Vesco, Ing. Alberto Carelli and Ing. Davide Margaria, the Cybersecurity team of LINKS Foundation, for their precious advice, continuous support and provided knowledge during the realization of this work.

I thank my family, who sustained me throughout my university career. In particular, I thank my father and my mother. Without them, this day would not have been possible. Finally, I would like to extend a special thanks to Elisabetta, who has always shown me her encouragement and love during these years of study.

Contents

Chapter 1

Introduction

The concept of digital identity has been evolving in the last decades. Digital identity is the expression and storage of one's identity in digital form, which is a set of claims, i.e. assertions of some truths, made about a subject, which can be a person, a thing, a device, etc. Despite this, only in recent years privacy, control and ownership over digital identity and personal data are being increasingly recognized as relevant factors by individuals [1]. Since the advent of the Internet, digital identity models have gone through several stages, from centralized identity to federated identity, becoming more user-centric over time reaching the definition of the Self-Sovereign Identity (SSI) paradigm [2]. In the Internet's early days, digital identity issuers and authenticators were designed as centralized authorities. Unfortunately, giving centralized authorities control over a user's digital identity has several drawbacks. For example, a single authority can deny a user's identity or even confirm a false one, so the user has no control over his own identity. SSI is the next development of digital identity where the main idea is that the user must be at the centre of identity management [2].

The Internet of Things increasingly involves the collection, processing and transmission of a wide variety of data to services and other devices [3]. Reasonably obvious privacy risks arise from IoT-connected devices when they exchange identifiable information, as this can reveal the activities and behaviours of users' devices and subtle risks arise when a considerable amount of data is available for analysis and linkage to additional data sets, since user identification or re-identification may happen as a result [3]. Digital identity, such as SSI, could be a solution for building a digital realm where heterogeneous objects and people interact securely. This will allow IoT devices to be unique and distinguishable from one another.

In this document, Self-Sovereign Identity is introduced, as well as Keystone enclave, an open-source framework for trusted execution environments based on hardware enclaves. Then is presented the solution, which has been designed to support a constrained device to create and manage its own digital identity.

Chapter 2

Self-Sovereign Identity

Self-Sovereign Identity (SSI) [4] is a new model for digital identity. In the SSI ecosystem, a user can fully control his own identity and can use it between any service. SSI is different from today's digital identities: it is anchored to distributed ledgers so is not controlled by any centralized services. One SSI innovation is the design and development of a common set of specifications: Decentralized Identifiers (DIDs) [5] and Verifiable Credentials (VCs) [6]. Using these specifications, user identity can be anchored to different distributed ledgers, but it will also be defined in the same standard way.

2.1 Decentralized Identifiers

DIDs [5] are identifiers referring to any subject determined by the controller of the DID. A DID's controller can demonstrate control over a DID by design allowing a verifiable, decentralized digital identity, which will be independent of any identity providers, and certification authorities.

Overview

A DID is a type of URI [7] scheme that links a DID subject with a DID document allowing trustable interactions associated with that subject. The subject of a DID is the entity identified by the DID and can be a person, a group or an organization, a device, etc. Typically the DID subject is also the controller, but a DID can have more than one controller.

Specifically, a DID is a simple text string, as shown in the figure 2.1, consisting of three parts:

- the DID URI scheme identifier
- the identifier for the DID method
- the DID method-specific identifier

Figure 2.1. A simple example of a DID [5].

A DID document contains information about a DID subject and cryptographic material that will be used to prove control of that DID. DID documents can be represented in JSON [8] or JSON-LD [9] format, as specified in W3C specification [5]. An example of DID document can be seen in Fig. 2.2. Only the controller of the DID has the right to make changes to the related DID document.

DIDs are generally stored in some underlying system or network for the resolution to DID documents. A verifiable data registry is a system that enables the recording of DIDs and the return of the data required to produce DID documents. Distributed ledgers, decentralized file systems, all types of databases, peer-to-peer networks, and other trusted data storage methods are some examples. The operations to create, resolve, update, and deactivate a DID and the related DID document are defined by DID methods and their specifications, which are generally coupled with a distinct verifiable data registry [5]. A DID *resolves* to a DID document by using the *read* operation of the applicable DID method [5]. A DID URL expands the syntax of a DID with other standard components of URI such as path, query, and fragment to find a specific resource, such as a cryptographic public key inside a DID document or a resource outside the DID document.

```
{
    "@context": [
        "https://www.w3.org/ns/did/v1",
        "https://w3id.org/security/suites/ed25519-2020/v1"
    ]
    "id": "did:example:123456789abcdefghi",
    "authentication": [{

        "id": "did:example:123456789abcdefghi#keys-1",
        "type": "Ed25519VerificationKey2020",
        "controller": "did:example:123456789abcdefghi",
        "publicKeyMultibase":
            "zH3C2AVvLMv6gmMNam3uVAjZpfkcJCwDwnZn6z3wXmqPV"
    }]
}
```

Figure 2.2. Example of a simple DID document from [5].

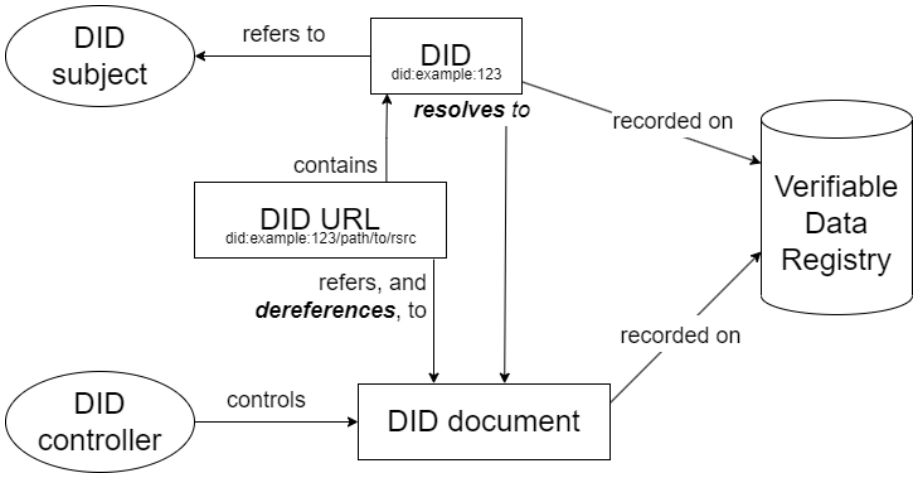

Figure 2.3. DID architecture overview and relationships between components [5].

2.2 Verifiable Credentials

Verifiable Credential [6] provides a standard method to express credentials on the internet in a way that is cryptographically safe, privacy-respecting, and machine-verifiable. Outside the technological domain, a credential could consist of:

- information related to identifying the subject of the credential (for example, a photo, name, or identification number)
- information related to the issuing authority (for example, a city government or a university)
- information related to the type of credential this is (for example, a passport or a driving license)
- information related to specific attributes or properties being asserted by the issuing authority about the subject (for example, nationality, the classes of vehicle entitled to drive, or date of birth)
- information related to constraints on the credential (such as expiration date, or terms of use).

In verifiable credentials, the additional inclusion of digital signatures makes them more trustworthy and more tamper-evident compared to physical credentials. These allow third-party verified machine-readable personal information usable on the Web for receiving services and benefits as in the physical world [6].

Overview

Distinct actors can be identified in the verifiable credentials ecosystem, which defines the roles and the relationships between them. The separation of roles allows the standardization of interfaces and protocols. In detail the existing entities that determine the so-called *trust triangle* [10] are:

- *holder*: his role is to request, possess or use verifiable credentials. Example holders include students, employees, and customers.
- *issuer*: his role is to create a verifiable credential and provide that to a holder by asserting claims about one or more subjects. For example, an issuer is a government.
- *verifier*: his role is to process verifiable credentials provided by holders. Example verifiers are whoever provides a service.

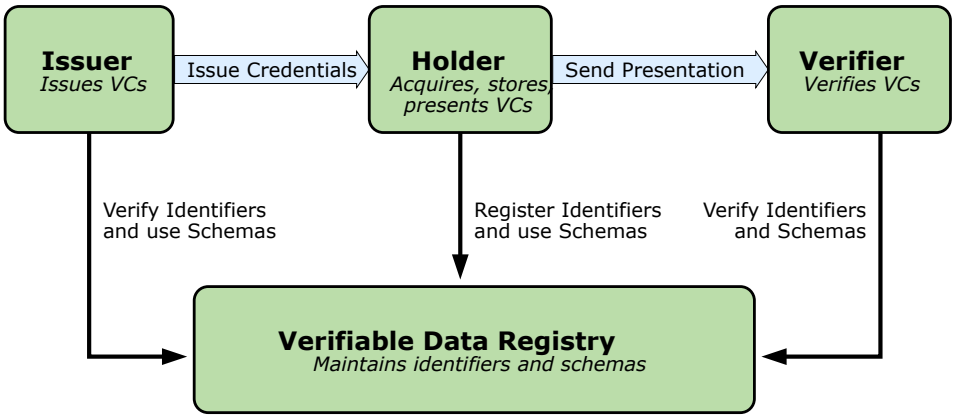

Figure 2.4. The roles and information flows of Verifiable Credential [6].

To use verifiable credentials with a specific verifier, a holder has to generate verifiable presentations. A verifiable presentation encloses one or more verifiable credentials, issued by one or more issuers. Before presenting the verifiable presentation to a verifier, a holder signs the verifiable presentation content, proving authorship of the data and possession of verifiable credentials. However, The fact that a credential is verifiable by the verifier does not mean that the claims it contains are true. Another aspect that verifiable credentials enhance is privacy. The usage of machine-readable credentials allows malicious actors to collect and correlate data, compromising holders' privacy. Verifiable credentials specification draft how to deal with these issues, by using privacy-enhancing technologies, such as *zero-knowledge proof* [11].

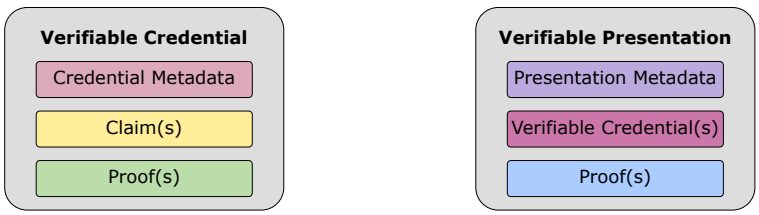

Figure 2.5. Basic components of a verifiable credential and a verifiable presentation [6].

Verifiable credentials and verifiable presentations can be represented in JSON [8] or JSON-LD [9] format, as specified in W3C specification [6]. Examples can be seen in Figure 2.6 and Figure 2.7.

```
{ "@context": ["https://www.w3.org/2018/credentials/v1"],
    "id": "http://example.edu/credentials/1872",
    "type": ["VerifiableCredential", "AlumniCredential"],
    "issuer": "https://example.edu/issuers/565049",
    "issuanceDate": "2010-01-01T19:23:24Z",
    "credentialSubject": {
        "id": "did:example:ebfeb1f712ebc6f1c276e12ec21",
        "alumniOf": {
            "id": "did:example:c276e12ec21ebfeb1f712ebc6f1",
            "name": [{
                "value": "Example University",
                "lang": "en"
            }]
        }
    },
    "proof": {
        "type": "RsaSignature2018",
        "created": "2017-06-18T21:19:10Z",
        "proofPurpose": "assertionMethod",
        "verificationMethod":
           "https://example.edu/issuers/565049#key-1",
        "proofValue": "eyJhbGciOiJSUzI1NiIsImI2NCI6ZmFsc2UsI..."
    }
}
```

Figure 2.6. A simple example of a verifiable credential [6].

```
{ "@context": ["https://www.w3.org/2018/credentials/v1"],
    "type": "VerifiablePresentation",
    "verifiableCredential": [{
        ...
    }],
    "proof": {
        "type": "RsaSignature2018",
        "created": "2018-09-14T21:19:10Z",
        "proofPurpose": "authentication",
        "verificationMethod":
           "did:example:ebfeb1f712ebc6f1c276e12ec21#keys-1",
        "challenge": "1f44d55f-f161-4938-a659-f8026467f126",
        "domain": "4jt78h47fh47",
        "proofValue": "Qy72IFLN25DYuNzVBAh4vGHSrQyHUGlc..."
    }
}
```

Figure 2.7. A simple example of a verifiable presentation [6].

2.3 Distributed Ledgers Technologies

Distributed Ledger Technology (DLT) is a new paradigm for collecting and sharing information between people. A distributed ledger is a database that is spread across several nodes or computing devices in a network. An exact copy of the ledger is replicated and stored on each node. The revolutionary aspect of distributed ledger technology is that no single administrator or central authority is responsible for maintaining the ledger. Each network's participant node keeps its state updated by constructing and recording updates to the ledger independently. The nodes then vote on these adjustments to ensure that the majority agrees with the conclusion reached. This voting and agreement on the state of the ledger is called consensus and is conducted automatically via a consensus algorithm [12]. The following criteria can be used to categorize DLTs: data structures, consensus algorithms, permissions, mining accessibility and so on. The main data structure types are blockchains and Directed Acyclic Graphs (DAGs). Although blockchains are the most popular and well-known DLT type, DLTs based on Directed Acyclic Graph (DAG) data structures are becoming more popular because they reduce transaction data size and transaction fees, and increase transaction speeds [13].

An Example of DAG DLT is *The Tangle* of IOTA [14], a cryptocurrency for the IoT industry.

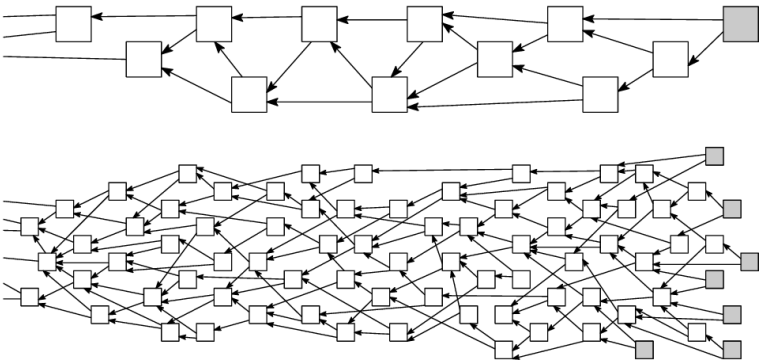

Figure 2.8. Visualizations of the IOTA tangle [14].

Chapter 3

Keystone Enclave

As a result of the increasing popularity of networked devices in recent years, device manufacturers are now taking security concerns more seriously than they previously did [15]. To adequately address security challenges, i.e. integrity and confidentiality properties of sensitive data when using connected devices, Trusted Execution Environments (TEEs) have been defined. Keystone [16] is an open-source framework for creating TEEs that are adaptable for use on a variety of platforms and are based on RISC-V hardware.

3.1 Trusted Execution Environment

A Trusted Execution Environment (TEE) is an execution environment that runs alongside but is isolated from the device's main operating system. It ensures that the confidentiality and integrity of the code and data loaded in the TEE are preserved [17]. Trusted applications running on TEE have access to the full capabilities of a device's main processor and memory, while hardware isolation shields these components from user-installed apps running in the main operating system. The various included trusted applications are protected from one another by software and cryptographic isolations within the TEE [15]. The two most common TEE implementations at the moment are ARM TrustZone and Intel SGX. All these TEEs make design decisions based on either the target applications or threat models and these choices are fixed since they are strictly hardware related. They were not designed to have flexibility or extensibility for enclave developers. If the hardware changes or has a new feature, the enclave developer has to redesign the TEE. All TEE platforms aim to reduce the enclave's trusted computing base, and they have managed to achieve different degrees of success [18]. The Trusted Computing Base (TCB) is a section of the system, which could include hardware, firmware and software. It is responsible for enforcing the security policy of the system [19]. Additionally, closed-source hardware and microcode implementations make it impossible for a third party to evaluate the security of TEEs.

3.1.1 Customizable Trusted Execution Environment

Customizable TEE is the solution to closed-source hardware-implemented TEEs problems. It has been designed to be flexible, and configurable and to have a small TCB. It has been designed with clear abstractions and a modular programming model which simplifies for others to extend and add features to the TEE. An example of a customizable TEE is Keystone [16]. Three logical actors, such as the manufacturer (who makes the hardware), the platform provider (runs the hardware, such as a cloud provider), and the enclave developer (who writes software that runs in the enclaves), were identified by Keystone developers as being a part of the customizable TEE ecosystem. In a customizable TEE, as opposed to a standard TEE, decisions made by all 3 actors together determine the security guarantees offered and the functionalities enabled [18]. Keystone offers security primitives that can be joined together via the software framework rather than creating a single instance of TEE hardware. The TEE can be modified by the creator of the enclave and the platform provider to suit their threat models or platform configurations. The Keystone project offers a general and formally proven interface for a variety of devices to create an open standard for TEEs [16].

3.2 RISC-V Background

RISC-V [20] is open-source, which provides Keystone with several benefits. The most noticeable is that anyone can see how it works, understand the threat model it can operate under, and verify how exploits/patches function [21]. Other advantages of RISC-V are security-oriented primitives, which provide efficient isolation, the most notable being Physical Memory Protection (PMP) [22]. RISC-V is an evolving and community-driven Instruction Set Architecture (ISA). Keystone has been designed and developed using RISC-V standard security features. Moreover, the ever-growing world of RISC-V gives Keystone a wide variety of potential platforms and different deployment scenarios to which it can adapt to [21].

3.2.1 RISC-V Privilieged ISA

RISC-V [20] has three software privilege levels (in increasing order of capability): user mode (U-mode), supervisor mode (S-mode), and machine mode (M-mode). Only one of the privilege modes can be active on the processor at once. The active privilege level determines what the software can do while it is running. These are typical applications for each level of privilege:

- U-mode: user processes
- S-mode: kernel (including kernel modules and device drivers) or hypervisor
- M-mode: bootloader and firmware.

When the processor is in the highest privilege mode, M-mode, it is in control of all physical resources and interrupts. As with microcode in Complex Instruction Set

Computer (CISC) ISAs (such as x86), M-mode is not interruptible and not affected by the interference of lower modes. M-mode is used in Keystone for executing the TCB of the system, the *security monitor* (SM).

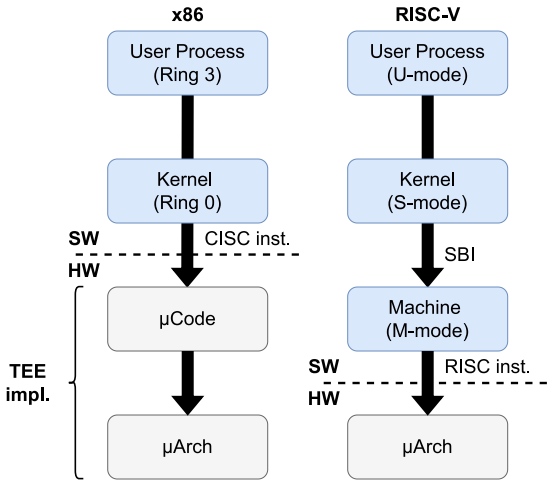

Figure 3.1. Architecture differences between x86 and Keystone

The following are some advantages of utilizing an M-mode software as the TCB:

- programmability: unlike microcode for x86, in RISC-V M-mode software can be written using pre-existing toolchains and programming languages, such as C
- agile Patching: since the TCB is purely software, bugs or vulnerabilities can be patched without updates, which are specific to a particular hardware
- verifiability: compared to hardware, the software is generally simpler to be formally verified.

3.2.2 Physical Memory Protection

Physical Memory Protection (PMP) [22] is a strong standard primitive that enables M-mode to control the access to physical memory from lower privileges modes. Keystone requires PMP to implement memory isolation of enclaves. Only software in M-mode can configure the PMP, which is controlled by a series of control and status registers (CSR) that limit physical memory access to the U-mode and S-mode. Depending on the platform design, PMP entries number can change.

Since PMP exclusively works on physical addresses, S-mode can continue to support virtual addresses without affecting the security of the system. Even though each processor may implement PMP differently in hardware, the basic guarantees are part of the standard. PMP is used by Keystone Security Monitor to create memory isolation.

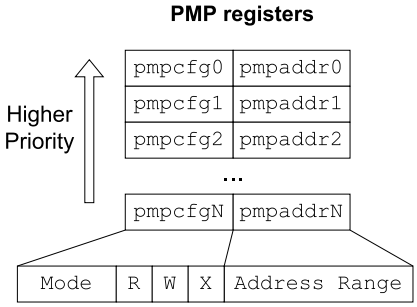

Figure 3.2. Image representing PMP registers [23].

3.3 Keystone components

A Keystone-capable system is made up of different modules operating in various privilege modes as shown in Fig. 3.3.

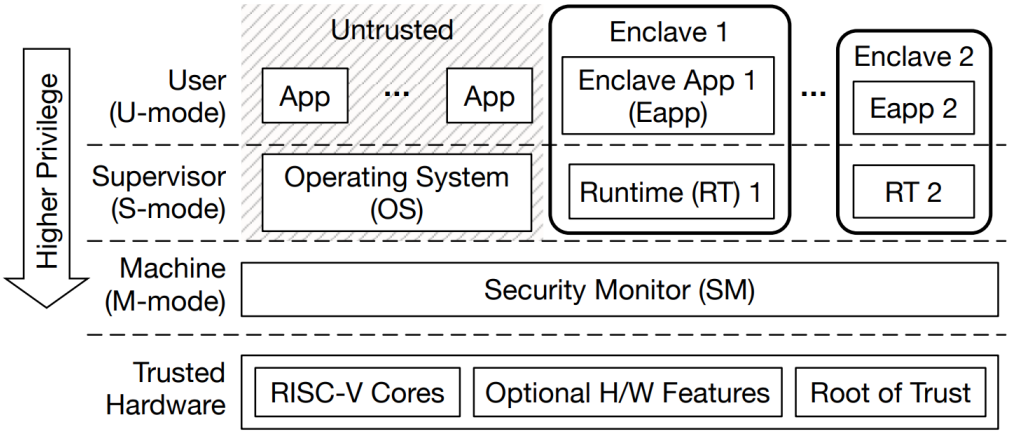

Figure 3.3. Keystone system with host processes, untrusted OS, security monitor, and multiple enclaves (each with runtime and eapp) [16].

Trusted Hardware

Trusted Hardware is a CPU package built by an honest manufacturer that must enclose standard RISC-V cores, which are Keystone compatible, and a root of trust. Optional features of the hardware could also include memory encryption, cache partitioning, a cryptographically safe source of randomness, etc. Platform-specific plug-ins are needed by the Security Monitor to support optional features.

3.3.1 Security Monitor

Security Monitor (SM) is a trusted software that runs in M-mode and works as the small trusted computing base (TCB) in the Keystone system. Before the SM can be considered trusted, it must be verified by the hardware root of trust. Then, the root of trust *measures* the SM, generates a keypair for remote attestation, signs the public key, and eventually can continue booting. The measurement of the SM consists in computing the hash of the SM firmware image. The SM manages isolation boundaries between the enclaves and the untrusted OS, therefore it implements the majority of Keystone's security guarantees. It serves as an interface for managing the enclave's lifecycle and utilising platform-specific features. The OS and enclaves may call SM functions using the Supervisor Binary Interface (SBI). Specifically, the SM provides the following functionality:

- *memory isolation* using RISC-V PMP
- *remote attestation* (signatures and measurement): the goal is to demonstrate to a remote client that the enclave contains the expected application, and is running on trusted hardware
- and other features, such as system PMP synchronization, enclave thread management and side-channel defences

3.3.2 Runtime

Keystone developers implemented the Runtime (RT) with the goal of minimal and flexible TCB. It is an S-mode software. As a result, enclave applications can use modular system-level abstraction (e.g., virtual memory management). It provides kernel-like functionality, such as system calls, trap handling, virtual memory management and so on. Although the RT functions similarly to a kernel inside an enclave, most kernel functionalities are not necessary for the enclave application. To allow enclave developers to include only the necessary functionality and minimize the TCB, Keystone developers created an example of RT called Eyrie. It enables reusability since it is compatible with multiple-user programs. And by adding RT modules, they expand RT functionality without changing user applications or without complicating the SM.

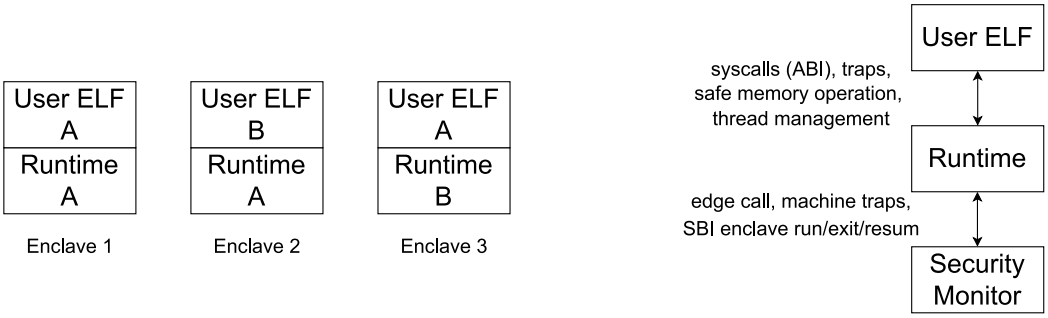

Figure 3.4. Example of runtime reusability on the left and its functionalities on the right [23].

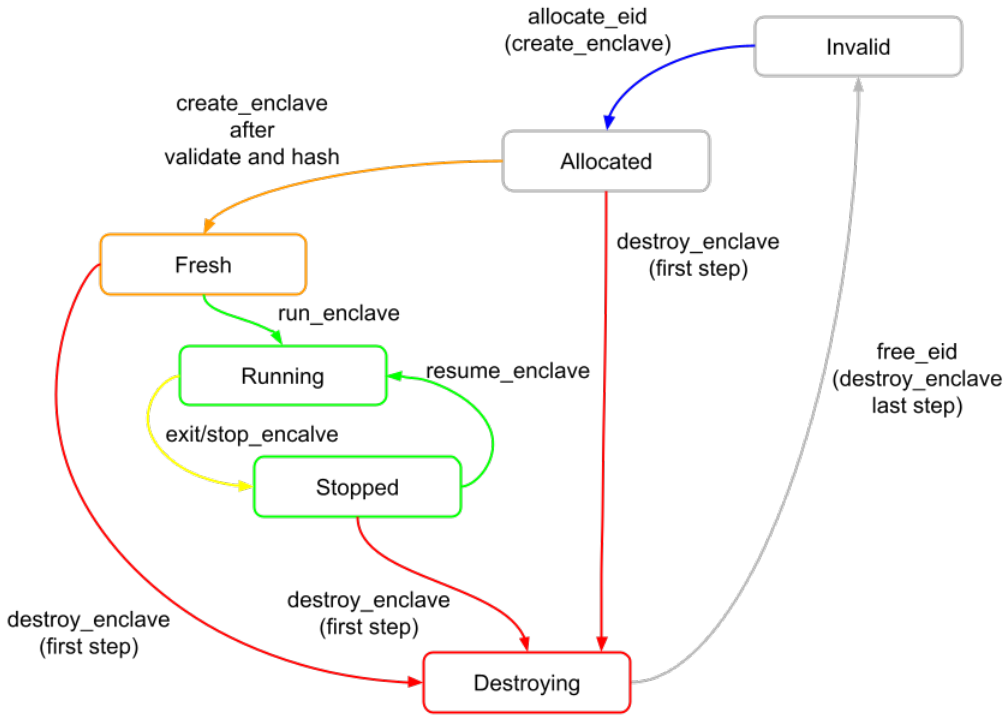

Figure 3.5. Enclave Lifecycle [23].

3.3.3 Enclave

An Enclave is an environment isolated from the untrusted OS and other enclaves. Each enclave is provided with a private physical memory region which is accessible by only the enclave and SM. Each enclave consists of a user-level enclave application called *eapp* and a supervisor-level runtime. An eapp is a user-level application that executes in the enclave. A developer can create a custom eapp from scratch, or just execute an existing RISC-V binary in Keystone. The enclave lifecycle can be seen in Fig. 3.5. The main phases are:

- *creation*: when an enclave is started it has a contiguous range of physical memory that is called Enclave Private Memory (EPM). In the beginning, the EPM is allocated by the untrusted host, which initialises it with the enclave's page table, the runtime and the enclave application. When the untrusted host calls the SM to create an enclave, the SM isolates and secures the EPM using a PMP entry, and then the PMP status is propagated throughout all of the system's cores. Subsequently, before the enclave execution, the enclave's initial state is measured and verified by the SM.
- *execution*: the SM enters the enclave on one of the cores as soon as the untrusted asks for it. The PMP permission is enabled to the core by the

SM, and the core starts running the eapp. The RT can exit or re-enter the enclave at any time depending on the execution flow of the eapp. The PMP permissions are switched to keep the isolation each time a core exits or enters the enclave.

- *destruction*: the untrusted host may want to destroy the enclave at any moment, when it happens, the EPM is cleared by the SM and the PMP entry is freed. The untrusted host then definitely reclaims the released memory.

3.3.4 Edge Calls

Function calls that enter or exit the enclave are known as *edge calls* in Keystone, as in other enclave systems. For instance, if an enclave wants to send a network packet, it must use an edge call to deliver the data to an untrusted host process. The current version of Keystone allows *enclave* → *untrusted host* calls, also known internally as *ocalls* (outbound calls, names under discussion). In the current version of Keystone, all ocall wrapping code uses shared memory regions to transfer data. When referencing data in these regions virtual address pointers are never used, instead, only offsets into the region are used [23].

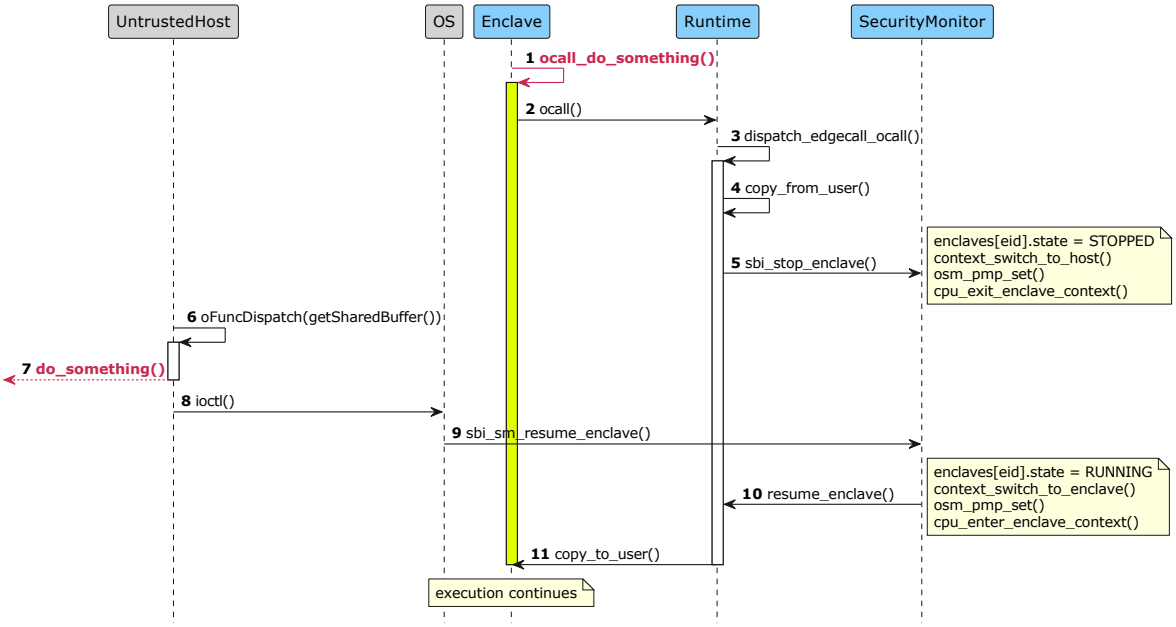

Figure 3.6. Simplified example of an ocall lifecycle [23].

Edge Calls Lifecycle

Consider for example a generic `ocall_do_something`, as represented in Figure 3.6. This call transfers some values passed as arguments from the enclave to be processed by the host process (it could be a value to be printed, a file to be stored and so

on). The enclave application calls `ocall_do_something(...)`, which is an edge wrapper function. `ocall_do_something(...)` uses the system-call-like interface to the runtime to execute an *ocalls* similar to `ocall(OCALL_DO_SOMETHING, &input, sizeof(input), &ouput, sizeof(output))`. The enclave passes a pointer to the value, the size of the argument and any necessary return buffer information. After allocating an `edge_call` structure in the shared memory region, the runtime fills out the call type, copies the value into another part of the shared memory, and sets up the offset to the argument value. Note that, in Keystone, edge calls employ offset values in the shared memory area, rather than pointers. The runtime subsequently exits the enclave with an `SBI_CALL`, i.e. `sbi_stop_enclave()`, passing a value indicating that the enclave is executing an *ocalls* rather than shutting down.

After resuming execution of the Keystone kernel driver, it checks the enclave's exit status, notes a pending *ocalls* and handles control to the userspace host process. The registered *ocalls* handler wrapper for `OCALL_DO_SOMETHING` is dispatched by the userspace host process, which also consumes the edge call. The wrapper generates a pointer to the argument value from the offset in the shared memory region and then calls `do_something` with the value as an argument. The host wrapper determines whether any return values must be copied into the shared memory region upon return and returns the control to the driver after setting the edge call return status to `SUCCESS`.

Through an `SBI_CALL`, the driver rejoins the enclave runtime. The enclave *ocalls* wrapper code is resumed once the runtime determines whether any return information has to be copied from the shared region into return buffers. Finally, the enclave function that has called at the beginning `ocall_do_something` receives any return values from the *ocalls* wrapper code [23].

3.4 Memory isolation using RISC-V PMP

In Keystone, developers refer to the memory section that an enclave uses as a *region* and each region is defined by a PMP entry. The SM employs two PMP registers for internal purposes (i.e. security monitor memory and untrusted memory). One active enclave may use one of the remaining PMP entries each. RISC-V PMP has several properties, the most relevant are:

- prioritization by index: the index of PMP entries statically determines the priority. Indices go from `0` to `N`, where `N` depends upon the platform. `0` is the highest priority, whereas `N` is the lowest
- default denies: if no PMP entry matches with an address, the memory access will be rejected by default.

For simplicity, in the following explanation PMP entries are denoted as `pmp[i]` where `i` is an index. Fig. 3.7 shows the memory at the initial state. At the start of the boot process, physical memory is not accessible by U- or S-modes.

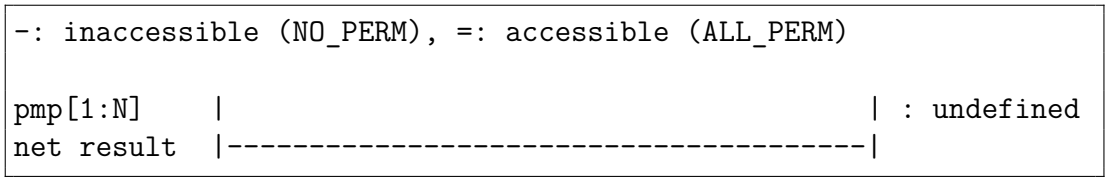

Figure 3.7. Memory state when booting start [23].

The SM sets the highest priority PMP entry to cover the address range containing itself and sets all permission bits to 0. Suddenly, the SM sets the lowest priority PMP entry to cover the full memory and sets all permission bits to 1, this will allow the OS to access the remaining memory and start booting. The result can be seen below in Fig. 3.8.

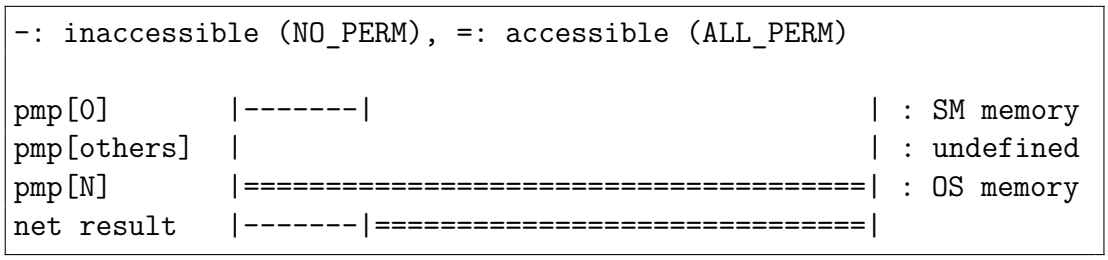

Figure 3.8. Memory state just after booting [23].

Every time the SM creates an enclave, it will select a PMP entry for the enclave to defend its memory from other U-/S-mode software. This can be seen below in Fig. 3.9.

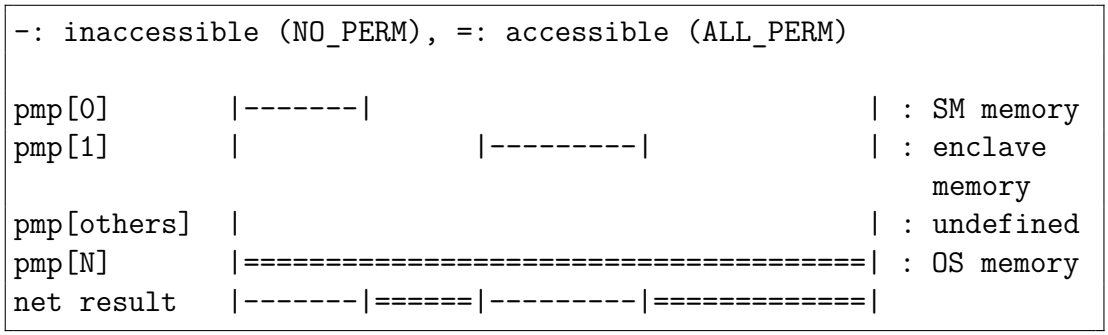

Figure 3.9. Memory accessible by the untrusted host [23].

When the SM enters the enclave and executes the eapp, it flips the permission bits of the enclave's PMP entry and the last PMP entry. Untrusted shared buffer is the term for the contiguous memory region that Keystone enables the OS to allocate in the OS memory space in order to use it as an enclave's communication buffer. This is shown below in Fig. 3.10. The SM just flips the permission bits back when it leaves the enclave. When an enclave is destroyed by the SM, the PMP entry is made available for usage by other enclaves.

```
-: inaccessible (NO_PERM), =: accessible (ALL_PERM)

pmp[0]        |-------|                                  | : SM memory
pmp[1]        |               |=========|                | : enclave
                                                           memory
pmp[others]   |                                          | : undefined
pmp[N]        |                                    |==|  | : untrusted
                                                           shared
                                                           buffer
net result    |-------|------|=========|-------|==|--|
```

Figure 3.10. Memory accessible by a running enclave [23].

3.5 Keystone key-hierarchy

Fig. 3.11 shows the key hierarchy of Keystone. The root of the key hierarchy is the asymmetric processor key pair (SK_D and PK_D). The asymmetric security monitor key pair (SK_SK and PK_SM) is derived from the measurement of the security monitor (H_SM) and the private processor key (SK_D) [23]. As a result, the computed security monitor key pair is bound to the processor and to the identity of the security monitor itself.

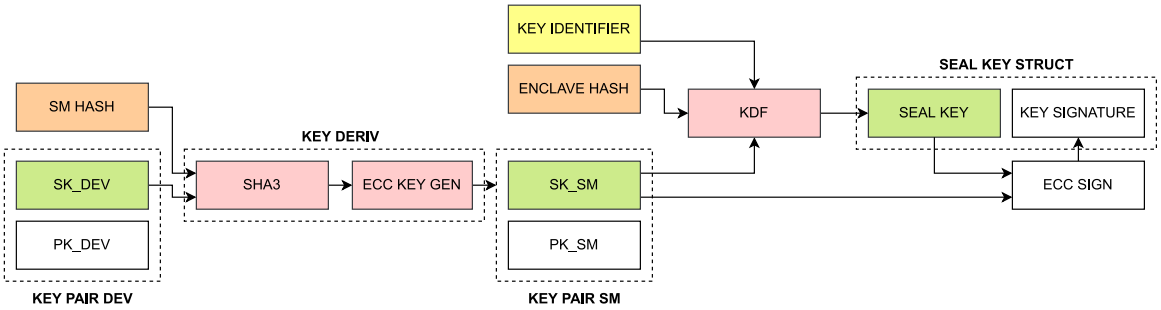

Figure 3.11. The key hierarchy of Keystone [23].

3.5.1 Sealing-Key Derivation

In Fig. 3.11 is also visible how sealing-keys are derived. An enclave can generate a key for data encryption using the data-sealing capability, enabling it to store data in untrusted non-volatile memory outside the enclave. This key is tied to the identity of the processor, the security monitor, and the enclave. As a result, only the same enclave using the same processor and security monitor can generate the same key. Data can be encrypted using this key and stored in unsecured, non-volatile memory. After an enclave restart, it can generate the same key once more,

retrieve the encrypted data from the untrusted storage, and then use the derived key to decrypt it [23]. A sealing key is computed starting from three inputs:

- the private security monitor key (`SK_SM`)
- the hash of the enclave (`H_SM`)
- a key identifier

The key identifier is an extra input for the key derivation function selectable by the enclave. A single enclave can generate several keys by giving the key identifier various values.

Chapter 4

Self-Sovereign Identity as a Service

The idea of a Self-Sovereign-Identity as a Service (SSIaaS) is to support a constrained device to create and manage its Self-Sovereign Identity. Since IoT devices cannot run natively the complete SSI stack there is a need to design and develop an edge device capable of providing such an identity to constraint devices as a service. Such a solution has the advantage of increasing the number of devices that can interact in such a secure digital ecosystem.

4.1 Use case analysis

The first step is to analyse and identify critical cryptographic operations involved in self-sovereign-identity management. The examples in Figs. 4.1, 4.2 and 4.3 illustrate a high-level procedure to create and use verifiable credentials.

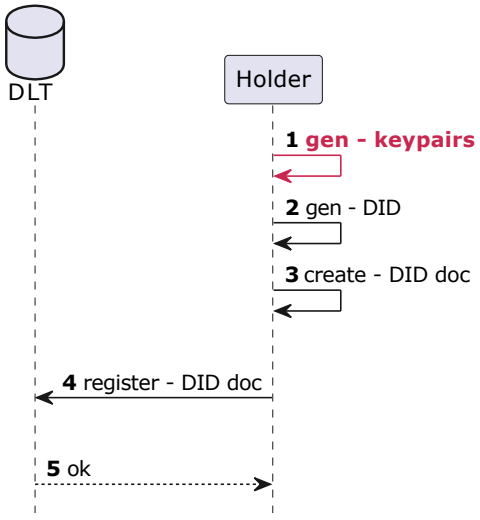

Figure 4.1. Creation of a DID

4.1.1 Creation of a DID

Independently from the chosen DLT, when a holder creates a DID, he uniquely binds cryptographic proofs with the DID identifier. In this process, the holder typically needs to generate public and private key pairs and then insert them in the DID document. The order of operations is presented in Fig. 4.1.

4.1.2 Verifiable Credential Issuance

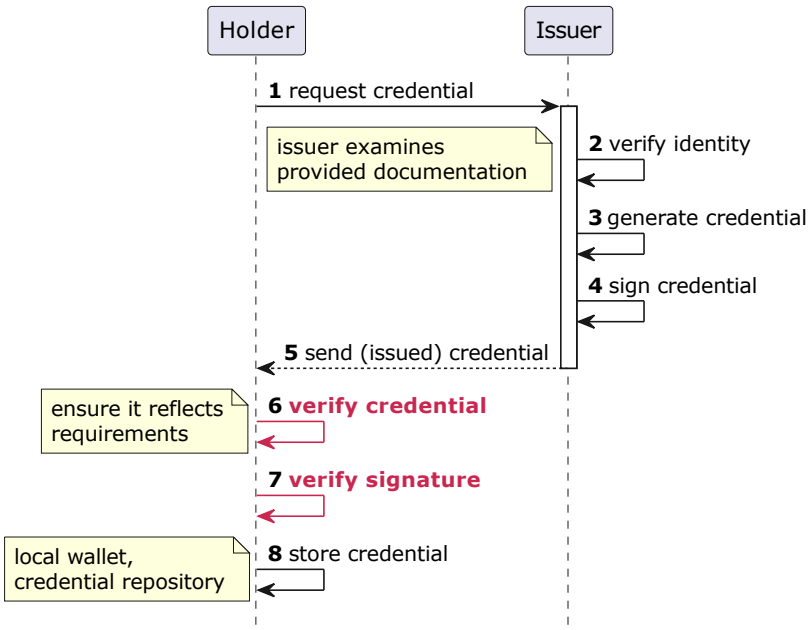

Figure 4.2. Issuance - verifiable credential creation

After the creation of a DID, a holder can get a verifiable credential from an issuer, that will verify the identity in some way, for example by examining some provided documentation. If the requirements are satisfied, the issuer will generate a verifiable credential by linking her identity information to DID. The holder receiving the verifiable credential will verify its validity and save it in his personal credential repository. The whole procedure is depicted in Fig. 4.2.

4.1.3 Verifiable Credential Verification

As can be seen in Fig. 4.3, once a holder has a DID and a verifiable credential, he can use them to access a service to a verifier. The holder will use the DID to prove to the requesting party that it is the controller of that DID through some sort of challenge-response. Then, the holder will create a verifiable presentation starting from one or more verifiable credentials. Whenever possible, to reduce correlation it is recommended to use selective disclosure, i.e. presenting proofs of claims without revealing the entire verifiable credential. Once created the verifiable presentation, the holder can send it to the verifier, which will check its validity and will authorize the holder if everything is correct.

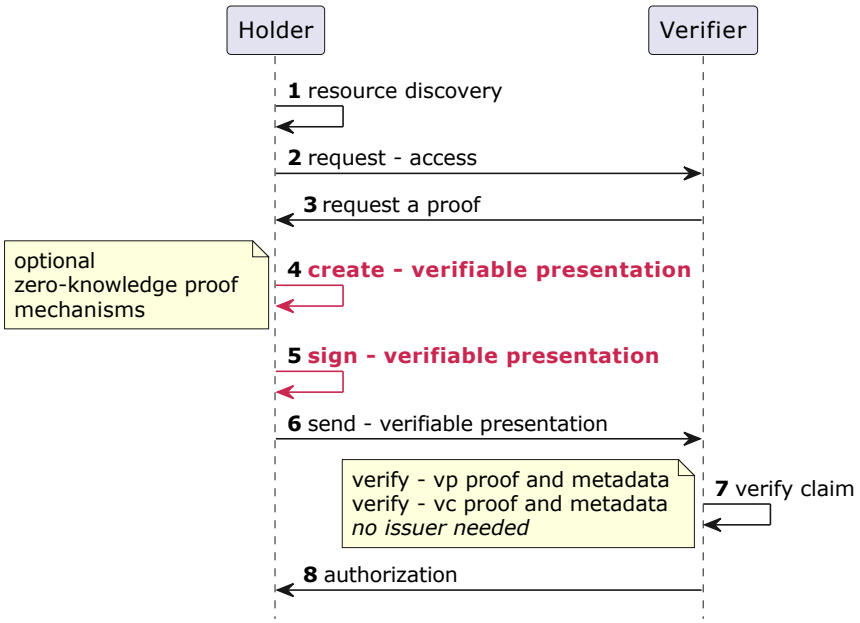

Figure 4.3. Verification - verifiable credential usage

4.2 Performance analysis

After analysing the different use cases, as highlighted in Figs. 4.1, 4.2 and 4.3, the cryptographic operations that could be critical for a constrained device are:

- keys generation
- signature generation and verification
- proof generation and verification

To understand how much critical these operations are on a constrained device it is necessary to analyse the differences between a non-constrained one by getting their execution times and comparing the result. STM32L4+ Discovery kit IoT node[24] has been used as a constrained device equipped with the `STM32L4S5 MCU @ 120 MHz`, 2 Mbytes of flash memory and 640 Kbytes of SRAM. The employed non-constrained device is instead more powerful and is equipped with an `Intel Xeon Silver 4110 CPU @ 2.10GHz`.

Mbed TLS [25] has been used as a library that implements the operation for key generation and signature (generation and verification). Mbed TLS is a C library that implements cryptographic primitives, it supports RSA, ECDSA, and other algorithms such as Ed25519. It has a small code footprint which makes it reasonable to use for embedded systems [25]. To test if a constrained device could be capable to implement and use some *zero-knowledge proof* mechanisms, it has been decided to use a library which implements the BBS+ signature scheme [26]. BBS+ signatures can be used to generate signature proofs of knowledge and selective disclosure zero-knowledge proofs [27] and they are implemented on top of BLS12-381 elliptic curve [28], which is a curve not supported in Mbed TLS. Since available BBS+ libraries

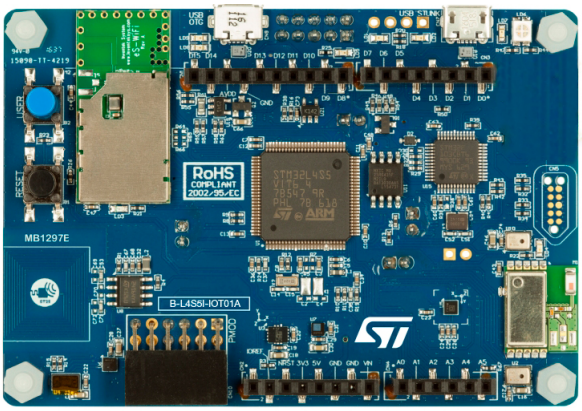

Figure 4.4. STM32L4+ Discovery kit IoT node[24].

are also not supported for STM32L4, it has been decided to take execution times only on the edge device. Then execution times between the ECDSA signature scheme and BBS+ signature scheme are comparable since they are been taken on the same architecture and they are both elliptic curve signature schemes.

4.3 Results

It is evident from the Table 4.1 that RSA is usable on constrained devices in real-life applications only if the key generation is precomputed, otherwise is unusable. On the constrained device, RSA signing and verification are faster than ECDSA, but ECDSA provides the same level of security as RSA but it does so while using much shorter key lengths. In applications where it could be useful to generate a keypair on the fly, ECDSA is a must. On the edge device, the ECDSA signing operation is faster than RSA, while the RSA verification process is faster than ECDSA. It can also be noted that the time difference between the constrained device and the edge device is extensive, due to the frequency of operation of the MCU and the CPU.

Operation	**constrained device***	**edge device***
EC-p256-keygen	318 ms	0.5 ms
ECDSA-p256-SHA256-sign	1 503 ms	0.6 ms
ECDSA-p256-SHA256-ver	6 031 ms	2.0 ms
RSA2048-keygen	622 749 ms	186 ms
RSA2048-SHA256-sign	1 305 ms	3.60 ms
RSA2048-SHA256-ver	331 ms	0.07 ms

*mbedTLS library

Table 4.1. Execution time comparison between constrained and non-constrained devices

Furthermore, in Table 4.2 ECDSA signatures and BBS+ signatures are compared on the same CPU architecture. The obtained results are that BBS+ signature scheme is much slower than the ECDSA one, even if they are both elliptic curve

based. The order of magnitude of the percentage increase is 3, which is a significant difference.

Operation	ECDSA*	BBS+*†	% increase
keygen	0.2 ms	39 ms	+7 800
sign	0.6 ms	27 ms	+4 500
verify	2.0 ms	166 ms	+8 300

*Xeon 2.10GHz †Rust bbs library

Table 4.2. Execution time comparison between ECDSA and BBS+

4.4 Design and implementation choices

The results obtained in Sect. 4.3 suggest that a constrained device cannot use the BBS+ signature scheme in real-world applications, since it can be supposed that execution times will be considerably high. In the SSIaaS paradigm, a new role, called *edge*, is defined. The interaction with the edge device allows IoT-constrained devices to join the SSI ecosystem. Roles and interactions of SSIaaS can be seen in Fig. 4.5.

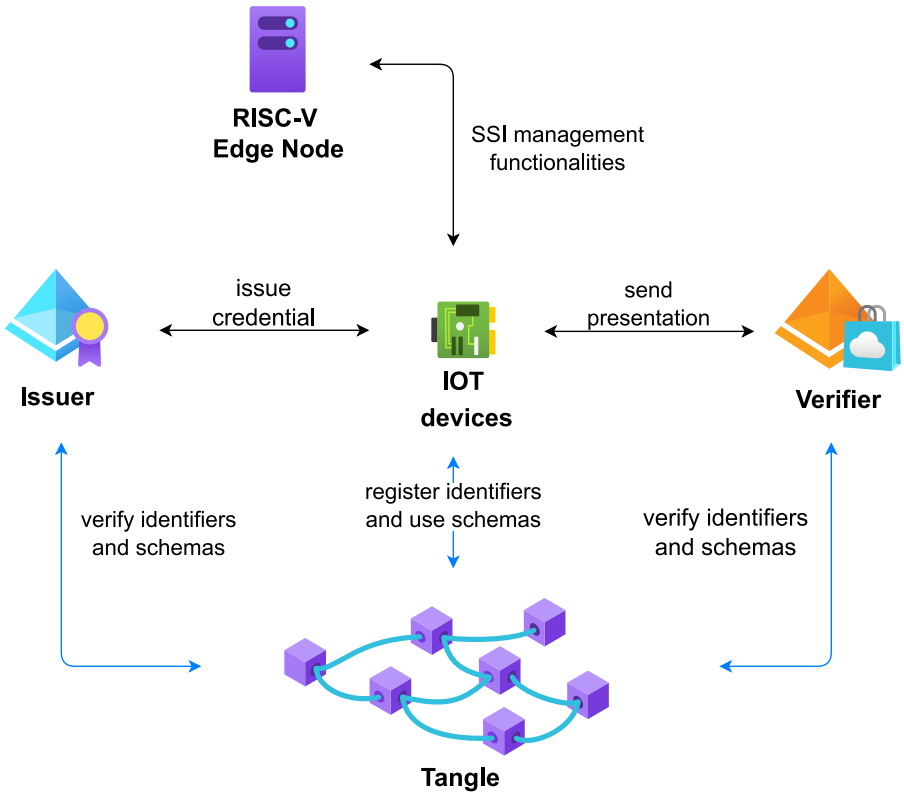

Figure 4.5. The roles and information flows of Self-Sovereign-Identity as a Service

4.4.1 Self-Sovereign-Identity as a Service Ecosystem

The edge device interacts with IoT-constrained devices and provides functionalities to create and support them in identity management. As shown in Fig. 4.6, the exposed operations by the edge device are:

- key pairs generation
- storage of a verifiable credential
- generation of a verifiable presentation

The IoT device can request the edge node to generate key pairs that could later use to generate a verifiable presentation. The IoT device can store on the edge node a verifiable credential got from an issuer. If the IoT device wants to interact with a verifier, it has to provide a verifiable presentation, which is generated by the edge device starting from the verifiable credential and signing the verifiable credential with a keypair previously generated for that specific IoT client. Since these are security-critical operations because they handle the identity of the IoT node, it needs the guarantee that the data shared with the edge device and the executed code are protected with respect to confidentiality and integrity. Hence, a TEE has been implemented using the Keystone framework. Figs. 4.7 and 4.10 show the architecture and the preliminary interactions between the IoT device and the edge device before the constrained device can ask for services.

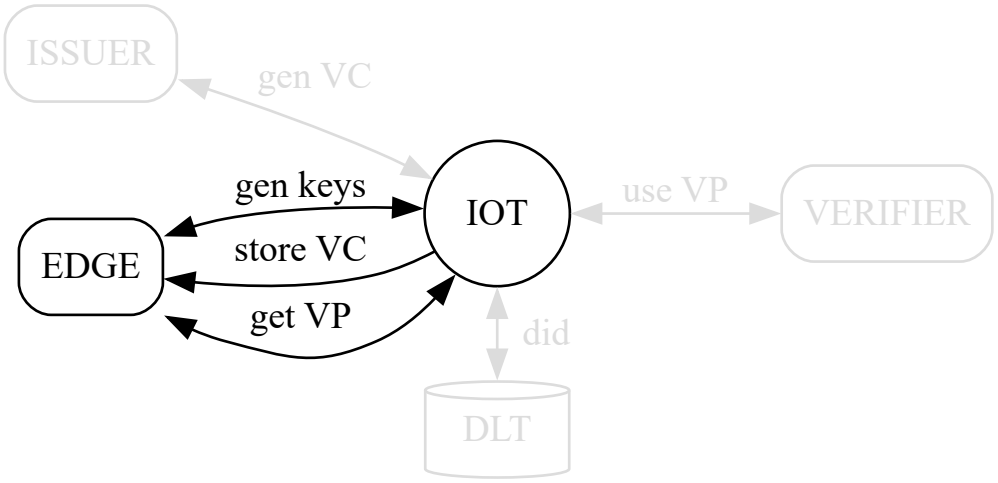

Figure 4.6. Basic components of Self-Sovereign-Identity as a Service

4.4.2 IoT Device Provisioning

Before deploying an IoT device in the network, it needs to be configured and initialised to be able to communicate and establish a trust relationship with the edge device. This operation is named *provisioning*. After the provisioning, the IoT device can be authenticable by the edge device and can communicate with it. In the implemented solution, a trustworthy system administrator provisions the client, i.e. the constrained device, with:

- a key pair (client secret and public keys) used to authenticate itself to the server, i.e., the edge device
- expected security monitor and enclave hashes to verify that the edge device is running the expected application in the enclave
- the device public key of the edge device to verify that is running on trusted hardware

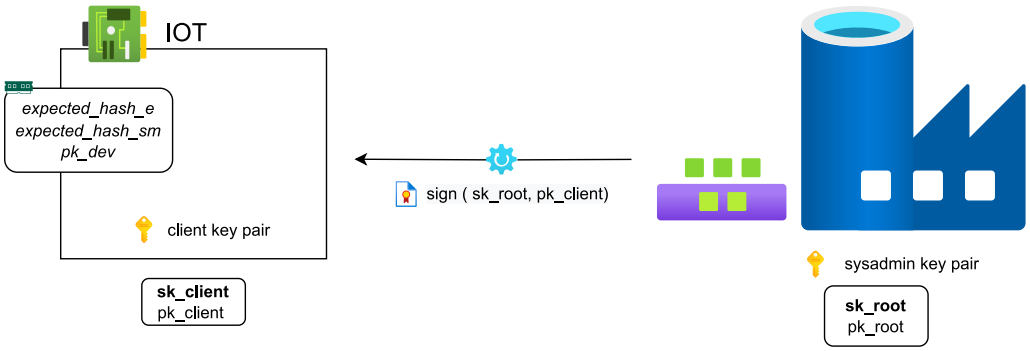

Figure 4.7. System administrator provisions IoT client with expected hashes, keys and signature of the public key

Then, the system administrator signs the public key of the IoT client and saves the signature. The signature is verifiable with the public key of the system administrator. It proves that the device with access to the client's secret belongs to a recognised IoT network group. This rest on the assumption of a pre-existent trust relationship between the system administrator and the edge device. The client authentication method just described was chosen because the edge device only needs to retain in memory the system administrator's public key, rather than keeping track of all the clients' public keys in a database, to determine whether a client is authorized to access edge SSI services on-demand.

4.4.3 Attestation Report and Session Context

Keystone supports a basic attestation mechanism, as introduced in Sect. 3.3 and uses ed25519 signatures [29] for computing hashes of the security monitor and enclave content. Once an enclave has started the execution, it requests the SM to generate a signed enclave report and a signed SM report. The SM Report contains a hash of the security monitor and the attestation public key, all signed by the device root key. The enclave report consists of a hash of the enclave at initialization and a data block from the enclave of up to 1Kbytes, all signed with the attestation public key of the SM.

A client will verify the signatures and values contained in the attestation report by using the device's public key, the expected SM hash, and the expected enclave hash. An example of an attestation report is shown in Fig. 4.8.

```
{
    "enclaveApplication": {
        "hash": "11ff35526c90c469ca6878dc22494703c554db654...",
        "enclaveData": "b07dfd3047fb5213b8af9b76594a06891...",
        "signature":
            "58ca380c6b4476ed7b27143d92dec14fff2858ffef8ed6ef4..."
    },
    "securityMonitor": {
        "hash": "e51749130b6036fe85b27409a4ea3e1c078fe4dcb76...",
        "publicKey":
            "cd98f4a28a8523ba8ecd31175aa0e2330b2f46e70...",
        "signature":
            "e738f4e708f73ffa4a0d3dc2199c9e0ac119bf14b32da33a..."
    },
    "devicePublicKey":
       "0faad4ff01178583baa588966f7c1ff32564dd17d..."
}
```


Figure 4.8. Example of an attestation report generated by the enclave.

When designing the demo, the session context has been introduced to authenticate the client on the server side. The session context includes two sections: the stub certificate and the homonym session context. The first section contains the client's public key and the signature of the client's public key, made at the provisioning phase by the system administrator and verifiable with the public root key. The enclosed session context contains a data block, all signed by the client public key.

The edge node will verify the signatures of this structure by using the root public key of the system administrator. An example of a session context is shown in Fig. 4.9.


```
{
    "sessionContext": {
        "data": "3a2992fe52a2082b26f8009117be53decff68...",
        "dataSignature": "02b0b554aecd929419a6c525ebfde..."
    },
    "stubCertificate": {
        "clientPublicKey": "e95f84f09574d8fe73c0e692905c6c8d...",
        "rootSignature": "a8b20a77877b79ca0d699091f8b303463b..."
    }
}
```


Figure 4.9. Example of a session context generated by the client.

In the demo implementation, the data part of the session context and the attestation report are both used for the Diffie-Hellman key exchange protocol so that the two parties can securely compute a shared key for creating a secure encrypted channel.

4.4.4 Demo Architecture

To ease the development of a first demonstration application, the architecture described in Sect. 4.4.1 has been simplified. In the presented implementation a single instance of edge device can handle requests only from one IoT node at a time. The developed application demonstrates how a remote-constrained device can request on-demand SSI services, by offloading computations to be performed on an untrusted edge server using an enclave. It has to be pointed out that the developed application code contains test keys, which are not safe and should not be used in production.

The demo architecture consists of:

- a server enclave application and an untrusted host application hosted on a RISC-V processor
- an IoT client provisioned by the system administrator

The demo enclave application has essential enclave capabilities (attestation report generation, data sealing, etc.). It uses Libsodium [30] for establishing a secure channel between the IoT client and the edge server. It exposes the three SSI-related services, as presented in the Sect. 4.4. The untrusted enclave host serves a few functions: starting the enclave, proxying network packets from and to the client and storing sealed data. The remote client establishes a connection with the untrusted host, verifies the enclave report, transmits the session context, and creates a secure channel before being able to communicate with the host to request offloaded SSI-related computations.

Running the demo

At the start, the client will connect to the enclave and perform the remote attestation. Expected hashes in the attestation report will be used by the trusted client to verify that the enclave is created with the right application and initialized by the known version of the security monitor. If the attestation is successful, the client can send its attestation, also called session context. The server will check the signature validity provided by the client verifying it with the public key of the system administrator (`root_pk`). On the other hand, if the edge device attestation report is invalid, the client will close the connection for security reasons, i.e. it is not the known edge device. Upon exchanging the session context and the attestation report, they establish a secure channel and the enclave-host waits for service requests.

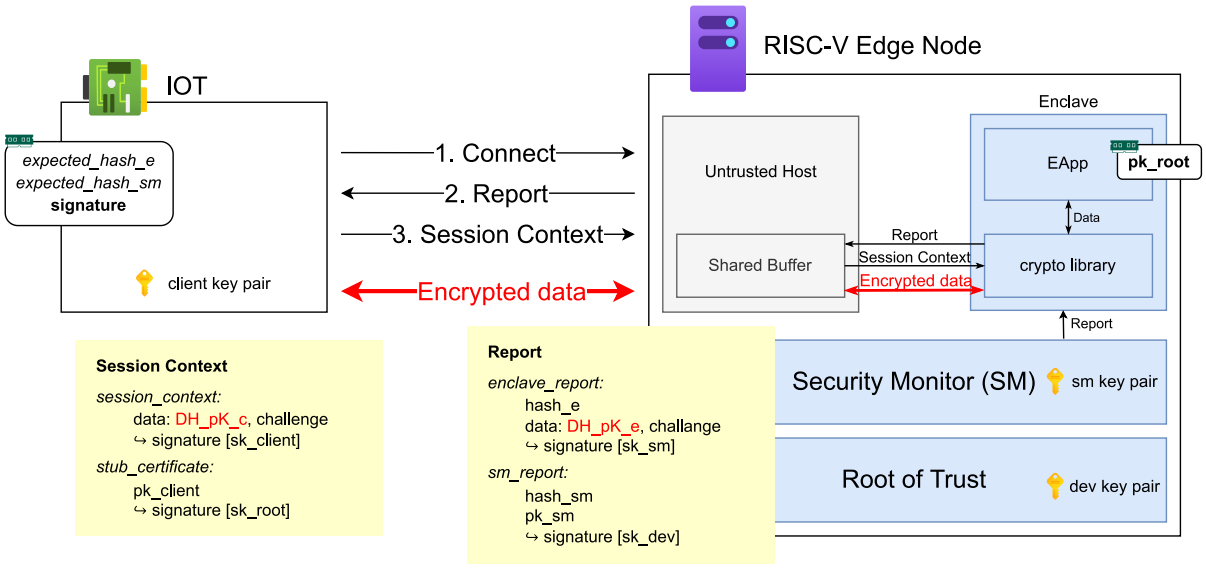

Figure 4.10. Demo architecture

4.4.5 Offloaded Operations

Once the edge node and the constrained devices have established a secure channel to communicate, the enclave waits for the client to request service. Once a message is received, the enclave authenticates and decrypts the message. If successful, it processes the message and passes the request to a specific *ocall*, i.e `ocall_wait_for_request(...)`, which dispatches the requests proxied by the untrusted host coming from the client. The enclave understands what service has been requested by checking the type of request sent by the IoT client. After processing the request, the enclave returns the result through the secure channel previously established. Figs. 4.12, 4.13 and 4.14 show a high-level view of the operations that have been implemented.

Setup phase

Using Keystone SDK, the generated sealing key size will be 128 bytes. In the demo implementation, the sealing key is used as a seed for generating an ed25519 key pair. The sealed data are signed with the ed25519 key pair. Whereas, the key used for encryption consists of the first `crypto_aead_chacha20poly1305_KEYBYTES` bytes, i.e. 32 bytes. of the sealing key. As is visible in Fig. 4.11, *libsodioum*, a cryptographic library written in C, is used for generating these keys.

Generate public keys

Consider the service of key pairs generation. The client sends this request along with the key type and a secret. The server processes the incoming request. It generates the sealing key, using as an identifier in the key derivation function the

```
// here sealing_key is used as a seed to gen sign keys
crypto_sign_seed_keypair(enclave_signing_pk, enclave_signing_sk,
                         sealing_material.key);

// sealing key here is used for the data encryption usage
memcpy(enc_key, sealing_material.key,
   crypto_aead_chacha20poly1305_KEYBYTES);
```

Figure 4.11. Setup of the encryption key and ed25519 keypair.

client's public key and the secret sent by the client. Then, it generates two key pairs of the type that the client indicates in the request. (To ease the development only `EdDSA` can be generated, in the future `BBS+` signature scheme will be implemented). Next, it encrypts the generated key pair with the obtained sealing key and signs the obtained ciphertext with an enclave signing key. The encryption of the data prevents the untrusted host to violate confidentiality. By checking the integrity and authenticity of the stored data with the signature, the enclave is able to verify that no alteration has been carried out from the untrusted host. Next, the enclave application calls `ocall_save_sealed_data(...)`, which is the edge wrapper function to the system-call `ocall(OCALL_SAVE_SEALED_DATA, buffer, len, 0, 0)`. This call exports the buffer, containing the ciphertext and the signature of the generated key pairs, from the enclave to be stored in the untrusted non-volatile memory by the host process. The file name is composed of the public key of the client and the type of data saved, in this case, keys. Hence, the file name will be `<client_pub_key>_keys`. The host saves the data received with the *ocall* and returns the control to the enclave. Finally, the enclave sends the response to the host, which will proxies the response back to the client. The response contains only the public part of the generated keys.

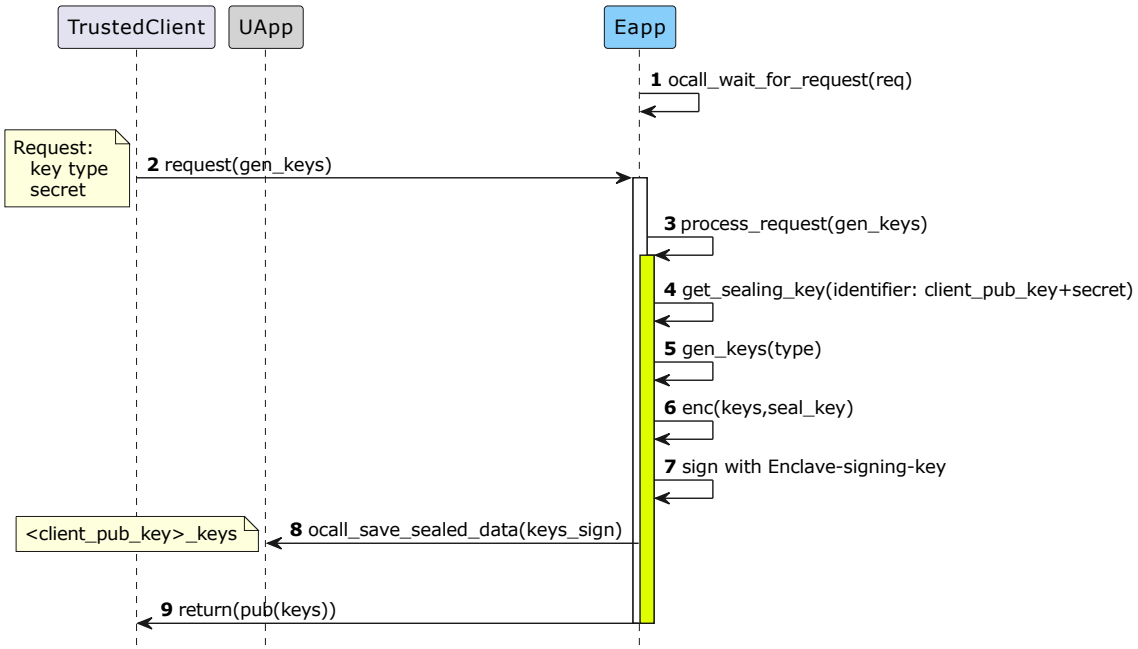

Figure 4.12. Key pairs generation operation (pseudo-code)

Store verifiable credential

The IoT client device can request the edge device for the storage service of a verifiable credential. The client sends this request along with the verifiable credential and a secret. The server processes the incoming request. It generates the sealing key, using as an identifier in the key derivation function the client's public key and the secret sent by the client. The sealing key employed for encryption will be identical for the same client. Then, it encrypts the received verifiable credential with the obtained sealing key and signs the obtained ciphertext with an enclave signing key. The encryption of the data prevents the untrusted host to violate confidentiality. By checking the integrity and authenticity of the stored data with the signature, the enclave is able to verify that no alteration has been carried out from the untrusted host. Next, the enclave application calls `ocall_save_sealed_data(...)`, which is the edge wrapper function to the system-call `ocall(OCALL_SAVE_SEALED_DATA, buffer, len, 0, 0)`. This call exports the buffer, containing the ciphertext and the signature of the verifiable credential, from the enclave to be stored in the untrusted non-volatile memory by the host process. The file name is composed of the public key of the client and the type of data saved, in this case, a verifiable credential. Hence, the file name will be `<client_pub_key>_vc`. The host saves the data received with the *ocall* and returns the control to the enclave. Finally, the enclave sends the response to the host, which will proxies the response back to the client. The response contains only feedback about the success of the storage operation.

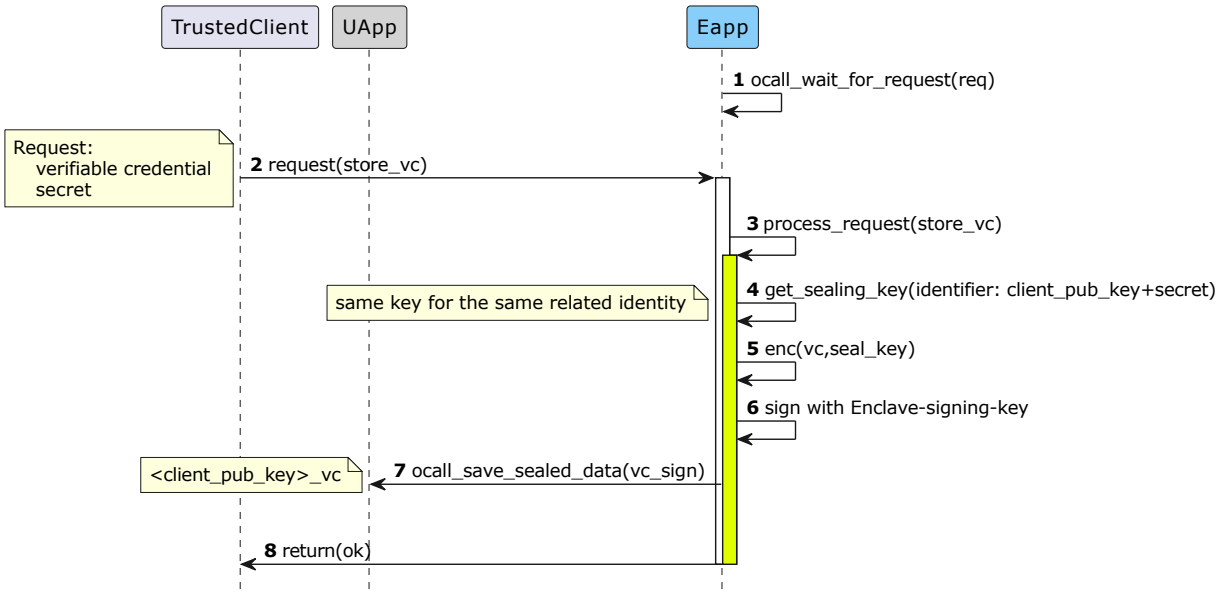

Figure 4.13. Storage of a verifiable credential operation (pseudo-code)

Get verifiable presentation

The IoT client device can request the edge device to create a verifiable presentation. The client sends this request along with a nonce and a secret. The server processes the incoming request. It generates the sealing key, using as an identifier in the key derivation function the client's public key and the secret sent by the client. The sealing key employed for encryption will be identical for the same client. Then, the enclave application calls `ocall_retrieve_sealed_data(...)`, which is the edge wrapper function to the system-call `ocall( OCALL_RETRIEVE_SEALED_DATA, buffer, len, msg, sizeof( struct edge_data ))`. This call exports the buffer, containing the file name to be retrieved, and the host once reading the correct file copies it into the shared memory region, sets the *edge_call* return status to SUCCESS and returns control to the enclave. The host saves the data obtained from the enclave and returns the control. The runtime copies the file from the shared region into return buffers and then resumes the enclave ocall wrapper code. The function will be executed for retrieving from the untrusted non-volatile memory the verifiable credential (the file name will be `<client_pub_key>_vc`) and for retrieving the previously generated keys (the file name will be `<client_pub_key>_keys`). The enclave will check the validity of signatures and decrypt the sealed data. Finally, the enclave using the retrieved keys signs the verifiable credential with the nonce received from the client and sends the response to the host, which will proxies the response back to the client. The response contains a freshly generated verifiable presentation.

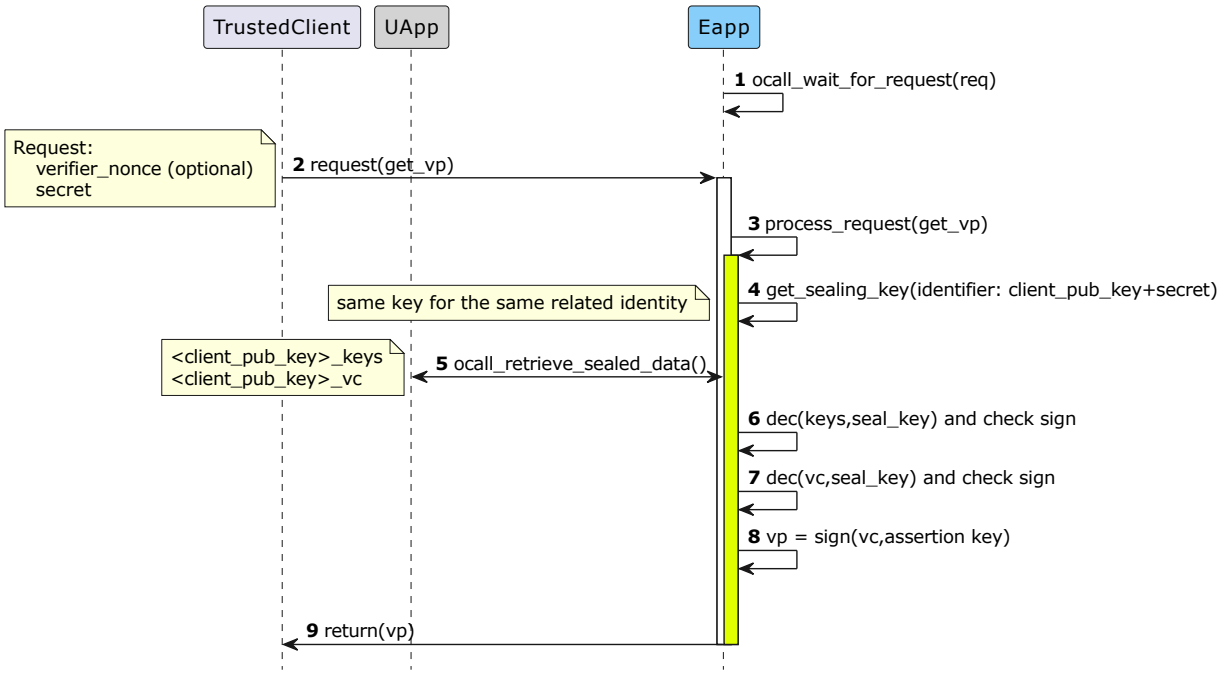

Figure 4.14. Generation of a verifiable presentation operation (pseudo-code)

Chapter 5

Conclusion and Future Work

In conclusion, this work allows IoT-constrained devices to establish a secure communication channel with the edge device and trusted SSI-related operations computation, which enables to access higher computational performances and more flexible hardware and software requirements. In the future, this opens up new scenarios employing additional and more modern cryptographic primitives for privacy-preserving credentials, which are yet not implemented in the current solution. Another future research topic could be the integration of a TLS connection between an external client and the enclave and how securely terminate TLS within the enclave, considering that the untrusted host proxies enclave messages. Additionally, another enhancement of the edge device could be the monitoring of the untrusted host integrity with a TPM. This is just an introductory work to enable and ease the adoption of the Self-Sovereign Identity as a Service paradigm.

Appendix A

User Manual

A.1 Constrained and edge device comparison

This section describes how to install and execute the code used for testing the cryptographic capabilities of constrained and non-constrained devices.

A.1.1 Test MbedTLS library

Mbed TLS [25] is a C library and it was used for testing RSA and ECDSA cryptographic primitives. The version used is `mbedtls-3.1.0`, available from this repository: https://github.com/Mbed-TLS/mbedtls.

Test on non-constrained device

First, MbedTLS needs to be installed on the chosen system.

```
$ wget
 https://github.com/Mbed-TLS/mbedtls/archive/refs/tags/v3.1.0.zip
$ unzip v3.1.0.zip
$ cd mbedtls-3.1.0/
$ sudo make install
```

Once the library has been installed, enter the Mbed TLS test directory and compile the program.

```
$ cd mbedtls-test
$ gcc -o test main.c mytest.c -lmbedtls -lmbedx509 -lmbedcrypto
```

Then tests can be launched and the results can be stored on a file with the following command.

```
$ ./test > results.txt
```

The output should be similar to the following shown in figure A.1.

```
Seeding the random number generator... ok

Test: ECDSA key gen
p256
0.000266988,
0.000238424,
...
Test: ECDSA sign-gen
p256 key gen time: 0.000260937
0.000004468,0.000309109,0.001057424, # (hash, sign, ver) times
0.000003286,0.000296264,0.001026635,
...

Test: RSA 2048 key gen
0.133808741,
0.087886941,
...

Test: RSA sign-ver
rsa2048 key gen time: 0.065950689
0.000003988,0.002522734,0.000039014 # (hash, sign, ver) times
0.000003206,0.001853610,0.000038684
...
```

Figure A.1. Example of MbedTLS tests output on a non-constrained device.

Test on constrained device

STM32CubeIDE [31] has been used (version `1.9.0`) for developing and flashing the binaries onto the STM32L4+ board [24].

For simplicity, the complete project is provided and it just needs to be imported into STM32CubeIDE as an existing project. Click `File > Import > Existing projects into Workspace`, then select the archive file `mbedtlsv1-stm32.zip` from the file system, select the project `Test1` and click `Finish`. Then plug the board into a USB port and click `Project > Build Project`. The code will be compiled. When it is done (and 0 errors appear in the console panel at the bottom), click `Run`.

The output on the board is very similar to the previous one, for debug messages the output of the `printf` function is redirected to one `UART` and it can be read with a terminal emulator that supports serial port, such as `Teraterm` or `Putty`, by reading the `COM` that correspond to the `STM32` debugger. The output should be similar to the following shown in figure A.2.

```
Seeding the random number generator... ok
Test: ECDSA key gen, sign-gen
p256
318,
316,
...
key gen time: 319
5;357;1257 # (hash, sign, ver) times
5;361;1264
```

Figure A.2. Example of MbedTLS tests output on a constrained device.

A.1.2 Test BBS+ signatures scheme

BBS+ rust library was used for testing BBS+ signatures that can be used to generate signature proofs of knowledge and selective disclosure zero-knowledge proofs. In these tests, only simple single-message signing and verification were tested.

Before starting, we need to install Rust. The installation of Rust is through a tool called Rustup, which is a Rust installer and version management tool. The way to install Rustup differs by platform, on Unix, run the next command in the shell. This downloads and runs `rustup-init.sh`, which in turn downloads and runs the correct version of the `rustup-init` executable for your platform. For other platforms check the Rust documentation [32].

```
$ curl https://sh.rustup.rs -sSf | sh
```

When Rustup is installed, it will also add the latest stable version of the Rust build tool and package manager, also known as Cargo. The following commands will install dependencies and build the project.

```
$ cd bbs-test
$ cargo build
```

To launch the test and take executions times of BBS+ signatures and verification, run:

```
$ cargo run
```

The output should be similar to the following shown in figure A.3.

```
AVG Time elapsed in key generation() is: 25 ms
AVG Time elapsed in sign() is: 18 ms
AVG Time elapsed in ver() is: 102 ms
```

Figure A.3. Example of BBS+ tests output.

A.2 Proof of concept

The proof of concept relies on the full Keystone SDK. The easiest way for building and try Keystone and the proof of concept is to use QEMU. QEMU is an open-source machine emulator, in this case, is used to emulate RISC-V architecture. The proof of concept has been tested with `Ubuntu 18.04`.

A.2.1 Keystone installation and requirements

The following dependencies must be installed before installing Keystone.

```
$ sudo apt update
$ sudo apt install autoconf automake autotools-dev bc \
  bison build-essential curl expat 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 python pkg-config libglib2.0-dev libpixman-1-dev \
  libssl-dev screen device-tree-compiler expect makeself \
  unzip cpio rsync cmake p7zip-full
```

Then check out the Keystone repository and install everything with the quick setup script, it will install the RISC-V toolchain and if `KEYSTONE_SDK_DIR` environment variable is not set, it will also install Keystone SDK.

```
$ git clone https://github.com/keystone-enclave/keystone.git
$ cd keystone
$ ./fast-setup.sh
```

If everything goes right, the following message is shown:

```
    RISC-V toolchain and Keystone SDK have been fully setup
```

After running `fast-setup.sh`, run the following command to temporarily set in the current shell relevant environment variables:

```
$ source source.sh
```

Otherwise for permanently store the environment variables, if bash is used this command will add the lines in `source.sh` to the shell's startup file:

```
$ cat source.sh >> $HOME/.bashrc
```

CMake [33] is used as a build system. As `<build directory>` the name `build` has been chosen. Then all components can be built, beware that this will take a while.

```
$ mkdir build
$ cd build
$ cmake ..
$ make
```

! It has been noted that under `Windows Subsystem for Linux (WSL)` the build of the image can fail. To solve this issue just modify the `PATH` to not include `/mnt/c/*` folders.

A.2.2 Build the demo

Extract the provided zip file that contains the demo of the proof of concept. The extracted folder should contain the developed code explained in this document. The `riscv-musl-toolchain` has been used for building the `server_eapp`, which can be set with the `./setup_musl.sh` script. Then, the `./quick-start.sh` will build the demo, the script will create two files: `demo-server.ke` and `trusted_client.riscv`.

```
$ cd keystone-demo-poc
$ source ./setup_musl.sh
$ SM_HASH=./include/sm_expected_hash.h ./quick-start.sh
```

Then once the demo is built, the binaries need to be copied into the Keystone build folder.

```
$ cp ./build/demo-server.ke ./build/trusted_client.riscv
   ../keystone/build/overlay/root/
```

Now the QEMU image can be re-generated.

```
$ cd keystone/build
$ make image
```

A.2.3 Run QEMU

The following script will run QEMU, and start executing from the emulated silicon root of trust. The root of trust then jumps to the SM, and the SM boots Linux.

```
$ cd keystone/build
$ ./scripts/run-qemu.sh
```

The script will start ssh on a random forwarded localhost port, which allows multiple test runs on the same development machine. The script will print what port it has forwarded ssh to on start. For example, to start a new shell in another window the next command can be used, and it is useful to run the client and the server in two different terminal windows.

```
$ ssh root@localhost -p <port number>
```

Login as `root` with the password `sifive`. You can exit QEMU by `ctrl-a '+' x` or using `poweroff` command.

A.2.4 Run the demo

Inside QEMU, run the following commands. The first one will load the Keystone kernel module and the second one will set up the loopback device.

```
> insmod keystone-driver.ko
> ifdown lo && ifup lo
```

On the server side run:

```
> ./demo-server.ke
```

On the client side run:

```
> ./trusted_client.riscv 127.0.0.1
```

The client will connect to the enclave and perform the remote attestation. If the attestation is successful, the client can send back the session context. Then, if server checks go right, the communication can start and the client can request operation to the server.

If the enclave server app will be modified, the expected hash values have to be regenerated, otherwise, it can be tested with an option that will ignore the validation of the attestation report (it will still print the status of the validation).

```
> ./trusted_client.riscv 127.0.0.1 --ignore-valid
```

A.2.5 Expected output

The output of the client and the server should be similar to the following figures A.4 and A.5.

```
            === Security Monitor ===
Hash: e51749130b6036fe85b27409a4ea3e1c078fe4dcb76eb697e7e3cdc5ff
313144628a1ec10558cea5ad94641de7fb31fda759758115caf1144b2c49603f
42db3a
Pubkey: cd98f4a28a8523ba8ecd31175aa0e2330b2f46e7034545254660126a
9f3b8cb9
Signature: e738f4e708f73ffa4a0d3dc2199c9e0ac119bf14b32da33afe842
66803c9ae0766eb6d9f83e6d3b2511cbe8443996feb0bf8714f35c0c678aa593
90846a8d50a
            === Enclave Application ===
Hash: 11ff35526c90c469ca6878dc22494703c554db65406dcc9942417be2d9
010722843d1bda7115604f518fabd47ec3ba79cd89a560847bb5b810c314201e
2217d6
Signature: 58ca380c6b4476ed7b27143d92dec14fff2858ffef8ed6ef40156
515d36da50a220e494d936cafedd23558b8d16ce66c6d5f00bdb7ec973ee46ec
f7a48c6fa0d
Enclave Data: b07dfd3047fb5213b8af9b76594a06891c893001c6c4be448a
d8b13f7eb02a19b27d0263bfd9aa8941345f837788159897a8aea2ecb33da926
b8d4747328eaace08a34d55fcdc41e2938838da173900485e99bcc1fb9eb7b00
dac1e2fe5a5174
              -- Device pubkey --
0faad4ff01178583baa588966f7c1ff32564dd17d7dc2b46cb50a84a69270b4c
[TC] Attestation signature and enclave hash are valid
[TC] Session keys established

       Available services
1. generate keys
2. store verifiable credential
3. get verifiable presentation
 . everything else to quit
Type the service to request:
> 1
```

Figure A.4. Example of output at client side.

```
Verifying archive integrity... All good.
Uncompressing Keystone Enclave Package
[EH] Got connection from remote client
[SE] NOT USING REAL RANDOMNESS: TEST ONLY
[SE] Stub certificate signature is valid
[SE] [C] Successfully generated session keys.
[EH] Got an encrypted message:
3e5d0575e8bccf73 eb5255dad244eee1 93fcae78f5ad0459 ...

[SE] Sealing key derivation successful!
generate_public_keys

EdDSA_keys_t: [192]
07FAC4B1522F06D916C55D321BB839F23AEF95AA48E051E53DB2AF284D7845...

Ciphertext [208]
AB35E2D0F9C3ADFDD10E448D4ABD7A42E00ED87CE569C4AC3809785CC9A028...

Sign [272]
FA72997814FA1F16D111DA0189D0440F0F245EB606BACACD0566DEB0767D44...
[EH] Saving sealed data
[EH] Filename: /root/075CFECFC6617F ... 26A116B1611_keys_sign
Response payload: [64]
8E120A739BA94B7E9AE2F590166682170570D7AFCB70026B513405E56E90C1...
```

Figure A.5. Example of output at server side.

A.2.6 Tools for updating enclave and sm hashes

If the SM or the eapp will be modified, the expected hashes need to be modified to generate a valid report for the client. The next command will create the necessary files in the `include` directory. The demo must be recompiled and the QEMU image rebuilt before and after this command is executed.

```
# build the demo
# copy the binaries and build qemu image
$ KEYSTONE_BUILD_DIR=<path/to/keystone/build path>
    ./scripts/get_attestation_modified.sh ./include
# build the demo
# copy the binaries and build qemu image
```

If the `sm_expected_hash.h` is not present in the `include` folder, can be generated for the first time following the instruction in the `README.md` file of the SM repository available at the following url: https://github.com/keystone-enclave/sm.git.

A.2.7 Tools for provisioning

The file `provision` can be used to generate a new `test_client_key.h`, which will contain the root key pair, the client key pair and the signature of the client public key made by the system administrator with its secrey key.

It could be modified to meet your purposes (for example to generate more client key pairs and signatures or to use a fixed root key). The program uses `libsodium` to generate an Ed25519 key pair for the system administrator. Then will generate another Ed25519 key pair for the client and the public part will be signed with the system administrator key pair. The server will use the system administrator public key to authenticate that the client is part of the system administrator network.

Once `libsodium` has been installed, compile `provision.c`, run it and redirect the output to a file with the name `test_client_key.h`. The file `test_client_key.h` is already present in the `include` folder, but it may be necessary to generate a new one and it can be done with the following commands.

```
$ git clone https://github.com/jedisct1/libsodium.git
$ cd libsodium
$ git checkout 4917510626c55c1f199ef7383ae164cf96044aea
$ ./configure
$ make && make check
$ sudo make install
$ sudo ldconfig

$ cd keystone-demo/provisioning
$ gcc -o provision provision.c -lsodium

$ ./provision > ../include/test_client_key.h
```

Appendix B

Developer Manual

B.1 Constrained and edge device comparison

B.1.1 Test MbedTLS library

`mytest.c` is the most relevant file for this subproject. It is equal for both non-constrained devices and the constrained device (STM32L4+ board [24]). The only difference is the function used for measuring the elapsed time of a cryptographic primitive. The function `clock_gettime()`, of the standard library `time.h`, is used on the non-constrained device, while on the STM32L4+ board is used the `HAL_GetTick()`. Figure B.1 describes in a pseudo-language how to measure the elapsed time of a cryptographic primitive. Then, the available tests are described.

```
start = clock_gettime( ) or HAL_GetTick()
// mbedtls cryptographic primitive
end = clock_gettime( ) or HAL_GetTick()
print ( end - start ) // in ms
```

Figure B.1. Pseudo algorithm for measuring the elapsed time.

`int test_RSA_keygen(...)`
It generates `N` RSA key pair and prints the elapsed for each generation in a CSV format.
Input:

- `int key_size`, size of the generated key (possible values: `2048`, `3072` or `4096`)
- `mbedtls_ctr_drbg_context *ctr_drbg`, reference to `CTR_DRBG` context structure for random number generation
- `int iterations`, number of times that the test will be executed

Output: Returns `1` if successful, or an `MBEDTLS_ERR_XXX_XXX` error code

int test_RSA(...)
It generates an RSA key pair, computes the RSA signature of a random hashed message (of 1024 bytes) and verifies the generated signature. Then, it prints the elapsed for each signature and verification in a CSV format. (Everything is repeated N times).
Input:

- int key_size, size of the generated key (possible values: 2048, 3072 or 4096)
- int sha_alg,
- mbedtls_ctr_drbg_context *ctr_drbg, reference to CTR_DRBG context structure for random number generation
- int iterations, number of times that the test will be executed

Output: Returns 1 if successful, or an MBEDTLS_ERR_XXX_XXX error code

int test_ECDSA_keygen(...)
It generates N EDSA key pair and prints the elapsed for each generation in a CSV format.
Input:

- int ecparams, the identifier of domain parameters (curve, subgroup and generator),(possible values: MBEDTLS_ECP_DP_SECP256R1, MBEDTLS_ECP_DP_SECP384R1 or MBEDTLS_ECP_DP_SECP521R1)
- int sha_alg,
- mbedtls_ctr_drbg_context *ctr_drbg, reference to CTR_DRBG context structure for random number generation
- int iterations, number of times that the test will be executed

Output: Returns 1 if successful, or an MBEDTLS_ERR_XXX_XXX error code

int test_ECDSA(...)
It generates an ECDSA key pair, computes the ECDSA signature of a random hashed message (of 1024 bytes) and verifies the generated signature. Then, it prints the elapsed for each signature and verification in a CSV format. (Everything is repeated N times).
Input:

- int ecparams, the identifier of domain parameters (curve, subgroup and generator),(possible values: MBEDTLS_ECP_DP_SECP256R1, MBEDTLS_ECP_DP_SECP384R1 or MBEDTLS_ECP_DP_SECP521R1)
- int sha_alg, digest algorithm (possible values: MBEDTLS_MD_SHA256, MBEDTLS_MD_SHA384 or MBEDTLS_MD_SHA512)
- mbedtls_ctr_drbg_context *ctr_drbg, reference to CTR_DRBG context structure for random number generation
- int iterations, number of times that the test will be executed

Output: Returns 1 if successful, or an MBEDTLS_ERR_XXX_XXX error code

B.1.2 Test BBS+ signatures scheme

For this subproject, the most relevant files are:

- `Cargo.toml` where dependencies are added, such as `bbs` and `rand`.
- `main.rs` where tests for BBS+ key generation, signature and verification are defined.
  - `fn key_gen_test(iterations: usize)`
  - `fn simple_sign_ver_test(iterations: usize)`

Figure B.2 describes in Rust language how to measure the elapsed time of a cryptographic primitive.

```
fn <primitive>_test(iterations: usize) {
  let mut sum: u128 = 0;
  for i in 0..iterations {
      let start = Instant::now();
      // bbs+ cryptographic primitive
      let duration = start.elapsed().as_millis();
      sum += duration;
  };
  println!("AVG Time elapsed is: {:?}", sum/iterations as u128);
}
```

Figure B.2. Pseudo algorithm for measuring the elapsed time.

The program does the following, it launches the test for key generation and the test for signature and verification, operations are executed `iterations` times and the average is displayed on the output video.

```
fn main() {
  let iterations: usize = 100;
  key_gen_test(iterations);
  simple_sign_ver_test(iterations);
}
```

B.2 Proof of concept

B.2.1 Guide to Keystone Components

The Keystone repository consists of several sub-components such as gitmodules or directories. This is a brief overview of them.

```
+ keystone/
|-- patches/
| # required patches for submodules
|-- bootrom/
| # Keystone bootROM for QEMU virt board, including trusted boot
   chain.
|-- buildroot/
| # Linux buildroot. Builds a minimal working Linux image for
   our test platforms.
|-- docs/
| # Contains read-the-docs formatted and hosted documentation,
   such as this article.
|-- riscv-gnu-toolchain/
| # Unmodified toolchain for building riscv targets. Required to
   build all other components.
|-- linux-keystone-driver/
| # A loadable kernel module for Keystone enclave.
|-- linux/
| # Linux kernel
|-- sm/
| # OpenSBI firmware + Keystone security monitor
|-- qemu/
| # QEMU
+-- sdk/
  # Tools, libraries, and example apps for building enclaves on
     Keystone
```

B.2.2 Guide to Keystone Demo Proof of concept

The designed solution has been developed starting from the officially `kestone-demo` repository available at this link: https://github.com/keystone-enclave/keystone-demo, commit `8c6c0565e44f3d0e00bc3e4a6e77fc84c9e6d343`. The demo uses test keys and is not safe for production use.

```
+ keystone-demo-poc/
|-- docs/
| # Contains read-the-docs formatted and hosted documentation,
   such as this article.
|-- include/
| # Contains shared files between eapp and client
|-- provisioning/
| # C program to generate new client and root key pairs
|-- scripts/
| # Contains a script for performing the attestation of sm and
   eapp
|-- server_eapp/
| # small enclave server that is capable of remote attestation,
   secure channel creation, and performing a simple
   word-counting computation securely
|-- sodium_patches/
| # Contains patch for libsodium that will run in the server eapp
+-- trusted_client/
  # simple remote client that connects to the host, validates the
     enclave report, constructs a secure channel, and then can
     send messages to the host for computation.
```

B.2.3 Relevant files of the demo

Below are explained relevant files that change from the official `kestone-demo` repository. Functions, types and variables not mentioned here can be deepened in the official `kestone-demo` repository and Keystone documentation.

include/eh_shared.h

This file is shared between the enclave and the untrusted host. It defines the following data structure for exchanging sealed (encrypted) data between the enclave and the untrusted host.

```
typedef struct stored_data_t{
  unsigned short file_type;
  unsigned char client_pk[crypto_kx_PUBLICKEYBYTES];
  size_t c_len; // content len
  unsigned char content[]; // Flexible member
} stored_data_t;
```

include/messages.h

This file is shared between the client and the enclave and it defines the messages (request and response) they exchange.

```
typedef struct request_message_t {
  unsigned short request_type;
  unsigned char secret[SECRET_LEN];
  size_t len;
  unsigned char payload[]; // Flexible member
} request_message_t;

typedef struct response_message_t {
  unsigned short response_type;
  size_t len;
  unsigned char payload[]; // Flexible member
} response_message_t;
```

include/session_context.c and include/session_context.h

The session context is the structure that the client provides to the enclave to prove that is built by a trusted manufacturer. With the function `session_context_from_buffer` the session context is extracted from the received buffer.

```
struct session_context_t {
  unsigned char dh_public_key[PUBLIC_KEY_SIZE];
  unsigned char challenge[CHALLENGE_SIZE];
  unsigned char data_signature[SIGNATURE_SIZE];

  unsigned char client_public_key[PUBLIC_KEY_SIZE];
  unsigned char root_signature_of_client_pk[SIGNATURE_SIZE];
};

void session_context_from_buffer(struct session_context_t*
   session_context, unsigned char* buffer);
```

`int session_context_verify(...)`
Input:

- `struct session_context_t session_context`, the session context to verify
- `unsigned char* challange`, the challenge that the server has sent to the client
- `const unsigned char* root_public_key`, the public key of the manufacturer

Output: an `int` value, 1 if it is a valid session context, 0 if not.

server_eapp/service.c and server_eapp/service.h

The file `service.h` just exposes the function that processes the request received by the client.

```
response_message_t* process_request(request_message_t *request,
   size_t *pt_finalsize) {

  setup_sealing_material(request->secret);

  switch (request->request_type) {
    case SERVICE_GEN_KEYS:
      return generate_public_keys(pt_finalsize, EdDSA);
      break;
    case SERVICE_STORE_VC:
      return store_verifiable_credential(pt_finalsize,
         request->payload, request->len);
      break;
    case SERVICE_GET_VP:
      return get_verifiable_presentation(pt_finalsize,
         request->payload, request->len, EdDSA);
      break;
    default:
      ocall_print_buffer("Invalid request type!\n");
  }
  return NULL;
}
```

`int store_data (...)`
It saves the sealed data in untrusted non-volatile memory.
Input:

- `unsigned char* buffer`, data to save
- `size_t len`, length of the data to save
- `unsigned short file_type`, type of the data to save (possible values: `FILE_CLIENT_KEYS_SIGNATURE` or `FILE_CLIENT_VC_SIGNATURE`)

Output: an `int` value, 1 if the operation is successful, 0 if not.

`unsigned char* seal_data_and_sign (...)`
It encrypts and signs the data to save in untrusted non-volatile memory.
Input:

- `unsigned char* data`, data to encrypt and sign
- `size_t data_len`, length of the data to encrypt and sign
- `size_t* sign_len`, a pointer to store the actual length of the signed message

Output: the signed message, which includes the signature plus an unaltered copy of the message.

`response_message_t* build_response (...)`
It builds the response to sand back to the client.
Input:

- `size_t* pt_finalsize`, pointer to store the final length of the response
- `unsigned short response_type`, type of the response (possible values: `SERVICE_GEN_KEYS`, `SERVICE_STORE_VC`, `SERVICE_GET_VP`, `MSG_EXIT`)
- `unsigned char *buffer`, response to send back to the client
- `size_t len`, length of the response

Output: the built response

`response_message_t* generate_public_keys (...)`
It handles the request of the client to generate two key pairs. It saves the key pairs sealed in the untrusted non-volatile memory and sends back to the client only the public part.
Input:

- `size_t* pt_finalsize`, pointer to store the final length of the response
- `int key_type`, the key type that the client chooses to generate

Output: the built response

`response_message_t* store_verifiable_credential (...)`
It handles the request of the client to store a verifiable credential that the client obtained from an issuer. It saves the verifiable credential sealed in the untrusted non-volatile memory and sends back to the client a `0` if everything goes right.
Input:

- `size_t* pt_finalsize`, pointer to store the final length of the response
- `unsigned char* vc`, verifiable credential to be stored
- `size_t vc_len`, length of the verifiable credential to be stored

Output: the built response

`response_message_t* get_verifiable_presentation (...)`
It handles the request of the client to generate a verifiable presentation to use when interacting with a verifier. It retrieves from the untrusted non-volatile memory the previously stored verifiable credential and key pairs and sends back to the client the sign of the verifiable credential with one using an assertion key of the type that the client chose.
Input:

- `size_t* pt_finalsize`, pointer to store the final length of the response
- `unsigned char* nonce`, *(optional)* nonce that the verifier asks the client to insert in the verifiable presentation
- `size_t nonce_len`, length of nonce
- `int key_type`, the key type that the client previously chose to generate

Output: the built response

server_eapp/server_eapp.c

In this file, the important functions to mention are:

void attest_and_establish_channel()
In this function, as explained in Chap. 4, the enclave sends its report to the client and waits for the session context of the client. Once received, it validates the session context and established the channel with the client. If some error occurs, the program terminates.

```
void attest_and_establish_channel() {
  // ...
  randombytes_buf(challenge, CHALLENGE_SIZE);
  // ...
  ocall_send_report(buffer, MAX_REPORT_SIZE);

  unsigned char session_ctx_buffer[SESSION_CTX_SIZE];
  ocall_wait_for_client_session_ctx(session_ctx_buffer,
     SESSION_CTX_SIZE);
  // signatures validity check
  validate_session_context(session_ctx_buffer, challenge);

  channel_establish(); // Ask libsodium to generate session keys
     based on the received pk
}
```

void handle_requests()
In this function, as explained in Chap. 4 and in Sect. B.2.3, the enclave will wait for client requests. Whenever a request arrives, the enclave copies the request from the shared region, processes the request (interacting with the enclave host for data sealing), and sends the response back to the client. If some error occurs, the program terminates.

```
void handle_requests() {
  struct edge_data msg;
  while(1){
    ocall_wait_for_request(&msg);
    // ...
    copy_from_shared(request, msg.offset, msg.size);
    // ...
    switch (request->request_type) {
      case MSG_EXIT:
        ocall_print_buffer("Received exit, exiting\n");
        EAPP_RETURN(0);
      break;
      default:
        response = process_request(request, &r_size);
      break;
    }
    if (response == NULL) {
      ocall_print_buffer("Response handling error\n");
      EAPP_RETURN(0);
    }
    // ...
    channel_send((unsigned char*) response, r_size, boxed_buffer);
    ocall_send_reply(boxed_buffer, boxed_size);
    // ...
  }
}
```

trusted_client/trusted_client.c and trusted_client/trusted_client.h

In this file, the important functions to mention are:

`int gen_session_context(...)`
It generates the client session context that will contain a data part and the signature of the client's public key made by the system administrator at the provisioning phase. The data part is composed of a key for the Diffie Hellam key exchange protocol and the challenge sent by the server, all the data part is signed by the client's public key.
Input:

- `byte* buffer`, a pointer to the buffer that will contain the session context

Output: an `int` value, 1 if the operation is successful, 0 if not.

`request_message_t* generate_request_message(...)`
It builds the request message to request a service to the server.
Input:

- `char* buffer`, a buffer of the data to send to the server (it can contain the key type or the verifiable credential)
- `size_t buffer_len`, length of the data
- `size_t* finalsize`, a pointer to store the final length of the request message
- `unsigned char* secret`, the secret that the server will use to generate the sealing key in the key derivation function
- `unsigned short request_type`, the request type that the client has chosen to send to the server (possible values: `SERVICE_GEN_KEYS`, `SERVICE_STORE_VC` or `SERVICE_GET_VP`)

Output: the built request message

`request_message_t* generate_exit_message(...)`
It builds the request message to close the connection with the server.
Input:

- `size_t* finalsize`, a pointer to store the final length of the request message

Output: the built exit request message

Bibliography

[1] The Laws of Identity, https://www.identityblog.com/stories/2005/05/13/TheLawsOfIdentity.pdf

[2] The Path to Self-Sovereign Identity, https://github.com/WebOfTrustInfo/self-sovereign-identity/blob/master/ThePathToSelf-SovereignIdentity.md

[3] S. Wilson, N. Moustafa, and E. Sitnikova, "A digital identity stack to improve privacy in the iot", 2018 IEEE 4th World Forum on Internet of Things (WF-IoT), Singapore, 2018, pp. 25–29, DOI 10.1109/WF-IoT.2018.8355199

[4] T. inevitable rise of self-sovereign identity, https://sovrin.org/wp-content/uploads/2018/03/The-Inevitable-Rise-of-Self-Sovereign-Identity.pdf

[5] Decentralized Identifiers (DIDs) v1.0, https://www.w3.org/TR/did-core

[6] Verifiable redentials Data Model v1.1, https://www.w3.org/TR/vc-data-model

[7] T. Berners-Lee, R. Fielding, and L. Masinter, "Uniform resource identifier (URI): Generic syntax," RFC-3986, January 2005, DOI 10.17487/RFC3986

[8] E. Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format," RFC-8259, December 2017, DOI 10.17487/RFC8259

[9] JSON-LD 1.1, https://www.w3.org/TR/json-ld11/

[10] Introduction to Trust Over IP (v2.0), https://trustoverip.org/wp-content/uploads/Introduction-to-ToIP-V2.0-2021-11-17.df

[11] A. survey of zero-knowledge proofs with applications to cryptography, http://www.austinmohr.com/work/files/zkp.pdf

[12] The Difference Between Blockchains and Distributed Ledger Technology, https://towardsdatascience.com/the-difference-between-blockchains-distributed-ledger-technology-42715a0fa92/

[13] Crypto FAQ: What is Distributed Ledger Technology (DLT)?, https://cryptocurrencyworks.com/faq/what-is-distributed-ledger-tech.html

[14] T. Tangle, https://assets.ctfassets.net/r1dr6vzfxhev/2t4uxvsIqk0EUau6g2sw0g/45eae33637ca92f85dd9f4a3a218e1ec/iota1_4_3.pdf

[15] Introduction to Trusted Execution Environment: ARM's TrustZone, https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html

[16] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments", Proceedings of the Fifteenth European Conference on Computer Systems, New York, NY, USA, 2020, pp. 1–16, DOI 10.1145/3342195.3387532

[17] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted execution environment: What it is, and what it is not", 2015 IEEE Trustcom/BigDataSE/ISPA, Helsinki, Finland, 2015, pp. 57–64, DOI 10.1109/Trustcom.2015.357
[18] Keystone Paper and Customizable TEEs, https://keystone-enclave.org/2019/07/22/Keystone-Paper.html
[19] S. of a trusted computing base (TCB), https://apps.dtic.mil/sti/pdfs/ADA108831.pdf
[20] RISC-V Specification Documentations, https://riscv.org/technical/specifications/
[21] What is Keystone and its first open-source release?, https://keystone-enclave.org/2018/12/13/what-is-keystone.html
[22] P. M. Protection, https://sifive.github.io/freedom-metal-docs/devguide/pmps.html
[23] Keystone Enclave's documentation, https://docs.keystone-enclave.org/en/latest/index.html
[24] Product Specifications of Discovery kit for IoT node, https://www.st.com/resource/en/data_brief/b-l4s5i-iot01a.pdf
[25] Mbed TLS, https://github.com/Mbed-TLS/mbedtls
[26] The BBS Signature Scheme, https://github.com/decentralized-identity/bbs-signature
[27] BBS+ signature scheme Rust library, https://crates.io/crates/bbs
[28] BLS12-381 for the rest of us, https://hackmd.io/@benjaminion/bls12-381
[29] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, "High-speed high-security signatures", Journal of cryptographic engineering, vol. 2, August 2012, pp. 77–89, DOI 10.1007/s13389-012-0027-1
[30] Libsodium, https://github.com/jedisct1/libsodium
[31] STM32CubeIDE, https://www.st.com/en/development-tools/stm32cubeide.html
[32] Install RUST, https://www.rust-lang.org/tools/install
[33] CMAKE, https://cmake.org/