POLITECNICO DI TORINO

Master degree course in Ingegneria Informatica

Master Degree Thesis

Integration of the DICE specification into the Keystone framework

Supervisor
Prof. Antonio Lioy
Ing. Sisinni Silvia
Ing. Bravi Enrico

Candidate
Valerio Donnini

Luglio 2023

To my family, that believed
in me since the beginning

Summary

The scope of this thesis is to implement a design that permits the correct implementation of the Device Identifier Composition Engine (DICE) in Keystone.

The first three chapters have been used to write the stare of the art of the different topics debated. The first one is related to the most important characteristics of Device Composition Engine (DICE), its hardware requirements and the different type of architecture on which it is based on. The second chapter talks about the Trusted Execution Environments, briefly explains what is and what is its feature. The rest of the chapter presents the different technologies that are actually on the market and a platform that produce standards about this topic. The third one, instead, is associated to the presentation of the Keystone Project, the project on which the thesis is based, showing how it works, its features and its weaknesses.

The practical part has been focused on the implementation of the DICE specification in the Keystone project making all the things needed to obtain the better results: the fourth chapter describes the design that have been chosen and in the fifth the implementation with description of this design has been provided.

The last part is associated to the tests that have been made to check the correctness of what have been done.

Acknowledgements

A special thanks to the professor Antonio Lioy that permits me to work on this topic and assists me over the entire thesis.

A lot of gratitude also to Dr. Silvia Sisinni and Dr. Enrico Bravi that follow me over the months and help me with the development of the thesis.

An enormous thanks go to my parents because without them, I will never be here because they give to me all the economic support and also they let me live free of pressure in my academic travel.

A big thanks also to all my friends, both the new ones meet in this two years and the old ones, that provide me a way to have fun and to be carefree.

A special mention also for my brother, Michele and his fianceé, Tania, that have filled my heart of happiness with their beautiful daughter, Sofia.

Last but not least, a gigantic thanks to Federica, that goes with me since the beginning of this year, that allows me to really understand what to be happy means and has changed my life in a way that I never expected.

Contents

Chapter 1

Introduction

During the last years, lots of new devices have been introduced in the market, most of them rely on the category of IoT. These products, differently from the standard ones, are characterized by hardware specifications that are not so powerful, due to different reason, for example, the minimal space or the necessity to maintain the cost as low as possible.

For these reasons to achieve strong security features in these devices is not so easy: a help to obtain them in this field can be the Device Identifier Composition Engine (DICE), that is a security standard that has been made by the Trusted Computing Group in order to obtain enhance security and privacy on systems with a Trusted Platform Module, but also to have viable security and privacy foundation in all the systems without a TPM. More in detail, it can be used to check the integrity of the software that is run in a device and also do some form of remote attestation.

This standard is not so easy to be implemented in most of the commercial Trusted Execution Environment (TEE), that have not the possibility to be modified, because they are proprietary of specific companies and for sure not open-source. This not happens with Keystone enclave, a framework that can be used to run customizable TEEs completely open source and so completely changeable.

Keystone is a framework that has been developed by the Linux foundation and one of its most important characteristic is that it can be run in general purpose processor that respects the standard of the RISC-V architecture, doesn't having the constraint to be associated to a specific one and so, it can be associated to systems like the IoT devices.

Implementing the DICE core in it, will be generated a package that can be used in lots of different platforms, from the most powerful to the light one, obtaining important security features that guarantee the correct execution of a system, or in the other case, provide a way to understand if something goes wrong and make operations to correct the situation: can be used for example to check if a specific TEE (called Enclave) is under the control of a malicious actor or also to check at boot if the platform is in a secure state.

In this work, all the arguments cited will be analyzed, showing the principal features of each one and talking related to the TEEs, also the differences from the actual products present in the marked are described. In the second part, a possible design and its implementation of the DICE specification in the Keystone project is proposed, comparing how it affects the performance with the respect to the original work.

Chapter 2

Device Identifier Composition Engine architecture

2.1 What DICE is

DICE stands for Device Identifier Composition Engine and it is a standard created by the Trusted Computing Group (TCG) [1]. This group associates to this topic the DICE Architecture Work Group and the goal of this project is to address the security problems related to the Internet of things, targeting products such as MCUs and system on a chip (SoCs).

This type of architecture can be integrated without increasing the silicon requirements using the hardware of security products during manufacturing. The most important thing of this type of architecture is that it can be used in systems where traditional Trusted Platform Modules may be unfeasible due to the limitations associated to the cost, power, space and so on.

2.2 Hardware requirements for DICE

This subchapter is used to specify what are the hardware requirements [2] and the process that are needed by DICE to create an identity value that is derived from a Unique Device Secret and the identity (a representation) of the first mutable code (see Fig. 2.1). The value that is derived from this process is called Compound Device Identifier (CDI) and it can be used for different application (for example it can be used to attest the trustworthiness of an embedded device). An important thing to say, is that for the DICE specification, the engine that performs this type of computation (the derivation of the CDI) can be updated, but those updates are not measured in the CDI and must be inherently trusted. Different from this, the first mutable code is the part of the code that is executed after the Device Identifier Composition Engine and for this reason it is not inherently trusted.

The CDI is obtained starting from the Unique Device Secret (UDS) and adding to it the measurement of the first mutable code that runs on the platform after the DICE. This value can also include some information about the hardware state and/or some configuration data that can change how the first mutable code is executed. The engine that generates the CDI is called Device Identifier Composition Engine (DICE). It is the only one that can access the UDS after the reset of the machine before giving the control to the first mutable code. The manufacturer provides the UDS in a way that has to be consistent with the specification. Changing for any reason the value of the UDS, implicate that also the CDI will change. The most important characteristic that the CDI must have is that it has to be calculated in a way that having it and the measure of the first mutable code is not possible to recover the UDS. To have this feature the DICE can be implemented using different types of techniques: the first one is to use a secure hash algorithm [3] to hash the concatenation of the two starting value (UDS and measure).

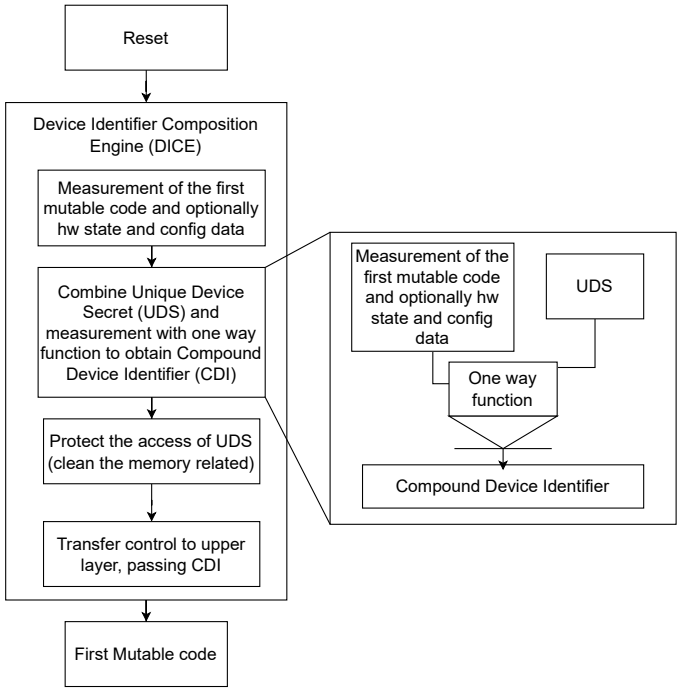

Figure 2.1. Compound Device IdentifierDerivation Process (source: [2])

$$\mathbf{H}(UDS \mid\mid \mathbf{H}(FirstMutableCode))$$

The second one is using a HMAC where the UDS is the key of the computation (using this method it will obtain twice the level of security with the respect to the level of security obtained with the simple hash, but at the same time the operation will take a little more time).

$$\mathbf{HMAC}(UDS, \mathbf{H}(FirstMutableCode))$$

How the two values have to be combined is chosen by the manufacturer and the choice will not affect the interoperability. In this scenario, the device is responsible to do all the things to be sure that the access to the CDI to do operations of reading, writing and changing is protection. One of the most important benefit of the CDI is that it changes each time the first mutable code changes, so for example if there is a malware that replace the first mutable code, the CDI that is computed by the DICE and probably passed to the malware is not the original CDI that were computed when the application were not affected by it. For the same reason the CDI obtained will be different from the original one if the first mutable code is updated with a security patch. The update process of the first mutable code has to be implemented in a way that it can be done only with the "assistance" of the manufacturer. It means for example that the updates of the first mutable code have to be signed with a private key, for which the public part is owned by the manufacturer. These requirements can be used in two different types of immutability of the DICE. In a simple device the DICE and all its dependencies can be invariant and not change after manufacturing. In more complex system, instead, can be implemented a DICE that can be manipulated directly or indirectly by the manufacturer. According to the DICE specification, these changes have not to be reflected in a modification of the resulting CDI. This changes can be needed to balance risks associated to complex systems. The changes to the DICE and to its dependencies are the basis for confidence to see if there are some modification in the UDS, in the first mutable code or in some resulting measurements. The types of protection mechanisms that are used to update the DICE and other stuff have to be inherently trusted. From the strength of this mechanism derives the ability of the customer to trust the CDI. The UDS has to respect these properties:

- Uncorrelated and statically unique

- Each entity has its own UDS
- The security strength of the UDS has to be at least the same of the attestation process
- If the attestation process is not under the control of the manufacturer, the UDS must have a length of at least 64 bytes
- Can not be rewritten

Instead, the DICE has to respect these types of properties:

- Has to be the only one that can access the UDS
- If there is a debug mode, it has to be active after the DICE
- If there is a debug mode, it can not be used to read the UDS

These are the basis properties that all the DICE must have, then, dependently from the type of DICE implemented other ones can be needed.

A first distinction that can be made of the DICE, is that if it is mutable or immutable. In the first case:

- The DICE updating process has to be secure and controlled by the manufacturer
- After the updating process, the new DICE has to respect the specification
- The UDS should not to be changed during the manufacturer controlled update process

In the second one:

- The DICE becomes immutable from the end of the manufacturing process of the device

So, making a recap, the different operations that the DICE can do are the following:

- It is executed without any interference after each reset of the device before any mutable code of the platform
- The UDS has to be combined with the measure of the first mutable code before its execution
- The CDI has to be created with a one-way function (at least the same security strength of the attestation process or 64 bytes in case this process is not present)
- After that the CDI is calculated and before the first mutable code is executed, the UDS shall be set in a way that is impossible to use it until the next reset.
- The DICE has to erase all the data that can used to obtain info of the UDS before the first mutable code is executed
- The CDI has to be written in a place in memory where the first mutable code has exclusive access until it requires exclusive access

2.3 DICE Layering architecture

This type of architecture [4] uses DICE as the basis of a multi-layered Trusted Computing Base to which is associated a hardware Root of Trust (RoT) that has to be inherently trusted because if it doesn't work properly, this type of failure can not be detected. One of the most important feature of a layered TCB is that each layer can provide trusted functionality to the upper layer and to obtain this, only a minimal set of functions are needed.

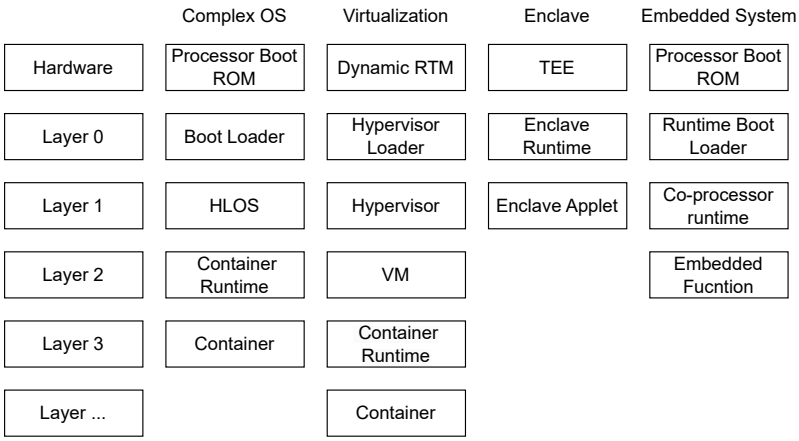

Figure 2.2. Examples of system layering (source: [4])

This type of architecture consider the different execution states started from a based hardware layer that is assumed to be in a trustworthy state before going to layer 0. The same thing is considered to be true when there is the transition from the layer 0 to the layer 1, so in general when there is a movement from a layer N to a layer N+1, the layer N is considered to be in a trustworthy state. Moving from a layer to another layer for the DICE architecture means that the associated CDI is computed and securely passed to the next layer.

To construct the next layer on this type of architecture, a set of TCB capabilities is needed and are protected in a hardened execution environment (as it is shown in Fig. 2.3). The passage from a layer to another one is protected because each layer uses interaction capabilities that are trusted by both the parts. Each TCB layer must have access to TCB capabilities used to:

- Produce TCB Component Identifier (TCI): measure that is component specific and describe it (for example a hash value)
- Calculate CDI: the CDI of a layer n has to be associated at least to two different values: the CDI of the previous layer and the TCI of the target TCB component. These two values are combined together using a one-way function. In this scenario the UDS is used to provide a statistically unique value to the DICE HRoT because no other previous context exists. Both the mechanism used by a TCB component to produce the CDI of another layer and how the CDI is given to the upper layer are to be trustworthy
- Use a one-way function [5]: it is a cryptographic pseudo-random function (PRF) that complies with NIST SP800- 56c recommendations. These types of functions work with a seed and a data. In the described scenario, the seed has to be the CDI obtained by a previous layer of the actual layer and the data has to be the TCI of a subsequent component.

An important concept to underlying in this scenario is the DICE layered identity. This type of identity lives only with a specific chain of TCB components due to the nature of how the different CDI of each layer are calculated and because it is from that the identity is derived. This approach is different with the respect to the randomly generated identity that can be associated to a component or to a device. The CDI of a specific component is obtained also from the CDI of the previous component, and the CDI of the previous component is obtained from the CDI of the previous component and so on, so the identity that is derived from a specific layer CDI represent not only TCB components but also their order because if two component will be swapped also the final CDI and the derived identity will change. The DICE HRoT has to respect some requirements:

- The security strength of the UDS must be sufficient for its usage
- The UDS and the measure of the layer 0 have to be used for the computation of the CDI of the layer 0

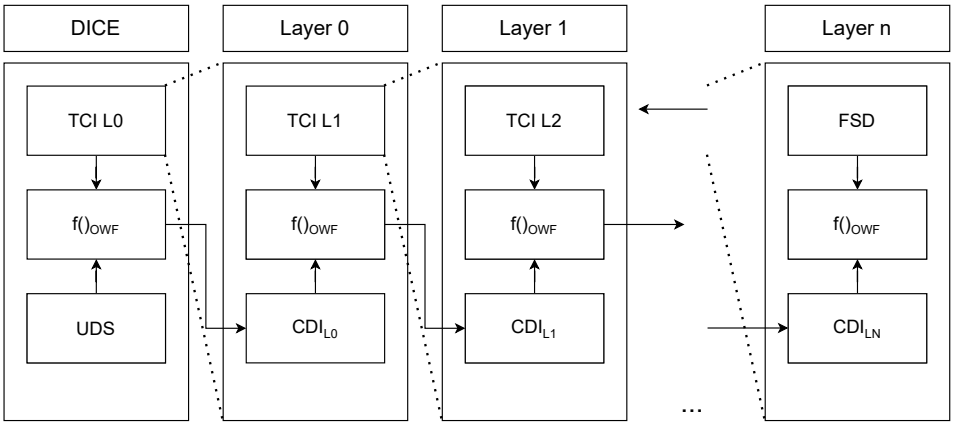

Figure 2.3. TCB layering architecture (source: [4])

- The DICE have to access only the CDI value, not other layer secrets
- The trustworthiness properties of the DICE HRoT is asserted by the DICE manufacturer of by the vendor
- DICE must have securely access to secrets not external visible
- If there is a process to generate keys, then the DICE HRoT must have a secure entropy source

Also the layer 0 has to respect some requirements:

- Include all the common requirements for the other layers
- The DeviceID key is generated with a process that is controlled by the manufacturer
- The DeviceID key is derived from the CDI value that describes the layer 0
- Associated to this key there should be a certificate issued by the manufacturer
- If the layer 0 is not modifiable out of a process controlled by the manufacturer, then also the DeviceID key and its certificate can not be changed
- If the layer 0 changes, then also the DeviceID key changes

In general all the different layers must respect these requirements:

- A DICE layer has to be placed in shielded location and it has to be built with protected capabilities
- A DICE layer has to be placed in shielded location and it has to be built with protected capabilities used by previous layers
- Each DICE layer has its own CDI and its own private keys. All of that have to be kept secret to other layers
- A DICE layer can not implicit trust a layer that is executed after it
- All the secrets that a layer has, have to be created by the layer itself or provided by the previous layer
- The keys associated to the layer n have to be derived directly by the layer n or given to the layer n by the previous layer

- The CDI of the layer n has to be used to compute the CDI value of the layer n+1
- If there is a layer that wants to implement a CA, then this CA has to be an Embedded Certification Authority (ECA), that can be used to certify the keys associated to specific layer following ECA defined procedures
- If the attestation process of a layer is guaranteed by a device, it means that the attestation is computed directly by the layer n or by some below layers

2.3.1 TCB capabilities

Some important TCB capabilities that must be present in a DICE layering architecture are the following: certification, attestation, authentication

Certification and token issuance

Certification of a DICE layered component means issuing a certificate or a token that can be used to bind the next layer TCB to the current one. A specific layer can generate the public keys for the next one and certify them. A layer must have the capabilities to certify keys if this feature is needed and the certification can be done in two different ways dependently from if the keys to certify are asymmetric or symmetric. In the first case an X.509v3 [6] cert has to be issued, in the second case the keys are issued in tokens.

If asymmetric keys are used, the certification can be done directly from the TCB layer n, that generates the keys for the TCB layer n+1 and signs the certificate, or the TCB layer n+1 generates the key pair and obtains a certification from the TCB layer n. If the certification process uses tokens (when the keys are symmetric), this token has to be built depending on the TCI of the next layer TCB, the next layer CDI and on a symmetric key that is derived from the current CDI. No specification for the generation of the symmetric key to use is given

Attestation

This procedure is associated to the attestation of a DICE layered component using a symmetric or asymmetric key that has been approved by the Embedded Certificate Authority for this porpoise. It is used to prove trustworthiness properties of a TCB layer of a component. Trustworthiness properties can be: implicit or explicit. Implicit means that these properties are inferred by a verifier and depend on some condition or state that wouldn't otherwise be possible. Explicit means that the properties are explicitly enumerated and encoded to be ready for the inspection made by an attestation verifier. When a DICE TCB layer supports this feature means that it will analyze a sequent layer TCB or a sequent component to obtain its trustworthiness properties. When the TCI associated to a specific layer is computed, in this measure there are trustworthiness properties associated to code and settings needed for the execution of a subsequent layer. An inspecting TCB layer can also create and certify with its ECA or sign with a specific attestation key some form of attestation evidence about a major layer that can be used later by the attester.

Authentication

Authentication of a DICE layered component means the usage of an asymmetric key or a symmetric key that has the porpoise to authenticate the device. This type of capability is inserted in a TCB layer if the DICE layer has an active role in the device authentication protocol.

2.3.2 Keys and credentials

Each TCB can use different types of keys for different porpoise: there can be asymmetric keys [7], symmetric keys [8] and each one of this category can be used for several targets. The asymmetric keys are divided in:

- **Embedded Certificate Authority keys**: these types of keys are used to issue (sign) a TCB component certificate for other keys that are derived for the current layer or the upper one. This keys can only be used to sign data that are known by the current TCB layer. Also, this type of keys can be considered to be an implicit statement of layered identity if they are generated from the CDI of the current layer with its TCI.
- **Attestation keys**: these types of keys are used to sign the attestation evidence that a layer can obtain about a major level. Like for the ECA keys, this keys can also be used to sign data that must be known by the TCB layer. This type of keys can be considered to be an implicit statement of layered identity if they are generated from the CDI value of the current layer.
- **Identity keys**: this type of key is used for signing authentication challenges
- **DeviceID key**: it is an asymmetric key that is obtained starting from the CDI computed by the DICE. It is strictly related to the UDS and to the measure of the Layer 0 and can be used to sign the certificates that can be issued for keys of upper layers. It is considered to be also an ECA key and an attestation key and is certified by the manufacturer with its private key and its certificate is provided (usually) in ROM.
- **Alias key**: it is an asymmetric key that is obtained starting from the last CDI value in the chain of the TCB components. Together with its certificate, it usually contains the information that can be used to attest top-level device firmware. It is also an attestation key because it can be used to sign attestation evidence.

If a key pair has to act as a proof of implicit layered identity, the seed starting from which the key pair is derived must contain the measurement of the TCB component that it identifies. The symmetric keys instead can be:

- **Symmetric Alias Key**: this type of key can be obtained from the CDI value and optionally from a PSK ID Hint that has to be chosen by the verifier. The pre-shared secret in this scenario is inserted for Symmetric Key Attestation and Layered Identity
- **Wrapping Keys**: this type of key can be used when it is not so comfortable to regenerate asymmetric keys on each boot. So, the wrapping key is used to persist previously derived asymmetric keys. For example, this type of key can be used to avoid that on each boot the DeviceID keys are regenerated: from the CDI, it can derive a symmetric key that can be used to store encrypted the value of the DeviceID.

Both the symmetric and the asymmetric keys can be created in different ways from the CDI values. Starting talking about the generation of asymmetric keys, this process (see Fig. 2.4) can generate keys that can be used to attest trustworthiness properties of a TCB layer. The keys are generated from the CDI of a specific layer, so they implicitly represent the layering semantics.

The most common algorithm that are used to derive asymmetric key pairs are:

- **ECDSA** [9]: this algorithm can be used to provide deterministic key pair choosing a seed that should be based on the specific TCB context and derived from the CDI of the layer. Doing so, a random number is obtained, and this number will be used like the seed of the ECDSA key generation function
- **RSA** [10]: the generation function can be full of a value that is generated starting from the current layer CDI value

The symmetric keys can be derived using a Key Derivation Function where the seed of this function is directly the UDS or the CDI obtained from it (see Fig. 2.5). The length of the CDI has to be enough to avoid overlapping problem with the derived symmetric keys.

Talking about the generation of the keys, some considerations have to be made according to the security, to the protection and to the management of them in a layered architecture. If an

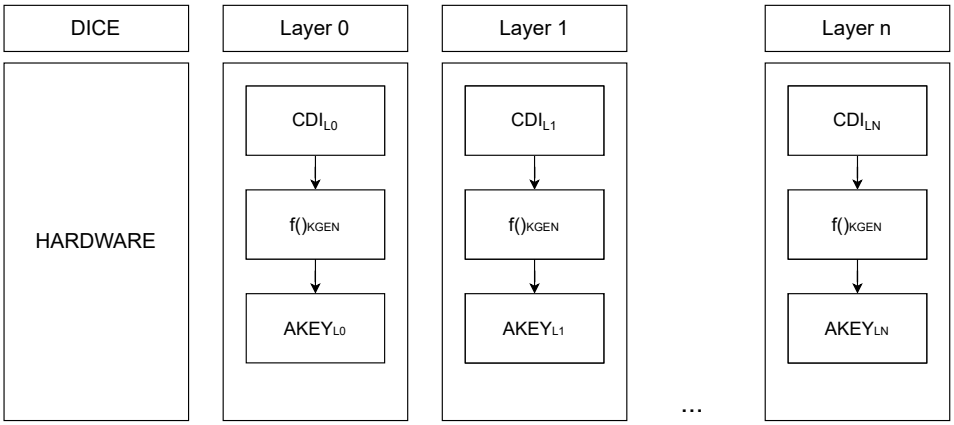

Figure 2.4. Asymmetric key generation example (source: [4])

attacker can access the CDI or the TCI or any other TCB context values, he can be able to derive or generate keys for a specific layer and impersonate it. For this reason all the stuff needed to protect the various TCB context values have to be implemented. In particular the private part of a key has never to be exposed above the layer that is considered to be trusted to protect it and all the location in memory that are used to store sensitive data have to be erased before the key protection responsibilities are passed to the next DICE layer.

There can also be some problems if there is storage of the private keys, because in this way also if there are some changes in the code, and the sequent CDI value changes, the persistent key that was stored derived from the previous value of the CDI, not change. It will represent a configuration that it is not more in use. This means that also the representation of the actual identity is not correct like the behavior and the trustworthiness state of the TCB layer. All the keys and the secrets used to generate them have to be placed in shielded locations and don't go outside of them. At the same time specific key pairs of a layer may not be regenerated on every reset cycle, due to computation reasons for example. To avoid this, wrapping keys can be used.

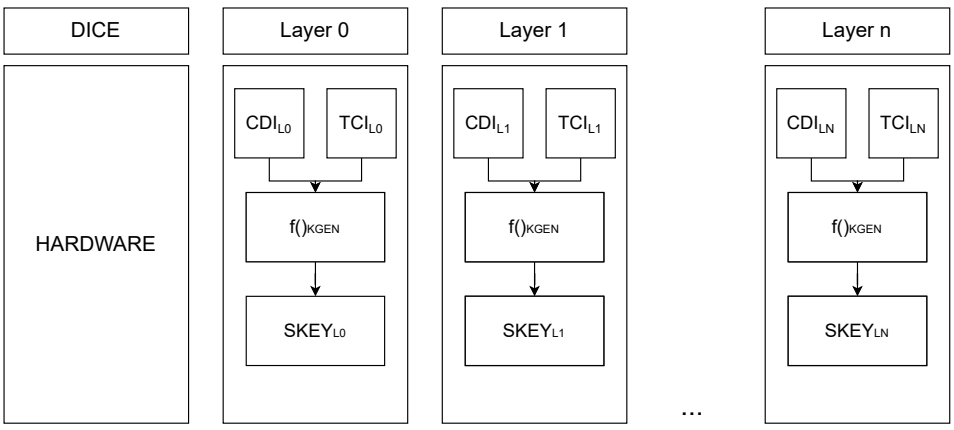

Figure 2.5. Symmetric key generation example (source: [4])

2.3.3 Layered Certification

Another important point of DICE layered architecture is the Layered certification. In this scenario is the manufacturer to certificate the device trust properties of the DICE HRoT. The manufacturers that are intended to produce DICE devices have to implement a CA hierarchy where the

manufacturer itself is considered to be the root CA and he can have one or more subordinate CAs that issue two different type of certificate:

- **Device identity**: certificate that is used to check that the device is not under the control of malicious actors because it is used to authenticate evidence (states of current configuration and so on). For example, it can be used by a verifier to be sure that it is interacting the expected device
- **Attestation certificates**: certificate used to assert that the manufacturer has embedded a cryptographic key in a device.

The Dice architecture may rely on Public Key Infrastructure (PKI) for device provenance. It is a duty of the device manufacturer to implement a certificate hierarchy that can used to generate certificates. It has to publish certificates for the device attribute or manifest containing trustworthiness assertion that are associated to DICE trustworthiness.

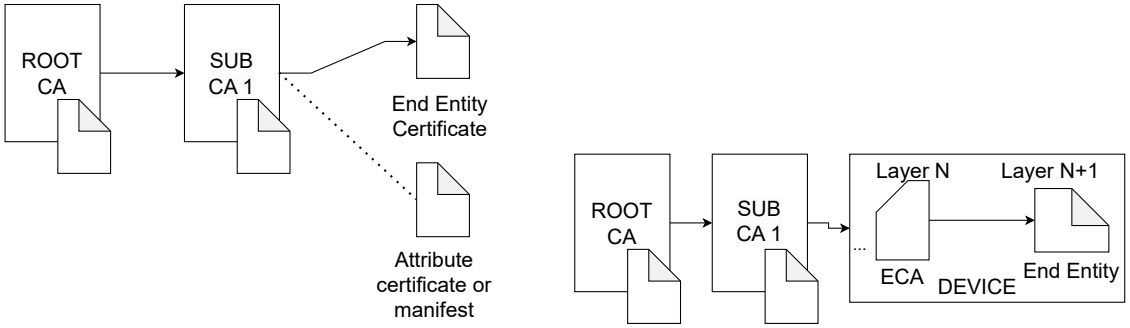

Figure 2.6. Certificate hierarchy with Attribute Certificate or Manifest and Certificate hierarchy with Embedded CA (source: [4])

The DICE TCB layering architecture has the possibility to produce certificate at any layer, starting from layer 0 using both external and embedded CAs. It also anticipates attestation as a precondition to certificate issuance. The ECA can issue certificates that contain attributes or sign manifest structures that have inside attestation information all of them associated to a layer specific end entity certificate (see Fig. 2.7).

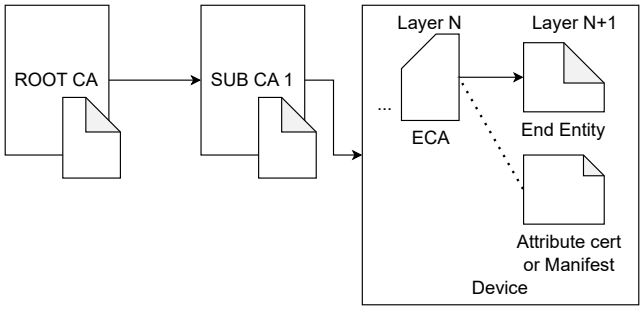

Figure 2.7. Certificate hierarchy with Embedded CA (source: [4])

The certification process can be done by ECA inside a DICE layer TCB and it is associated to keys that have to be certified for allowing higher DICE layer and external entities to verify trustworthiness at or below the DICE layer in which there is the Embedded Certification Authority. If there is a consumer of an ECA issued certificate, he needs to trace all the cert chain of the DICE layers until the DICE manufacturer and he expects also to have the manufacturer's certificate.

A specific example of the layered certification can be the scenario presented in the Fig. 2.8 where in the layer 0 there is an ECA that is used to issue a cert for the layer 1. The same thing

is repeated between the layer 1 and the layer 2. In the layer 0 the manufacturer also provide his certificate that can be used as initial device identity.

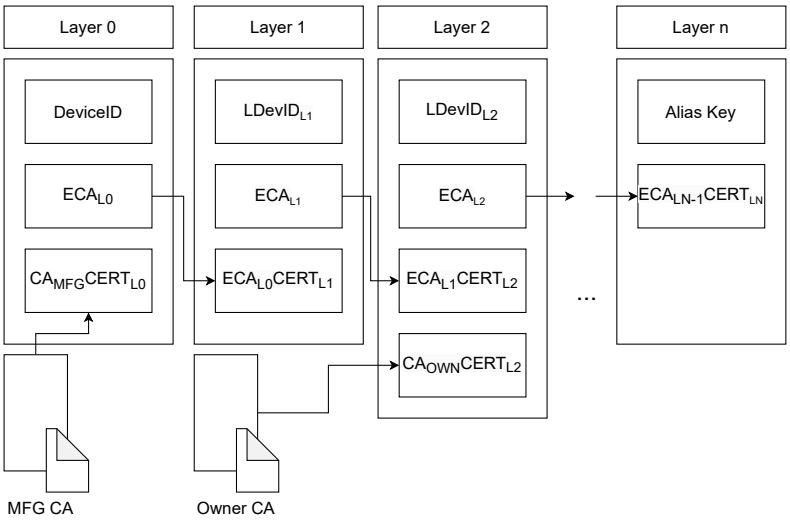

Figure 2.8. Layered certification example (source: [4])

The certification can be done in two different ways:

- using an Embedded CA: a layer issues a certificate for a higher layer to extend trust. There are two different models of embedded CAs: the ECA can decide when issue a certificate or the upper layer asks for the sign of a specific certificate (Certificate Signing Request CSR)
- using an External CA: the DICE TCB layer interact with some external CA to obtain device identities that can be provided during manufacturing or when the device is into a network. The most commonly used approach is the first one where the manufacturer can provide both the device keys and the device identity. Another option is that the device keys are not provided by the manufacturer but generated from the device and the identity credential is obtained after that the device made a specific credential creation request.

Certification with Embedded CA: direct layered Certificate

With this specific configuration the ECA can emit certificates respecting some form of policy that can be associated directly to the ECA firmware or that was configured previously in a secure way. This type of policy is used to specify how and when the ECA can issue specific layer certificates. What happens is that the ECA generates the key pair that has to be certified and then it will certify them giving the possibility to the upper layer to secure access to the key pair or providing it directly. The different steps to take this are the following (see Fig. 2.9):

1. the layer n has to measure the upper layer n+1 to obtain TCB identifier TCI n+1
2. After that the TCI is computed, the layer n uses it to obtain the CDI value for layer n+1
3. Starting from the CDI n+1 the layer n derives a key pair for the layer n+1
4. The layer n emits a certificate for the key pair generated with its ECA
5. The layer n gives all the data to the layer n+1 (CDI, certificate and optionally the private key, because the layer n+1 can re-derive it starting from its CDI)

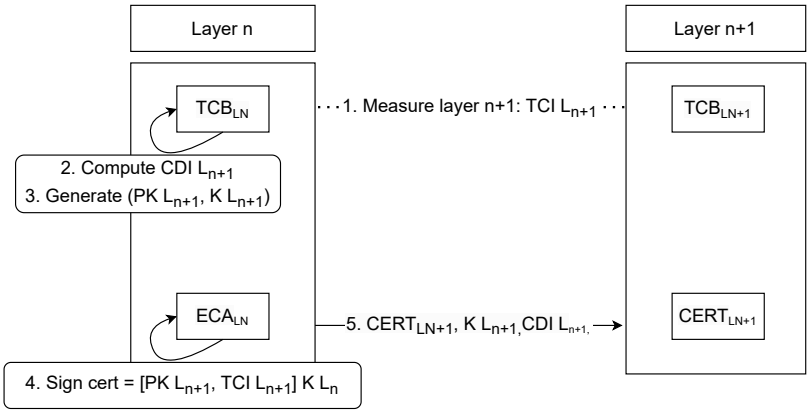

Figure 2.9. Direct Layered Certification by an ECA (source: [4])

Certification with Embedded CA: Layered TCB Certification using CSR

With this specific configuration is an upper layer that asks a previous layer ECA to certify a key pair after that the previous layer ECA has verified that the TCB component is the same of the CSR subject inserted in the request. The different steps needed to do this are the following (see Fig. 2.10):

1. the layer n has to measure the upper layer n+1 to obtain TCB identifier TCI n+1
2. After that the TCI is computed, the layer n uses it to obtain the CDI value for layer n+1
3. The layer n has to give the CDI previously computed to the layer n+1
4. The layer n+1 uses its CDI value to create a key pair
5. The layer n+1 has to build a CSR containing all the information that can be useful for the layer n to verify the TCB component of the layer n +1
6. The layer n, after that the CSR is received, has to verify that the signature of this request has been made with the private key derived from the CDI of the layer n+1 using the public key inserted in the CSR
7. The layer n recompute the CDI of the layer n+1 and check that it is the same that the layer n+1 has inserted in the CSR
8. The layer n generates the key pair starting from the CDI value of the layer n+1 and check that the public key is the same that the layer n+1 has inserted in the CSR
9. The layer n issue a certificate for starting from what is has received inside the CSR

Some considerations can be made talking about the ECA certificate issuance:

- an ECA has not the duty to remind the list of the certificates that have been issues. To avoid the re-issue of the same certificate, the issuance can be based on some attribute that make the process deterministic like for example the serial number and so on
- The usage of the ECA signing key has to be limited because in contrary its lifetime will be decreased due to cryptanalysis
- An ECA has the opportunity to emit certificates for keys that have different usage related to the RFC5280 [6] KeyUsage constraint
- An ECA has not the duty to proper manage a certificate revocation request

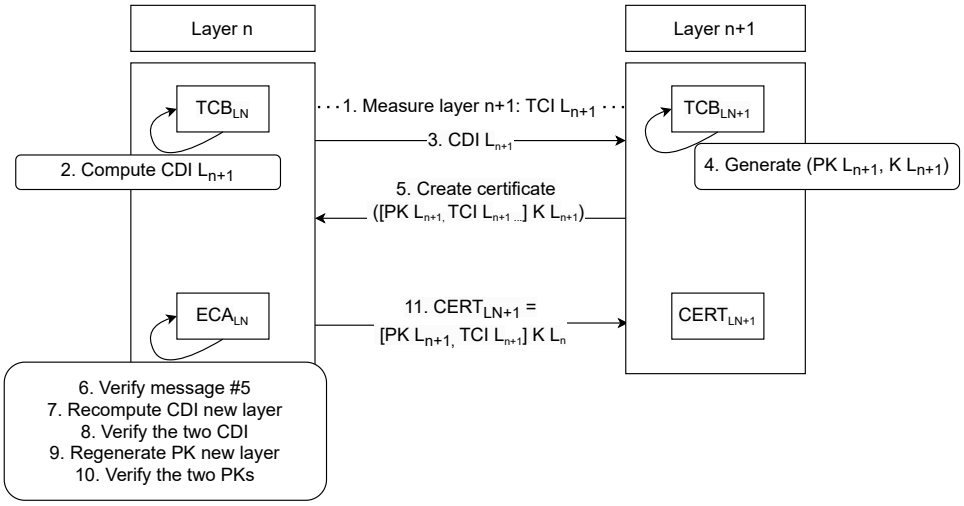

Figure 2.10. Layered TCB Certification using a CSR (source: [4])

Certification with external CAs: issuance of LDevID certificate by the owner

This is the scenario where there is a device owner that wants to issue a local device identity using a CA that is under his choice after having taken the possession of the device, from a supply chain entity. The owner can also choose the layer that has to be used for the operation and depending on it, a CSR is created using a local public key and all the chain until the RoT has to be given to the owner to supply attestation evidence. If they are sufficient, then the owner produces, using its CA, a certificate for the supplied public key.

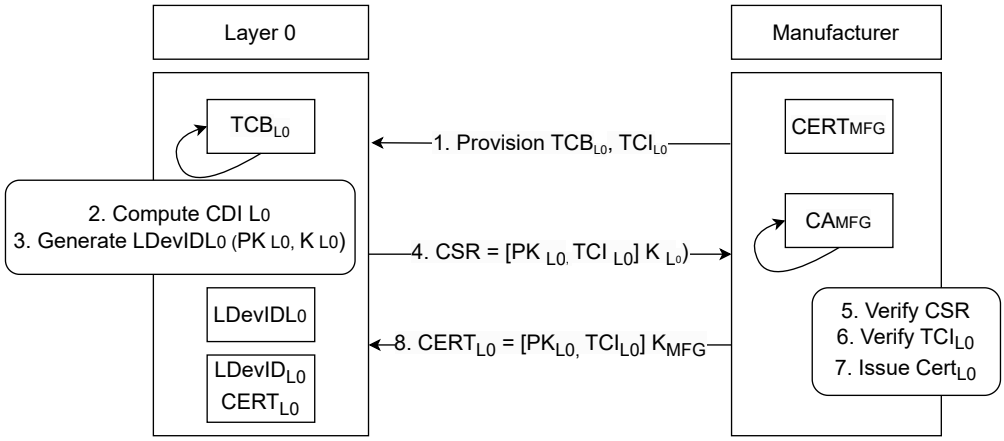

Figure 2.11. Example initial device identity (IDevID) certification by a manufacturer (source: [4])

The onboarding and ownership acquisition steps are (see Fig. 2.11):

1. A specific layer creates a public key (lDevLN) that has to be certified
2. A nonce is given by the owner to the device
3. The specific layer gives back to the owner all the certificate chain generated by the device's ECA and the nonce. This message, that is the attention message, is signed with the specific layer attestation key
4. The CSR signature is verified by the owner

5. The certificate chain is verified by the owner
6. A certificate for the public key provisioned to the owner is issued by the owner CA
7. The owner CA gives back to the specific layer the issued certificate for the lDev pk given before

Design consideration

When a layered DICE architecture has to be implemented some guidance have to be kept in mind related to:

- **Privacy**: some applications need to interact with a single cloud infrastructure during their lifetime that is explicitly aware of the identity of each device. In other case this type of scenario can be unacceptable for some vendors and users and for this reason the firmware has to be built in a way that the possibility to be tracker are the lowest as possible. Some strategies to minimize tracking are for example to continuing the key derivation and certificate chain beyond the Alias key or recycling Alias key hiding the device certificate.
- **External communication**: the communication over the network is not so simple to be implemented in the first layer of the DICE architecture and for this reason the DICE HRoT and layer 0 should be kept as simple as possible without having this feature
- Possibility to do a **factory reset**: it can be implemented, knowing that has not to be possible to do a rollback after that a device has been re-provisioned. There are 4 options to implement a factory reset:
  - Change the uds in the device
  - Change the layer 0
  - Both the first two options
  - If the UDS cannot be changed, the CDI has to be derived from some other information that can be modified implementing the factory reset.

2.4 DICE attestation architecture

The DICE attestation architecture [11] is a form of architecture that include both implicit and explicit attestation, defined by the Trusted Computing Group. In this type of architecture there is a set of roles associated to actors that are needed to complete the attestation process. Dependently on the deployment model used, an actor can combine together or separate different roles but they not change the attestation roles or the responsibilities of each one (see Fig. 2.12).

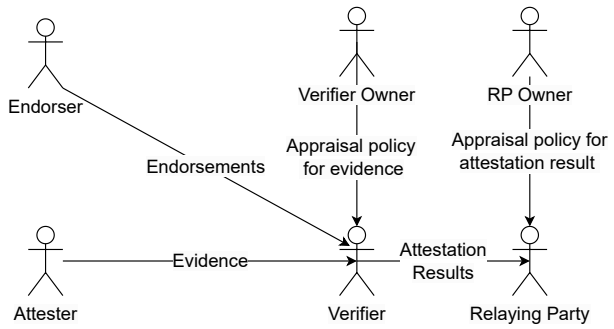

Figure 2.12. Attestation Roles and message flow (source: [11])

To work properly, some certificate extensions need to be defined to construct some evidence or some reference values. For this type of architecture the functions used are:

- Creation of attestation evidence
- Conveyance of attestation evidence
- Appraisal of attestation evidence

The different roles that interact together are the following:

- **Attester**: is the role that provides attestation evidence to the verifier (see Fig. 2.13). It is associated to a specific identity, called attestation identity established during the manufacturing process, that is used to authenticate the Evidence that has to be sent to the Verifier.

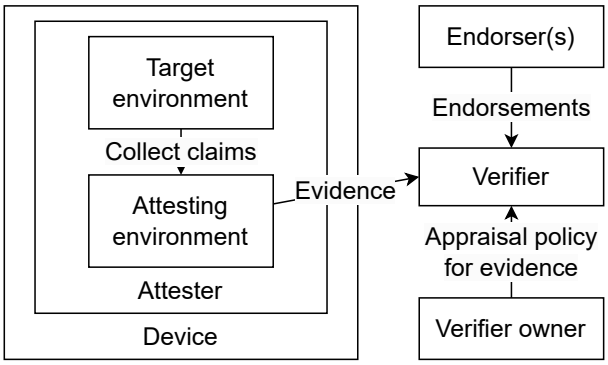

Figure 2.13. Device with Attesting Environment and Target Environment (source: [11])

  It is composed by an Attesting Environment and a target Environment. The first one is used to obtain assertion called claims about the second one associated to its trustworthiness properties. Implementing DICE architecture each TCB layer can be an attesting environment that can be used to generate some Evidence. If to the layer N is associated the attester role, the layer N-1 attests the state of the layer N, the layer N-2 attests the state of the layer N-1 and so on until the layer 0. The Target Environment is the layer N+1 with the respect to the layer N that is the attesting environment

- **Endorser**: a role that is usually built with a supply chain that is needed to have reference endorsements each one that contains some form of assertions about the device's intrinsic trustworthiness properties. This role is used to implement all the stuff needed to establish the trustworthiness properties of the testing device. The association between the DICE layers and the Endorser is not for sure 1 to 1, for example there can be more than 1 endorser for the same DICE layer
- **Verifier**: used to collect Endorsements and Evidence and redirect Attestation result to Relying party/ies. Usually it is a service provider entity
- **Verifier Owner**: role used to set the policy that have to be followed by the verifier. It sets the reasons to choose between an acceptable and unacceptable Evidence and Endorsements. It also may have a type of storage where the different endorsements are saved
- **Relying party**: role used to accept Attestation Results from a Verifier. It evaluates the Attestation result following the Appraisal policies defined by the Relying party Owner and can decide to do something
- **Relaying party Owner**: the role used to decide the policies to determine which attestation results are acceptable and unacceptable and to give these policies to the relaying party

Associate to the different roles, also different type of role messages can be exchanged in this type of architecture (role message stands for message that consist of assertions about the trustworthiness properties):

- **Evidence**: messages sent by the Attester containing claims
- **Appraisal Policy for Evidence**: messages contains policy used as input in the Verifier to decide what to do associated to trustworthiness Claims in Evidence
- **Endorsements**: messages contain assertions that are signed by an actor that is playing the endorser role
- **Attestation Results**: messages containing the result of the attestation Evidence appraisals and following the policies defined by the Verifier Owner. The verifier protects with authenticity, integrity and confidentiality this type of message
- **Appraisal Policy for Attestation Results**: messages containing policy used as input in the Relaying party to decide what to do associated to trustworthiness Claims in Attestation Result

Also, different type of message exchange pattern can be used:

- Passport model: it simulates the real case of the emission of a passport (see Fig. 2.14). There is a passport holder with his identity credentials and gives them to the passport issuer that builds the passport document. The different steps in this type of topology are the following:
  - The Evidence message is presented by the attester to the verifier. What is inside the message is validated with the specific policies and then the verifier produces an attestation evaluation result. This result is signed by the verifier and or is built with something that allows the Relaying Party to authenticate the results arrived from the verifier
  - The results are sent to the Attester and then later delivered to the Relaying party. It authenticates and then process them.

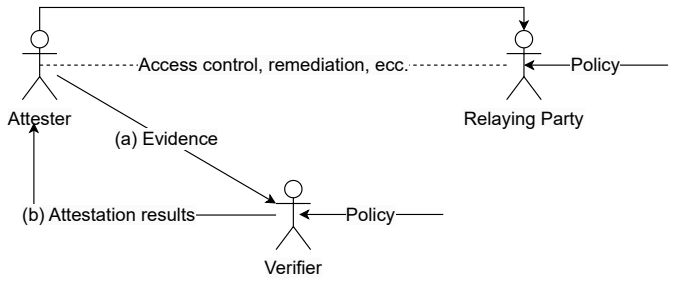

Figure 2.14. Passport Topology Model (source: [11])

- Background Check Model (see Fig. 2.15): it simulates the case where who receives the credentials, has not the possibility to directly process them. So they have to be passed to a background entity that does the job. Differently from the previous topology the steps are the following:
  - the Evidence message is received by the Relaying party, that ca only check its freshness, integrity and origin so sends it to the verifier
  - the verifier checks the evidence message like in the passport model and then gives the attestation results to the Relying party
- Multipart Background check Mode: topology similar to the second one presented except if there are more than one Relaying party. The interaction between the different parts preserves the pattern described in Attestation Roles Architecture diagram.

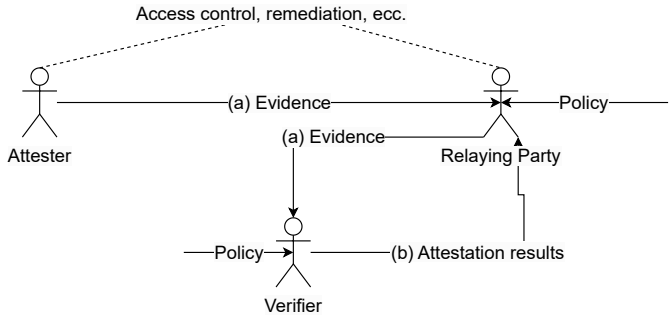

Figure 2.15. Role Interactions - Background Check Topology Model (source: [11])

In this type of architecture different strategies can be used to assign roles to actors (see Fig. 2.16). Usually the different actors use some form of interface or protocol that are needed to ensure the correct communication of the exchanged messages. This conveyance mechanisms can be local or remote. The first one is when the same actor is associated more than a role, so the communication of messages to different roles are internally managed by the same actor. It means also that the co-resident roles trust protocol for the authentication, protection and transmission of the role messages.

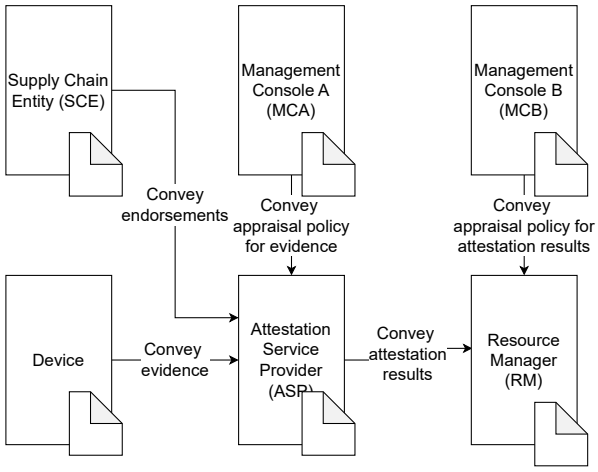

Figure 2.16. Attestation Actors (source: [11])

Another situation that can happen, is the situation where two or more Actors are combined or co-located and the roles play by each one don't form collectively a new hybrid one. Some examples can be:

- Co-located Verifier and Relying Party Example
- Composite Attestation Example
- Local Verifier Example
- Layered Device Attestation Example

2.4.1 Layered device attestation

Type of attestation that is performed by the different DICE layers, where a layer n attests the state of the layer n+1 using some form of Evidence associated to the layer n+1 signed by the layer n (see Fig. 2.12).

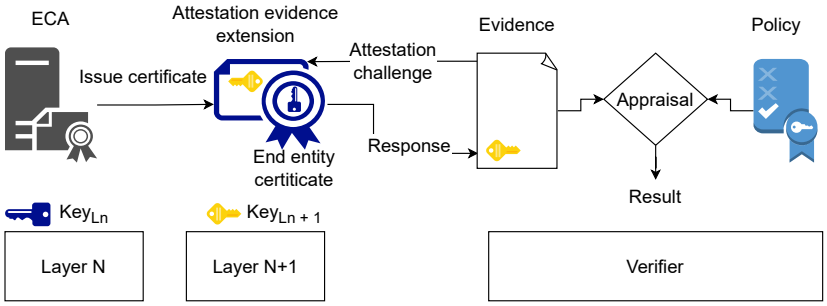

Figure 2.17. Layered Attestation (source: [11])

There are different possible solution to provide evidence to a Verifier; three different approaches can be used:

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

An extension that can be used is the TCB Info Evidence Extension that contains attestation Evidence about the DICE layer to which the subject key is associated. If this extension is used, it has to be marked s critical. An alternative that can be used when the initial state of a DICE TCB is associated to multiple measurements, can be the Multiple DiceTcbInfo Structures Extension where there is a sequence of DiceTcbInfo structures, one for each measurement. Another extension is UEID Evidence Extension that can be used when the content of this extension is used in the generation of the CDI. An evidence can be created starting from an X.509 certificate attribute, signed it with a specific key called attestation key. It is generated by the Attestation environment, the environment that controls the attestation key, starting from the target environment, or alternately can be created starting from a manifest always signed with the attestation key. A verifier to check the proof of the Evidence, usually uses Endorsements, the reference value made by the manufacturer or the supplier for the checking process. Like the evidence, the Endorsements can also be encoded with different techniques:

- X.509 identity certificate extensions containing Endorsement values
- X.509 attribute certificates containing Endorsement values
- CoSWID (Concise Software Identification Tags) manifest containing Endorsement values
- SWID (Software Identification (SWID) Tagging) manifest containing Endorsement values.

Talking about the attesting environments, it must ensure that if there are non-constant fields, each one of this, is usually used to derive the associated CDI value to be sure that there is the consistency between the actual state described by the Evidence and what has been evaluated.

Chapter 3

Trusted Execution Environment (TEE)

3.1 Introduction to Trusted Execution Environment

Nowadays, the traditional security technology are no longer sufficient to satisfy the different security requirements of various architectures. This is the main reason for why there is a new trend that is characterized by the integration of trusted computing concepts in different types of systems, like for example embedded systems [12].

Trusted computing was born to help the systems to obtain secure computation, privacy and the protection of the data. At the beginning, the trusted computing was associated to specific separate hardware module that exposes interfaces to obtain platform security. The Trusted Platform Module (TPM) is used to provide a proof of the integrity of a specific system and also allows a secure management of cryptographic keys (they are stored in tamper-evident hardware module).

The principal problem of the TPM is that it can't be used by a third party, so it can expose only a predefined set of APIs. To overcome this problem, a new approach based on the execution of arbitrary code within a specific environment that guarantees tamper-resistant execution has been developed. This type of environment can be called in different ways, for example closed-box VM, operator virtual machine (OVM), TrustZone [13] software (TZSW), and trusted language runtime, but the most used one is become trusted execution environment (TEE).

A more specific definition of Trusted Execution Environment can be the following: a TEE is a secure, integrity-protected processing environment, consisting of memory and storage capabilities. Nevertheless, there is no official definition associated to this term and for this reason during the years TEE has been bind with different explanations:

- Ben Pfaff, Terra [14], 2003, "The TEE is a dedicated closed virtual machine that is isolated from the rest of the platform. Through hardware memory protection and cryptographic protection of storage, its contents are protected from observation and tampering by unauthorized parties."
- OMTP, Advanced Trusted Environment [15], 2009, "The TEE resists against a set of defined threats and satisfies a number of requirements related to isolation properties, lifecycle management, secure storage, cryptographic keys and protection of applications code."
- GlobalPlatform, TEE System Architecture [16], 2011, "The TEE is an execution environment that runs alongside but isolated from the device main operating system. It protects its assets against general software attacks. It can be implemented using multiple technologies, and its level of security varies accordingly."
- Jonathan M. McCune, Trustworthy Execution on Mobile Devices [17], 2013, "The set of features intended to enable trusted execution are the following: isolated execution, secure storage, remote attestation, secure provisioning and trusted path."

Comparing the different reported sentences, it can be clear that the world *isolated execution* and *secure storage* are directly related to the Trusted Execution Environments. But they are not aligned in total because for example in the first definition the secure storage is lined with the feature of having *states cryptographic protection*, while in the third sentence the concept associated to the secure storage is more general and it refers only to the needed to have some space to protect the assets. It can be also seen that the first definition also say something about the isolation and how it has to be bind with the integrity and the confidentiality of the TEE runtime states. Other consistent differences are with the respect to the threat model: there is no specification in the first and the fourth definition, instead in the third one there is an ambiguous reference to all the software attacks while in the second one the threat model is precisely detailed defining all the attacks that must not damage a TEE.

All these definitions not center the most important aspects and on the contrary they seem to be a little ambiguous in some parts. For this reason a new definition of TEE has been given, but before that, some concepts have to be defined.

3.1.1 Prerequisite: Separation Kernel

One of the most important component related to the TEE is the separation kernel. It is needed because it is the part that provides the isolated execution. Its main goal is to allow the presence in the same platform of different systems that needed different level of security. What it does is essentially the division in different partition of the entire platform and provides strong isolation between the different parts.

The security requirements that are needed to implement this are defined in "the Separation Kernel Protection Profile (SKPP)". Differently from the traditional security kernels, the separation kernel is more simple and it is used to guarantee the division in terms of space and time. The security requirements are characterized by four main security policies:

- **Data separation**: each partition can access only its data
- **Sanitization**: if there are some shared resources, they can't be used to achieve information between different partitions
- **Control of information flow**: the communication between different partitions have to be explicitly allowed
- **Fault isolation**: a security breach in a partition has to remain in the specified partition and doesn't go the other.

3.1.2 TEE Definition

The newer definition that can be done associated to a TEE is the following: a tamper-resistant processing environment that is associated to a separation kernel. It is used to guarantee that the code that has to be executed is authentic, the runtime states are right (associated to the integrity) and the code, data and runtime states are saved in persistent storage securely with confidentiality. A TEE may be able to do remote attestation that can be used by the environment to prove its trustworthiness to third parties. It can be also updated in a secure way and has to be able to resist to all the software or physical attacks that can be done on the main memory of the system. Backdoor security flaws cannot be used to attack the system.

The TEE protects its runtime states and stored assets and it can be updated changing its code and data. It has also to be associated to something that can be used to securely attest its trustworthiness to third-parties. All the attacks that can be made or in the main memory or on its non-volatile memory are inserted in the threat model.

3.1.3 What "trust" means

A crucial aspect talking about the TEEs is the concept of *trust*, because for example it can be used to compare two different TEE. In the computer world, this word assumes the meaning of something that is behaving as expected. Trust can be:

- **static**: means that an evaluation based on a specific set of security requirements has been made
- **dynamic**: means that the states associated to a running systems have to be associated to states that can be considered to be trust

More in detail, dynamic trust can be linked to the concept that there are some proof about the trust state of a specific system. So, in this particular scenario, the concept of trust can be associated to the concept of secure. However, to achieve this, there is the necessity to have an entity that is called Root of Trust (RoT) that gives trustworthy evidence associated to the state of a platform. The RoT has the duty to:

- do trusted measurements
- compute trust score

For sure the Rot has to be tamper-resistant hardware module and can be implementing using different types of know-how. How it is implemented is directly bind with the hardware platform that has been chosen to provide the isolation properties inside the separation kernel.

Specific talking about the TEE, the trust can be considered hybrid: before the real deployment, following a protection profile, a TEE has to be certified and after this on each boot, the RoT controls that the TEE that is has been loaded is the same with the respect to the one that was certified by the manufacturer. When the TEE is running, there is the separation kernel that guarantees its integrity and for this reason the trust associated to the TEE is semi-dynamic: the trust level associated to the TEE is not supposed to change during execution. The measurements that are done are integrity measurements and the trust score is simply a boolean flag that is used to know if the state has been modified or not (true if the TEE is trust, false otherwise).

3.1.4 TEE building blocks

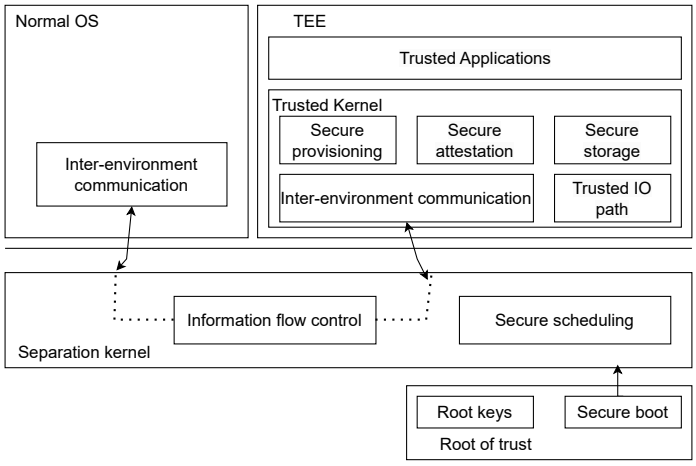

Figure 3.1. TEE building blocks (source: [12])

A TEE is composed by a set of building blocks (see Fig. 3.1):

- **Secure boot**: process that is used to ensure that only code with a certain property is executed. If a change in the code is captured, the boot process is stopped. One of the most used techniques to implement the secure boot is to check the integrity of the following component comparing the measurement with a reference value, usually provided by the manufacturer. Designing secure boot means implementing chain of trust because it is usually composed by a set of different stages. This chain can be represented as follows:

$$I_0 = true$$
$$I_{i+1} = I_i \wedge V_i(L_{i+1})$$

  Where I_i is associated to the integrity of the layer i, instead V_i is the function that has to be used for the verification. The cryptographic hash of a layer has been made with the specified function and then the results are compared with the value provided. Because of the nature of this implementation, if the integrity of the initial boot code can not be verified, also any other integrity check goes wrong. For this reason to the initial boot is associated a tamper-evident hardware module.

- **Secure scheduling**: used to be sure that the execution of the TEE does not affect the performance of the main OS

- **Inter-Environment Communication**: the interface that is needed to allow the communication between the TEE and the rest of the system. It is a necessary component but at the same time introduce new threats: for example it introduces message overload attacks. There are different mechanism that can be used to implement this component, the important thing is that each one of this mechanism has to respect this attribute:
  - reliability
  - minimum overhead
  - protection of communication structures

- **Secure storage**: storage that provides confidentiality, integrity and freshness of the data inserted. The access to this storage is allowed only to authorized entity. This is usually implemented with sealed storage

- **Trusted I/O Path**: paths that are used to have a secure communication between the TEE and the peripherals. Also, the data that are exchanged have to be protected from some types of attacks like for example sniffing attacks or tampering attacks. More in detail, this type of component is used to obtain the protection against:
  - screen-capture attack [18]
  - key logging attack [19]
  - overlaying attack
  - phishing attack [20]

3.2 CPU based Trusted Execution Environments

In this section of the second chapter, different solutions to implement a TEE in a CPU will be presented. The full list of the various technologies that will be analyzed are the following:

- **x86 System Management Mode** [21]
- **ARM platforms**: TrustZone [13]
- **Intel platform**s: Software Guard Extention (SGX) [22]
- **AMD platforms**: Secure Encrypted Virtualization (SEV) [23]
- **IBM Z** [24]

- **RISC-V Keystone** [25]

Will be also discussed something about the Standards and frameworks to provide unified Application Program Interfaces (APIs) like for example:

- **Global Platform specifications** [26]
- **Open Portable Trusted Execution Environment** [27]

3.2.1 x86 System Management Mode

One of the most important feature to have in a system like Windows 10 is the guarantee about the healthy and the trustworthiness of the firmware platform. If it is true, also other feature like Hypervisor-protected code integrity (HVCI) and Windows Defender Credential Guard will behave as expected. To obtain this, Windows use a hardware-based RoT that protects from the execution of code like Unified Extensible Firmware Interface (UEFI) malware before the bootloader launches.

A way to obtain that the hypervisor and the rest of the system is protected is to avoid that the System Management Mode is compromised. The System Management Mode [21] is an execution mode in x86 based CPUs that runs in a level that is higher than the level of the hypervisor. It is usually used to make interactions with some specific type of hardware, like NV RAM, or to emulate functions associated to hardware, to manage hardware interrupts and so on.

To avoid attacks from obtaining the control of the SMM, the OS must have the guarantee about the correct SMM's behavior. Intel and AMD have developed mechanism that are used to enforce the isolation of the SMM with the respect to the OS and to understand to which resources the SMM has access to.

The isolation of the SMM is composed by three parts (see Fig. 3.2):

1. Original Equipment Manufacturer's (OEMs) are associated to specific policy that are used to understand to which resources they require access
2. This policy are enforced by the chip vendor on System Management Interrupts (SMIs)
3. The compliance to this policy is reported to the OS by the chip vendor

Figure 3.2. SMM isolation (source: [21])

Inside the policy gives by the OEM there is a list that contains the different resources that the SMI handlers requires access to. This policy is not under the control of the OS, it only enforces the policy stated. The chip vendor's reporting mechanism provide the enforced policy to the Trusted Computing Base (Tcb) Launch and it compares the OEM's SMM access policy with different layers of Windows SMM isolation requirements in a way that it is able to understand the level of isolation provided. This level obtained is later used for attestation and it is given to the OS.

Isolation levels means the type of restrictions that an SMI has related to what it can access and can be associated to:

- SMM page configuration lockdown
- Static page tables

- Model-Specific Register (MSR) access
- IO port access
- Processor state save access

The Dynamic RoT Measurement (DRTM) is strictly related with the SMM isolation because if the first one is not present, what has been evaluated during the boot can not be trusted by the OS because it is not protected from the influence of the SMM. During the DRTM the different SMIs are not working because, doing this the DRTM can establish a new RoT and with this, the evaluation of the SMM access policy can be done.

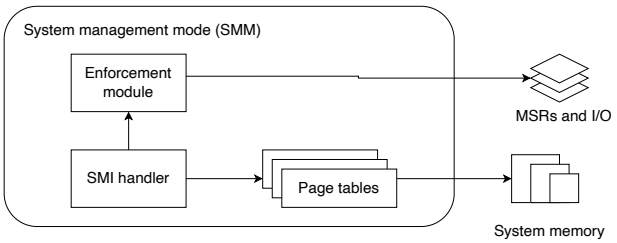

Figure 3.3. SMM interactions (source: [21])

3.2.2 ARM platforms: TrustZone

ARM TrustZone [13] is a hardware security extension technology, that is used to obtain secure execution environment dividing the resources in two different execution world, one that is considered to be trusted, the secure world, and another one that is not considered to be trusted, the normal world. This type of extension is not specific for a particular ARM architecture: it can be implemented in targets that running normal applications, like for example the smartphone or in microcontrollers.

Making a very general overview about the ARM architecture [28], it can be said that it is a RISC (Reduced Instruction Set Computer) architecture and the principal features are the following:

- only a large uniform register file
- the operations associated to data can only be done using registry, not memory
- the addressing mode is simple
- it has some instructions that combine both the shift operation and the arithmetic one
- auto-increment and auto-decrement addressing modes are implemented to obtain better performance with program loops
- can load and store multiple instructions to obtain the best data throughput
- the execution throughput is boost with the conditional execution of many instructions

Talking about TrustZone, it can be said that it is an optional hardware security extension of the different ARM processor architectures. To implement this type of extension, that is based in the division in two different execution world, one secure and the other one not trusted, hardware barriers are implemented in a way that the normal component, situated in the normal world, can not access directly the secure world. More in detail, the implemented memory system has the following features, regarding what the normal world can do with the respect of the secure one:

- if a memory region is designed to be secure, the normal world can't access it

Processor mode	Abbr.	ARM v7 Priv. level	ARM v8 Exc. level	Security state
User	usr	PL0	EL0	Both
Supervisor	svc	PL1	EL1	Both
System	sys	PL1	EL1	Both
Abort	abt	PL1	EL1	Both
IRQ	irq	PL1	EL1	Both
FIQ	fiq	PL1	EL1	Both
Undefined	und	PL1	EL1	Both
Monitor	mon	PL1	EL3	Secure only
Hyp	hyp	PL2	EL2	Non-secure only

Figure 3.4. The different processor modes in ARM v7-A architecture (source: [13])

- system controls related to the secure world, can't be access by the normal one
- the normal world can't access state switching if it is not included in few approved mechanisms

An other thing to say is that, this type of division can be both physical and virtual.

Hardware Architecture of TrustZone

TrustZone has been implemented with some system additions that are used to have the guarantee about the security restrictions but at the same time not consume so much power and respects other advantages ARM's design.

The AMBA3 AXI to APB Bridge is used to have secure communication between a CPU and the peripherals because the Advanced eXtensble Interface (AXI) bus has a bit (NS bit) that specify where the read/write operations are going to be done (secure/normal world). Also, the Cache Controller also looks for the same bit that is usually the 33rd of the address. According to the fact the physical cache is one for the two different world, there will be two different set of addresses that are related respectively one to the normal world and the other one to the secure world. The Direct Memory Access (DMA) can manage at the same time events related to the two different world, giving full support for interrupts and peripherals. The TrustZone Address Space Controller (TZASC) is used to associate dynamically to each AXI slave memory-mapped device if it is secure or not. It is controlled directly by the secure world and it permits the splitting of a unit of memory instead of asking separate secure and non-secure units. The number of partitions that can be created is arbitrary. The TrustZone Memory Adapter (TZMA) is used to split on-chip static memory in secure and not secure, instead the Generic Interrupt Controller (GIC) manages the secure and not secure prioritized interrupts. The last component is the TrustZone Protection Controller that is a signal-control unit.

Software Architecture of TrustZone

From the point of view of the software the secure world can be implemented as an operating system that can be customized Linux or OP-TEE. So the functionality that the device must have, are the following:

- implementation of the proper boot of the two different systems
- implementation of the proper way to make the two different worlds communicate

With the respect to the boot, ARM has implemented a secure boot to avoid that a malicious version can be booted by the device. The secure boot is implemented building a chain of trust, where each step can be cryptographically verified, usually starting with a vendor-specific public key. Recapping the different steps that are done during the boot:

1. the most important peripherals are initialized by the ROM-based bootloader
2. the secure world can be load from a flash drive
3. the other OS boots

After the booting phase, the communication between the two world can be done using a secure monitor that acts as a normal context switch (see Fig. 3.5). The only way that the normal world has to access the secure one is through a hardware interrupt, an external abort signal or the software instruction SMC (guarantee the possibility to pass message without a complete changeover) To simplify the life of the TrustZone software at application level, ARM has also

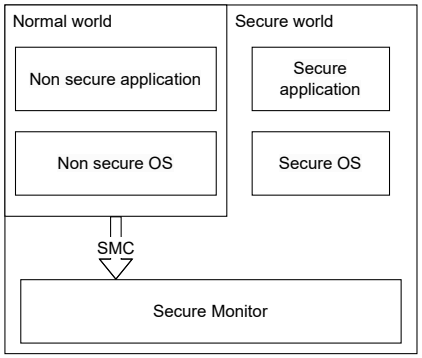

Figure 3.5. Interaction from the normal world to the secure one (source: [13])

published some specification for TrustZone API that is strictly related to the communication from and to the secure world.

In the implementation of the TrustZone for the ARMv8-M architecture, the monitor mode is not provided (see Fig. 3.6). In this way the interrupt latency is reduced because a transition mode has been removed. In this type of implementation, the secure state is defined not with the NS bit but according to where the code that has to be executed is placed. This provides the feature that a non-secure application can call a secure application, simply jumping to a specified memory location.

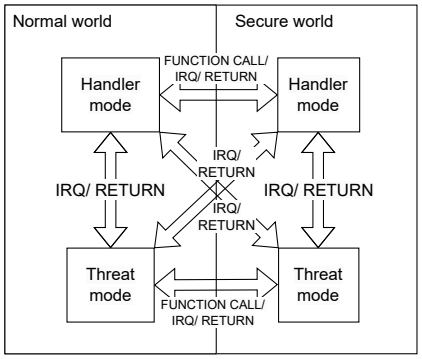

Figure 3.6. TrustZone on ARM cortex-M (source: [13])

3.2.3 Intel platforms: Software Guard Extention (SGX)

Intel SGX [22] is composed by CPU instructions and a set of hardware and it is used to provide user-level applications with specific hardware-enforced confidentiality and integrity protections. It permits to the developer to split their app in different secure containers, referring to them like

enclaves, each one is hardware-protected. An SGX enclave can be considered like an isolated container that is placed inside the running application's address space. The main feature related to an enclave are the following:

- the tamper-resistant property is associated to the code that is executed inside an enclave
- all the data that are associated to an enclave are protected from snooping or disclosure

The memory associated to an enclave has some features that guarantee that it is separated from the rest of the system memory:

- hardware-enforced checks does not permit that non-enclave code can read or modify the data inside the enclave
- hardware-based SGX Memory Encryption Engine (MEE) is used to encrypt and authenticate all the data associate to the enclave before writing them to the untrusted memory

The principal benefits of this type of TEE are the following (see Fig. 3.7):

- Protection from Higher Privilege Levels: all the data and the code associated to a specify enclave is protected from higher privilege levels, like for example OS, VMM and so on. The only entity trusted by the enclave is the CPU hardware and the code that has to be executed in the enclave itself. It derives is in a massive reduction of the attack surface because the application's trusted computing base decrease its size. More over, also if the machine will be compromised, all the data that have been stored in some parts of the non-trusted memory, have the property of confidentiality and integrity, so they can not be disclosure
- Attestation and Sealing: SGX includes some mechanism that can be used to check that the running enclave is legit, both in local or remotely. This guarantees the trustworthiness of the enclave to the challenging entity, that for example has to be known as the entity wants to send to the enclave some sensitive data. All the data are encrypted before storing using a sealing key derived from the SGX hardware and they can be restored only by the same instance of the same enclave running on the same machine

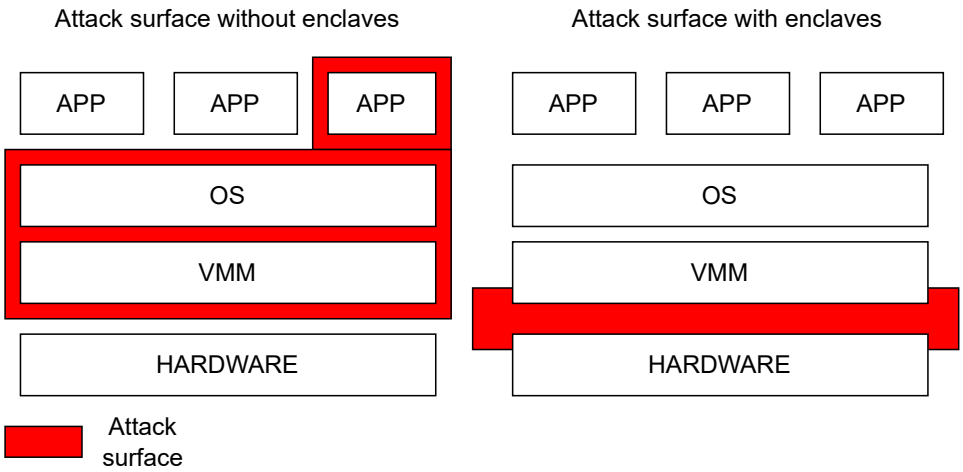

Figure 3.7. Comparison of attack surface between application running with and without SGX enclaves (source: [22])

The principal caveats associated to this technology are the following:

- Ring 3 (no syscalls): certain instruction will be illegal inside an enclave, due to the fact that the protections are only related to user-level (ring 3). All the instruction that are directly related to a VMEXIT can not be executed in an enclave.

- Limited Memory: there is the limitation of 128Â MB to the size of the enclave memory that can be protected
- Overheads: it is associated principally to the operation of encryption and decryption of enclave data when they have to be saved in memory. These operations don't come for free.
- Licensing: to use SGX hardware, the developer has to but a license from Intel. If the license is not bought, the developer has no access to the full functionality of the SGX, because he can only run the enclaves in software simulation or on hardware that has the support, but without having the full confidentiality and integrity protection of the hardware.

3.2.4 AMD platforms: Secure Encrypted Virtualization (SEV)

Secure Encrypted Virtualization [23] is a feature that has been added in the AMD architecture which has been properly built to manage in the proper way the complexity and isolation needed in the modern systems. It is used to boost the isolation through the usage of cryptography and encrypting code and data. Also, this type of technology provide a way to securely protect code from higher privileged code.

The usual security model used in tradition computing is the model associated to different rings each one of them has the possibility to access specific resources, higher is the level, more are the resource that the level can access. The scenario presented is different from what happens in the SEV model (see Fig. 3.8), where code that has to be executed at different level is isolated and for this reason there is no part that can access the resource of the other one. The higher level, such that the hypervisor level, has not the possibility to access the resource associated to lower level because it is protected with cryptographic isolation. This permits the lower level to be more secure without trusting the higher level code. In this scenario the hypervisor can still interact with the guest, but the communication path is tightly controlled.

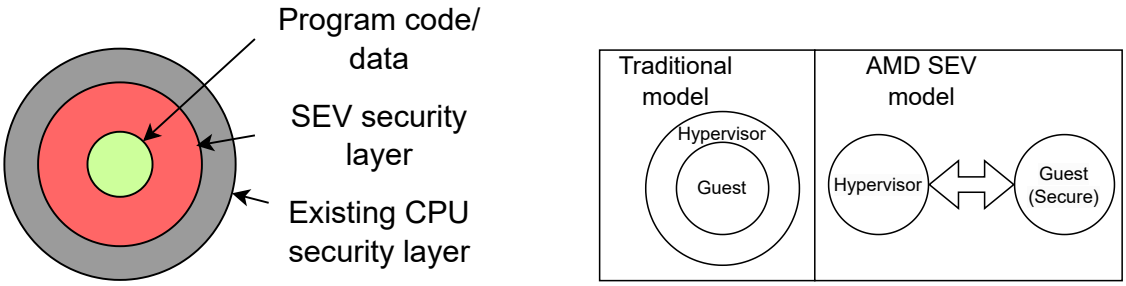

Figure 3.8. Security layers and SEV security model (source: [23])

In the threat model built around the SEV model, consequently, an attacker can also have the possibility to execute malware at higher level code, for example at the level of hypervisor. It can be also assumed that the attacker can have full access to the DRAM and more in general to the physical machine because SEV provides assurance about the protection of the guest from this type of attacks.

SEV use cases

SEV can be used in different scenario, the principals are the following:

- Cloud: SEV technology can be used in the scenario of Infrastructure as a Service, where there is the possibility that different owners of VMs are associated to the same physical machine and if the isolation between the VMs fails, there will be the possibility that there is a leakage of sensitive data. SEV can be used to boost the security features providing

better isolation rooted in the hardware itself. The workloads associated to each customer will be protected cryptographically from the other ones and at the same time it will be also protected from the hosting software. If SEV will be used, it can also help with the situation where there is a malicious administrator that wants to access data related to a specific VM.

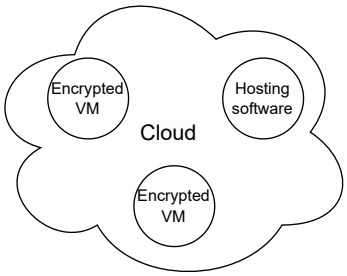

Figure 3.9. Encrypted VMs in clouds (source: [23])

- Sandboxing: SEV hardware can be used to isolate a very large portion of code, like for example a full VM, or to do a better fined isolation, like for example, protecting a container.

SEV architecture

The SEV architecture (see Fig. 3.10) is based on the usage of a specific tags, called VM ASID, that associate data and code to the VM that has generated them in a way that only the specific VM can access both the parts. This tag is associated to the data when inside the SOC, and avoid non owner VMs to access it.

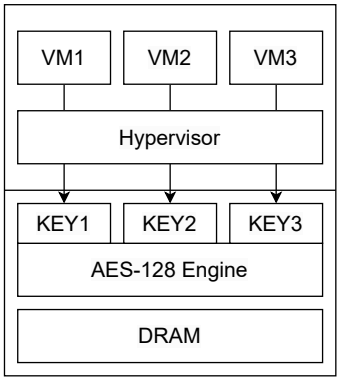

Figure 3.10. SEV architecture (source: [23])

Outside the SOC the data are encrypted with AES-128 [29] using a specific key that is associated to the tag, related to the VM. Each VM will have its key, also the hypervisor, and depending on the tag in the SOC, each VM can only access a specific key. If it tries to access data that are associated to another VM, the key that will be used to decrypt the data is not correct, so the VM will not be able to see the correct data. This ensure that there is strong cryptographic isolation between the VMs.

To understand which are the pages that are encrypted the SEV architecture use a bit called C-bit that has that functionality. One of the most important feature of SEV is that all VMs are free to choose which data have to be private and which not. this decision is full under the control of the guest and in the first case, the data is encrypted with the specific key of the VM, in the

second one the hypervisor key is used to encrypt them. In this situation each VM can liberally decide which are the pages that they want to share and the other that instead have to be kept private (see Fig. 3.11).

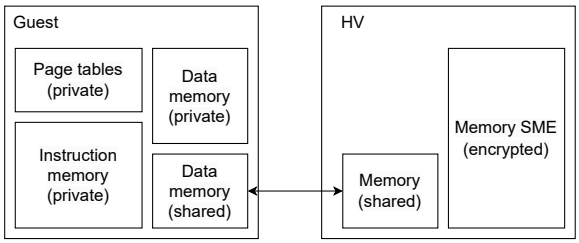

Figure 3.11. Guest-VM communication example (source: [23])

The last aspect to cover about the SEV architecture is the key management. This is one of the core aspect to obtain the proper level of security, because if for this reason an attacker would find the different keys, all the data protected with them associated to different VMs will be disclosed. To avoid that the hypervisor has to directly managed the different keys, there is a secure key management interface that is provided by the SEV firmware that runs inside the AMD-SP. This interface is used by the hypervisor to enable SEV for secure guest and perform usual activities. To obtain the protection of SEV enabled guest, the firmware is built for guaranteeing three main security properties:

- authenticity of the platform: used to avoid that a malicious software or a rogue can not say to be a legit platform. The proof is given with an identity key that is signed by AMD platform key with SEV capabilities and signed by the owner of the platform to underlying who is that has the control on the device for the guest owner's.

- attestation of a launched guest (see Fig. 3.12): used to ensure to the secure guest's owner that their guest has securely launched with SEV enabled. The firmware provides to the guest owner a signature of some components of the SEV associate to the guest state. This is used by the owner to check if the hypervisor has interfered with the initialization of SEV.

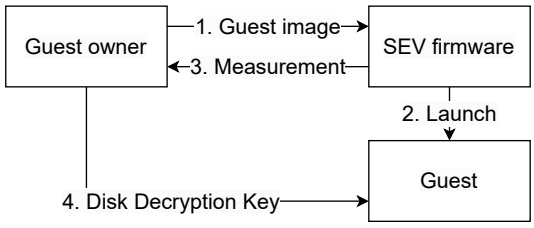

Figure 3.12. Guest attestation example (source: [23])

- confidentiality of the guest's data: encrypting the memory associated to the guest with a specific key that is managed by the SEV firmware and it is never exported outside if the recipient is not authenticated. This avoids that the hypervisor will take the control of the key and consequently of the guest's data

An other feature that can be implemented is the migration of the guest data to another SEV capable platform. All the data are passed encrypted and when the remote platform is authenticated, also the guest encryption key are sent in a secure way.

SEV software implications

With the respect to the hypervisor, SEV continues to be associated to it for many VM functions, but at the same time the reliance on the hypervisor is reduced for security. A guest with SEV enabled, uses the hypervisor as usual, but at the same time it is protected marking what the pages that have not to be shared as private. The management of the encryption is demanded to the AMD-SP with which the hypervisor communicates during the runtime. This interaction between these two entities is also done when the hypervisor has to attest the guest to create a secure mechanism. The ASID used to run a VM is also under the control of the hypervisor, and consequently also the selection of the encryption key.

From the point of view of the guest, the OS that is placed in an SEV-enabled guest have to know the new hardware feature for configuring properly the page tables. An important aspect to underlay is that the DMA has not the possibility to access guest encrypted memory. All the DMA has to be directed to shared guest memory. It is a choice of the guest to select which pages can be accessed by the DMA or to configure a special buffer for DMA purposes. Also, there are no performance penalties with multicore guests supported by SEV because the hypervisor must simply use the same ASID for all virtual CPU.

3.2.5 IBM Secure Execution (IBM Z)

IBM Secure Execution [24] for Linux is a z/Architecture security technology that is used to add to the data of a KVM guest, protection from being inspected or modified by the server environment; no entity can access the data associate to a guest that has been run as an IBM Secure Execution. This technology is used to add pervasive encryption to the data that protect them while they are at-rest, in-flight and at-use.

If the scenario is where the KVM guest is run in cloud, the principal security risks, related to the workload are the following:

- malicious actors that might gain root privileges if there are some security problems in the administration of the hypervisor
- Code associated to the hypervisor that has been introduced by a malicious actor
- virtual machine out of the control of the hypervisor that try to obtain hypervisor privileges

Introducing the pervasive encryption, data stored are protected and are also protected during processing (see Fig. 3.13).

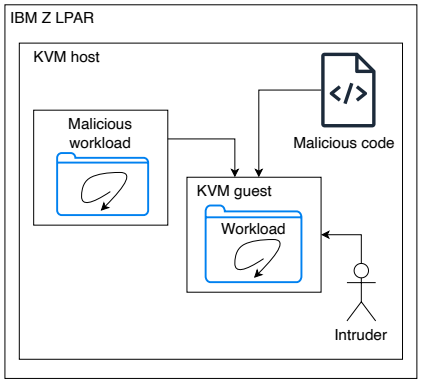

Figure 3.13. Protection of workloads with IBM Secure Execution (source: [24])

IBM Secure Execution components

IBM Secure Execution for Linux provides technology to defend against different security threats. The IBM Secure Execution implements the following feature:

- Boot image protection: specific image built to be run by the ultravisor. This image is encrypted and a crypto hash of it is calculated, with its IBM Secure Execution header. The image encryption keys and the hashes are located there encrypted. The header also includes the customer root key that is encrypted with the host's public key, this is due to the fact a host key document has to be associated to a specific host system. Inside the boot image, can be inserted also some sensitive data, because it is encrypted.
- Memory protection: in a normal situation the hypervisor can access the data of the virtual servers. Instead, if IBM Secure Execution has been used to run a secure way a virtual server, the hypervisor can not access its virtual memory. If it tries to do that, its request will be rejected and redirected to the ultravisor. After that the boot image is decrypted by the ultravisor, it is placed in the secure memory and in this way all the memory that will be used by the virtual server continues to be secure
- State protection: all the information that are used to describe the state of a virtual server are protected by the ultravisor

The ultravisor is the entity that controls the execution instead of the KVM. It has the duty to do all the stuff to maintain secure a virtual server, so for example secures its memory and manages it and so on. The first thing that the ultravisor does is to load the image of a guest that has to be secure ad check for its integrity after that it was decrypted. After that it set all the memory associated to the created virtual server as secure and do the operations needed to secure the state of it. The ultravisor is also used to protect against of manipulation of the workload and changing of memory pages because when a page has to be swapped out by the hypervisor, before that, the ultravisor computes the hash of this page and encrypts it without that the hypervisor can access the page. Once the page can return to the memory, the ultravisor checks for its integrity (see Fig. 3.14).

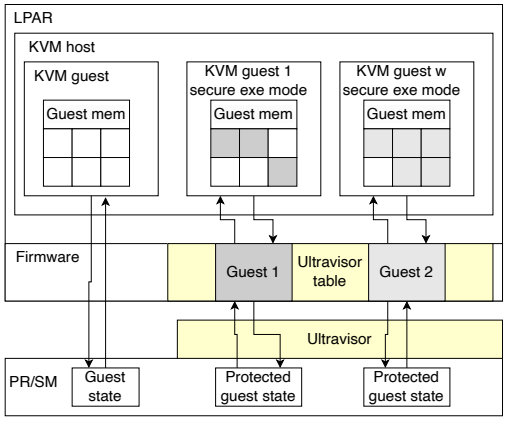

Figure 3.14. IBM Secure Execution protects guest memory and state (source: [24])

To understand which pages are associated to a specific virtual server, the ID of the server is used to label them. Obviously each virtual server can only access its memory pages and if it tries to access pages related to something else, the IBM Z memory management hardware and the firmware avoid that.

3.2.6 RISC-V Keystone

This section is used only to present a general overview of the RISC-V Keystone [25], because the next chapter is totally used to explain in detail this type of TEE and the principal characteristics

of the RISC-V architecture, architecture used to run TEE associated to Keystone.

Keystone is considered to be an open-source Trusted Execution Environment that has been developed for RISC-V processors. Keystone can be tested in different platforms:

- QEMU
- FireSim
- SiFive HiFive Unleashed board.

Keystone Enclave can be migrated to a generic RISC-V [30] processor, knowing that there will be the necessity to make some changes on hardware to plant the silicon RoT. To check the current capabilities of Keystone Enclave the Keystone Demo can be used. It is placed in this repository `https://github.com/keystone-enclave/keystone-demo` and inside also a documentation of the things can be found.

The principal components of which the Keystone repository is composed are the following:

- patches: something needed for to change the submodules includes in Keystone repo
- bootrom: the bootrom for Qemu virt board. It also includes the trusted boot chain
- buildroot: used to build a Linux image for testing in platforms
- docs: contains the manual
- riscv-gnu-toolchain: used to build riscv targets. It is needed also to build all the other components
- linux-keystone-driver: the loadable kernel module for the different Keystone enclave
- linux: Linux kernel
- SM: contains the Keystone Security Monitor (SM) and the OpenSBI firmware
- qemu: contains all that is needed to properly run QEMU
- sdk: contains all the stuff needed for testing, running and building examples Enclave on Keystone

3.2.7 Standards and frameworks to provide unified Application Program Interfaces (APIs)

The most common standards and specification that can be used to provide Application Program Interfaces, talking about the Trusted Execution Environment, are the following:

- Global platform specifications [31]
- Open Portable TEE [32]

Global platform specification

Global Platform [31] is an organization that has the objective to make technical standards used for the efficient launch and management of digital services and devices that are innovative and secure-by-design delivering to the users end-to-end different features like security, privacy and simplicity. Its main goal is to public standardized technologies and certifications that can be used by the various technology and service provider to realize their device and services fitting the best their business, security and so on. One of the most important feature that is provided by the Global Platform is the secure component specification:

- **Device Trust Architecture**: to access securely services from a device
- **IoTopia Framework**: to secure launch and manage device that are connected through each other
- **SESIP Methodology**: used to certify IoT devices

More related to the Trusted Execution Environment, the Global Platform has also published some papers about the specification of:

- **Secure Element** (SE) [26]: is a tamper-resistant platform that can be used to obtain the secure execution of an application giving to it the confidentiality of the data. The security features that are provided by the SE have to follow the rules and security requirements defined by the trusted authorities. There can be different types of Secure Element:
  - embedded SE
  - smart microSD
  - SIM/UICC

  The SEs can be considered as an evolution of the traditional chip that are located in smart cards, that have been adapted to address the new security necessities in the different devices. Global Certification can release a certification that guarantees that the functional behavior of a product compared with the requirements outlined by GlobalPlatform Se configurations and spec is compliant with market interoperability. Dependently from the specific needs, GlobalPlatform has released different documents for the SE community. Some of the topics of these documents are the following:
  - End-to-End (E2E) Frameworks
  - Confidential Card Content Management (Amendment A)
  - NFC Managing Entity Specification
- **Trusted Execution Environment** (TEE) [27]: papers in which the following security features have been defined and have to be respected by the TEEs:
  - Isolation from the Rich OS
  - Isolation from the other Trusted Applications
  - Application management control
  - Identification and binding
  - Trusted Storage
  - Trusted access to the peripherals
  - State-of-the-art cryptography
- **Trusted Platform Services**(TPSs): papers used to specify the different mechanism that can be used to access platform services that have been offered by secure components like SE, or TEE, from an internal or external device. A secure component inside a device can be considered to be trustworthy and also the service offered is the same, thanks to a Chain of Trust directly to the application that can be attested. The principal objective of these papers is to simplify the linkage between strong security technology offered by secure components in products built by service providers or application developers. This is obtained through GlobalPlatform's Device Trust Architecture (DTA).

Open Portable TEE

Open Portable TEE [32] is a Trusted Execution Environment that has been developed has the set of a non-secure Linux kernel running on ARM and Cortex-A cores that use TrustZone technology. It has the Tee Internal Core API v1.1.x that is the API that can be used by the Trusted Applications and the TEE Client API v1.0, that is the API that describes how the communication with the TEE can be made. The non-secure OS is called Rich Execution Environment (REE) and it is usually a Linux OS flavour.

The design of OP-TEE is principally related to the ARM TrustZone technology and so based for example to the hardware isolation mechanism. However, each type of isolation technology suitable for the TEE concepts and goals can be used in OP-TEE due to how it has been defined.

OP-TEE has been developed to achieve:

- **Isolation**: the TEE guarantees the isolation from the non-secure part and the secure one and the uses the underlying hardware supports to defend the security of the loaded Trusted Application from the other ones.
- **Small footprint**: the size of the TEE has to kept as small as possible, to be sure that it can store in a reasonable amount of on-chip memory
- **Portability**: the TEE can be easily moved from a type of architecture to another one and from a type of HW to another one. It must have also the support to be associated with different client OSes or different TEEs.

The different components that compose OP-TEE are the following:

- secure privileged layer
- secure user space libraries used by the Trusted applications
- Linux kernel TEE framework and driver
- Linux user space library
- Linux user space supplicant daemon used for the remote services
- test suite
- examples
- build scripts

3.3 An example of coprocessor-Based TEEs in one SoC: Apple Secure Enclave Processor (SEP)

Secure Enclave [33] is an integrated subsystem built into Apple systems on chip (see Fig. 3.11). There is the isolation between the secure enclave and the main processor in a way that, also if the Application Processor kernel is compromised, the sensitive user data continue to be secure. It is composed by different parts:

- **bootrom** to establish RoT
- **AES engine** for crypto operations
- **protected memory**: it does not include a secure dedicated storage but implements mechanism to store data securely on the attached storage in a space that can't be used by the Application Processor and operating system

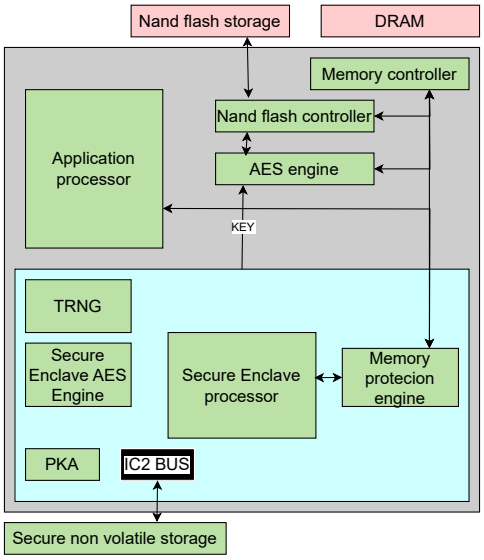

Figure 3.15. Overview of the secure enclave (source: [33])

The most important components that compose this TEE are the following:

- **Secure Enclave processor**: is the processor used by the secure enclave to do operations. It can not be used by the normal applications and this prevents some type of attacks like for example the side-channel attacks. It runs a specific version of microkernel, the Apple-customized version of the L4 microkernel, that works at a lower clock speed to avoid power attacks. It is composed of memory-protected engine, encrypted memory, secure boot, AES engine and random number generator.

- **Memory protection engine**: the mechanism by which the memory of the Enclave is protected is the following:

  – at the starts, an ephemeral memory protection key is generated by the Secure Enclave Boot Rom

  – if the Secure Enclave has to write something, the key is used to encrypt data and also an authentication tag is generated (Cipher-based Message Authentication Code)

  – if the Secure Enclave has to read something, first the authentication tag is verified, the blocks that have to be read s decrypted, otherwise an error is sent to the Secure Enclave and the Secure Enclave is stopped until the next reboot.

  The Memory protection engine is completely transparent to the Secure Enclave. It writes and reads from the memory like if this operation were made in the normal way, not knowing what the engine will do. This permits to not have software or performance complexity tradeoffs.

- **Secure Enclave Boot ROM**: it is used to establish the hardware RoT. It associates to the Secure Enclave specific memory region and starts the Memory protection engine

- **Secure Enclave Boot Monitor**: it is used to be sure that there is strong integrity on the has representing the booted sepOS (Secure Enclave Processor Operating System)

- **Root Cryptographic Keys**: each Secure Enclave has associated a unique ID root cryptographic key. The UID that is generated during manufacturing is inserted in the SoC. It is generated directly by the Secure Enclave TRNG and written to the fuses using a software that is run directly in the Secure Enclave. This makes the UID invisible outside the Secure Enclave.

- **AES Engine**: an AES256 crypto engine is associated to each Apple device that has a Secure Enclave and it is directly integrated in the DMA path. During the boot the sepOS created an ephemeral wrapping key, derived with the TRNG and it is given to the AES Engine with dedicated wires. This key can be used to mask other keys that have to be used by the Application Processor file-system driver. When an operation of reading or writing has to be made, the wrapped key is sent to the AES Engine that unwraps it. An important thing to say is that the key no more unwrapped is never exposed to the software (see Fig. 3.16).

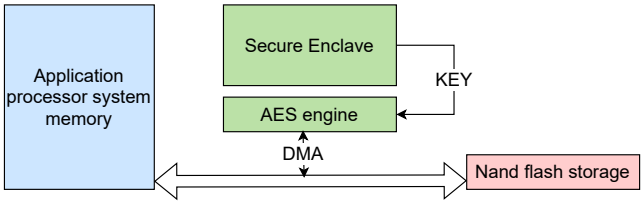

Figure 3.16. AES Engine (source: [33])

- **Secure nonvolatile storage**: the different encryption keys that are used are rooted in entropy saved there. In the latest devices there is also the Secure Storage Component that is used for entropy storage. The communication between the Enclave and the Secure Storage Component is made with a protocol that guarantees authentication and encryption to access the entropy.

3.4 An example of Coprocessor-Based TEEs in external SoC: Microsoft Azure Sphere: Pluton

Microsoft Pluton [34] security processor is a security technology that follows the Zero Trust principles (security strategies in designing and implementing security principles like verify explicitly, use the least privilege access, assume breach). Pluton implements different security services:

- hardware-based RoT
- secure identity
- secure attestation
- crypto operations

Pluton can be thought as a secure subsystem that run in a system on chip that is combined with Microsoft authored software that is the software that runs on it.

More in detail, Pluton can be considered as a secure crypto-processor used to have the code integrity and always the latest update provided by Windows Update. It can use also to store securely credentials, identities, and so on. It is compliant with all the functionalities that a Trusted Platform Module must have and also with the specification of TPM 2.0.

3.4.1 Architecture overview

Pluton is composed of three different layers (see Fig. 3.17):

- **Hardware**: secure element that provides the TEE and all the crypto services needed
- **Firmware**: Microsoft firmware used to make possible the communication between the software, the application and Pluton. During the Pluton Hardware initialization, it is loaded
- **Software**: drivers and app that can be used by end user to use the hardware capabilities provided by Pluton

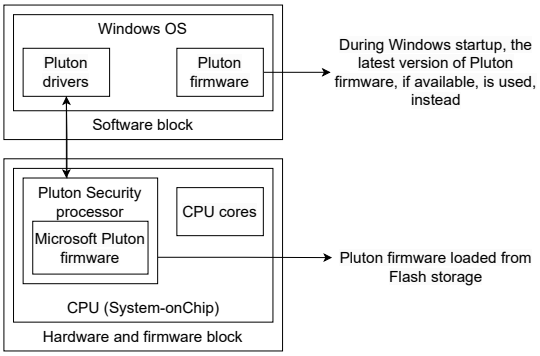

Figure 3.17. Pluton architecture (source: [34])

3.4.2 Firmware load flow

The firmware load flow is described as follows:

1. initialization of Pluton hardware and ROM
2. Loading of Pluton firmware from SPI storage
3. UEFI boot
4. UEFI handover to boot manager
5. if the version of Pluton is updated, it is loaded, otherwise the firmware is loaded from the SPI Flash
6. Boot on Windows

Chapter 4

Keystone Enclave

4.1 RISC-V overview

As mentioned in the previous chapter, Keystone enclave is based on the RISC-V [30] architecture. This is a type of instruction set architecture, and one of its most important feature is that it is open and free, so each one that has sufficient knowledge can decide for sure to use it, but also to modify and extend. Some important characteristics of RISC-V are the following:

- Physical Memory Protection: security oriented primitives that allow efficient isolation
- the ISA is evolving and all the changes are community driven. Keystone can decide to use only the security features that are considered to be useful and at the same time integrates the good idea in the standard itself
- all the stuff like open-source cores and products are associated to RISC-V. For this reason, Keystone can be used in a very large set of platform.

At the moment Keystone is compatible with a specific subset of the RISC-V ISA:

- rv64gc-lp64d (Sv39 virtual addressing mode)
- rv32gc-ilp32d (Sv32 virtual addressing mode)

4.1.1 RISC-V Privilieged ISA

RISC-V is composed of three different privilege levels (the order used is to specify an increasing level of the operations possible):

- **user mode** (U-mode)
- **supervisor mode** (S-mode)
- **machine mode** (M-mode)

It is not possible to have at the same time more than one level associated to the processor.

What a running software can do while it is in execution in the processor, is strictly associated to the privilege level. The privilege levels are usually associated to the following usage:

- U-mode: user processes
- S-mode: kernel and also the hypervisor

- M-mode: bootloader and firmware

M-mode is the most powerful privilege level. In this mode, there is the full control of all the physical peripherals and of the interrupts. In Keystone the Security monitor (SM) runs in this mode, that is the Trusting Computing Base of the system. There are different advantages when the M-mode software is the TCB:

- **Programmability**: existing programming languages and toolchain can be used to compose M-mode software
- **Agile patching**: adding patches is very simple due to the fact that the SM is all software, so there is not the needed to make hardware-specific updates
- **Verifiability**: usually the hardware is more complex to check instead of the software part

4.1.2 Physical Memory Protection (PMP)

Physical Memory Protection is a strong standard primitive that has been included in RISC-V and permits the control of the physical memory access by the M-mode. It decides which part of the memory can access the lower privilege levels. This feature is required by Keystone to provide the isolation between the different enclaves. This feature is implemented using a control status register (CSR) and setting it properly. The platform design influences the number of PMP entries. It works on physical addresses, so the capability of the S-mode to configure the virtual addresses is not influenced.

4.1.3 Interrupt, exceptions and virtual address translation

The interrupts and the exceptions are by default received by the M-mode that can decide to delegate the CPU scheduling and configuration to the S-mode.

Each one of the interrupts can also be enabled or disabled by the M-mode and referring to the traps, they can be redirected to the S-mode simply change the setting of a bit of a specific register. Making this a more efficient managing of the frequent traps, for example, can be done, avoiding the interaction with the M-mode handler.

In RISC-V there is a memory management unit that is used to make the virtual address translation. This component is composed of:

- page table walker
- translation look-aside buffer

In RISC-V architecture multi-level page table is implemented and an important thing to say is that the Keystone enclaves cannot be attacked trying to modify the page table, because its enclave has its page table that is protected from being accessed by the OS.

4.2 Customizable TEEs

Customizable TEE means that the TEE uses a common software framework to build a specialized TEE for a porpoise, that has multiple stakeholder's input. When the platform provider decides to realize a specific TEE instance, it has the duty to choice different aspects, like the hardware interface, the trust model and what are the requirements associated to the enclave programmer's feature.

One of the most important reason of the customizable TEE is that there is not a defined threat model associated to different use cases, applications or the platform chosen. Implementing

customizable TEE, each enclave has the possibility to define what are the security features that have to be implemented.

Differently from Keystone enclave, the other TEE systems that are on the market, have a specified threat model, that can't be changed and that is directly linked with the hardware platform. Using instead RISC-V architecture, introduces the possibility to have multiple concurrent and potentially multithreaded enclaves that each one of them has associated a memory region while at the same time there is the open of the supervisor-mode and also the MMu for the enclave use. In this way an enclave can liberally choose to have inside both a lightweight o a full supervisor-mode OS.

A platform that has the support for Keystone, must have the following hardware requirements:

- device specific secret key (visible by only the boot process)
- hardware source of randomness
- trusted boot process

4.3 Keystone overview

A system that is compliant with Keystone [25] is associated to different components each one as specific privilege mode (see Fig. 4.1):

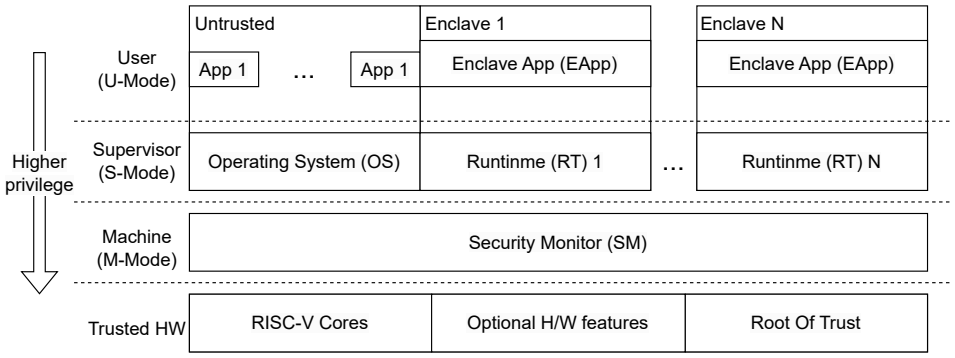

Figure 4.1. Different components in a system that implements Keystone enclaves (source: [25])

- **Trusted hardware**: hardware compliant with the Keystone-compatible standards RISC-V cores that has been built by a trustworthy vendor that also provided the RoT. Some optional features like cache partitioning, memory encryption and so on can be included.
- **Security monitor**: software that runs in M-mode that has a small TCB. It is used to provide all the functions that are needed to manage the lifecycle of enclaves and to use features related specific to the platform.
- **Enclaves**: environments totally isolated from all the other enclaves and the non-secure part. Each one has associated a specific memory region that has been protected with PMP
- **Enclave application**: the application that runs in the enclave.
- **Runtime**: it can be thought as the OS of the enclave that is used to manage the system calls, trap and so on.

Associated to Keystone, there are two different workflows (see Fig. 4.2) due to the fact that the elements that compose the systems can be edited and modified at the same time both by

the platform provider and by the enclave developer.The first one is who provides where to run Keystone enclaves. During the provisioning stage the SM is built by it and deployed to the device.

Some duty of the platform provider are the one, two and third presented in the figure below that are associated to the configuration, the building and the deployment of the SM with the hardware. The second one, instead, builds the enclave using the Keystone SDK. It has to develop

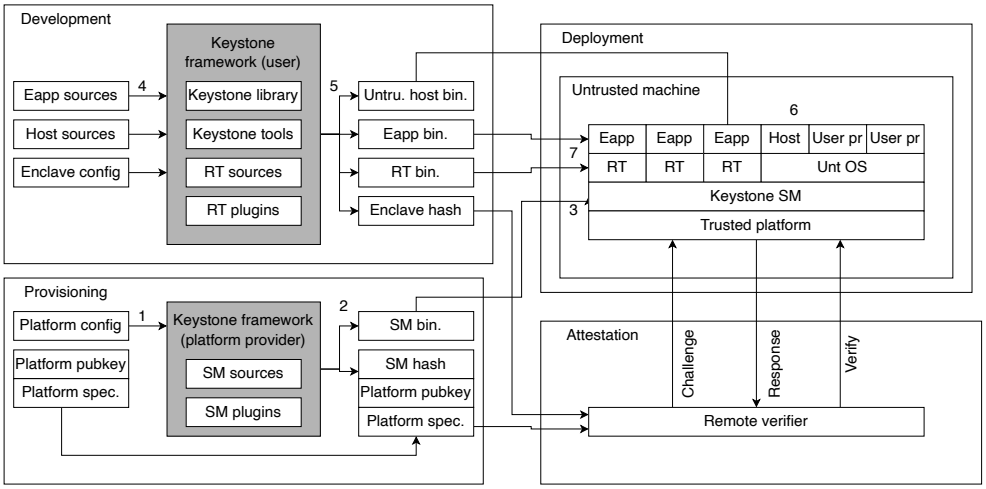

Figure 4.2. Keystone workflow (source: [25])

the eapp, the host and the runtime binaries (point 4 in the figure). The remote machine that runs Keystone receives the different enclaves components and build them. (point 5,6,7 in the figure).

Another feature that is supported by Keystone is also the remote attestation where the enclave is measured by the SM and this measure can be provided to a remote attester for the validation (point 8).

With the respect to the Enclave, the workflow associated to its life (see Fig. 4.3) is the following:

1. **creation**: when the enclave has to be started, a contiguous range of physical memory called enclave private memory (EPM) is associated to it. This allocation is made by the untrusted host that after it makes other operations like the initialization of enclave's page table, of the runtime and of the eapp. After that, the untrusted OS calls the SM asking for the creation of a new enclave and it answers isolating and protecting the EPM adding a new PMP entry. The status associated to the PMP is delivered to all the cores, that knowing that, guarantee the protection of the EPM. Then, before the execution, the enclave is also measured and verified.

2. **execution**: to enter into the enclave using a core, the host has to ask for and after that the permission to access the PMP have been released by the SM to the core, that become allowed to execute the enclave. The PMP permissions have to be changed all the time that the core does an enter/exit operation for continues having the property isolation

3. **destruction**: the enclave can be destroyed at any time by the host. If it has to be done, the EPM associated to the enclave are cleaned by the SM that also has the duty to free the PMP entry.

4.4 Security monitor

It is the most important part of the Keystone TEE. It can be easily ported from different platforms all implementing RISC-V architecture. The default implementation of the SM is used to provide the isolation and ensure that some security-critical features are respected, this to reduce as possible the attack surface.

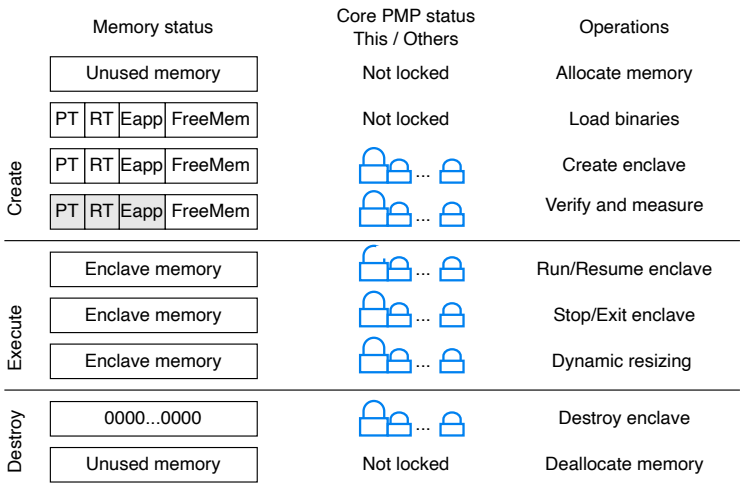

Figure 4.3. Enclave lifecycle (source: [25])

4.4.1 Memory isolation

This feature is provided by the SM using the PMP provided by the RISC-V architecture where S-mode and U-mode can access only the memory region that are associated to them with PMP entry (see Fig. 4.4). During the SM boot, Keystone configures the first PMP entry to be applied to its memory region, don't allow the U-mode and the S-mode to have the possibility to access it.

When an enclave has to be created, the SM creates a new PMP entry, that has higher priority than OS PMP entry, with all the permission disabled. When the Enclave has to take the control, the SM enables the PMP permission bits associated to the specific enclave and at the same time disable all the OS PMP entry permission in a way that the enclave can only access its memory region. When there is a context-switch that consist of exiting from the enclave, the reverse process is done, to protect the memory of the enclave and to permit the OS to access its memory regions. Other important features related to the SM are all the actions that have to be made for the managing of the enclave lifecycle (creation, execution, destruction) and the related operations that have to be made for the proper assigning and managing of the enclave memory.

4.5 Keystone Modular Runtime

The Keystone Modular Runtime is the private code that is run by the enclave in S-mode. Its functionalities can be compared to the functionality that has a kernel, but with not so much effort. The runtime that has been developed is called Eyrie. Due to the fact that the runtime is executed in S-mode, it can be easily modified to implement new features needed without affecting user applications.

Some additional modules can be added to the Eyrie RT to allow it to manage in a flexible way the memory:

- Module for Free memory: used to allow the RT to do page table management in an unmapped physical memory that is not included in the enclave measurement
- Module for In-Enclave Self paging: used to implement the swapping paging mechanism.
- Module for Protecting the Page Content Leaving the Enclave: used to protect with integrity and confidentiality the pages that have to be swapped out where there is a page fault

Some functionality modules included in Keystone are the following:

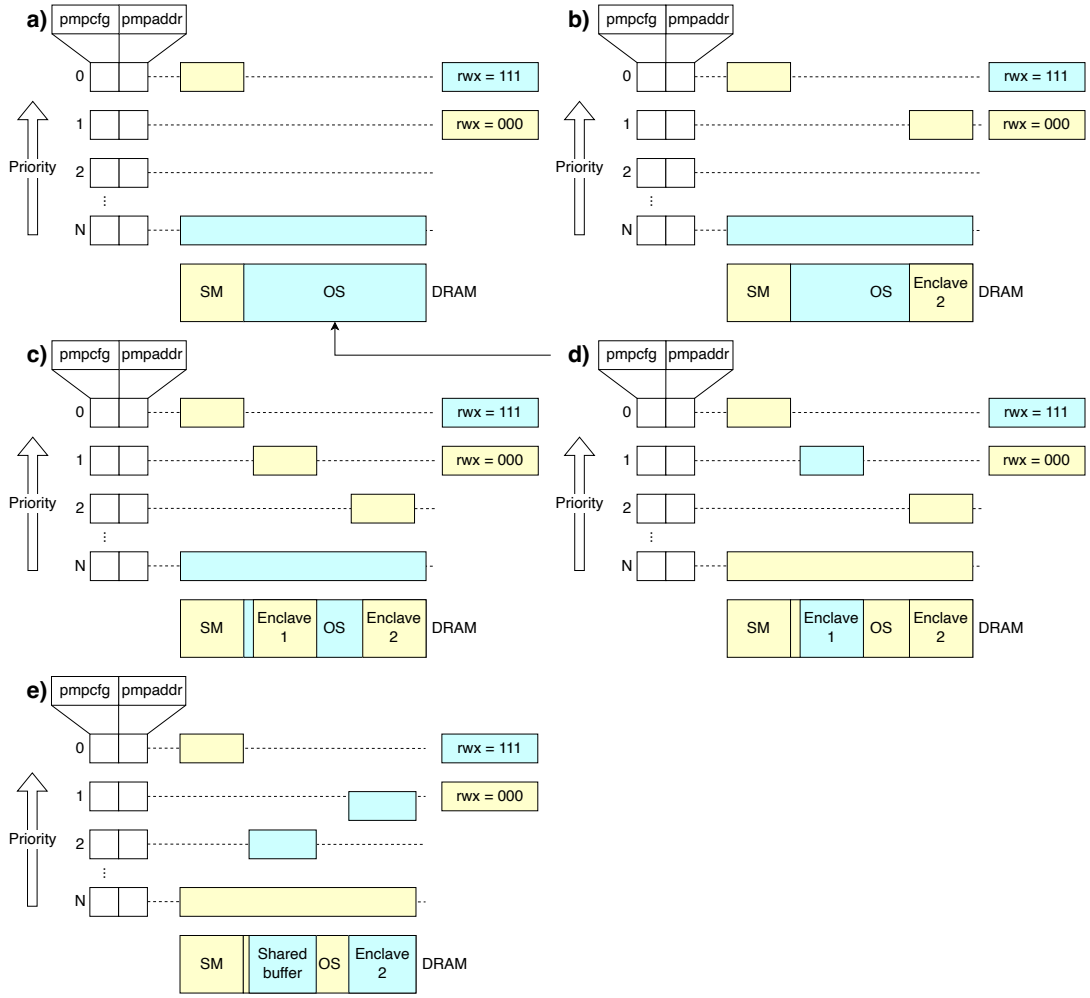

Figure 4.4. How Keystone uses RISC-V PMP for memory isolation

- **Edge call interface**: mechanism that is used to access memory that is not associated to the enclave. To perform this type of operations (read/write) there must be a buffer shared from Eyrie RT and the host
- **Multi-threading**: the Eyrie RT manages the different threads associated to the eapps.

4.6 Security analysis and weaknesses

4.6.1 Protection of the Enclave

A Keystone Enclave is protected from the following types of attacks:

- **Mapping attacks**: The RT is considered to be trusted, so malicious virtual to physical addresses mapping can't be created and this means that also the mapping will be valid. While the enclave is running, the RT also check if the layout is not compromised when the mapping is updated. The RT also checks if the new pages that can be associated to the enclave during the dynamic resizing are safe.
- **Syscall Tampering attacks**: Keystone is designed to use shielded systems to protect against Iago attacks and system call tampering attacks.

- **Side channel attacks**: due to the fact that the enclaves are completely isolated with the respect to the state with the OS or the user application, there is not the possibility to controlled channel attacks. All what happens in the Enclave can only be seen by the SM, not the OS

4.6.2 Protection of the Host OS

The host cannot be attacked by the Enclave because it can access only its memory region due to the PMP protection, can't change the page table that is not associated to him, modify the state associated to the host because when the control is passed to the enclave, the SM does a complete context switch.

4.6.3 Protection of the SM

It is certain that the SM memory cannot be attacked by lower level components such as the Eyrie RT and the Host OS due to the PMP protection. A DOS attack cannot be done because the SM is only a monitor that is used as reference and it also uses techniques to be protected from Cache attack and Time attack

4.6.4 Protection against Physical Attackers

The enclave is protected from physical attackers due to the presence of the on-chip memory connected with RT's paging module that guarantee the integrity and the confidentiality of the pages that leave the on-chip memory. In the backing store there are pages that are encrypted and also protected with the PMP.

The SM code should have placed entirely in the on-chip memory in a way that all that is situated outside that is not considered to be trusted or there are the protections of the encryption and the integrity.

4.6.5 Weaknesses

Keystone does not come only with some advantages, but present also some weaknesses that are explained as follows:

- Keystone is pretty young as a project
- at the moment it can only be used in RISC-V architecture
- it is strictly related to the PMP
- there are a number of maximum enclaves that can be created
- all the project is based on the fact that the SM, the RT and the eapps are bug free: the RT is not so easy to verify, because it is not so small as the SM and adding feature will be always more difficult to ensure that it is without bugs.
- there are no defenses against speculative attacks

Chapter 5

DICE specification in Keystone: design

5.1 Root of Trust requirements and keys generation in Keystone

To implement the hardware requirements needed for the DICE core, the RoT has been designed in this way: on the real device, the platform has to be associated with a secure root device keystore, that will be used to store the UDS in a way that is has not to be rewritable and only the DICE can access it. For testing porpoise, it is supposed that the UDS is statically defined in the `bootloader.c` file and used when there is the necessity. When the boot process ends, the memory associated to the store of the uds variable is erased in a way that no other layer can use or access it.

To calculate the CDI value, the simple hash function has been chosen because the standard implementation of the Keystone project comes with already defined all the methods related to the SHA3 family function.

$$\textbf{SHA3}(UDS\ ||(SM\_Measure))$$

This family of functions is considered to be perfect for the embedded systems because it is:

- **strong**: due to the fact the multi-round permutation f is intricate (the operation that is used to change the state of the hash algorithm)
- **cost-effective**: if it is compared with other algorithms (provide better protection with the respect to the SHA2 family algorithm)
- **efficient**: not so expensive to be implemented both in silicon that in software

It is based on the KECCAK cryptographic function where the input has not a defined length and at the same time it is the user that chooses the length of the output. This function can be used for different aims, from the support of symmetric cryptographic functions to the authenticated encryption. Once the CDI has been calculated it is passed to the SM securely: the variable is located in a part of the memory that is shared only by the DICE core and the SM. This is done because the SM is considered to be trusted, and also because not having the access to the UDS, it cannot obtain the CDI differently.

The key generation in Keystone comes for free with the default implementation that is provided on GitHub and it is associated more in detail to the following files:

- `ed25519.h`

- `keypair.c`

in which there is the definition and then the implementation of the function:

```
void ed25519_create_keypair (unsigned char *public_key, unsigned
        char *private_key, const unsigned char *seed)
```

This function is based on the SUPERCOP "ref10" implementation and it is a portable version so all the files needed to make the function properly works are the only included in the folder *ed*25519 of the Keystone project.

It creates a keypair starting from the seed that is provided of 32 bytes and provides in the two buffer *public_key* and *private_key* respectively the public key and the private key, the first one with a length of 32 bytes and the second one with a length of 64 bytes.

More in general, Ed25519 is a public-key signature system that has the following features:

- Fast single-signature verification: the number of cycles that are needed to verify a signature is not so big
- Even faster batch verification: the time needed to batch 64 separate signature verifications are not so big
- Very fast signing: very slow numbers of cycles to make the signature of a message
- Fast key generation: this operation is as fast as the signature operation
- High security level: its security target is of 2^{128}
- Foolproof session keys
- Collision resilience: if some hash-function collisions have been founded, they cannot break the system
- No secret array indices: the secret addresses in RAM are never read from the implementation of the algorithm
- No secret branch conditions: all the jumps made due to specific conditions are never based on secret data
- Small signatures: the size of the signatures are only of 64 bytes
- Small keys: the keypair that can be created is composed of the public key of 32 bytes and the private key of 65 bytes

5.2 DICE concepts applied to Keystone TEEs: proposed design

The figure below (Fig. 5.1) shows what is the design chosen to implement the DICE specification in the Keystone project.

More in detail, the DICE core is associated to the RoT, the layer 0 of the DICE specification is associated to the SM and the layer 1 of the architecture is related to the different Enclaves that can be present in the platform (the enclave level embeds both the Runtine RT and the EApp). The RoT comes provided with:

- **Unique Device Secret** (UDS in the figure)
- **Signature of the Measure of the SM** (SMSM in the figure): signature of the SM measure provided by the manufacturer obtained with its private key

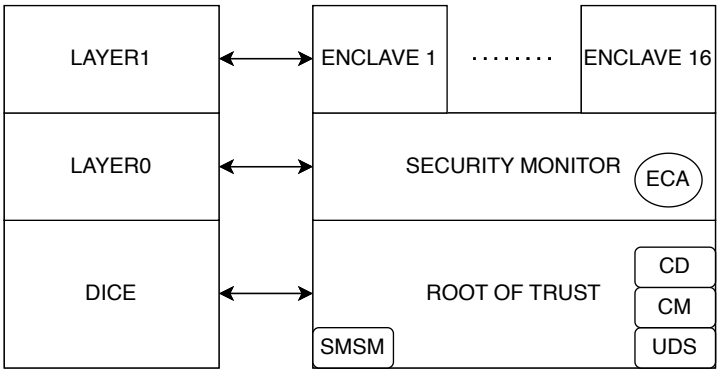

Figure 5.1. Proposed design to implement DICE specifications in Keystone

- **Certificate of the Manufacturer** (CM in the figure): X.509 certificate in DER format of the manufacturer issued by certification authority
- **Certificate of the Device Root Key** (CD in the figure): X.509 certificate in DER format of the Device Root Key signed by the manufacturer

All these data are stored securely in memory and the Unique Device Secret can only be accessed during the booting process. The Signature of the Measure of the SM is used to implement the so called *secure boot*: a way that avoids the completion of the boot and stops the system if the SM has been compromised. It is implemented calculating the hash of the SM and then, using the public key of the manufacturer certificate, verifying the SMSM with the specific method. If the verification goes well, the boot process continues, otherwise all the system is stopped and a message is shown to the user to specify the error (see Fig. 5.2).

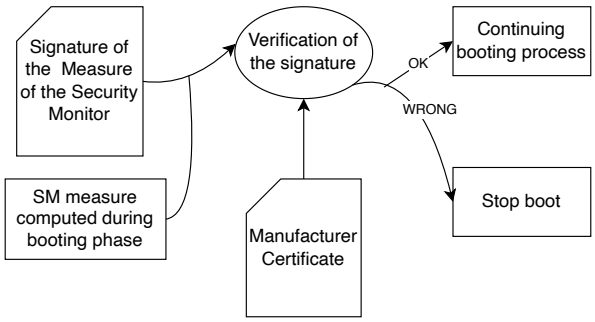

Figure 5.2. Secure Boot

The Certificate of the Device Root Key is the certificate that has been issued by the manufacturer associated to the public part of the Device Root Key. This keypair is derived from the CDI of the level 0 that is calculated from the measure of the SM and the Unique Device Secret. The Certificate can be issued by the manufacturer in advance because the SM must have a specific measure if it is not compromised, the UDS is provided directly by the manufacturer, so he can calculate the same Device Root Key keypair and produce an X.509 cert associated to the public part.

The SM is also the Embedded Certification Authority (ECA in the figure) that provides the X.509 certificates associated to the Local Attestation keys that are generated with the creation of the enclaves each one with its keypair. The ECA keypair is derived from a seed that is obtained hashing together the CDI of the level 0 and the SM measure.

The starting point of Keystone Project does not come with already implemented the managing of the X.509 certificates. To respect the DICE specification, this feature has been added from

scratch in Keystone creating a custom library called X509_custom. This library has been built starting from MBed_tls, another library that offers cryptographic functionality and specific for the porpoise, embeds all the functions that are necessary for the correct managing of the X.509 certs. All the functions needed to the creation, population and translation of an X.509 certificate have been taken and inserted in X509_custom, doing all the stuff to make compatible them with Keystone; a change made, for example, has been to replace all the calls made to the functions $free$ and *malloc*, because Keystone does not have the possibility to use them.

5.3 Hardware layer: keys and certificates

All the keys that are provided and/or generated in the hardware layer are the following:

- **Device Root Key keypair**: keypair that is derived from the CDI associated to the level 0. The CDI is computed starting from the Unique Device Secret and the Trusted Compound Identifier (TCI) of the SM. This keypair is needed to sign the X.509 certificate that is issued during this process associated to the public key of the Embedded Certification Authority because the public part is certified by the manufacturer and the manufacturer is certified by a specific certification authority so a trusted chain of X.509 certs can be created.
- **ECA keypair**: keypair that is associated to the Embedded Certification Authority that is derived from combining the CDI of the level 0 with the measure of the SM using a hash function. An X.509 certificate is issued during the boot process associated to the public part of this keypair

All the certificate that are provided/generated in the hardware layer are the following:

- **Manufacturer certificate**: certificate that is provided using a secure part of the memory that is needed to check the signature made by him on the measure of the SM and also to create with the two following certificate a trusted chain of certificates
- **Device Root Key certificate**: certificate that is provided that contains the public part of the Device Root Key keypair that is signed by the manufacturer
- **ECA keypair certificate**: certificate that is created during the booting process related to the public part of the associated keypair where the issuer is set to be the "Root of Trust" and the subject is the "Security Monitor". This cert is the first one in the chain that has to be used in the different enclaves to check if the signature of the certificate related to their local attestation keys are correct or not.

5.4 Security Monitor: keys and certificates

From the hardware layer no keys are directly provided to the SM. This is due to the fact the only keys necessary to the SM is the keypair related to the Embedded Certification Authority. The public part is provided in the associated certificate and the private key can be calcuated directly by the SM because its measure is inserted as extension in the certificate above-mentioned and the CDI of the level 0 is securely provided. Instead, the lower level provides the following certificates:

- X.509 Certificate related to the manufacturer public key
- X.509 Certificate related to the public part of the Device Root Key keypair
- X.509 Certificate related to the public part of the ECA keypair

The keys that are generated in this layer are the following:

- **Local Attestation keys**: keypair that is created when a new enclave has to be created. It is obtained using as a seed the first 32 bytes related of the CDI of the enclave. The CDI of the enclave is calculated hashing the CDI of the level 9 from the measure of the enclave that come for free from the standard implementation of Keystone Project. It is used for the attestation report of the enclave
- **Local Device ID**: keypair that is created and it is used to create an identity of the enclave. It is certified by a remote verifier that, after checking if the enclave has the excepted measure or not, issue for this key an X.509 certificate

The X.509 certificate that are generated in this layer is the following:

- **Certificate of the local attestation keys**: certificate that is associated to the public part of the keypair created during the creation of the enclave used to have the chain of certs until the RoT

5.5 Trusted Applications: keys and certificates

Differently from what is said in the DICE specification, in the proposed design, each enclave has not direct access to the keys associated to it, more in detail to the private part of each keypair, so for this reason it cannot create them in any way.

What an enclave can do is to ask the SM to provide him all the keypairs that it needs but knowing that the private part will not be under its control. When it has to do specific operation related to the private part, for example the signature of some data, there are specific interfaces exposed by the SM to permit the operations in a way that they are totally transparent for the enclave: it has only to provide the public part of the keypair and the data that have to be signed. Is the SM that will have the duty to recognize the key provided, to find the related private part, and to do all the operations needed to make the signature and at the end to provide the signature to the enclave. This has been made to avoid that an enclave can be moved from a machine to another machine. If it can be done, it is not correct because each keypair is derived from the CDI of the enclave and some other data. But the CDI of the enclave depends on the CDI of the SM and so on until the RoT. So if the enclave can be moved from a platform to another platform there will be a consistency problem due to what has been described above. Implementing each enclave without that it can have the possibility to directly control its keypair, prevent this type of error and this is the main reason for why there is the requirement for the enclave to pass from the SM to do stuff related to the private key of some keypairs.

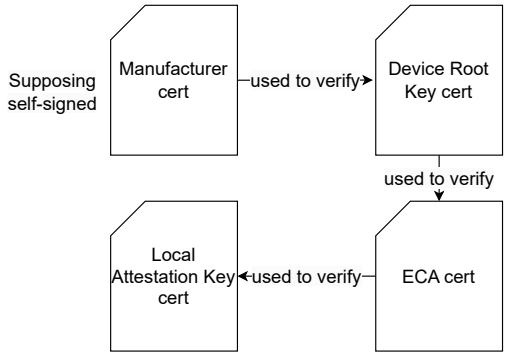

Figure 5.3. The chain from the Local Attestation Key to the Manufacturer cert (supposing it self-signed)

The mechanism provided to make possible the interaction between the SM and the Enclave is similar to the mechanism that is used by an operating system to manage the system call: what happens is that an enclave can call a specific function that is associated to an SBI_CALL (SBI

stands for Supervisor Binary Interface and is the interface between code that runs at different operational levels). This SBI_CALL is managed by the SM sbi handler that dependently from the code used, passes the control to a function implemented at SM level and after that the operations are done, returns the control to the enclave if all goes well. More in detail the execution flow is (see Fig. 5.4):

1. the EApp calls a Software Development Kit (SDK) function
2. the SDK function makes a syscall passing a specific value
3. dependently from the specific value provided, the Runtime calls a sbi_function
4. this sbi_function makes a SBI_CALL with a specific flag that is intercepted by the sbi ecall enclave handler that is present at SM level
5. a specific function exposed by the SM to the enclave is called dependently from the flag provided by the Runtime

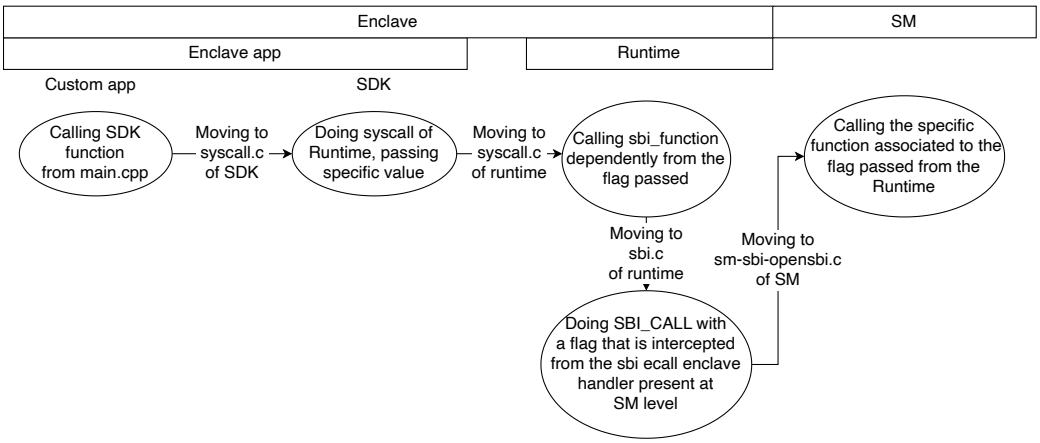

Figure 5.4. The communication flow between the EApp and the SM

With the respect to the generation of different keypairs, the SM exposes a specific interface to do that and it is for this reason that each enclave can have associated a variable number of keypairs dependently on what it has to do to. However, each enclave has at least associated two different keypairs that are the following:

- Local Device Id Key
- Local Attestation Key

At the same time, also the creation of X.509 certificates is forbidden in the enclave and what it can do is only obtaining the cert chain of the Local Attestation keys passing through specific SBI to check it.

Chapter 6

DICE specification in Keystone: implementation

6.1 X509_custom library

Before starting analyzing the parts that have been implemented in the different layers of the Keystone project, a brief overview of the `X509_custom` library is needed.

This library has been created to solve the problem that in the native Keystone project there is not the possibility to manage the X.509 certificates. The only things that were made related to this topic are the following code lines shown in the below figure (Lis. 6.1), that is something very far from the "real" X.509 certificate managing:

Listing 6.1. How is the original "X.509 management"

```
void sm_print_cert()
{
        int i;

        printm("Booting from Security Monitor\n");
        printm("Size: %d\n", sanctum_sm_size[0]);

        printm("============ PUBKEY =============\n");
        for(i=0; i<8; i+=1)
        {
                printm("%x",*((int*)sanctum_dev_public_key+i));
                if(i%4==3) printm("\n");
        }
        printm("================================\n");

        printm("=========== SIGNATURE ===========\n");
        for(i=0; i<16; i+=1)
        {
                printm("%x",*((int*)sanctum_sm_signature+i));
                if(i%4==3) printm("\n");
        }
        printm("================================\n");
}
```

The library has been built taking from an existing library called *MBed_TLS* all the structures and the methods that are necessary for the creation and parsing of an X.509 certificate (making

all the stuff needed to make the code portable). The most important structures that have been introduced in Keystone are:

- `mbedtls_x509write_cert`: structure that is used to embed together all the info that have to be inserted in an X.509 certificate like the subject, the issuer, the key used to the signature, the public key that has to be put in the certificate, the validity and the extensions.
- `mbedtls_x509_crt`: structure used to parse an X.509 cert in DER format inserting all the info in the specific field. This is the structure associate to an X.509 cert. It contains all the fields that a cert must have. With the respect to the standard implementation, a new important field called *hash* has been inserted. This field contains the measure of what layer of the Keystone architecture the certificate is associated to: for example if the certificate is associated to the Local Attestation key of an enclave, this field contains the hash value of the enclave.

The principal methods that are used to manage the X.509 certs are the following:

- `void mbedtls_x509write_crt_init(mbedtls_x509write_cert *ctx)`: this method is used to clean the memory of the `mbedtls_x509write_cert` variable and to set the version of the certificate to 3
- `int mbedtls_x509write_crt_set_issuer_name_mod(mbedtls_x509write_cert *ctx, const char *issuer_name)`: this method is used to set who is the issuer of the certificate
- `int mbedtls_x509write_crt_set_subject_name_mod(mbedtls_x509write_cert *ctx, const char *subject_name)`: this method is used to set who is the owner of the certificate
- `void mbedtls_x509write_crt_set_subject_key(mbedtls_x509write_cert *ctx, mbedtls_pk_context *key)`: this method is used to set the public key that will be inserted in the certificate
- `void mbedtls_x509write_crt_set_issuer_key(mbedtls_x509write_cert *ctx, mbedtls_pk_context *key)`: this method is used to set the private key that will be used to sign the certificate, so the key of the issuer
- `int mbedtls_x509write_crt_set_serial_raw(mbedtls_x509write_cert *ctx, unsigned char *serial, size_t serial_len)`: this method is used to set the serial of the certificate
- `void mbedtls_x509write_crt_set_md_alg(mbedtls_x509write_cert *ctx, mbedtls_md_type_t md_alg)`: this method is used to set the hash algorithm used to make the hash of the certificate before making the signature
- `int mbedtls_x509write_crt_set_validity(mbedtls_x509write_cert *ctx, const char *not_before, const char *not_after)`: this method is used to set the validity of the certificate
- `int mbedtls_x509write_crt_set_extension(mbedtls_x509write_cert *ctx, const char *oid, size_t oid_len, int critical, unsigned char *val, size_t val_len)`: this method is used to set the hash value of the component as extension of the X509 certificate
- `int mbedtls_x509write_crt_set_basic_constraints(mbedtls_x509write_cert *ctx, int is_ca, int max_pathlen)`: this method is used to add an extension to the final X.509 cert, that is needed when that certs are used to sign another certificate, so the subject acts as a Certification Authority
- `int mbedtls_x509write_crt_der(mbedtls_x509write_cert *ctx, unsigned char *buf, size_t size, int (*f_rng)(void *, unsigned char *, size_t), void *p_rng)`: this method is used to create an X.509 certificate in DER format, starting from a variable of type `mbedtls_x509write_cert` and signing it with the private key of the issuer provided in the structure

- `int mbedtls_x509_crt_parse_der(mbedtls_x509_crt *chain, unsigned char *buf, size_t buflen)`: this method is used to parse an X.509 certificate in DER format in a variable of type `mbedtls_x509_crt`

This library has been introduced in all the layers of the Keystone architecture where it is necessary: the booting phase is one of them but also at SM level.

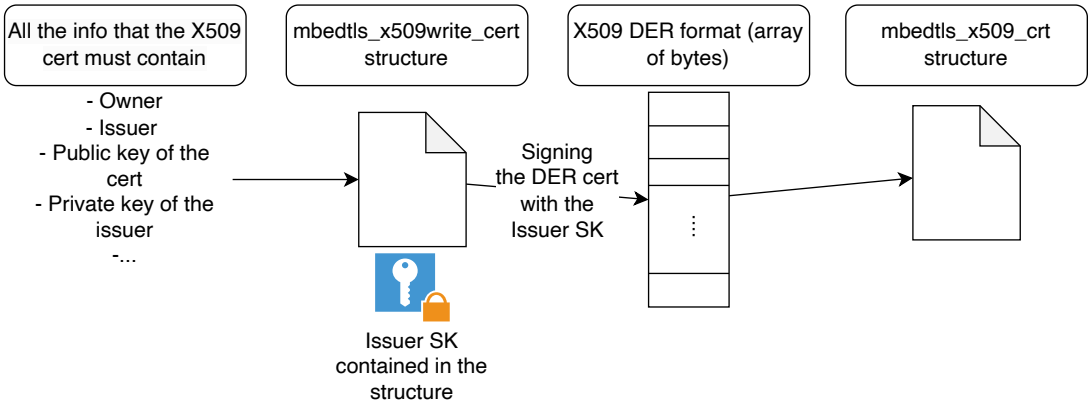

Figure 6.1. The flow to obtain a `mbedtls_x509_crt` variable

One of the most important changes was to make compatible the original files with the embedded functions of signature and verification already present in Keystone. In fact, the standard implementation of MBed_TLS does not foresee the usage of ed25519 and to make it possible some adjustments have been made to allow the correct functioning.

With this has been also defined the new type of extension that can be attached to the X.509 certs to allow the insertion of the hash value, extension not previously present and consequently also the methods of creating and parsing the certs and the structure above described have been modified accordingly. The other type of extension ported in the custom library is the extension related to the possibility of using a private key to sign other certs, so considering the subject like a Certification Authority.

6.2 DICE Engine

This section is used to describe all the stuff that have been made to implement the DICE Engine in the Keystone project. The first thing that can be said is that the original Keytone project comes with already defined a file, called `test_dev_key.h` that contains a keypair that is used in the project. The private part of this keypair as been associated to the so call in the DICE architecture, Unique Device Secret, instead the public part has been considered to be the public key inserted in the X.509 certificate of the manufacturer. (see Fig. 6.2)

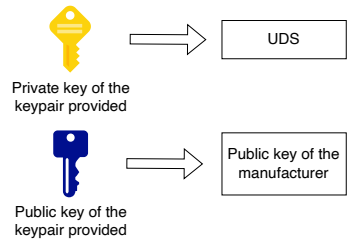

Figure 6.2. How the keypair of the original Keystone project is used

With the respect to the DICE specifications, the platform has to come with already the certificate of the manufacturer and the certificate associated to the Device Root Key. The first one has been created using a specific script that produces the DER format of the cert. After that it has been saved and inserted in a specific pre-defined variable that is provided to the booting stage. In this test situation, it is supposed that this cert is self-signed by the manufacturer, but in the real case it is signed by a Certification Authority of which the certificate can be easily found and so on until the real RoT. In the same way the certificate of the Device Root Key can be calculated and provided and this is done simulating the real scenario, but in the booting phase this certs in DER format is calculated again because the public key inserted on it, depends on the measure of the CDI of level 0 that depends on the measure of the SM. So, until the SM is modified, to make all the things working there is necessity to compute "on the fly" this cert (see Fig. 6.3).

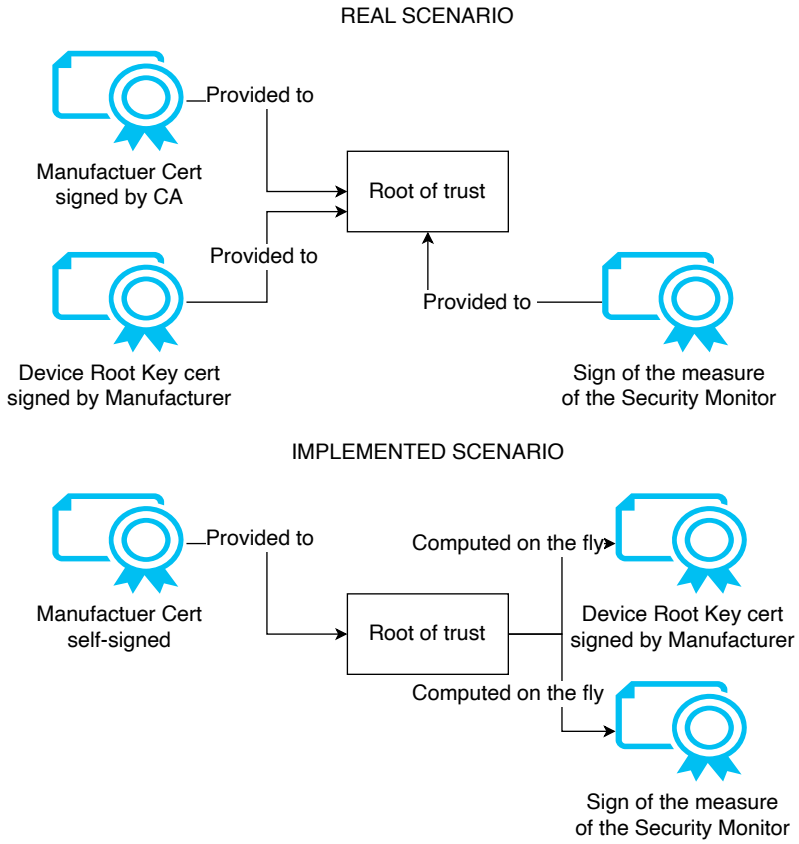

Figure 6.3. The real scenario vs the implemented scenario

Entering more in detail on what happens during the booting phase, that is the hardware layer that implements the DICE engine, can be said that the first thing that is done is the implementation of the secure boot mechanism (the signed measure of the SM is calculated "on the fly" for the same reason written above) (see Fig. 6.4):

1. the SM is measured
2. the signature is verified
3. if the signature is ok, the booting process continues and the end, if no other errors arrive, it returns 0, otherwise the value 1 is returned and the process is stopped with the next step:
   (a) the `bootloader()` function called by the `bootloader.S` file, ends and the return value, that is stored in the a0 register is saved in the s10 register.

(b) the file `fw_base.S` is called and it does all the operations that have to do, until the end, where the content of the s10 register is controlled and if it is wrong (1), it calls a specific function present in the file `sbi_init.c`, `void to_be_stopped(struct sbi_scratch *scratch, bool flag)`

(c) this function resumes the normal start calling the function `sbi_init()`, but passing it a flag with value 0 that symbolizes that the boot has to be stopped

(d) the function `sbi_init` sends the same flag to `init_coolboot()`, that is the first place where the process can be stopped calling the function `sbi_hart_hang()` and printing to the terminal an error message.

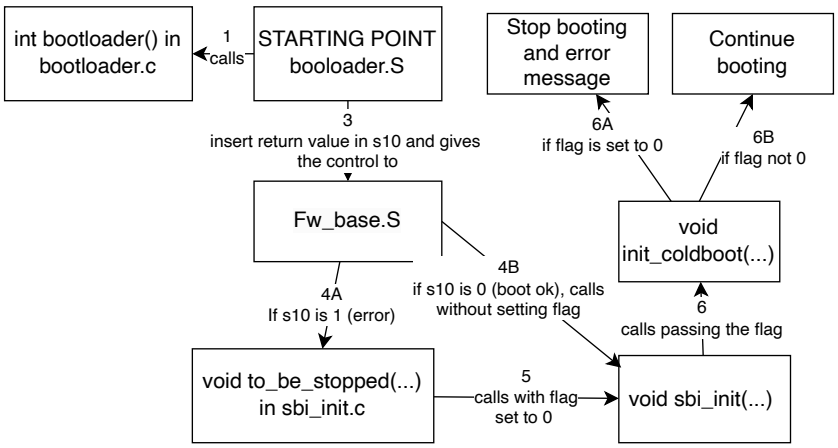

Figure 6.4. Secure boot flow

After that what is done is the following (see Fig. 6.5):

- The CDI of level 0 is calculated from the UDS and the measure of the SM
- the Device Root Keys are derived from the CDI
- the keypair associated to the Embedded Certification Authority is generated from the hash obtained concatenated the CDI value with the measure of the SM
- the X.509 certificate in DER format related to the public key of the ECA keypair is issued and signed with the private key of the Device Root Key keypair
- what has to be passed to the SM is inserted in the specified variables
- all memory related to the secret data (UDS, DRK_sk, ECA_sk) is freed

Due to the fact that each layer of Keystone is built independently, there is a specific mechanism (Lis. 6.2) that is used to make possible the communication between them, more in detail associated to the transfer of shared variables. This mechanism is related to some specific `.ldS` files (`.lds` is the extension of Linker Script file) that are used to specify where in memory the variables have to be stored and how much space they need. In this way, specifying the same linking option both in the bootloader and in SM, allow the two different layers to share variables. The variables that have to be passed from the booting stage to the SM are:

- CDI of layer 0
- the X.509 certificate of the manufacturer
- the length of the certificate of the manufacturer
- the X.509 certificate of the Device Root Key

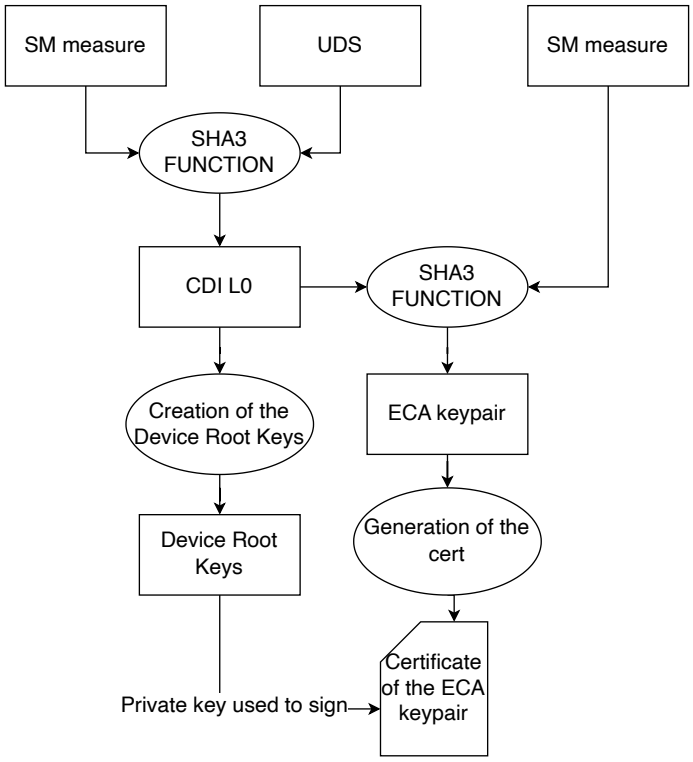

Figure 6.5. What happens during booting stage

- the length of the certificate of the Device Root Key
- the X.509 certificate of the ECA key
- the length of the certificate of the ECA key

Listing 6.2. The configuration file for the linker

```
. = 0x801ff000; /* the last page before the payload */
.
.
/* 64 Bytes : security monitor's signature by device */
PROVIDE( sanctum_CDI = . );
. += 0x40;
/* 512 Bytes : security monitor's signature by device */
PROVIDE( sanctum_cert_man = . );
. += 0x200;
/* 512 Bytes : security monitor's signature by device */
PROVIDE( sanctum_cert_sm = . );
. += 0x200;
```

6.3 Security Monitor

When the control passes to the SM, with the respect to the original implementation, at the beginning it stores the variable that the Booting phase as passed to it (if all goes well) in some

internal variables and then parses all the certificate in DER format inside variables of type `mbedtls_x509_crt`. All the process stops, if there are some problems with the parsing of the certificates (see Fig. 6.6).

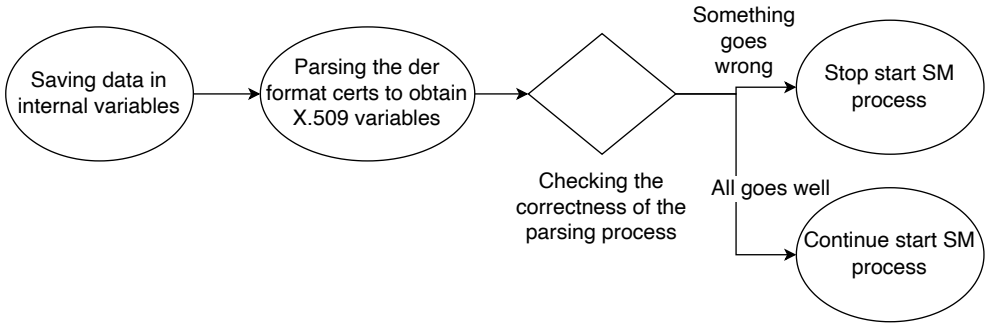

Figure 6.6. The initial operations at SM level

At this point the content of the different certificates can be print on the screen to check for example if some fields inserted in the original `mbedtls_x509write_cert` has been correctly inserted and passed. After this a more important thing is that the SM has to do are:

1. To check that all the X.509 certificates are formally correct and does not have some needed fields missing

2. To verify the signature of each certificates until the certificate of the manufacturer that it is supposed to be the RoT in the implementation. To make this thing possible, each variable of type `mbedtls_x509_crt` has a specific field called *tbs* that contains the data on which the signature has been made. This field is built during the parsing process and if the certificate has been manipulated what has been inserted in the field is not more the original data that have been used to make the signature, so the verification will fail. So, this field has to be used in the first moment to do the hash and this hash will be used to verify the signature of the certificate using the correct public key. If for some reason the verification of each one of the certificate does not go well the all the system stops working (see Fig. 6.7).

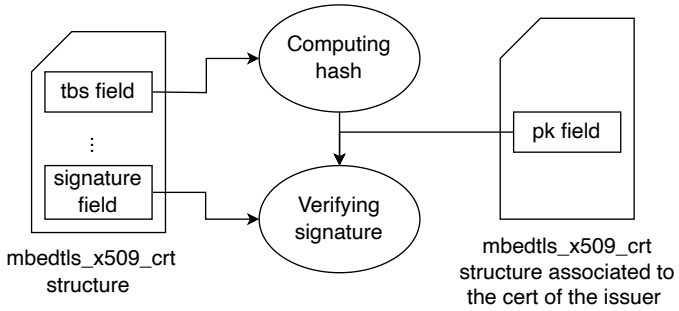

Figure 6.7. How the signature verification process works

3. To derive the private key of the Embedded Certification Authority, starting from the CDI of the level 0 and itself measure that can be found in the field called *hash* of the `mbedtls_x509_crt` variable obtained parsing the ECA key certificate (see Fig. 6.8).

After this, other important operations that have been implemented at the SM layer are the operations that have to be done when there is the creation of a new enclave. In fact, for each enclave that has to be created, the associated CDI has to be calculated starting from the CDI provided from the Booting stage to the SM and the TCI of the specific enclave. This value

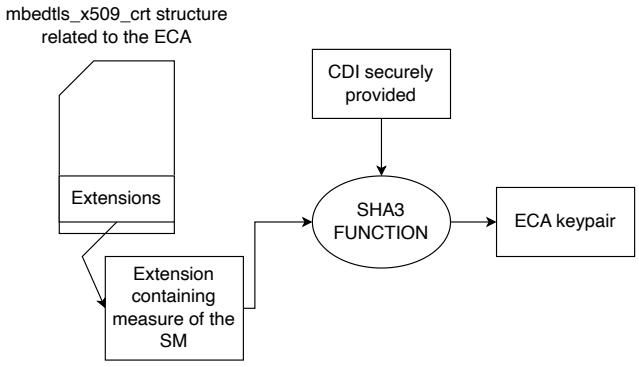

Figure 6.8. How the private key of the ECA keypair is calculated

is stored in a variable added in the structure used to manage the information related to the enclave, not present in the starting implementation. Moreover, also a keypair has been created by default for each enclave, the Local Attestation Key that has also to be certified by the Embedded Certification Authority always during the creation process. The seed used to obtain this keypair are the first 32 bytes of the CDI of the enclave (see Fig. 6.9).

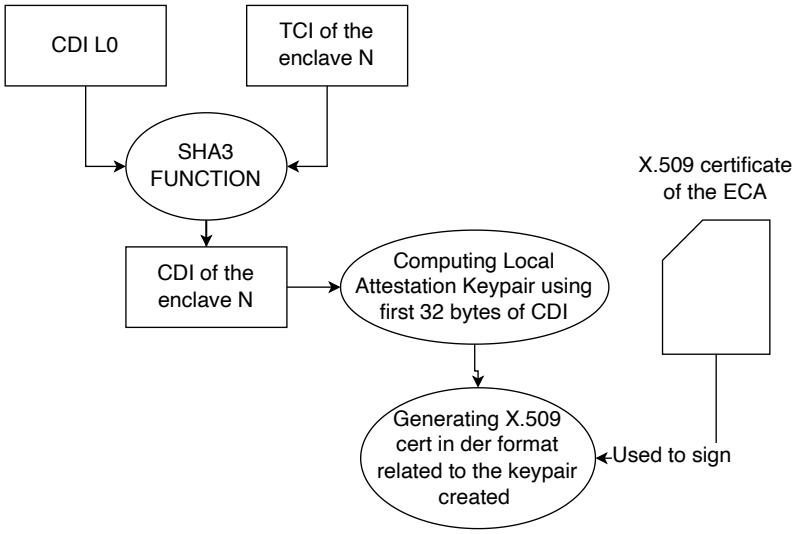

Figure 6.9. Computing the CDI of the enclave and its Local Attestation key

Other variables added to the enclave structure are the following:

- `pk_ldev` and `sk_ldev`: keypair used to store the Local Device keypair, that is a keypair that have to be certified by a remote attester
- `sk_array` and `pk_array`: array used to store all the keypairs associated to the specific enclave

The last things that have been implemented to satisfy the proposed design are two different functions, that can be called by the different enclaves to satisfy their necessities in terms of the management of their keypairs. These methods are called following the schema explained in the previous chapter and are:

- `unsigned long create_keypair(enclave_id eid, unsigned char* pk, int seed_enc)`: function called by the enclave to ask the SM to create a new keypair that will be stored in the two related array (see Fig. 6.10).

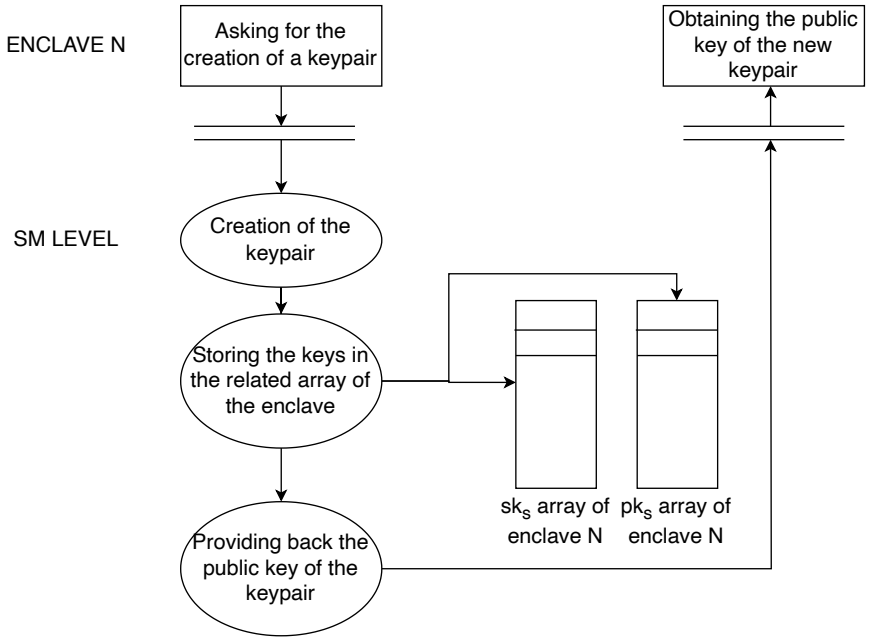

Figure 6.10. Creation of a keypair for a specific enclave

- `unsigned long get_cert_chain(enclave_id eid, unsigned char** certs, int* sizes)`: function used to get the cert chain and all the related information.
- `unsigned long do_crypto_op(enclave_id eid, int flag, unsigned char* data, int data_len, unsigned char* out_data, int* len_out_data, unsigned char* pk)`: function that acts as an interface that integrates some cryptographic operations that can be made with private keys. The enclave specifies what has to be one passing a specific flag. One of the function is the creation of the signature related to the TCI of the enclave and the public part of the Local Device key used to obtain the remote certification. The SM knows what is the enclave that asks for the operation, so can easily find the related Local Attestation key that has to be used for the porpoise. The other one is used to make a generic signature of some data given by the enclave in hash format, using the private key associated to the public one always provided by the enclave. Other functions can be implemented if they will be needed.

Chapter 7

Test sets for the proposed solution

This chapter is used to show some tests that have been made on the proposed solution to integrate the DICE specifications into the Keystone project. Two types of tests are performed:

- Functional tests: tests used to show the features that have been added in the Keystone project
- Performance tests: tests used to see what is the overhead added into the Keystone project from the features introduced to satisfy the DICE specifications

7.1 Testbed description

All the features that have been introduced from the booting phase to the SM, have been successfully tested using the QEMU emulator and the script provided by the original Keystone project, that allow the user to launch it and check what has been done.

So, all the test can be done in a single physical machine, that must support the configuration and the installation of the Keystone project. More in detail the machine used to perform all the tests is the following: Lenovo Ideapad 720s 13-ARR. This machine has:

- CPU: AMD Ryzen 5 2500u with radeon vega gfx x 8
- Storage: 1TB
- RAM: 8GB
- OS: Ubuntu 20.04.6 LTS 64 bit

7.2 Functional tests

Different functional tests have been done to check if the features added work properly.

The first test presented here is the test used to see the Secure Boot mechanism: to simulate the failure of the verification of the signature, provided by the manufacturer, of the SM, the measure computed during the booting process has been modified changing a byte with a random value. In this way when the function provided by the `ed25519.h` to check the signature is called, it returns the 0 value that means that there was a problem verifying it. After this the mechanism discussed in the previous chapter related to the Secure Boot when there is a problem, starts, and what can be seen launching QEMU is the shown in the Fig. 7.1.

Arriving to the SM level, the first test that has been implemented is the test related to a possible failure that can happen when the X.509 certificate generated during the booting process

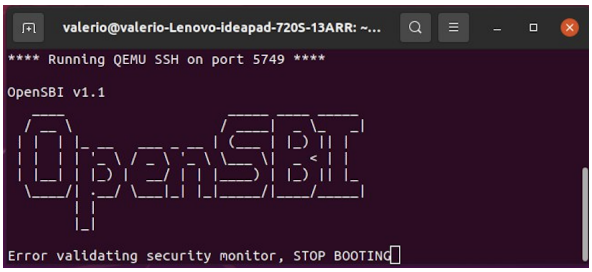

Figure 7.1. Secure Boot test

associated to the keypair of the Embedded CA has to be parsed from the DER format into the specific structure. The test has been achieved changing in the related parsing function the field related to the version of the certificate, simulating that there was an error in a field when the certificate has been produced. If this type of error occurs, like in the other cases, the system stops, otherwise if all the certificate has been correctly parsed, a message on the screen is printed (see Fig. 7.2).

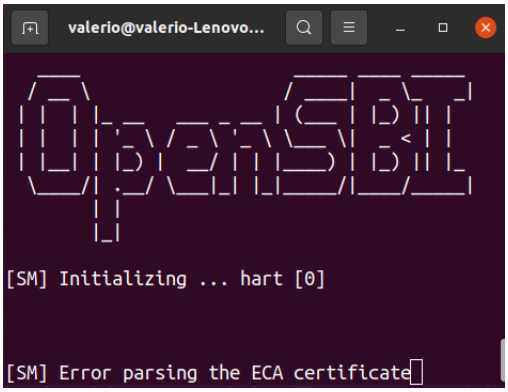

Figure 7.2. Failure during parsing

With the same philosophy, also a test to see if there is an error in a field of the certificate after that it has been parsed, has been made. The certificate structure has lots of fields, but someone of these are mandatory, for example the public key, the subject, the issuer, the expiration date and so on. It can happen that the certificate is correctly parsed but some of these fields are not present. So a function used to check the correct fulfillment is called after the parsing process and if there is at least one error, a message related to that is printed on the screen and the booting process stops (Fig. 7.3)

If no error occurs, a message that the validation process has gone well is printed on the screen. The most important test made is the test related to the verification of the different signatures of the certificates from the cert of the ECA to the manufacturer certificate, a test made to be sure that no changes have been done to the X.509 certificates. Also in this situation an error determines the stop of the system and otherwise a message is printed (Fig. 7.4). To be in this scenario, a byte of one of the provided certs has been changed in a way that the signature cannot be verified.

After this, a test that has been made is related to check if the value inserted in the extension of the Embedded CA certificate is the same value of the SM measure passed from the original implementation of Keystone. This test is very useful because allow to understand if the different features have been correctly made (see Fig. 7.6 and Fig. 7.5); to obtain equivalent values, the functionalities that have to work properly are:

- creation of X.509 extension with the proper value

Figure 7.3. Failure during validating fields

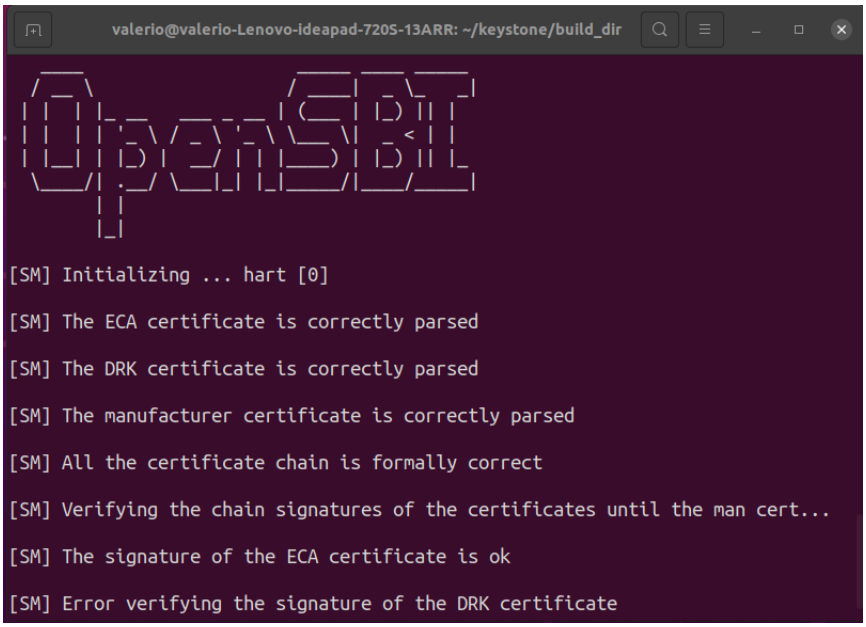

Figure 7.4. Failure during verifying certs

- creation of the structure `mbedtls_x509write_cert` with the associated value
- parsing in DER format of the above structure
- parsing into `mbedtls_x509_crt` structure
- exchanging of shared variables between different layers

The last test done is related to check if the different SBIs exposed by the SM to the enclave work properly. What has been implemented are three different interactions between the EApp and the SM, each one used to check a specific function provided:

- the first one have been done to see the creation of a new keypair giving to the SM a specific seed
- the second one have been done to obtain from the SM the full chain of the cert in DER format with their length

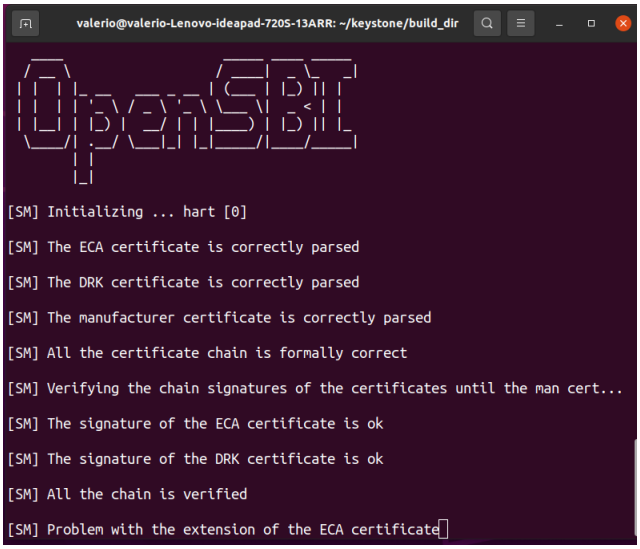

Figure 7.5. Failure during checking the equivalence of the SM measure with the value of the original implementation of Keystone

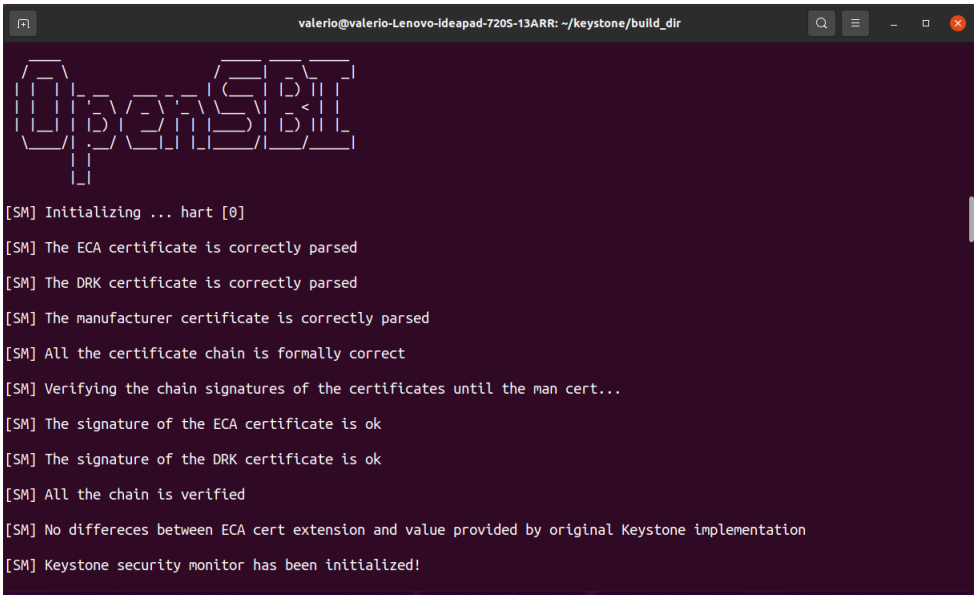

Figure 7.6. Correct start of the system

- the third one have been done to test all the methods of the crypto interface implemented in the SM

Each one of this test as associated to a *.sh* file present in the scripts folder of the build directory where the Keystone project has been installed except of the last one that is associated to a package (*.ke*). The different files are:

- `error_sb.sh`: to see what happens if the SM is compromised
- `error_parsing_crt.sh`: to see what happens if the certs cannot be correctly parsed
- `error_validating_crt.sh`: to see what happens if the parsed certs don't have the same required fields
- `error_verifying_crt.sh`: to see if the chain cannot be entirely verified

- `correct.sh`: to see what happens if the system can be correctly launched
- `hello-native.ke`: to check the new SBIs

More detail on how to launch this scripts and the package will be given in the user's manual.

7.3 Performance test

To test how the performance have been downgraded due to the introduction of all the features needed to implement the DICE specifications, the number of ticks necessary to enter a specific situation have been taken. This is possible thanks to a specific library, called `sbi_time.h` that is provided with the original Keystone project and gives the user to know the number of ticks passed from when the timer has been activated using a specific function called `u64 sbi_timer_value(void)`. Using these, the time has been taken in three different situations:

- when the process to start the SM is initialized
- when the SM has been initialized
- when the process to start the enclave is initialized
- when the enclave has been initialized

These numbers have been used to generate three different comparisons that show the difference from the original project and the custom one in terms of how much time is needed to: making the booting process, starting the SM and starting an enclave.

In the figure (Fig. 7.7), it can be seen that there is a performance penalty due to the different operations that have been introduced in the booting stage; the order of magnitude of the value doesn't change but at the same time it increases of 40% more or less. This is due to the fact that an X.509 certificate has to be made and parsed in DER format, but the most expensive operation is the measurement of the SM and the verification of the signature provided by the Manufacturer, as it can be seen in the second situation.

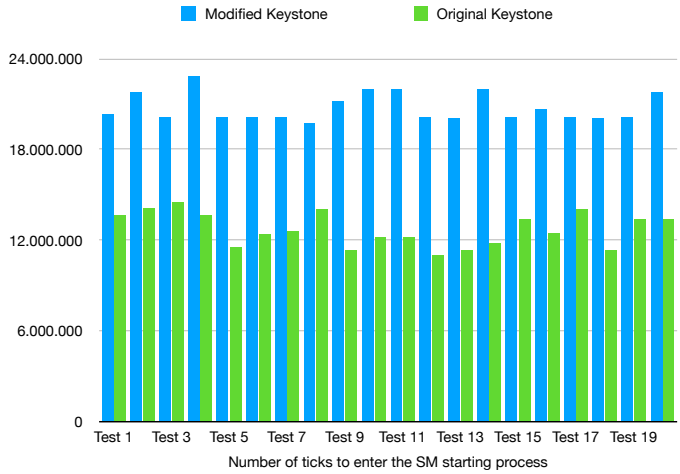

Figure 7.7. Comparison of the time needed to complete the booting process

Differently from the previous comparison, the performances are subject to an important deterioration (Fig. 7.8): the order of magnitude of the ticks changes and this is due to the fact that during the starting of the SM all the chain of the certificates, from the cert of the ECA

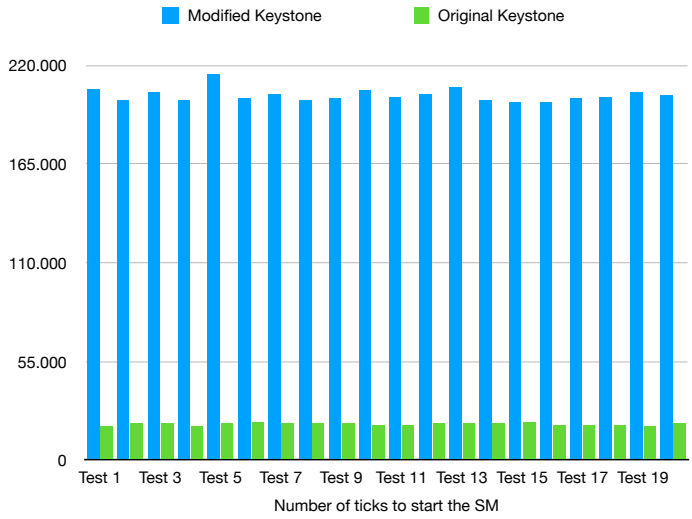

Figure 7.8. Comparison of the time needed to start the SM

to the manufacturer cert, has to be verified. Consequently, two times have to be made hashing operations and for two times a signature has to be verified. Checking how far are the values in the tests, can be deduced that these types of operations are very expensive, differently from the operations of issuing X509 cert, that have not done in this phase: comparing the results of the first test with these, it can be deduced that these types of operations are not so CPU-consuming.

In the last comparison (Fig. 7.9) associated to the time needed to create an enclave, it can be seen that the performances are not so affected after the introduction of the new operations. These are related principally to the creation of the X.509 certificate in DER format of the Local Attestation Key associated to the enclave. The order of magnitude doesn't change and the values don't defer so much, other proof that both the issuing of an X.509 cert and the generation of keypairs are operations CPU-friendly.

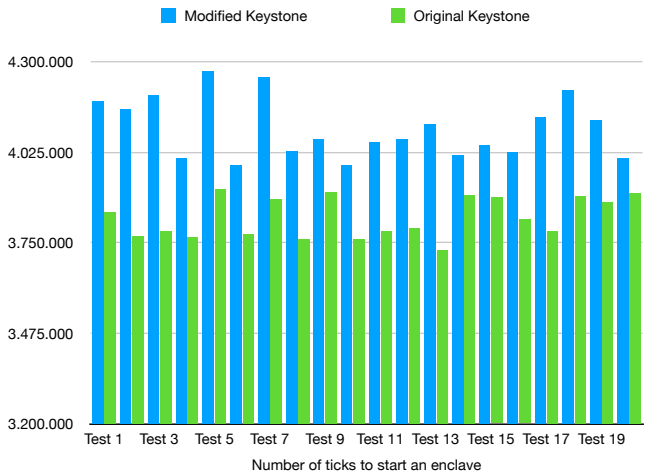

Figure 7.9. Comparison of the time needed to start an enclave

Chapter 8

Conclusions

All that has been presented in this work has been done to show the state of the art of the actual technologies present in the market, related to the TEEs and also to exhibit what are the main characteristics of the DICE specification and of the Keystone enclave.

The design proposed can be implemented in the reality to obtain all the security features that are needed to be sure about the integrity of a platform and its correct operation. Using the Keystone project as the basis for the introduction of the DICE Engine, allow the users to have under their hands secure environment that are completely modifiable depending on the different needed, property not present using proprietary technology like for example Apple Secure Enclave.

The RISC-V architecture, used to run the Keystone project, is perfect to be used on the Internet of Things devices, due to its flexibility and the property to be adjustable. All that has been introduced, compared with the original Keystone project, is something that can be managed also in devices that are not so powerful, also if the performances a little make worse, because to be properly implemented the DICE specification foresees the presence of X.509 certificates that have to be verified in the different layers.

The goal presented in the Introduction can be considered completed and this work can be the starting point for other projects due to what has been inserted to respect the DICE specification; an example can be the dynamic attestation: process implemented while the different enclaves are running that measures them, certifies the measures with the LAK and then provides the TCIs to a remote verifier that knows the excepted value and check if what has been obtained from the platform is correct or not, implementing a way to stop the specific enclave if it is under the control of a malicious actor. Others future works can be for example the implementation of a mechanism that check statically when an enclave is created if it is compromised or to make a remote attestation of a specific key created by the SM for the enclave.

Bibliography

[1] "About TCG", https://trustedcomputinggroup.org/about/

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

[3] Rohit, S. Kamra, M. Sharma, and A. Leekha, "Secure Hashing Algorithms and Their Comparison", 2019 6th International Conference on Computing for Sustainable Global Development (INDIACom), New Delhi (India), March 13-19, 2019, pp. 788–792

[4] The TCG group, "DICE Layering Architecture", https://trustedcomputinggroup.org/wp-content/uploads/DICE-Layering-Architecture-r19_pub.pdf

[5] L. A. Levin, "The tale of one-way functions", Problems of Information Transmission, vol. 39, Jan 2003, pp. 92–103, DOI 10.1023/A:1023634616182

[6] S. S. D.Cooper, NIST, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC-5280, May 2008, DOI 10.17487/RFC5280

[7] Z. Meng and Y. Wang, "Asymmetric Encryption Algorithms: Primitives and Applications", 2nd International Conference on Electronic Technology, Communication and Information (ICETCI), Changchun (China), May 27-29, 2022, pp. 876–881, DOI 10.1109/ICETCI55101.2022.9832032

[8] S. Chandra, S. Bhattacharyya, S. Paira, and S. S. Alam, "A study and analysis on symmetric cryptography", International Conference on Science Engineering and Management Research (ICSEMR), Chennai (India), November 27-29, 2014, pp. 1–8, DOI 10.1109/ICSEMR.2014.7043664

[9] D. Johnson, A. Menezes, and S. Vanstone, "The Elliptic Curve Digital Signature Algorithm (ECDSA)", International Journal of Information Security, vol. 1, Aug 2001, pp. 36–63, DOI 10.1007/s102070100002

[10] X. Zhou and X. Tang, "Research and implementation of RSA algorithm for encryption and decryption", 6th International Forum on Strategic Technology, Harbin (China), August 22-24, 2011, pp. 1118–1121, DOI 10.1109/IFOST.2011.6021216

[11] The TCG group, "DICE Attestation Architecture", https://trustedcomputinggroup.org/wp-content/uploads/TCG_DICE_Attestation_Architecture_r22_02dec2020.pdf

[12] M. Sabt, M. Achemlal, and A. Bouabdallah, "Trusted Execution Environment: What It is, and What It is Not", IEEE Trustcom/BigDataSE/ISPA, Helsinki (Finland), August 20-22, 2015, pp. 57–64, DOI 10.1109/Trustcom.2015.357

[13] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", 2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC), Pittsburgh (PA,USA), November 1-3, 2016, pp. 445–451, DOI 10.1109/CIC.2016.065

[14] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A virtual machine-based platform for trusted computing", Operating Systems Review (ACM), vol. 37, September 2003, pp. 193–206, DOI 10.1145/945445.945464

[15] OMTP Limited, "Advanced Trusted Environment", http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

[16] Global platform, "TEE System Architecture v1.3", https://globalplatform.org/specs-library/tee-system-architecture/

[17] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone Explained: Architectural Features and Use Cases", 2016 IEEE 2nd International Conference on Collaboration

and Internet Computing (CIC), Pittsburgh (PA, USA), November 1-3, 2016, pp. 445–451, DOI 10.1109/CIC.2016.065

[18] MITRE, "Screen capture attack", https://attack.mitre.org/techniques/T1113/

[19] M. Srivastava, A. Kumari, K. K. Dwivedi, S. Jain, and V. Saxena, "Analysis and Implementation of Novel Keylogger Technique", 5th International Conference on Information Systems and Computer Networks (ISCON), Mathura (India), October 22-23, 2021, pp. 1–6, DOI 10.1109/ISCON52037.2021.9702433

[20] M. A. Ivanov, B. V. Kliuchnikova, I. V. Chugunkov, and A. M. Plaksina, "Phishing Attacks and Protection Against Them", IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), St. Petersburg and Moscow (Russia), January 26-29, 2021, pp. 425–428, DOI 10.1109/ElConRus51938.2021.9396693

[21] Microsoft corporation, "System Management Mode deep dive: How SMM isolation hardens the platform", https://www.microsoft.com/en-us/security/blog/2020/11/12/system-management-mode-deep-dive-how-smm-isolation-hardens-the-platform/

[22] J. Sobchuk, S. O'Melia, D. Utin, and R. Khazan, "Leveraging Intel SGX Technology to Protect Security-Sensitive Applications", IEEE 17th International Symposium on Network Computing and Applications (NCA), Cambridge (MA, USA), November 1-3, 2018, pp. 1–5, DOI 10.1109/NCA.2018.8548184

[23] David Kaplan, Jeremy Powell, Tom Woller, "AMD MEMORY ENCRYPTION", October 18, 2021, https://www.amd.com/system/files/TechDocs/memory-encryption-white-paper.pdf

[24] IBM corporation, "Introducing IBM Secure Execution for Linux 1.3.0", November, 2022, https://www.ibm.com/docs/en/linuxonibm/pdf/l130se03.pdf

[25] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: an open framework for architecting trusted execution environments", Heraklion (Greece), April 17, 2020, pp. 1–16, DOI 10.1145/3342195.3387532

[26] Global Platform, "Introduction to Secure Elements", May, 2018, https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Secure-Element-15May2018.pdf

[27] Global Platform, "Introduction to Trusted Execution Environments", May, 2018, https://globalplatform.org/wp-content/uploads/2018/05/Introduction-to-Trusted-Execution-Environment-15May2018.pdf

[28] P. B. Hemanthkumar, R. A. Shreekar, F. T. Josh, and R. Venkatesan, "Introduction to ARM processors, its types and Overview to Cortex M series with deep explanation of each of the processors in this Family", International Conference on Computer Communication and Informatics (ICCCI), Coimbatore (India), January 25-27,2022, pp. 1–8, DOI 10.1109/ICCCI54379.2022.9740768

[29] S. Arrag, "Design and Implementation A different Architectures of mixcolumn in FPGA", International Journal of VLSI Design and Communication Systems, vol. 3, August 2012, pp. 11–22, DOI 10.5121/vlsic.2012.3402

[30] A. Raveendran, V. B. Patil, D. Selvakumar, and V. Desalphine, "A RISC-V instruction set processor-micro-architecture design and analysis", 2016 International Conference on VLSI Systems, Architectures, Technology and Applications (VLSI-SATA), Bengaluru (India), January 10-12, 2016, pp. 1–7, DOI 10.1109/VLSI-SATA.2016.7593047

[31] Global Platoform, https://globalplatform.org

[32] Open Platform project, "About OP-TEE", https://optee.readthedocs.io/en/latest/general/about.html

[33] Apple corporation, "Secure Enclave", https://support.apple.com/pl-pl/guide/security/sec59b0b31ff/web

[34] Microsoft corporation, "Microsoft Pluton security processor", May 12, 2023, https://learn.microsoft.com/en-us/windows/security/information-protection/pluton/microsoft-pluton-security-processor

Appendix A

User's Manual

A.1 System requirements

A.1.1 Keystone enclave

To deploy the same environment proposed in the previous chapter, the Keystone project has to be compatible with the physical platform that will be used because a modified instance of it and the original one have to be installed in order to repeat the performance and functional tests proposed. (The two different projects are needed only for the performance tests)
All the operations needed to do that are the following:

1. install dependencies

```
$ 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
        python3 \
    pkg-config libglib2.0-dev libpixman-1-dev libssl-dev screen \
    device-tree-compiler expect makeself unzip cpio rsync cmake
        ninja-build p7zip-full
$ curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
$ rustup toolchain install nightly
$ rustup +nightly component add rust-src
$ rustup +nightly target add riscv64gc-unknown-none-elf
$ cargo +nightly install cargo-xbuild
```

2. install the original version of the Keystone project (skip if don't want to redo performance tests)

```
$ git clone https://github.com/keystone-enclave/keystone
$ cd keystone
$ ./fast-setup.sh (once it has finished do the operation written
    in the screen)
$ source source.sh
```

3. build original Keystone framework

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

```
$ make
$ make image
```

4. install the modified version of the Keystone project (the root directory has to be different)

```
$ git clone https://github.com/valerio1805/my_keystone.git (and
    rename the folder in keystone)
$ cd keystone
$ ./fast-setup_edit.sh (once it has finished do the operation
    written in the screen)
$ source source.sh
$ sudo cp ./original_files/boot.c ./qemu/hw/riscv/
$ sudo cp ./original_files/virt.c ./qemu/hw/riscv/
$ sudo cp ./original_files/virt.h ./qemu/include/hw/riscv/
$ sudo cp ./original_files/fw_base.ldS ./sm/opensbi/firmware
$ chmod 777 -R ./sm
```

5. build modified Keystone framework

```
$ mkdir build_dir
$ cd build_dir
$ cmake ..
$ make buildroot KBUILD_MODPOST_WARN=1
$ make qemu
$ make linux KBUILD_MODPOST_WARN=1
$ make sm
$ make bootrom
$ make driver
$ cp ../hello-native.ke ./overlay/root/
$ make image
```

A.2 Performing tests

A.2.1 Functional tests

To run functional tests associated to *.sh* scripts (using the edited version of Keystone):

1. move all the content of *tests_folder* into the folder *./build_dir/scripts*
2. open each *.sh* file and change the installation directory of the keystone project of the field *rom*, *bios*, *kernel* and *file* according to your setup
3. give the permission to the *.sh* files to be executed (if needed)
4. go to the build directory (*build_dir* according to this guide)
5. run each test [example: *./scripts/error_sb.sh*]

To run the functional test related to the *hello-native.ke* package (using the edited version of Keystone), from the build directory (*build_dir* according to this guide) do:

```
$ ./script/run-qemu.sh (username: root password: sifive)
$ insmod keystone-driver.ko
$ ./hello-native.ke (if it is not executable, give to it the permission
    with the chmod 777 command)
```

A.2.2 Performance tests

To run performance test:

1. replace the files *sm.c* and *enclave.c* of the original Keystone project with the two provided in the folder *edited_files* of the modified version of Keystone in the path [*installation_path*]/*keystone*/*sm*/*src*/
2. go to the build directory of the original Keystone project and do:

```
$ make sm
$ make hello-package
$ cp examples/hello/hello.ke ./overlay/root
$ make image
$ ./scripts/run-qemu.sh (username: root password: sifive)
$ insmod keystone-driver.ko
$ ./hello.ke
```

3. go to the build directory of the edited Keystone project and do:

```
$ make hello-package
$ cp examples/hello/hello.ke ./overlay/root
$ make image
$ ./scripts/run-qemu.sh (username: root password: sifive)
$ insmod keystone-driver.ko
$ ./hello.ke
```

4. comparing the two different terminals to see what has been presented in the seventh chapter

Appendix B

Developer's Guide

B.1 How the manufacturer cert is created

The code below has been used to create the cert associated to the manufacturer, that in the reality comes provided with the platform. (Lis. B.1)

Listing B.1. How the manufacturer cert is created

```
 mbedtls_x509write_cert cert_man;
 mbedtls_x509write_crt_init(&cert_man);
 // Setting subject and issuer of the issuer of the cert
  ret = mbedtls_x509write_crt_set_issuer_name_mod(&cert_man,
      "O=Manufacturer");
  ret = mbedtls_x509write_crt_set_subject_name_mod(&cert_man,
      "O=Manufacturer");
...

 // Parsing the private key of the embedded CA that will be used to sign the
     certificate of the security monitor
 ret = mbedtls_pk_parse_public_key(&issu_key_man, sanctum_dev_secret_key,
     64, 1);
 ret = mbedtls_pk_parse_public_key(&issu_key_man, sanctum_dev_public_key,
     32, 0);

 // Parsing the public key of the security monitor that will be inserted in
     its certificate
 ret = mbedtls_pk_parse_public_key(&subj_key_man, sanctum_dev_public_key,
     32, 0);

 // Setting serial, validity, algorithm that has to be used fot the
     signature and the keys inside the structure
mbedtls_x509write_crt_set_subject_key(&cert_man, &subj_key_man);
mbedtls_x509write_crt_set_issuer_key(&cert_man, &issu_key_man);
mbedtls_x509write_crt_set_serial_raw(&cert_man, serial_man, 3);
mbedtls_x509write_crt_set_md_alg(&cert_man, KEYSTONE_SHA3);
ret = mbedtls_x509write_crt_set_validity(&cert_man, "20230101000000",
    "20240101000000");

// Setting manufacturer cert as a certificate used to sign other certs
mbedtls_x509write_crt_set_basic_constraints(&cert_man, 1, 10);
```

```
// The structure mbedtls_x509write_cert is parsed to create a X.509 cert in
    DER format, signed and written in memory
ret = mbedtls_x509write_crt_der(&cert_man, cert_der_man, 1024, NULL,
    NULL);//, test, &len);
if (ret != 0)
{
  effe_len_cert_der_man = ret;
}
unsigned char *cert_real_man = cert_der_man;
// effe_len_cert_der stands for the length of the cert, placed starting
    from the end of the //buffer cert_der
int dif_man = 1024-effe_len_cert_der_man;
// cert_real points to the starts of the cert in DER format
cert_real_man += dif_man;
```

B.2 How the SM cert and CDI are created

The code below has been used to create the cert associated to the Embedded Certification Authority, placed at SM level and to derive the CDI of the level 0. (Lis. B.2)

Listing B.2. How the SM cert and CDI are created

```
// Combine hash of the security monitor and the device root key to obtain the
    CDI
  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, sanctum_uds, sizeof(*sanctum_uds));
  sha3_update(&hash_ctx, sanctum_sm_hash, sizeof(*sanctum_sm_hash));
  sha3_final(sanctum_CDI, &hash_ctx);

  // The device root keys are created from the CDI
  // This keys are certified by the manufacuter and the cert is stored in
      memory, like the cert of the manufacturer
  ed25519_create_keypair(sanctum_device_root_key_pub,
      sanctum_device_root_key_priv, sanctum_CDI);

  // The ECA keys are obtained starting from a seed generated hashing the CDI
      and the measure of the SM
  unsigned char seed_for_ECA_keys[64];
  sha3_init(&hash_ctx, 64);
  sha3_update(&hash_ctx, sanctum_CDI, 64);
  sha3_update(&hash_ctx, sanctum_sm_hash, 64);
  sha3_final(seed_for_ECA_keys, &hash_ctx);
  ed25519_create_keypair(sanctum_ECASM_pk, sanctum_ECASM_priv,
      seed_for_ECA_keys);

  // The process is the same with the respect to the previous described to
      the generation of the manufacturer certs, the difference are related to
      what parameters are passed to the different functions
  ...

  // Define and set the two extensions of the certificate: the hash of the
      security monitor and the possibility to use this cert to sign other
      certs
```

```
mbedtls_x509write_crt_set_extension(&cert, oid_ext, 3, 0, sanctum_sm_hash,
    65);
mbedtls_x509write_crt_set_basic_constraints(&cert, 1, 10);

// The creation process ends in the same way of the previuos creation
    proposed
...

// Erasing the memory
memset((void *)sanctum_ECASM_priv, 0, sizeof(*sanctum_ECASM_priv));
memset((void *)sanctum_device_root_key_priv, 0,
    sizeof(*sanctum_device_root_key_priv));
memset((void *)sanctum_dev_secret_key, 0, sizeof(*sanctum_dev_secret_key));
```

B.3 How the variables have been copied and how the formal structure of the X.509 DER format is controlled

The code below has been used to moving in internal variables of the SM level all the data that have been passed from the Booting stage; after this, the verification of the correct parsing of the certificate provided in DER format is controlled. (Lis. B.3)

Listing B.3. Copying variables and verifying the correct parsing of the certs at SM level

```
// All the variables passed from the boot stage are copied in sm variables
sbi_memcpy(CDI, sanctum_CDI, 64);
sbi_memcpy(cert_sm, sanctum_cert_sm, sanctum_length_cert);
sbi_memcpy(cert_root, sanctum_cert_root, sanctum_length_cert_root);
sbi_memcpy(cert_man, sanctum_cert_man, sanctum_length_cert_man);
length_cert = sanctum_length_cert;
length_cert_root = sanctum_length_cert_root;
length_cert_man = sanctum_length_cert_man;

// The different certs are parsed if there are no problems
if ((mbedtls_x509_crt_parse_der(&uff_cert_sm, cert_sm, length_cert)) != 0){
    // If there are some problems parsing a cert, all the start process is
        stopped
    sbi_printf("\n\n\n[SM] Error parsing the ECA cert created during the
        booting process");
    sbi_hart_hang();
}
else{
  sbi_printf("\n[SM] The certificate of the security monitor is correctly
      parsed\n\n");
}
// For the other certs is the same
```

B.4 How the different certificates have been verified and how the keys of the ECA are derived

The code below has been used at SM level to check the signatures inserted in the different X.509 certs provided and in the second part the derivation of the ECA keys has been done. (Lis. B.4)

Listing B.4. Verifying the signatures and deriving ECA SM private key

```
// Check that all the certs in the chain are formally correct usually the
    defined function
char* str_ret = validation(uff_cert_sm);
if(my_strlen(str_ret) != 0){
  sbi_printf("[SM] Problem with the ECA certificate: %s \n\n", str_ret);
  sbi_hart_hang();

}
else
{
// The same for the other two certs
...
}

// Once the cert in DER format is parsed, there is a field inserted in the
    structure that represents the raw data of the cert that is used to
    compute the hash
// Using this field, the sm can verify the signature inserted in his cert,
    using the public key of the issuer (in this case the issuer is the RoT)

sha3_init(&ctx_hash, 64);
sha3_update(&ctx_hash, uff_cert_sm.tbs.p, uff_cert_sm.tbs.len);
sha3_final(hash_for_verification, &ctx_hash);
sbi_printf("[SM] Verifying the chain signature of the certificates until
    the man cert...\n\n");

if(ed25519_verify(uff_cert_sm.sig.p, hash_for_verification, 64,
    uff_cert_root.pk.pk_ctx.pub_key) == 0){
  sbi_printf("[SM] Error verifying the signature of the sm
      certificate\n\n");
  sbi_hart_hang();
}
else{
 // The same is repetaed for the cert associated to the DRK
 }

// From the CDI and its measure inserted as extension in the ECA keys
    certificate,
// the sm can directly obtain the keys associated to the emebedded CA
// that are used to signed the cert associated to the attestation key of
    the different enclaves
sha3_init(&ctx_hash, 64);
sha3_update(&ctx_hash, CDI, 64);
sha3_update(&ctx_hash, uff_cert_sm.hash.p, 64);
sha3_final(seed_for_ECA_keys, &ctx_hash);

ed25519_create_keypair(ECASM_pk, ECASM_priv, seed_for_ECA_keys);
```

```
//Checking the equality between the SM TCI inserted in the extension of the
    ECA cert with the value provided to the SM by the original Keystone
    implementation
if(my_memcmp(uff_cert_sm.hash.p, sm_hash, 64) != 0){
  sbi_printf("[SM] Problem with the extension of the ECA certificate");
  sbi_hart_hang();
}
else
    sbi_printf("[SM] No differeces between ECA cert extension and value
        provided by original Keystone implementation\n\n");
```

B.5 How the CDI of each enclave, its Local Attestation key (with the certificate) are created

The code below has been used to derive for each enclave that is created its CDI and to compute the Local Attestation key, releasing also its X.509 certificate in DER format. (Lis. B.5)

Listing B.5. Computing CDI of the enclave and deriving its local attestation key with its cert

```
// The CDI of the sm is combined with the measure of the enclaves to obtain
    the CDI of the enclave
sha3_init(&hash_ctx_to_use, 64);
sha3_update(&hash_ctx_to_use, CDI, 64);
sha3_update(&hash_ctx_to_use, enclaves[eid].hash, 64);
sha3_final(enclaves[eid].CDI, &hash_ctx_to_use);

unsigned char seed_for_local_att_key[32];

for(int i = 0; i < 32; i ++)
  seed_for_local_att_key[i] = enclaves[eid].CDI[i];
// The CDI of the enclave is used to create the local attestation keys of
    the enclave
ed25519_create_keypair(enclaves[eid].local_att_pub,
    enclaves[eid].local_att_priv, seed_for_local_att_key);


// The process needed to generate the DER format of the X.509 certificate
    related to the local attestation key of the enclave is the same
    described when the keys associated to the security monitor have been
    created during the booting process
// The difference is that this cert has not set the extension related to
    the basic constraint because it is not associated to a key of a
    Certification authority
...
// Once the DER format has been created, it and its length are stored in
    the specific variables of the enclave structure
enclaves[eid].crt_local_att_der_length = effe_len_cert_der;
my_memcpy(enclaves[eid].crt_local_att_der, cert_real, effe_len_cert_der);

// The number of the keypair associated to the created enclave that are not
    the local attestation keys is set to 0
enclaves[eid].n_keypair = 0;
```

B.6 The functions exposed to the enclave

The code below has been used to expose to each enclave three different functions:

- the first one is the function that the enclave can call to ask the SM to create a keypair (Lis. B.6)

Listing B.6. Function exposed to create keypairs for the enclave

```
unsigned long create_keypair(enclave_id eid, unsigned char* pk, int
    seed_enc){
     unsigned char seed[PRIVATE_KEY_SIZE];
     unsigned char pk_app[PUBLIC_KEY_SIZE];
     unsigned char sk_app[PRIVATE_KEY_SIZE];
     unsigned char app[65];
     // The new keypair is obtained adding at the end of the CDI of
         the enclave an index, provided by the enclave itself
     my_memcpy(app, enclaves[eid].CDI, 64);
     app[64] = seed_enc + '0';

     sha3_ctx_t ctx_hash;

     // The hash function is used to provide the seed for the keys
         generation
     sha3_init(&ctx_hash, 64);
     sha3_update(&ctx_hash, app, 65);
     sha3_final(seed, &ctx_hash);
     ed25519_create_keypair(pk_app, sk_app, seed);

     // The new keypair is stored in the relatives arrays
    ...
    // The first keypair that is asked to be created is the Local
        Device Keys, that is inserted in the relative variables
    ...
     // The location in memoty of the private key of the keypair
         created is clean
     my_memset(sk_app, 0, 64);
     return 0;
    }
```

- the second one is the function that the enclave can call to obtain all the certificates chain (Lis. B.7)

Listing B.7. Function used to provide certificates chain

```
unsigned long get_cert_chain(enclave_id eid, unsigned char**
    certs, int* sizes){
  // Providing the X.509 cert in DER format of the ECA and
      its length
  my_memcpy(certs[0], cert_sm, length_cert);
  sizes[0] = length_cert;
```

```
  // Providing the X.509 cert in DER format of the Device
      Root Key and its length
  my_memcpy(certs[1], cert_root, length_cert_root);
  sizes[1] = length_cert_root;
  // Providing the X.509 cert in DER format of the
      manufacturer key and its length
  my_memcpy(certs[2], cert_man, length_cert_man);
  sizes[2] = length_cert_man;
  return 0;
}
```

- the third one is the function that the enclave can call to ask the SM to do some crypto operations (Lis. B.8)

Listing B.8. Function used to expose signature functionality to the enclave

```
unsigned long do_crypto_op(enclave_id eid, int flag, unsigned char*
    data, int data_len, unsigned char* out_data, int* len_out_data,
    unsigned char* pk){

  sha3_ctx_t ctx_hash;
  unsigned char fin_hash[64];
  unsigned char sign[64];
  int pos = -1;

  switch (flag){
    case 1:
        // Sign of TCI|pk_1Dev with the private key of the ECA .
         // The sign is placed in out_data. The attestation pk
             can be obtained calling the get_chain_cert method
        ...
          ed25519_sign(sign, fin_hash, 64, ECASM_pk, ECASM_priv);
          my_memcpy(out_data, sign, 64);
          *len_out_data = 64;
          return 0;
    break;

    case 3:
      // Sign of generic data with a specific private key.
      // In this case the enclave provides directly the hash of
          the data that have to be signed
      // The same of the cae 2, without have to compute the hash
          that is provided by the enclave
     ...
    break;
    default:
      return -1;
    break;
  }
  return 0;
}
```