MASTER OF SCIENCE
IN ENGINEERING

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering

Orientation: Information and Communication Technologies (ICT)

Leveraging TPM-based Remote Attestations to secure Edge-to-Cloud applications

Realised by

Ludovic Gindre

Under the supervision of
Prof. Nabil Abdennadher
Prof. Tewfiq El Maliki
At Haute école du paysage, d'ingénierie et d'architecture de Genève

External expert Cristóvão Cordeiro

Lausanne, HES-SO//Master, 2021

Accepted by HES-SO//Master (Lausanne, Switzerland) on a proposal from

Prof. Nabil Abdennadher, master thesis advisor

Prof. Tewfiq El Maliki, master thesis supervisor

Cristóvão Cordeiro, principal expert

Lausanne, April 22, 2021

Prof. Nabil Abdennadher Prof. Tewfiq El Maliki **Advisors**	Philippe Passeraub **Head of Department**

Ludovic Gindre
April 22, 2021

Acknowledgments

This project was a real challenge. I had to tap into my inner resources to bring it to completion. However, it would never have been possible to accomplish it without the precious help of many people. I would like to express my deepest gratitude to all of them.

I would especially like to thank Prof. Nabil Abdennadher and Prof. Tewfiq El Maliki for their invaluable advice and supervision throughout this project.

I would especially like to thank Prof. Nabil Abdennadher for his invaluable advice and supervision throughout this project.

I would like to express my very great appreciation to Cristovao Cordeiro and the entire SixSq team for their assistance and their advice.

My thanks also go to Marco E. Poleggi and my colleagues, who greatly assisted me.

Finally, I would like to deeply thank my girlfriend, parents, sister, and grandparents for their unconditional love and support, especially during this COVID period.

Abstract

As Internet of Things (IoT) is moving towards Edge-to-Cloud solutions, Edge device became easy targets for attackers as they are deployed in adversarial environments. Remote Attestation protocols became a high potential solution for Edge computing systems as security mechanisms that detect adversarial malware presence and verifies the state of Edge devices (EDs). Hardware-based Remote Attestations (RAs), which provide the best security guarantees, Leverage the presence of Trusted Platform Modules (TPMs) inside EDs. A TPM is a cryptographic co-processor which issues attestations about the state of an ED and can be trusted by a remote party that verifies attestations. We developed a RA framework that uses TPM version 1.2. We validated our framework on two use cases and were able to detect adversarial presence.

Contents

Acknowledgments **i**

Abstract **iii**

Introduction **1**

1 Remote attestations to secure Edge Devices **5**

1.1 Security in Edge Computing 5

1.2 Remote Attestations 8

1.2.1 Software-based Remote Attestation 10

1.2.2 Hardware-based Remote Attestation 11

1.3 Remote attestation use cases 12

2 TPM **15**

2.1 introduction 15

2.2 TPM applications 16

2.3 TPM version 1.2 16

2.3.1 Endorsement Key 17

2.3.2 Root of trust 19

2.3.3 Storage Root Key 19
2.3.4 Ownership 21
2.3.5 Platform Configuration Registers 21
2.3.6 Measured boot and Root of trust for measurement 24
2.3.7 Quote & Attestation Identity Key 26

3 Nuvla.io and NuvlaBox **29**

3.1 NuvlaBox architecture 30
3.2 Specification 32

4 Integrating RAs using TPM in NuvlaBox **35**

4.1 TPM and Linux 35
4.2 Architecture 36
4.3 Validation 38
4.3.1 Test protocol 38
4.3.2 The noise sensors and spatial accuracy use case 39
4.3.3 the Nuvla.io and NuvlaBox use case 42
4.4 Conclusion 46

Conclusion **47**

Bibliography **49**

Abbreviations **54**

List of Figures **57**

List of Tables 59

A PCRs usages 60

Introduction

Over the past few years, the IoT has significant and fast growth. According to a leading provider of market and consumer data, Statista[1], the market reached USD 151 billion in 2018 and forecast to reach USD 1567 billion in 2025. Several vertical and horizontal players create and improve products and applications at a steady pace.

IoT is an internet extension that consists of a new paradigm in which objects are connected to the internet. These objects are equipped with sensors and actuators that enable them to collect data and interact with their environment. They build a bridge between the virtual and the physical world. Numerous new applications exploiting the extensive amount of data collected by these objects have been developed. Several domains are targeted, such as smart agriculture, smart city, and smart grid.

Nowadays, connected objects cover cars, refrigerators, thermometers, weather stations, lights, blinds, smart meters, and many others. We can assume that there is a connected digital version of any existing object. The emergence of the IoT even led to the creation of new connected objects. Despite the dynamism in the IoT market, some obstacles stand in its way of progression. Among these obstacles, we can highlight three prominent challenges: costs (Total Cost Ownership (TCO)), centralized architectures of current IoT applications, and security.

The sensors are developed to have a low unit price. However, at scale, purchase costs increase drastically. Furthermore, developing, configuring, and maintaining the underlying infrastructure increases operational costs. Several solutions exist to reduce costs. Firstly, Micro-services applications must be preferred over siloed applications as it offers more flexibility and decouples development into several smaller projects that can be distributed among actors. Secondly, the development of multi-tenant Generic platforms that allow sharing resources among applications and users to share costs between actors. Several actors are already proposing solutions to reduce TCO.

The maturity of Cloud Computing (CC) contributed to the growth of IoT. It provided a centralized solution for storing collected data and CPU power to process data creating value for businesses. With CC, IoT applications are centralized in the cloud, and data has to travel all the way up to the cloud to be processed, creating bottlenecks and increasing latency. Furthermore, sending a large amount of raw data is expensive and inefficient. These issues led to the rise of Edge Computing (EC). In contrast to CC, EC is a model in which part of the computing power is shifted to the object itself. We will, therefore, enlarge the notion of connected objects to EDs that are connected objects with more computing power. In this way, data are processed directly on the ED, while reports are sent to the cloud. This paradigm is called Edge to Cloud (Edge-to-Cloud) architecture because it does not replace the cloud with the Edge but combines both advantages. Edge-to-Cloud architecture has four main advantages: (1) respect for privacy, (2) reduces the amount of data passing through the network, (3) offloading the cloud, and (4) eliminating the single point of failure that the cloud represented. These advantages explain why recent IoT applications are moving towards an Edge-to-Cloud architecture rather than a Cloud-only architecture.

This shift elicits new security questions as Edge-to-Cloud architectures increase the attack surface. Indeed, in an Edge-to-Cloud architecture, we now need to secure EDs. The virus "Mirai"[2] demonstrates the open security issues. In 2016, the virus spread across IoT objects such as IP cameras or printers and exploited their weak security policies. The virus used a table with the most common default usernames and passwords to take root privileges and install malware on the objects. These infected objects participated in several DDoS attacks against OVH, GitHub, Twitter, Reddit, Netflix, Airbnb, and many other companies [3]. Besides, ED can be deployed anywhere; they are much more accessible than conventional servers secured in locked rooms. They can therefore be more easily manipulated. It is frequent to hear news about successful attacks exploiting weaknesses of connected objects. As these attacks emphasize, it is imperative to build IoT applications with reliable and robust security solutions.

SixSq is a Geneva-based company that develops Nuvla.io, a multi-tenant generic Edge-to-Cloud platform. Nuvla.io allows its users to simplify the deployment, monitoring, management, and operation of their Edge-to-Cloud IoT applications. SixSq also develops NuvlaBox, a software stack that turns any ARM or x86 platform into smart ED for Nuvla.io. They aim to provide a solution to the TCO, centralized architecture, and security obstacles that could face IoT as they promote Edge-to-Cloud architectures in which resources and costs could be shared. With their NuvlaBox, they aim to improve the security aspect. However,

the authentication of the software running on the Nuvlabox-based edge devices is still an open problem.

RA are promising techniques that could improve ED security. They allow remotely detecting malicious state changes in a device, hence, providing strong evidence of ED trustworthiness. Some of these techniques make use of TPM, a hardware component, to improve their quality. This work aims to enhance NuvlaBox with a TPM-based RA framework to secure Nuvla.io and NuvlaBox Edge-to-Cloud applications. It has several objectives: Design and develope the RA framework; Allow the framework to leverage the TPM; Interface a TPM through docker containers.

This document is structured as follows. The first chapter describes security applied to EDs with a focus on Remote Attestation (RA). The second chapter explains the underlying principles of TPMs and how they are used in RA. The third chapter presents Nuvla.io and NuvlaBox, describes the platform and the NuvlaBox architecture, and specifies our TPM-based RA solution. Finally, The fourth chapter presents the architecture of our solution, its implementation, its validation with two use cases, and discusses the validation results.

Chapter 1

Remote attestations to secure Edge Devices

1.1 Security in Edge Computing

In recent years, the rapid development of IoT led to the emergence and the rapid growth of Edge Computing. This rapid development has made possible security threats that were no longer possible in cloud computing. Furthermore, EDs are easy targets for attackers as their attack surface is more extensive than regular devices' attack surface due to their locations. Indeed they are likely to be exposed to various physical attacks. This attack surface is larger because of four angles [4]:

Computation Power EDs have less computation power than regular servers or desktop computers. Attacks that were ineffective against those computers because they could protect themself are effective on ED.

Attack Unawareness EDs are often remotely monitored. Thus, their global running state is not always known by its administrators. It gives the attacker the upper hand. An attack may be detected after a long time. In worst-case scenarios, administrators could never detect it.

Protocol heterogeneity OSs use different communication protocols. The high diversity of communication protocol leads to difficulties bringing new and unified security mechanisms.

Access Control Most of the current Access Control Systems (ACSs) are coarse-grained, and their design does not allow fine-grained access control of a resource. These systems can not represent Edge Computing applications because they have different types of interactions. For instance, a specific ACS for Edge-computing should be able to grant read-only permissions to a given ED, for a given group of users, during a period, or periodically. Bad access control design leads to security breaches as it could allow unauthorized users to access resources.

As Their attack surface is large, they are often subject to attacks. Each one is different, but we distinguish six classes of attacks [4]:

Denial-of-Service (DoS) Attacks This type of attack aims to cause a service or resource to become unavailable. As the service or the resource is unreachable, regular users can no longer use it. Distributed Denial-of-Service (DDoS) is a distributed version of DoS attacks. Attackers use botnets, a network of zombie computers, to flood a target from several sources. These attacks are prevalent because they do not require high knowledge and are difficult to mitigate. Indeed it is tough to distinguish legitimate traffic. Attackers can then ask for a ransom to stop their attack. Mitigation of such an attack is not a trivial problem and is still a research topic.

Authentication and Authorization Attacks ACSs consist of two main steps: first, they authenticate, then they authorize. Authentication is the action of verifying someone's identity. When a user requests a non-public service, they must prove their identity by proving they know a shared secret such as a password or by proving they own something unique, such as a fingerprint, a private key, or a credit card. Some systems have two-factors authentication. It means that their users must first prove they know the shared secret and then prove they own a unique thing. These systems are often used in online bank accounts authentication since they are more secure than those using only one authentication factor. After authentication, authorization comes. A user that is authenticated can gain rights and privileges. The authorization step grants these rights and privileges. This type of attack aims to gain privileges and rights and pass the ACS maliciously. This attack is made either by bypassing the ACS to guess authorized users' credentials or bypassing it. The former

Malware Injection Attacks This type of attack aims to take advantage of exploits to insert malicious code into a system. The attacker will try to steal data or install malware by injecting the code in a breach. These attacks can come in many forms

and are therefore difficult to prevent. Furthermore, the heterogeneity of devices on both hardware and software makes it even harder to reduce the number of breaches. Many different malware injections have been detected on several devices such as printers, IP cameras, network routers, or even Android systems in recent years [5]–[9]. When such an attack succeeds, the attacker's goal is to install persistent malicious software such as a rootkit or modify the firmware.

Side-channel attacks These attacks are based on the information a system inherently leaks. In this scheme, the attacker does not try to exploit a weakness in an algorithm or a protocol. Instead, it uses information such as power consumption, websites connections, and many others that are not sensitive by design but can be exploited to infer sensitive information. These attacks often require physical access to the target and high expertise and skill in the type of information analyzed. For instance, it requires high electronics knowledge to infer a private key from a device's power consumption. Restricting access to side-channel is a solution to protect against those attacks, but they are often hard to identify and impossible to hide. The example of power consumption is the perfect example of an impossible to hide side channel.

Bad-Data Injection Attacks These attacks aim to corrupt a system by corrupting data collected or received by an entity. These attacks often aim to smart power-grids in which actors monitor the power consumption to predict future consumption. In this scenario, the attacker will try to change the estimated transmitted power to act on the future prices resulting in a financial profit.

Man-in-the-Middle Attacks This type of attack tries to intercept communications between two entities without both entities knowing they are attacked. Given three entities, Alice, Bob, and Charlie, Charlie, the attacker tries to hide and make the two entities believe they are the other entity. When this attack succeeds, the attacker can read all messages sent by both entities. When Alice sends a message to Bob, she sends a message to Charlie because she thinks Charlie is Bob. Charlie can then rewrite the message and send it to Bob. Bob will think Alice sent the message and will respond to Charlie, thinking he is Alice. Charlie must often intercept the first messages between Alice and Bob for this kind of attack to work. If he succeeds in doing so, he will pass wholly undetected and steal the messages' information. Even cryptographic techniques do not protect against this kind of attack. Consider that Alice wants to talk to Bob, so she asks him for his public key and provides him with hers. Charlie intercepts this message and sends her public key. Alice thus begins to communicate in an encrypted way with Charlie by thinking that he is

Bob. Meanwhile, Charlie pretends to be Alice and asks for Bob's public key. Bob sends his public key to Charlie, thinking he is Alice. Thus, Alice has Charlie's public key, thinking it is Bob's public key. Bob also has Charlie's public key, thinking it is Alice's public key. Finally, Charlie has the public keys of Alice and Bob. Bob will intercept each message during future message exchanges and pass it on to the other party without being detected. When Alice wants Bob's information, she will request it from Charlie, thinking he is Bob, who will pass it on to Bob. Bob will reply to Charlie, thinking he is Alice, who will then pass it on to Alice. To protect against this type of attack, Trent, a trusted third party, is introduced. This third party issues certificates that state that Alice and Bob's public keys belong to them. So instead of just exchanging their public keys, they exchange their certificates. Trent's private key signs these certificates. Therefore, Charlie cannot change the certificates signed by Trent without Trent's private key. However, Alice can verify Bob's certificate with Trent's public key and vice versa for Bob.

Replay attacks are particular Man in the Middle (MITM) attacks in which an attacker intercepts legitimate communications containing sensible information such as passwords or credit card numbers to reuse this communication later to impersonate its victim.

As shown in the figure 1.1, it is the most frequent attack against ED with DDoS attacks. Among these attack vectors, the biggest threat in terms of damage to ED is malware injection. Indeed, the attacker can then steal sensible data, attack other targets with DDoS, or even encrypt all data and ask for a ransom in exchange for the encryption key. A promising counter-measure against Malware Injection attacks is RA.

1.2 Remote Attestations

In recent years RA emerged as a promising security solution to secure ED. In RA, a trusted entity, the verifier, tries to establish another untrusted entity's trustworthiness, the prover. It allows detecting prover's software compromised states remotely. It is, therefore, not a defense mechanism as such but a detection mechanism. As attackers have many advantages over defenders in the case of ED, the aim is to detect an attack as quickly as possible to take the necessary action.

The basic scheme of RA is simple, as shown in the figure 1.2: the verifier sends an attestation request to the prover (step 1). Then the prover computes the attestation (step

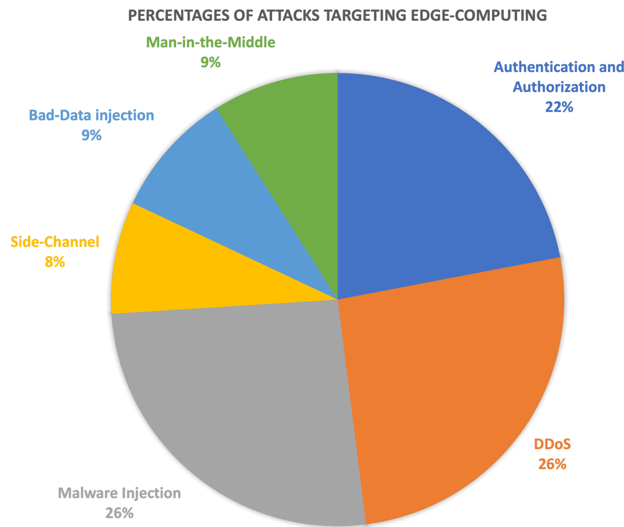

Figure 1.1: Percentage of attacks targeting Edge-computing [4]

2) and sends the attestation back to the verifier (step 3). Finally, the verifier verifies the attestation and establishes the trustworthiness of the prover (step 4). This scheme is simple because it hides the complexity that lies in the prover and verifier's verification. Indeed how does the prover computes its attestation so he can not lie about its current state? As the prover is untrusted, the malware could force the prover to lie if an attacker succeeded in a malware injection attack.

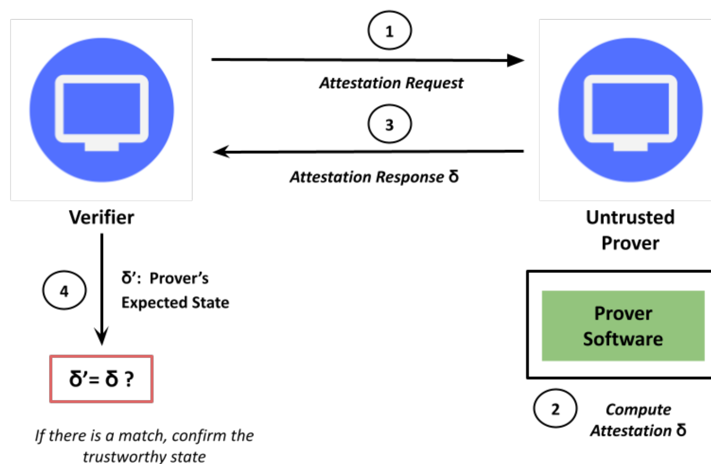

Figure 1.2: General Remote Attestation scheme.

Generally, RAs schemes are divided into two distinct phases. During the first phase, the measuring and initializing phase, the trustworthy state is described to the verifier. This description can have several forms, such as a description of all software running on the prover or a list of possible measures. Since each RA protocol behaves differently, the measuring phase depends on what the protocol measures. The second phase is the attestation phase described above.

The literature distinguishes two families of RA: Software-based RA and Hardware-based RA. A third family of hybrid RA exists, but we will focus on the two prominent families.

1.2.1 Software-based Remote Attestation

SoftWare-based ATTestation (SWATT)[10], the first RA scheme proposed to add a tiny software in the prover that will randomly analyze its memory and checksum the analyzed memory region. When the verifier wants the prover to attest, it sends a random number that acts as a seed for the random memory analysis. Thus even though the prover's memory is randomly checked, the verifier knows exactly which memory location the prover's software analyzes. This random number protects against replay attacks as each attestation is different. Hence the attacker can not respond with an old valid attestation to a new request. To not be maliciously modified, the prover software is written and optimized so that a single instruction added by the attacker significantly increases the execution time. In this way, the verifier can detect that the software responds slower than expected and thus detect a malicious modification of the software. When the prover finally computed the attestation and sent it to the verifier, the verifier must verify it. Consequently, the verifier must know the system's actual state, i.e., all the possible states of each software running on the platform. This task, although theoretically possible, is very complicated. Originally this type of remote attestation was intended for embedded systems that only ran simple firmware on simple mono-core CPU architectures without virtualized memory or branch prediction. Therefore, it would be impossible to apply this scheme to EDs that run a rather complicated Operating System (OS) on multi-core architectures and whose states are therefore non-deterministic. Furthermore, since it directly analyzes the device's memory, the running time of the prover software overgrows with devices with more memory. Hence this solution is not good at scale.

This scheme suffers from the difficulty use time as a source of trustworthiness [11]. This scheme suffers from overclocking issues. Indeed, if the attacker overclocks the CPU, the running time could seem valid for the verifier. Moreover, it has been proven ineffective at

detecting rootkits that attack the prover software itself to produce legitimate attestations of an attacked device [11]. Every software-based RA scheme suffers from these issues, which make them inherently insecure because they are software-based [11]. Hardware-based RA has been introduced to solve this security issue.

1.2.2 Hardware-based Remote Attestation

In hardware-based RA, an assumption is made that provers feature a trusted component. In its most basic definition, a trusted component is a hardware-protected memory. The fig 1.3 shows the Hardware-based RA general attestation scheme in which the prover features à trusted component. In this scheme, the Trusted component is responsible for computing the attestation.

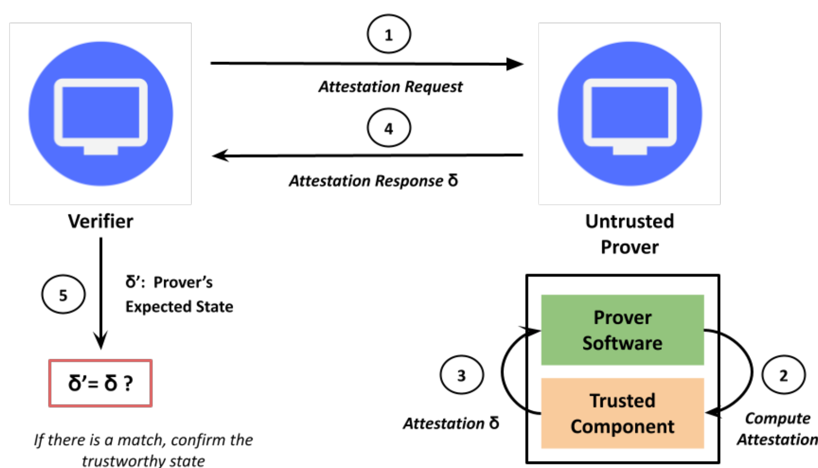

Figure 1.3: General Hardware-based Remote Attestation scheme.

Three major, trusted components exist Intel SafeGuard Extension[12], ARM TrustZone[13], and TPM[14]–[16]. Even though they aim to perform the same task, they are entirely different technologies. Intel SafeGuard and ARM TrustZone are embedded in their respective CPUs while TPMs are independent of the CPU. The main difference is that ARM TrustZone and Intel SGX are trusted execution environments, while TPMs are passive components. The former can securely run code, while the latter is a chip that offers secure and trusted functionalities. Hardware-based RA protocols depend on which technology is chosen.

Many state-of-the-art RA protocols require trusted execution environments as they offer flexibility to implement different protocols, while TPMs are not as flexible and force their RA scheme.

TPM-based RA must comply with trusted functionalities TPM. A TPM allows performing integrity measurements that can be used in attestations. As a result, trusted execution environments seem to have an advantage over TPMs. However, as trusted execution environments share the same CPU cores and cache, many attacks such as side-channel attacks have been proven effective on these [17]–[21]. Hence as long as these trusted execution environment present vulnerabilities, RAs implemented on top of them are pointless. As long as trusted execution environments share the same hardware as the regular environment, they will be subject to this type of attack [22]. Therefore we will focus on TPM-based RA. The TPM will be described in detail in the next chapter.

1.3 Remote attestation use cases

RA's main purpose is to create a trust bond between a prover and an entity that needs to communicate with the prover. The trust bond is the result of two predominant elements: The trust that lies in the prover side's attestation protocol, *e.g.*, the TPM in the prover and the trust that lies in the ability of the verifier to verify the authenticity of the attestation. This trust bond can have several use cases that are not yet fully exploited. Because RA is still a new technology, there are not many real use cases. Furthermore there are many different RA protocols that are tightly linked with their domain. However, potential use cases are numerous.

The first proposition is the attestation of a laptop against its user's smartphone. The goal was to provide the user with a notification about the laptop's trustworthiness. Based on that notification, the user knows whether its laptop has been tampered with or not. That information could be useful in a situation in which the user is a diplomat or someone with critical secret information. IoT also brings new needs in terms of trust. Indeed, since devices are deployed in adversarial environments, they need to be monitored. RA could be a potential monitoring tool that could work as a heartbeat and also provide that trust bond between devices and applications deployed on top of them. As a more concrete use case, in 2012, RA where implemented on Google Chromebook to attest the platform booted a trustworthy Chrome OS version [23]. Recently, companies have started to implement RA at a company scale. It allows to verify their user to connects to the

company internal services on verified platforms and replace two-factor authentication. Google is currently working towards this goal and is one of the pioneer in the field of RA[24].

Chapter 2

TPM

2.1 introduction

Since the early days of IT, private key management has always been a big concern. If a private key is leaked, the attacker can usurp the identity of the attacked entity. Since private keys are often stored on hard drives, it is possible for an attacker who gained root access to a machine to steal the private key. In 2003 Intel, AMD, Microsoft, and Cisco, among others, created a consortium named the Trusted Computing Group (TCG). Its role is to "develop, define and promote open, vendor-neutral, global industry standards, supportive of a hardware-based root of trust, for interoperable trusted computing platforms."[25] In response to the private key management issue, the TCG worked on creating a dedicated safe zone in which private keys could be stored and used without being exposed.

In 2009, the TCG released the first version of the specification of this dedicated safe zone named TPM. In 2011, the TCG released the specification version 1.2, a major revision that solved version 1.0 vulnerabilities. The 1.2 specification states that a TPM must handle the Rivest-Shamir-Adleman (RSA) cryptosystem and Secure Hash Algorithm 1 (SHA-1) hash function. In 2016 they released the 2.0 version, which must handle RSA and Eliptic-Curve Cryptography (ECC) cryptosystems and Secure Hash Algorithm 256 (SHA-256) hash function. In this document, we will focus on version 1.2 since it is the version our EDs feature.

2.2 TPM applications

In addition to being used by RA protocols, TPM is used for other applications. One of the prominent use is disk encryption. By encrypting the disk with a key stored in the TPM, the disk can not be read on another platform. Disk encryption tools such as BitLocker [26] for Windows and dm-crypt [27] for the Linux kernel are able to encrypt and decrypt a disk. They are able to leverage the presence of a TPM to store the encryption key in it.

Another application of TPM is the system integrity validation. The TPM can store measurements about the system's integrity. Bitlocker is able to use the TPM to verify the integrity of the system before decryption of the system [26]. Intel Trusted Execution Technology is another example of system integrity validation system. The intel technology uses TPM in the platform to validate the boot chain [28] and ensure the booted system is trusted. The TPM has also been virtualized by most of the major hypervisors such as VMWare, Xen and KVM so that physical TPM can be used within virtual machines[29]–[31]. This allows to extend those TPM applications to virtualized environments.

2.3 TPM version 1.2

As TCG only provides a specification. Anything that implements the specification is considered a TPM. It can therefore take several forms such as a hypervisor (software, virtualized), emulator (software), or hardware (dedicated chip), among others. In this thesis, we will refer to the hardware solution when using the term TPM.

This TPM is not only a secured memory zone as one needs to use private keys without retrieving them. It is, in fact, a passive co-processor capable of responding to commands by carrying out cryptographic operations. It can generate RSA key pairs and random numbers, store encrypted values, or store privates keys and store measurements among all its functionalities. Although it is specialized in cryptographic operations, it is not a cryptographic accelerator. Operations carried out by a TPM are very slow. Along with the cryptographic processor, a TPM is made up of several basic blocks we will explain, as shown in the figure 2.1.

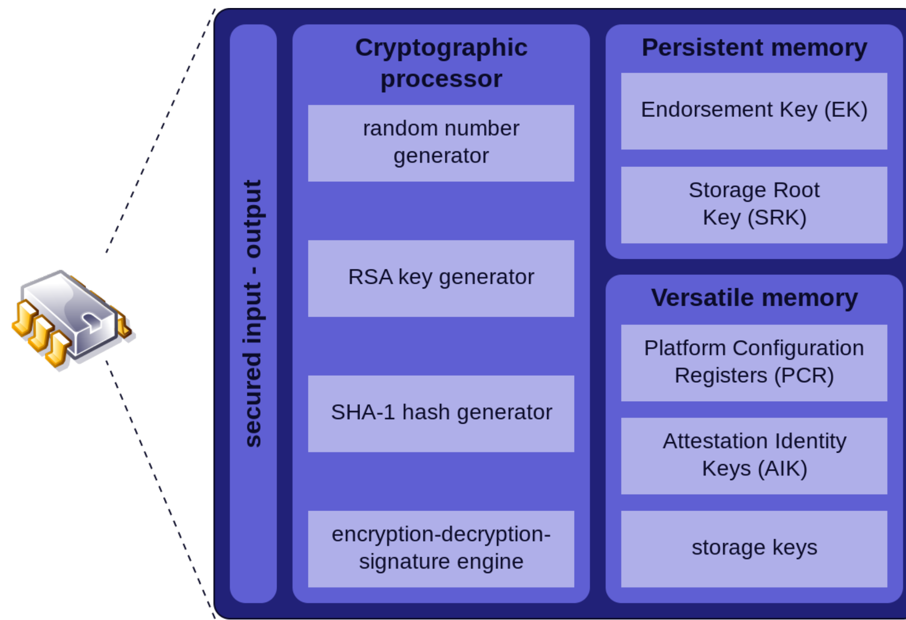

Figure 2.1: Components of a Trusted Platform Module complying with the TPM version 1.2 standard

2.3.1 Endorsement Key

The main component is the Endorsement Key (EK). It is a special purpose RSA key pair created by the manufacturer before shipping the TPM. The term "created by the manufacturer" means that the manufacturer executes the command that generates an EK on the TPM. The TPM responded by using its cryptographic processor to generate a RSA key pair. Then it stored the private part of the key in a persistent memory that is tamper-resistant and only readable from inside the TPM. Finally, it returned the public key to the manufacturer. It implies the manufacturer does not know the private key. It also implies the manufacturer did not hardcode the private key in the TPM's persistent memory. The tpm is designed so that one can only read in or write to its memory through specific commands. The advantage of commands is that it allows validating before writing to the memory. The only command that exists is the one to generate an EK. This security measure forces the manufacturer and anyone else to run the command to generate a new one. However, the system the TPM could check if an EK already exists and return an error code. As the manufacturer should always run this command first, an attacker could not overwrite the EK.

Along with the EK, an EK certificate is written into the TPM memory by the manufacturer. The manufacturer signed the EK's public part with their private key and issued a certificate that attests the TPM returned the EK's public part. This certificate is essential

for the TPM, as it proves that the manufacturers initialized the TPM and that it was not altered during the shipping. With this certificate, the manufacturer acts as a third party that certifies the identity of the TPM. Since the manufacturer's private key signed the certificate, if an adversary tried to replace the EK to setup a MITM attack, it would require the manufacturer's private key to re-sign the certificate with the new EK. Without the manufacturer's private key, the adversary cannot forge a new valid certificate without causing the certificate verification to fail. In other words, a certificate binds the EK's public key to the TPM's identity.

The manufacturer also has a certificate stating that they own the key pair that signed the EK's certificate. This latter certificate has been issued by a Certification Authority (CA), which also has a certificate and so on, until a root certificate. This is a chain of trust, as shown in the figure 2.2. It starts with the root CA, an entity such as DigiCert or IdenTrust, which is well known and trustworthy. The root CA signed itself its certificate since it is the root. Then it ensured that the intermediate CA (the manufacturer) owns the private key they claim to own. After ensuring they own the private key, the root CA can issue a new certificate for the intermediate CA. The trust granted to the root CA is therefore extended to the intermediate CA. The intermediate CA can now issue new certificates to extend its trust to other entities such as TPMs. For example, STMicroelectronics, a major TPM manufacturer, have a certificate signed by GlobalSign, a trustworthy root CA; therefore, they can issue certificates for TPMs they manufacture. Thus they are an intermediate CA.

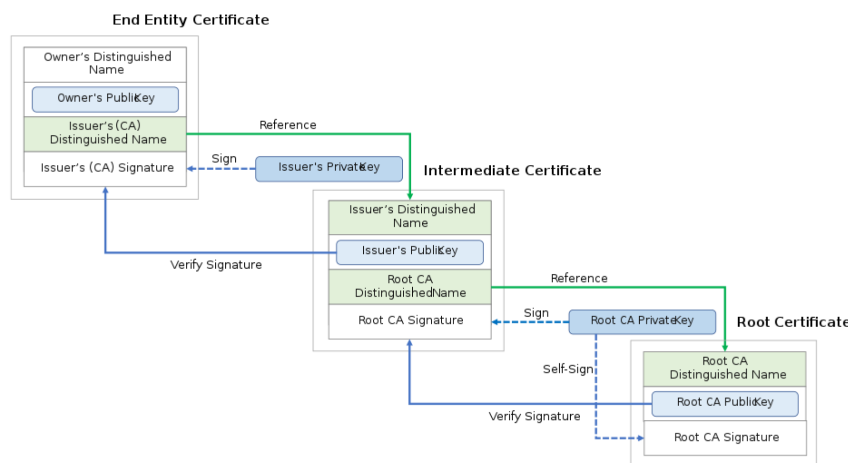

Figure 2.2: Example of a chain of trust with a root certificate, an intermediate certificate, and an end-entity certificate

The EK is a special purpose RSA key. Indeed this key has a restricted scope. As this key is deeply linked to its TPM by the certificate, this can raise privacy concerns. Indeed, this

key being unique could be used to identify the TPM and, by extension, the platform to which it belongs. Therefore, it is not possible to sign with this key. Its only scope is to encrypt and decrypt messages.

2.3.2 Root of trust

As the EK has been signed by the manufacture whose public key/certificate has been signed by another CA and so on until a root CA, we now know the EK is stored inside the TPM. Therefore, the trust placed in the root CA and in the manufacturer was extended up to the TPM. As a result, it has become a root of trust from the point of view of the entity leveraging it. It is responsible for providing the ability to verify the trustworthiness of the whole system. In other words, it provides tools to extend the trust placed in the TPM to the system. This root of trust can be described as three basic blocks of trust: The Root-of-trust-for-storage (RTS), the Root of Trust for Measurement (RTM), and the Root of Trust for Reporting (RTR). The first is about the storage functions provided by the TPM. It answers the following question: "Can we trust that secrets stored by the TPM are really secret ?". The second is about measuring the state of the system to decide whether it is in a legitimate and trustworthy state. In order to decide it, the whole system is measured. It answers the following question: "Can we trust that the measurements stored in the TPM have not been modified?" Finally, the last, which works in tandem with the second one, is also about the measurement. When one wants to verify the measurements to decide whether the system is in a legitimate state, one must ask oneself the following question: "Are reported measures trustworthy?". The RTR answers that question. Together they constitute the root of trust and allow the TPM to extend this trust to the whole system.

2.3.3 Storage Root Key

The second component is the Storage Root Key (SRK) which represents the RTS. It is stored in the same tamper-resistant memory as the EK. As its name suggests, it is used for storage purposes. The TPM's primary purpose is to store private keys, but its tamper-resistant storage has limited capacities. The SRK is used to encrypt other private keys.

The operation of encrypting a key pair with another key pair is called wrapping. Given two key pairs A and B, we say that A wraps B if B's private key has been encrypted by

A's public key, creating B'.

$$B' = wrap(A_{pub}, B_{pri}) \iff wrap(A_{pub}, B_{pri}) = encrypt(A_{pub}, B_{pri}) \tag{2.1}$$

The opposite operation called unwrapping is the action of decrypting a wrapped key with a private key. Let B' being the result of a wrapped key pair B by the key pair A; the unwrap operation is defined as follows:

$$B_{pri} = unwrap(A_{pri}, B') \iff unwrap(A_{pri}, B') = decrypt(A_{pri}, B') \tag{2.2}$$

By extension, wrap is the inverse operation of unwrap. We can describe this relationship as follows:

$$B_{pri} = unwrap(A_{pri}, wrap(A_{pub}, B_{pri})) \tag{2.3}$$

These operations allow creating key pairs hierarchies using the SRK as the root of the hierarchy. In the TPM, the hierarchy starts with the SRK, which can wrap other keys. As shown in the figure 2.3, all key pairs stored by the TPM are wrapped by the SRK. Then, other keys can also wrap daughter keys and so on. An application that requires a key can then scroll down the hierarchy by unwrapping each parent key starting from the SRK.

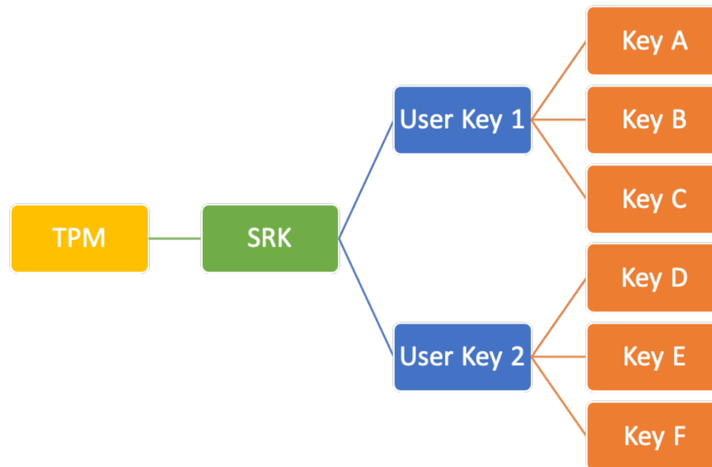

Figure 2.3: Example of key pairs hierarchy with SRK as the root key

The TPM is resource-constrained to be usable by low-powered devices. As a result, it cannot afford to store many private keys in the tamper-proof memory. As the SRK wraps all other private keys, they can be stored outside this memory zone while being encrypted. The SRK is therefore considered as the TPM's RTS.

2.3.4 Ownership

When shipped, the TPM is in a specific state that offers a limited set of commands. In order to use the full set of commands, one must become the TPM owner. The owner has full control over it. He can enable or disable it, create keys, and set policies. As per the specification, the TPM must ship with no owner installed. Therefore, setting up the ownership is a critical first step that requires tight controls and particular attention.

When in an unowned state, any process can set up the ownership. It can be taken by executing the command TPM_TakeOwnership and by providing a new owner passphrase. This command creates a new SRK and a proof value. Therefore, when the owner is not set, the TPM does not contain an SRK created when the ownership is taken.

When one wants to use the SRK through a command, one has to provide the secret owner's passphrase to prove he has ownership along with the command. The TPM can thus verify the provided passphrase against the proof value. If the passphrase does not match, the command will fail with an error TPM_AUTHFAIL. Any entity knowing the secret is then considered the TPM's owner.

When in an owned state, the owner can choose to clear the ownership. It means that the TPM will return to its original unowned state. The command TPM_OwnerClear clears the ownership, which means it also clears the SRK. As a result, the keys wrapped by the SRK will no longer be exploitable as they can no longer be unwrapped. After the clear ownership command succeeds, the TPM is disabled. When disabled, it can only be (re-)enabled in the BIOS/UEFI settings. This security mechanism ensures that no one takes ownership without rebooting.

2.3.5 Platform Configuration Registers

Platform Configuration Registers (PCRs) are special memory registers. They are shielded locations inside the TPM and are part of its volatile memory, which is reset to a predefined value at each boot. PCRs are designed to store integrity measurements so that they can not be tampered with. The specification requires a minimum of 16 independent registers, but in practice, there are often 24. Each of them has a dedicated purpose, such as PCR[0] which is dedicated to storing BIOS measures; PCR[1] which is dedicated to storing BIOS configuration measures; or PCR[16] which is dedicated to debugging. The table A.1 shows in detail each PCR's purpose.

Performing an integrity measurement is the action of storing a standardized measure identifying a component and its integrity. Given a system that should run a program (the component), if one wants to verify the integrity of the system, they must also verify the integrity of this program, *i.e.*, check that the program is the one it claims to be and not another program that an attacker maliciously replaced. To verify this statement, one needs to store a measure that identifies the program, such as its binary content, before loading it into memory to proceed to the verification. Indeed, a program that tries to usurp another will not have the same binary content even though having the same name: check-summing the binary helps to detect a usurper program.

A PCR is made to store those integrity measurements. It is a 160 bits register that is designed to store SHA-1 digests[1]. In cryptography, a cryptographic hash function such as SHA-1 must satisfy requirements, *i.e.*, it must have some properties. Some of these properties are the reason PCRs have been designed as such. The first property is that it must be deterministic, *i.e.*, one input hashed always yields the same output. This property is vital for measures to be consistent.

The second property is the impossibility of retrieving the input from the digest, *i.e.*, there exist no reverse hash functions. It is also known as the one-way-ness property. This second property is handy from a security point of view. By storing a hash instead of a value, an eavesdropper can not retrieve the value and thus replay the measurements. In practice, this property is never fulfilled as it is always possible to brute-force the digest to find the corresponding input. However, hash functions are designed to require much effort to brute-force one single digest. In fact, due to successful attacks against SHA-1, this hash function has been replaced by SHA-256 in version 2.0 of the TPM specification.

The third property is the impossibility of finding two (or more) messages that yield the same digest (collision). This property allows the creation of a signature of the measure. By combining this property with the previous one, this signature becomes a footprint that cannot be traced back to the original measure. Again, due to attacks against SHA-1, the first collision was made public in 2017[32]. They were able to create two different PDFs with the same digest. Even though SHA-1 shown vulnerabilities, It required $2^{63.1}$ SHA-1 comparison, representing 6500 years in terms of CPU time and 100 years in terms of GPU time for only one digest.

Finally, the fourth property is that a small change in the input changes the resulting digest so that the two digests appear uncorrelated. A single-bit switch should com-

[1] A digest is the result of a hashing function such as SHA-1.

pletely change its digest. Given a small message $m_0 = F0_{16} = 11110000_2 = 240$ and another one $m_1 = F1_{16} = 11110001_2 = 241$ which is only one bit different from m_0, their digests should be completely different. SHA-1 has this property. Therefore, $\textit{SHA-1}(m_0) = efe43def97eb295fe99c3753f2d740d7b36df689_{16}$ while $\textit{SHA-1}(m_1) = 07b7255eacbc81c051445ebe4f8c74fc8892dd3e_{16}$

PCRs can not be written directly; instead, they can be extended. Extend is the only possible operation to write in a PCR. This operation is similar to writing, but it contains specificity. The value that will extend the register is not directly written into it. Before storing the measure, the TPM will concatenate the previous register value with the measure and then hash it. Finally, the TPM will store the resulting digest in the register. The extend function of a register with index n for a value v is defined as follows:

$$PCR[n] = \textit{SHA-1}(PCR[n] \mid\mid v) \tag{2.4}$$

Where || is the concatenation operator.

Successive calls to extend create a chain of measurements. A chained list is created, starting from the initial PCR value. As the initial value is well defined, the first call to extend will create the first node of the list and so on.

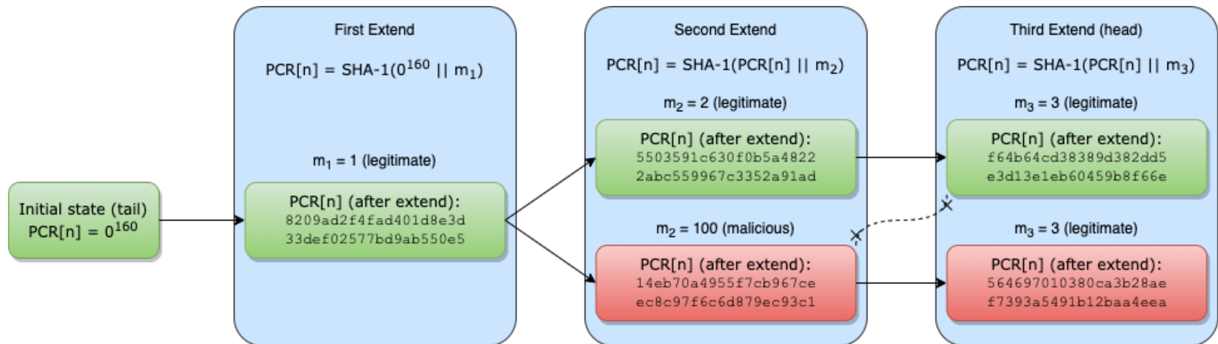

Figure 2.4: Example of a hash tree with two branches, one legitimate and one malicious

This list can be represented as a path in a tree of the possible values as shown in the figure 2.4, where two branches are created during the second extend: one is a legitimate measure while the other is a malicious one. Of course, this tree has an infinite number of possible branches, but only a finite subset of them are legitimate. Thus a verifier only needs to know the legitimate subset of the tree to detect illegitimate states.

Cryptographic hash function properties also apply to this tree. The first property guar-

antees the tree will not change each time the PCRs are reset. Because of the one-way-ness property, this list can not be read from the head to the tail. The only operations possible are to read the head or extend and add a new node. Finally, it is impossible (very hard in practice) for a malicious branch to rejoin a legitimate one because of the third property, as shown in the figure 2.4 between the second and the third extend.

Because of the concatenation with the previous result, this tree also guarantees the order of measures. Indeed swapping two measures will yield another result which can be expressed as a new branch. This new branch could either be legitimate if the measure order does not matter or malicious if the order is important. For example, swapping legitimate m_2 with legitimate m_3 would yield the following result:

$$\begin{aligned} PCR[n] &= \textit{SHA-1}(8209ad2f4fad401d8e3d33def02577bd9ab550e5 \,||\, m_3) && 2^{\text{nd}} \text{ extend } (\mathrm{m}_3) \\ &= 397f4aad13b76df48bfc5ddbcdfd561907d1ef7f && \text{Stored in PCR[n]} \\ PCR[n] &= \textit{SHA-1}(397f4aad13b76df48bfc5ddbcdfd561907d1ef7f \,||\, m_2) && 3^{\text{rd}} \text{ extend } (\mathrm{m}_2) \\ &= b4cc898d09ea680dc72d4ef487033948fb544e56 && \text{Stored in PCR[n]} \end{aligned} \tag{2.5}$$

These properties make PCRs the RTR. Indeed, They ensure that an attacker cannot modify the reported measures while stored by the PCRs. However, they do not guarantee that measurements are trustworthy as a malicious component could lie about the measurement. Thus, they do not constitute the RTM.

2.3.6 Measured boot and Root of trust for measurement

For integrity measurements to provide trustworthy results, they must be performed by a trusted component. Indeed, if an untrusted component is a malicious component, it could make a fake measurement. This is where trusted boot takes place: it allows to extend the trust from the TPM to the OS.

The measured boot aims to establish trust in a booted environment. During the boot process, several components are run step by step. However, at first glance, each component is untrusted as each of these components could have been replaced with a malicious one. A malicious component performing the same task as the original would go undetected by the user, but the computer could be compromised.

In a measured boot process, each component measures the next component before launching it. The BIOS measures the Master Boot Record (MBR) containing the bootstrap code and the partition table before loading the code in it. Then the bootstrap code measures the next component before loading and running it. Continuing this way until the bootloader that measures the operating system before loading and running it as shows figure 2.5. A downside of this process is that each component must be TPM aware to extend the PCRs. The firmware of a platform is supposed to be TPM aware as the manufacturer that created the platform chose to include one in it. However, some bootloaders such as GRUB are not by default TPM aware. Thus deploying a measured boot requires a relatively substantial workload on an existing installation. Indeed, one needs to replace the existing bootloader with a TPM aware one, such as the TPM aware GRUB version.

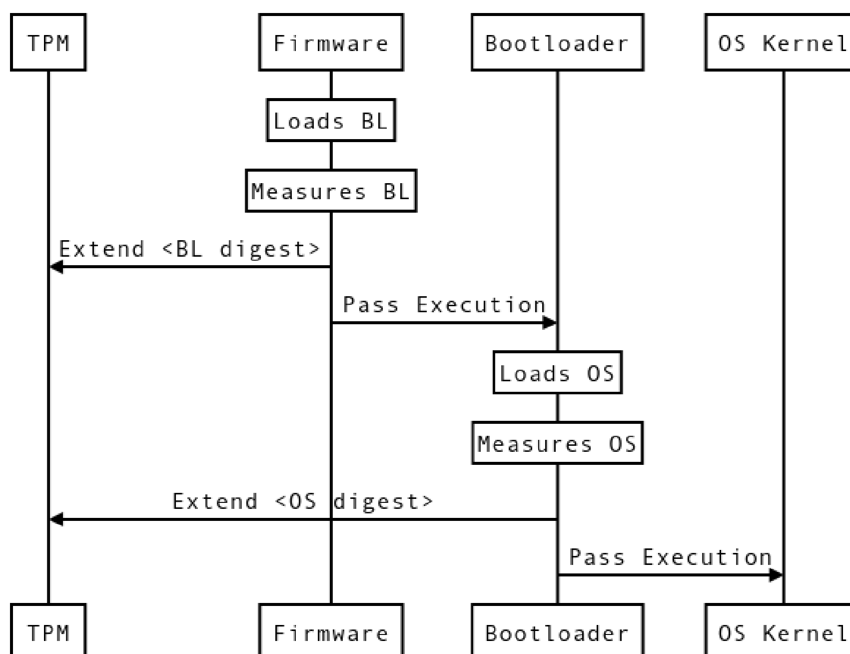

Figure 2.5: Each measured boot process step is detailed as a sequence diagram.

However, this process, as explained above, does not solve the trust problem. Indeed, to trust the first measure, we need to trust the first component that makes the measure, but to trust it, we need to measure it. There is a chicken-and-egg problem that the TPM can not solve since it is passive, *i.e.*, it only responds to commands but does not act independently.

The Core Root of Trust Measurement (CRTM) solves this problem. It represents the very first step in a measured boot process. In a normal boot process, the bios is the first component that is loaded on the CPU. However, in a measured boot process, the CRTM

is responsible for performing the first measure of the BIOS then loading it. This software is implicitly trusted as it is static, *i.e.*, it is engraved in the memory and can not be modified. As this code is run on the CPU, it is actually the RTM. The measured boot process creates a chain of trust. As shown in the figure 2.6, this chain of trust is created by the CRTM and goes through all boot steps. When the chain of trust reaches the OS, it can be extended to applications; however, it could cause the subset of legitimate branches to grow fast.

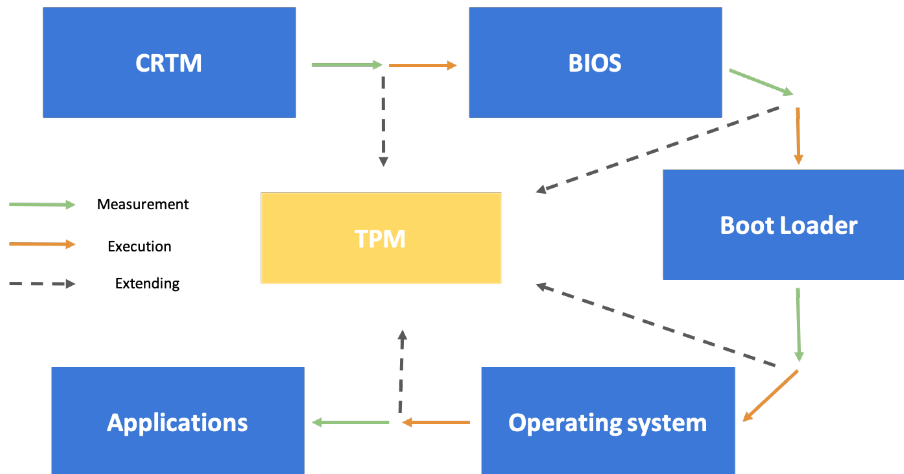

Figure 2.6: Chain of trust starting from CRTM to the OS and extending to applications.

2.3.7 Quote & Attestation Identity Key

Although the TPM stores the measurements in its PCR in an immutable state, it is only a passive co-processor. It means that it only performs actions the CPU asks for. If a malicious program asks to read PCRs values or somehow intercepts those values, it can modify the values before sending them to a verifier. Therefore, for a verifier to check the system's integrity, direct reading of the PCR is not the right solution as it can be falsified.

The TPM must provide something that cannot be falsified to remedy this problem. In practice, it is impossible to guarantee that data read is immutable. Indeed, since the data is copied from the PCRs to the CPU, the copy of the data is no longer in a PCR and is therefore no longer immutable. However, it is possible to sign these measures. Indeed, if they are signed, a verification of this signature makes it possible to detect that a modification of the data has been made.

This operation of signing PCRs data is named a quote. The TPM provides the command TPM_Quote that allows asking for a quote. With this command, it is possible to provide

a subset of PCRs one wants to quote. The TPM will only quote this subset. The command also takes an extra data input which can be provided by the user. This extra input allows providing a nonce: a random value that will be signed along with the PCRs. This nonce's purpose is to mitigate replay attacks. Indeed, by signing a random value, each quote has a different signature even though PCRs values did not change. Thus, an eavesdropper observing the communication will not be able to record the quote and reuse it later on as a valid quote. The nonce guarantees the "freshness" of the quote.

A quote is not only a signed version of the PCRs values. It also provides essential information: it proves that data comes from inside the TPM, as it was signed using a stored key. Indeed, keys that can sign quotes are also special keys. The EK can not sign anything because of privacy concerns. Thus, Attestation Identity Keys (AIKs), a type of key dedicated to signing quotes, exists. As the SRK wraps these keys, they are protected by the TPM.

The command TPM_MakeIdentity allows creating such a key. It returns the public part of the key along with a "blob", *i.e.*, the newly created private key encrypted by the SRK. When one wants to use this key, they load the blob into the TPM. Loading the blob will decrypt the private key using the SRK, and the TPM will be able to use it to sign a quote.

Even though AIKs purpose is to sign a quote so that another entity can verify it, nothing proves the key signing a quote is stored in the TPM. Thus, proving that the AIK that will sign quotes belongs to the same TPM as the EK is mandatory. A solution to this step is the credential activation protocol[33].

This protocol, shown in the figure 2.7, has two entities involved: the platform with the TPM (as the prover) and the Attestation Certification Authority (ACA), whose role is to deliver a certificate for the AIK. After creating an AIK, the prover sends a certification request containing the AIK's public key and the EK's public key to the ACA. In return, the ACA calculates a challenge, *i.e.*, a random value, which they will encrypt with the two public keys (AIK and EK). Thus, only an entity in possession of the two private keys will be able to decrypt the challenge. The TPM has a built-in command ActivateCredential which fulfills this task. Once the prover has decrypted the challenge, they can send the plaintext challenge back to the ACA, proving they own both private keys. The ACA can then issue and reply with the certificate.

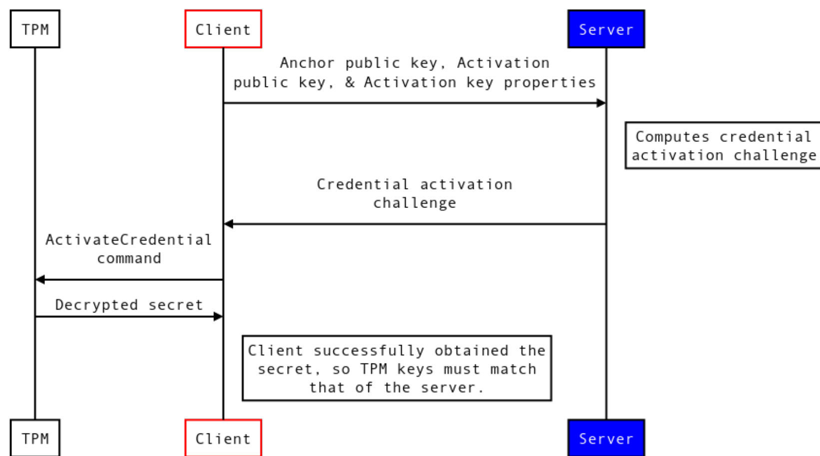

Figure 2.7: Credential activation protocol's sequence diagram.

Chapter 3

Nuvla.io and NuvlaBox

Nuvla.io is an open management platform as a service developed by SixSq to quickly deploy, manage, monitor, and update Edge-to-Cloud applications. The platform is cloud and edge agnostic, which means components of applications can either be deployed in the cloud or edge devices. It supports all forms of infrastructure: public cloud, private cloud, and bare-metal infrastructures. The platform is built around the container technology Docker. Thus any containerized application can be deployed through Nuvla.io. It allows high flexibility for its users to either deploy their own applications or reuse existing ones.

The platform uses container orchestration technologies such as SWARM or Kubernetes to create an abstraction layer between itself and cloud providers and edge devices. As long as either a SWARM or a Kubernetes infrastructure is deployed somewhere and registered, Nuvla.io can deploy containers on this infrastructure. Hence, the deployment of all components of an application can be done in a few clicks.

As shows figure 3.1, Nuvla.io acts as a bridge between the cloud and the edge devices. From this bridge, one can manage its whole Edge-to-Cloud application from a central point. This is enabled by the NuvlaBox software, an ED software stack, also developed by SixSq, that allows managing Edge Devices through Nuvla.io. Thus any device that runs the NuvlaBox software stack becomes a NuvlaBox edge device.

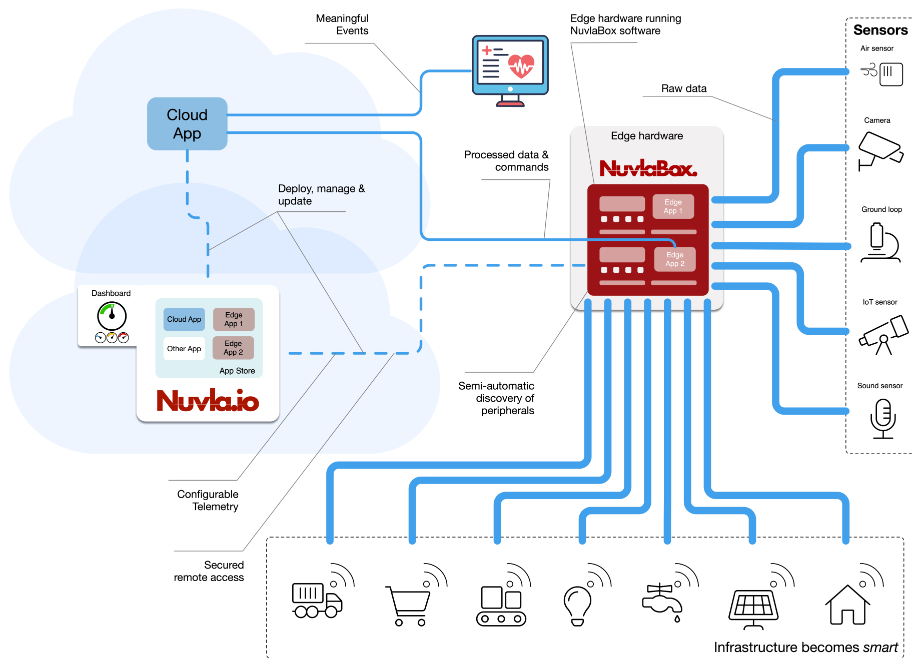

Figure 3.1: Nuvla.io's integration in its environment.

3.1 NuvlaBox architecture

As previously stated, a NuvlaBox is an ED that runs the NuvlaBox software stack. This software stack turns any ARM and x86 hardware platform into a smart ED that can be remote-controlled from Nuvla.io. A non-technical operator can quickly and securely deploy it. NuvlaBox software is certified to work on a range of hardware platforms, including Hewlett Packard Enterprise, Dell, OnLogic, and Raspberry Pi. It is made of several modules that are Docker microservices. The fig 3.2 shows the general architecture of the NuvlaBox Engine that contains several different modules.

agent The main module is responsible for the NuvlaBox activation, the monitoring, and all outgoing communication with Nuvla.io.

system manager It is responsible for checking whether the ED has enough resources to run the whole stack during the NuvlaBox bootstrap. Then it is responsible for checking the health of the whole system. It achieves this by scanning containers looking for defective containers and trying to fix them automatically.

API It is an interface responsible for receiving remote commands from Nuvla.io and forward them to the service, which can respond to the command. It also provides a relay to the docker Application Programming Interface (API) so that Nuvla.io can interact with Docker installed on the NuvlaBox.

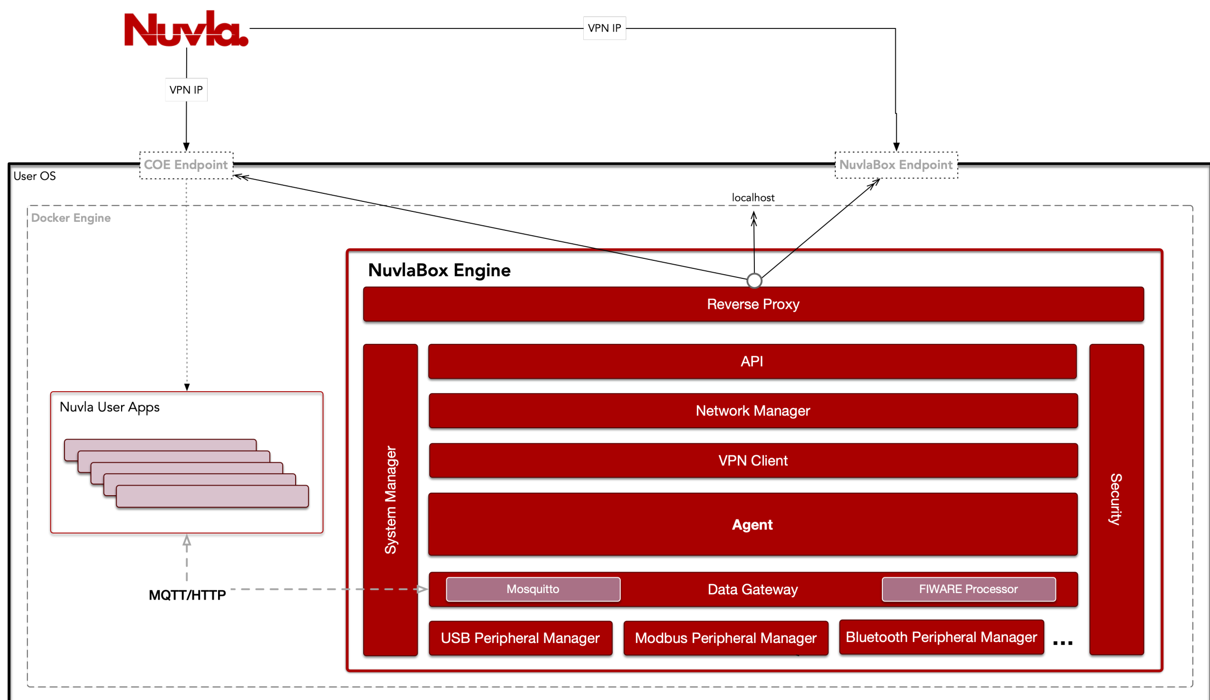

Figure 3.2: NuvlaBox's software stack.

network manager It is responsible for setting up the network configuration the NuvlaBox requires. It ensures this configuration is always valid so Nuvla.io can communicate with the NuvlaBox.

VPN client It is responsible get a configuration from the Network Manager and connecting to a Virtual Private Network (VPN) server. Nuvla.io provides a VPN server, but the client can connect to any server. Hence the NuvlaBox is always remotely accessible.

security It is a standalone agent which periodically scans the NuvlaBox for vulnerabilities is the NuvlaBox. The vulnerability database is also periodically updated by open sources of vulnerabilities.

data gateway It is a gateway between applications deployed on the NuvlaBox and sensors connected to it. It is responsible for collecting sensor data and serve it through HTTP. Thus applications deployed on the NuvlaBox have do not need to bundle sensor-specific data acquisition logic.

peripheral manager Peripheral managers are optional modules that detect and categorize peripherals. They work in pair with the Data Gateway to provide data to applications. Examples of peripherals managers are: *peripheral-manager-usb*,

peripheral-manager-bluetooth, *peripheral-manager-modbus*, *peripheral-manager-gpu*, *peripheral-manager-network*, and others

As Docker and containers are native Linux functionality, these microservices are designed to run on Linux distributions. In theory, the NuvlaBox Engine should work with any Linux distribution. However, SixSq tested it on Ubuntu, CentOS, and Debian and suggested one of these to their customers. In the past few years, Windows and macOS docker versions have been developed. However, these versions work with a tiny Linux VM behind the scene. Hence, NuvlaBox features are not guaranteed on Windows and macOS Docker versions.

3.2 Specification

Although the NuvlaBox security module scans the NuvlaBox for vulnerabilities, one angle of attack is not protected by their solution: The host OS, Docker, and the hardware. Indeed, since the stack is deployed in a containerized environment, each container is individually secured, and the security module ensures this. However, the host system and the Docker installation are not secured. In addition to this, as the stack is made to run on any OS that can run Docker, some likely have unknown security flaws. Thus, from a strict security standpoint, the whole system should not be trusted as it does not provide strong evidence of trustworthiness.

Currently, when a Nuvla.io/NuvlaBox user deploys a NuvlaBox somewhere, even though the Agent monitors the system, they do not know the current global state of the system. For instance, an attacker could replace the NuvlaBox Docker binaries with one they maliciously modified to work the same, but that also collects private data. As the NuvlaBox would still work as expected, the malicious Docker would go undetected by Nuvla.io because the NuvlaBox containers would still send reports. Nevertheless, these reports could probably be falsified.

RAs and measured boot described in the chapter 2 could fill this gap between the hardware and the NuvlaBox software stack. In collaboration with SixSq, we want to develop a TPM-based RA framework that could be integrated into the Nuvla.io and NuvlaBox workflow. It implies we must containerize the framework. It is a challenge as the TPM must be accessed through a Docker container. The framework must allow conducting RA to detect any change in the NuvlaBox host platform. This work will focus on the attestations phase,

i.e., the prover and the verifier, and will not define a measuring policy. The measure part must be carefully defined as it can impact how our framework will integrate into the workflow. Measured boot requires TPM aware components, so they create the chain of trust. However, with our RA framework, it could create a starting point for further work to integrate our framework to provide Nuvla.io and NuvlaBox users strong trust evidence.

Although Nuvla.io and the NuvlaBox engine are very flexible when it comes to the code executed thanks to Docker containers, SixSq still needed to introduce a constraint. The developed solution must not be written in Java. Indeed, Java is a language that needs a virtual machine to run. This leads to a large overhead in terms of memory requirements. This overhead is not suitable for Edge devices with limited resources. In addition, the JVM's memory usage policy is not compatible with Docker by default for some versions of Java. Finally, Docker images for Java are relatively greedy in memory size. For all these reasons, Java is not desired by SixSq.

Chapter 4

Integrating RAs using TPM in NuvlaBox

4.1 TPM and Linux

The Nuvla.io and NuvlaBox solution is built on top of Docker. Thus the framework we develop must be containerized. Containers being a native Linux technology, it is necessary to interface the TPM with Linux. In response to this need, the TCG specified the TCG Software Stack (TSS), also known as TrouSerS, for TPM 1.2[34]. Its role is to be the bridge between applications and the TPM. Since the TPM only has one I/O bus, the stack takes care of several tasks between applications and the TPM like serializing and deserializing the commands and data sent and received. Among the TSS, the TCG Service Provider (TSP) module provides TPM services for applications. On top of it lies the TCG Service Provider Interface (TSPI). This interface provides C headers that allow programmers to develop programs using the TPM. These headers provide a secure and standardized way of interacting with the chip.

To develop our solution, we chose the GO language. GO is a recent statically typed and compiled language developed by Google. It is a high-performance language that enforces good-practices and standard formating. By its statically typed and compiled nature, it allows the development of reliable and stable software, which is imperative in security. It also has a Foreign Function Interface (FFI) feature to interface with C and C++. Finally, GO has the edge over other languages: a TSPI bindings library exists developed by Google employees. Hence, interfacing with C headers is already done. For all these reasons, we

chose the GO language to create the solution.

4.2 Architecture

In the context of remote attestations, two distinct entities, the prover, and the verifier, work together to establish a bond of trust. For the former, it must prove that it is trustworthy using its TPM. For the second, it must validate that the information provided is both correct and legitimate. In other words, the prover has not falsified the information received and that the information received corresponds to that expected.

The figure 4.1 presents the general architecture of the prover side of the application and its environment. As pictured, it depends on TSS and lies in a container. To access the TPM within the container, dockers can share a host device with its containers as if it was a USB key or a keyboard. The application comprises three main components: the TPM interface, the Prover Engine, and the REST API.

Prover Engine is the main component as it implements the prover side logic of the RA and the RA protocol. It is also responsible for handling the registration process. Indeed, as previously stated, RAs have two distinct phases: The registration and the attestations. During the former phase, the prover must send his EK certificate and prove the AIK belongs in its TPM.

TPM interface This module decouples the framework from the TPM version. We focused on TPM version 1.2, but in the optic of improving with new versions, this module only will change while the other modules will remain unchanged. The TPM interface is responsible for offering primitives for the prover to communicate with the TSPI bindings. Indeed, the TSS is made for low-level use of the TPM 1.2. It does not provide higher-level commands such as AIK creation or EK certificate retrieving that the prover needs. Thus, the TPM interface creates an abstraction layer of the TPM. It also provides the structures that allow interaction with the abstraction, such as a PCR, a quote, an EK, or an AIK. Functions using these structures make it easy to perform actions with these objects. In this way, it is easy to manipulate these objects. For example, these functions allow serializing one of these objects in JSON to send them or verify a quote's signature.

REST API is responsible for receiving queries from the verifier and forwards them to the Prover Engine. It also takes care of sending the response back to the verifier.

In the case of the prover, only the attest route is available. This route allows the verifier to ask the prover for a quote.

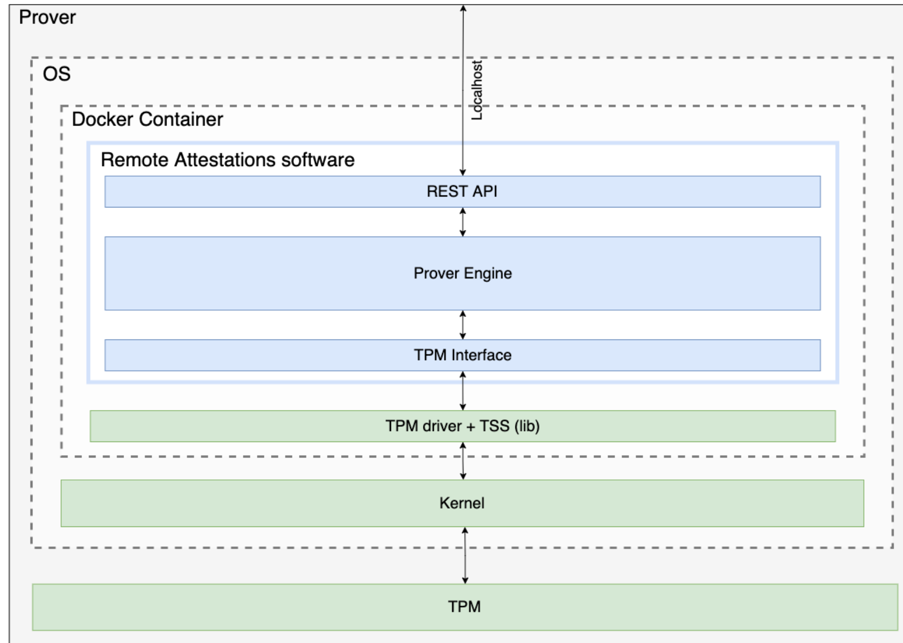

Figure 4.1: Prover's architecture

The opposite side of the prover is the verifier. Its architecture is similar to that of the prover, as shown in the figure 4.2. As the prover, the verifier lies in a container. However, as it does not need a TPM, the verifier software does not depend on the TSS. Thus, its container is lighter. The verifier software comprises four main components: the TPM Utils, the Verifier Engine, the Attestation DB interface, and the REST API.

Verifier Engine is the core of the application. During the registration phase, it verifies the EK certificate sent by a prover. Then he verifies the TPM stores the AIK. Finally, it stores the AIK in persistent storage with some metadata about the prover, such as its IP address. During the attestation phase, it is responsible for periodically asking its provers to attest. When a prover sends back its attestation, the verifier engine is responsible for checking the attestation's authenticity and verifying its content is legitimate, *i.e.*, comparing the PCRs values with those saved.

Attestation DB interface This interface is an abstraction layer that allows using any storage method. It allows the Verifier Engine to store and retrieve pieces of information about provers such as their legitimate states, their AIK, or their IP addresses.

TPM Utils The verifier does not use the TPM but uses structures tightly linked with the TPM version 1.2. This module is separated from the verifier engine to allow the

framework to handle future new TPM versions without changing the verifier engine. The TPM Utils module provides structures representing quotes, PCRs, AIKs, and EKs, and functions on these structures. For instance, this module provides the logic to verify a quote and the logic to deserialize a prover's quote.

REST API is responsible for receiving queries from provers. It is only used during the registration phase so that provers can register.

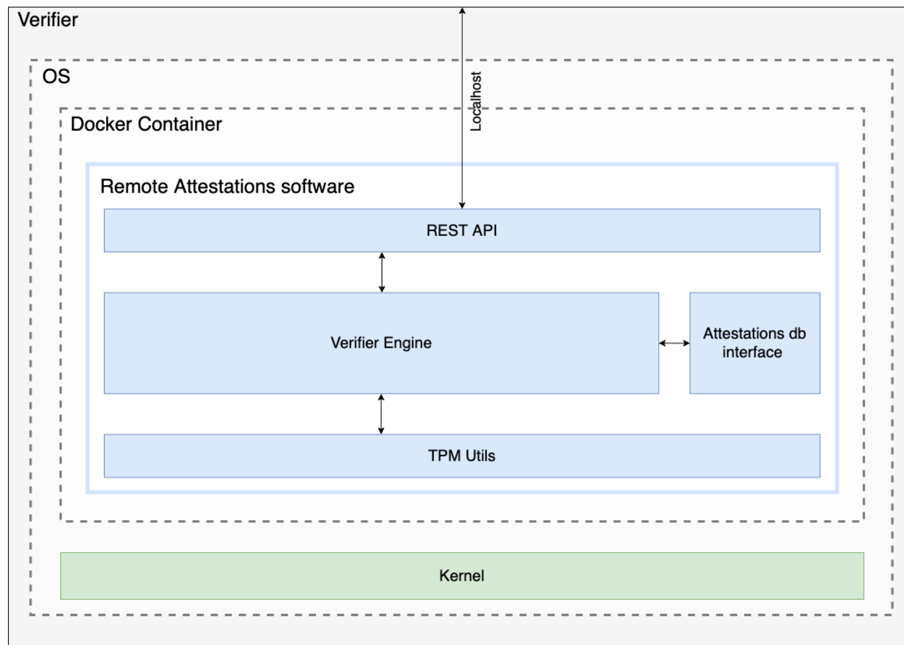

Figure 4.2: Verifier's architecture

4.3 Validation

4.3.1 Test protocol

We defined a test protocol that allowed us to reproduce experiments and get consistent results to validate the software. The protocol is designed in two phases that simulate a legitimate program being altered by an attacker. The first phase is the measuring phase and attesting phase. During this phase, the program is measured, and the resulting measure extends the 23rd PCR. Indeed, the 23rd PCR is dedicated to measuring applications. By measuring, we mean hashing the program with a cryptographic hash function such as SHA-1. Then the values of PCRs are dumped and stored in the verifier database as legitimate. Thus, the verifier starts, and the prover starts and registers. The verifier starts to

ask for attestations. From now on, the prover's attestations should be valid as its PCRs are in a legitimate state. Any other outcome means either the expected state was not stored correctly or that something does not work in the program.

The second phase is the measuring and attesting of an illegitimate program. A program must be maliciously altered to become illegitimate. Any change to the program should be detected, even a single bit added, removed, or flipped. Thus we defined a standard alteration of the program: we append a byte (8 bit) of zeroes at the end of the binary file with the following command:

```
\$ dd if=/dev/zero bs=1 count=1 >> program
```

After the command has altered the program, we re-apply the first phase. The program is measured, and the resulting measure extends the 23rd PCR. The verifier should detect a change in the program because this change was noticed in the prover's PCRs.

As described, the test protocol does not contain measured boot verification for several reasons. As stated in the chapter 2, deploying a measured boot adds a substantial workload that is not required to validate our work. We are trying to validate the RA software that must detect illegitimate states. Even though measured boot would be unavoidable in a production environment, it is unnecessary to validate the RA software. Indeed, as long as a state transitions from a legitimate one to an illegitimate one is detected, our experiment is validated.

We applied our test protocol to two different use cases, including the Nuvla.io and NuvlaBox one. We will present those use cases in detail in the next section.

4.3.2 The noise sensors and spatial accuracy use case

The first use case in which we validated our solution is an application that helps detect misbehaving sensors within a network of sensors. As part of a research project within the HES-SO in collaboration with SABRA, noise sensors have been deployed in the city of Carouge by SABRA. These sensors send reports every 15 minutes. Each report consist of noise levels recorded during the 15 minute period such as :

L_{min}: The minimum recorded level.

L_{max}: The maximum recorded level.

L_{eq}: The equivalent continuous recorded level also sometimes known as Average Sound Level

L_{10}, L_{50}, L_{90}, L_{95}: the level exceeded for 10%, 50%, 90% and 95% of the period (percentiles)

These noise sensors have some data integrity issues. A spatial correlation system is considered to detect problematic sensors. This system compares the data sent by a given sensor with that of its neighbors. Through a Linear Least Square Estimator, the system can estimate the value that a sensor should have sent based on those sent by its neighbors. This estimator's result is then normalized between zero and one to give an objective score of the data's quality. A program written in python computes this score. For the sake of simplicity, we will call this program the precision estimator throughout this work.

Currently, this deployment is not an Edge to Cloud one, but work is done to move towards an Edge to Cloud architecture. This work raised some questions on how to secure those future EDs. RAs are a promising solution to protect the devices from tampering and other software attacks. Thus, our work could be applied to these EDs to secure them. Securing the ED would require measuring each piece of software executed on the ED, including the precision estimator. Assuming these EDs only run an OS and the precision estimator, we can apply our test protocol with the precision estimator as the subject program.

Since Python is not a compiled language, the precision estimator is divided into several python files. Therefore, all files composing it must be measured to detect a malicious alteration of the program.

In reality, the actual program is the python interpreter, which reads the files and acts accordingly. In a real-life situation, it should therefore also be measured. However, in our case, we will only measure the files for the sake of simplicity. To get one hash, we hashed each file individually. Then we gathered the results and hashed them into one single digest to extend the PCR as shown in figure 4.3.

During the first phase of our test protocol, the verifier received attestations and classified those as valid as expected as shown in figures 4.4 and 4.5. Hence, we could move to the second phase. We altered the main file with the dd command and re-measured it as shown in figures 4.6 and 4.7. As in the first phase, we measured all the files and extended the 23rd PCR. At this moment, the verifier started to detect the illegitimate state of the program as shown in figure 4.8. This result validated our solution with the noise sensors and spatial accuracy use case. As we validated our solution with the first use case, we

moved to the second use case: the Nuvla.io and NuvlaBox use case.

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# ./measure 23 accuracyTF/
ab3dd04c0767dafeba78beadb643f017899e1325
Extending PCR 23 with hash: ab3dd04c0767dafeba78beadb643f017899e1325
New PCR value: {23 [91 223 211 243 24 68 191 251 19 2 235 67 21 18 3 220 127 187 83 9]}
```

Figure 4.3: First measure stored in the 23rd PCR.

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File  Edit  View  Search  Terminal  Help
(base)  ludovic@ludovic-mint-private  ~/Cours/Master/TM  docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name ve
rifier ludognd/verifier:latest
{Rest:{Address:0.0.0.0 Port:8080} Verifier:{Init:{OwnerPassword:tpmOwnerPassword UserPassword:tpmUserPassword} Attesta
tionInterval:15s}}
time="2021-03-26T12:53:32Z" level=info msg="starting up REST API on 0.0.0.0:8080\n"
time="2021-03-26T12:53:32Z" level=info msg="Starting attestations"
```

(a) Verifier container starts

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# docker run --rm -it --device=/dev/tpm0 -v /var/lib/tpm:/var/lib/tpm -v ${PWD}/ak.json:/opt/RemoteA
ttestation/ak.json -p 8080:8080 --name prover ludognd/prover:latest
&{Rest:{Address:0.0.0.0 Port:8080} Prover:{Name:prover AKFile:ak.json OwnerPassword:tpmOwnerPassword UserPassword:tpmU
serPassword VerifierAddress:http://10.42.0.1:8080}}
INFO[0001] starting up REST API on 0.0.0.0:8080
INFO[0007] /attest
```

(b) Prover container starts

Figure 4.4: Verifier and Prover containers start. Verifier sends attestation request to prover. Prover receives it and processes it

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File  Edit  View  Search  Terminal  Help
(base)  ludovic@ludovic-mint-private  ~/Cours/Master/TM  docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name ve
rifier ludognd/verifier:latest
{Rest:{Address:0.0.0.0 Port:8080} Verifier:{Init:{OwnerPassword:tpmOwnerPassword UserPassword:tpmUserPassword} Attesta
tionInterval:15s}}
time="2021-03-26T12:54:44Z" level=info msg="starting up REST API on 0.0.0.0:8080\n"
time="2021-03-26T12:54:44Z" level=info msg="Starting attestations"
time="2021-03-26T12:54:53Z" level=info msg=/registerNewEK
time="2021-03-26T12:54:53Z" level=info msg=/registerNewAK
time="2021-03-26T12:54:59Z" level=info msg="Starting attestations"
time="2021-03-26T12:55:00Z" level=info msg="prover(10.42.0.245:8080): Valid Quote"
time="2021-03-26T12:55:00Z" level=info msg="prover(10.42.0.245:8080): Valid PCR state"
```

Figure 4.5: The prover responded the request. The verifier verified it and declared it as valid.

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# dd if=/dev/zero bs=1 count=1 >> accuracyTF/accuracyTF.py
1+0 records in
1+0 records out
1 byte copied, 0.000166806 s, 6.0 kB/s
```

Figure 4.6: On the prover, a malicious modification by an attacker is simulated with the dd command.

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# ./measure 23 accuracyTF/
3ee7aa483ad177b0ca49a0444f742e342a354d31
Extending PCR 23 with hash: 3ee7aa483ad177b0ca49a0444f742e342a354d31
New PCR value: {23 [238 197 184 176 66 54 46 131 18 237 168 246 214 125 188 253 62 218 232 168]}
[root@localhost ~]# docker run --rm -it --device=/dev/tpm0 -v /var/lib/tpm:/var/lib/tpm -v ${PWD}/ak.json:/opt/RemoteA
ttestation/ak.json -p 8080:8080 --name prover ludognd/prover:latest
&{Rest:{Address:0.0.0.0 Port:8080} Prover:{Name:prover AKFile:ak.json OwnerPassword:tpmOwnerPassword UserPassword:tpmU
serPassword VerifierAddress:http://10.42.0.1:8080}}
INFO[0001] starting up REST API on 0.0.0.0:8080
INFO[0010] /attest
```

Figure 4.7: A periodic measure occurs and remeasures the same file and stores it in 23rd PCR.

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File  Edit  View  Search  Terminal  Tabs  Help
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier   ludovic@ludovic-mint-private:/media/ludovic/Data/Cours/Maste...
time="2021-03-26T12:59:23Z" level=info msg=/registerNewEK
time="2021-03-26T12:59:23Z" level=info msg=/registerNewAK
time="2021-03-26T12:59:32Z" level=info msg="Starting attestations"
time="2021-03-26T12:59:33Z" level=info msg="prover(10.42.0.245:8080): Valid Quote"
time="2021-03-26T12:59:33Z" level=error msg="prover(10.42.0.245:8080): Illegitimate PCR state: PCRs don't match Parsed
Quote"
```

Figure 4.8: The verifier re-sends an attestation request. The prover responds. The verifier detects the incorrect state and signals it.

4.3.3 the Nuvla.io and NuvlaBox use case

As stated in the chapter 3, Nuvla.io is a platform developed by SixSq and deployed as a service. This web platform allows the deployment of applications on NuvlaBoxes. NuvlaBoxes are platforms such as EDs that run SixSq's NuvlaBox software stack. This software stack comprises several containers that allow their owners to deploy, monitor, manage, and update applications through Nuvla.io.

NuvlaBox features several security mechanisms such as periodic scans for vulnerabilities, resulting in a high-security level. However, EDs are often deployed in adverse environments such as streets, which exposes them to attacks. These attacks could potentially try to replace components such as the kernel with malicious ones. As the NuvlaBox software stack relies on Docker, attackers could be tempted to replace the docker binary with a malicious one that could hide its malicious side from Nuvla.io. The attacker could also be tempted to replace the kernel as it is the only piece of software shared between the host and the containers. Thus, our solution could help prevent these attacks. For this reason, we are going to validate our solution using this use case.

Assuming the NuvlaBox software stack is deployed on top of an ED, this ED runs a light OS that only features docker. We can therefore validate our solution using our test protocol

with Docker as the measured program. Indeed, if the entire platform is in a legitimate state, the NuvlaBox software stack's security policies take care of the security of the platform.

In reality, Docker does not work by itself to operate containers. In recent years Docker has begun breaking down its monolithic architecture into several small blocks, each with its responsibility, such as the container runtime *containerd* or *runc* which is a Command line Interface (CLI) for spawning containers. A production environment would require measuring each block. However, for the sake of simplicity, we will focus on Docker. Docker is made of three main components: the Docker binary, the Docker daemon and Containerd. The Docker Binary is a CLI that sends commands to the Docker Daemon. The Daemon responds to the commands either by itself or by sending commands to the container runtime Containerd. Hence, we adapted our test protocol to this situation. As shown in figure 4.9, we measured the Docker binary first, then extended the PCR and repeated for the daemon and containerd and stored the resulting PCRs states as a valid state. As a result, the verifier received valid attestations as expected as show figures 4.10 and 4.11. Thus we reached the second phase of the protocol. During the second phase, we started by altering the program. We choose to alter the docker CLI and leave the Docker daemon and containerd untouched as show figure 4.12. Then we re-measured both binaries (figure 4.13). Finally, the verifier detected the change in the prover's state as shown in figure 4.14. This result validated our experiment for the Nuvla.io and NuvlaBox use case.

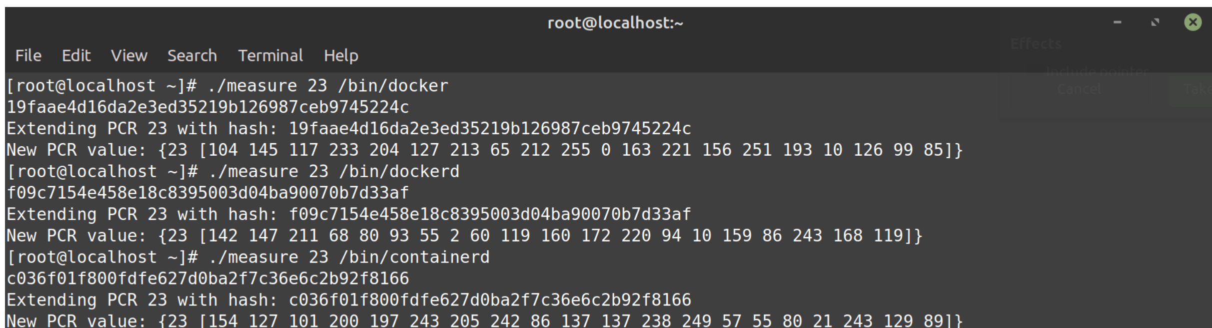

Figure 4.9: First measure stored in the 23rd PCR.

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File  Edit  View  Search  Terminal  Tabs  Help
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier    ludovic@ludovic-mint-private:/media/ludovic/Data/Cours/Maste...
(base)  ludovic@ludovic-mint-private  ~/Cours/Master/TM  docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier ludognd/verifier:latest
{Rest:{Address:0.0.0.0 Port:8080} Verifier:{Init:{OwnerPassword:tpmOwnerPassword UserPassword:tpmUserPassword} AttestationInterval:15s}}
time="2021-03-26T13:07:11Z" level=info msg="starting up REST API on 0.0.0.0:8080\n"
```

(a) Verifier container starts

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# docker run --rm -it --device=/dev/tpm0 -v /var/lib/tpm:/var/lib/tpm -v ${PWD}/ak.json:/opt/RemoteAttestation/ak.json -p 8080:8080 --name prover ludognd/prover:latest
&{Rest:{Address:0.0.0.0 Port:8080} Prover:{Name:prover AKFile:ak.json OwnerPassword:tpmOwnerPassword UserPassword:tpmUserPassword VerifierAddress:http://10.42.0.1:8080}}
INFO[0001] starting up REST API on 0.0.0.0:8080
INFO[0010] /attest
```

(b) Prover container starts

Figure 4.10: Verifier and Prover containers start. Verifier sends attestation request to prover. Prover receives it and processes it

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File  Edit  View  Search  Terminal  Tabs  Help
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier    ludovic@ludovic-mint-private:/media/ludovic/Data/Cours/Maste...
(base)  ludovic@ludovic-mint-private  ~/Cours/Master/TM  docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier ludognd/verifier:latest
{Rest:{Address:0.0.0.0 Port:8080} Verifier:{Init:{OwnerPassword:tpmOwnerPassword UserPassword:tpmUserPassword} AttestationInterval:15s}}
time="2021-03-26T13:07:11Z" level=info msg="starting up REST API on 0.0.0.0:8080\n"
time="2021-03-26T13:07:11Z" level=info msg="Starting attestations"
time="2021-03-26T13:07:17Z" level=info msg=/registerNewEK
time="2021-03-26T13:07:17Z" level=info msg=/registerNewAK
time="2021-03-26T13:07:26Z" level=info msg="Starting attestations"
time="2021-03-26T13:07:27Z" level=info msg="prover(10.42.0.245:8080): Valid Quote"
time="2021-03-26T13:07:27Z" level=info msg="prover(10.42.0.245:8080): Valid PCR state"
```

Figure 4.11: The prover responded the request. The verifier verified it and declared it as valid.

```
root@localhost:~
File  Edit  View  Search  Terminal  Help
[root@localhost ~]# dd if=/dev/zero bs=1 count=1 >> /usr/bin/docker
1+0 records in
1+0 records out
1 byte copied, 0.000155084 s, 6.4 kB/s
```

Figure 4.12: On the prover, a malicious modification by an attacker is simulated with the dd command.

```
root@localhost:~
File Edit View Search Terminal Help
[root@localhost ~]# ./measure 23 /usr/bin/docker
80f8156caa908930272aca6cb5d2cbc57df8ed2c
Extending PCR 23 with hash: 80f8156caa908930272aca6cb5d2cbc57df8ed2c
New PCR value: {23 [242 207 8 145 91 197 10 237 4 219 52 228 30 74 104 183 213 4 81 158]}
[root@localhost ~]# ./measure 23 /usr/bin/dockerd
ae04c9e41119777489c9e97d0e136f387bd72a9f
^[[AExtending PCR 23 with hash: ae04c9e41119777489c9e97d0e136f387bd72a9f
New PCR value: {23 [174 119 177 209 31 214 112 213 8 13 35 33 51 41 159 158 230 21 103 63]}
[root@localhost ~]# ./measure 23 /usr/bin/containerd
004c5dde30119b984f919962d02c1dc79214db00
Extending PCR 23 with hash: 004c5dde30119b984f919962d02c1dc79214db00
New PCR value: {23 [32 37 254 98 60 107 238 14 77 161 8 245 107 212 22 53 80 156 239 187]}
```

Figure 4.13: A periodic measure occurs and remeasures the same file and stores it in 23rd PCR.

```
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier
File Edit View Search Terminal Tabs Help
docker run --rm -p 8080:8080 -v ${PWD}/pcrs:/pcrs --name verifier   ludovic@ludovic-mint-private:/media/ludovic/Data/Cours/Maste...
time="2021-03-26T13:12:13Z" level=info msg=/registerNewEK
time="2021-03-26T13:12:13Z" level=info msg=/registerNewAK
time="2021-03-26T13:12:25Z" level=info msg="Starting attestations"
time="2021-03-26T13:12:26Z" level=info msg="prover(10.42.0.245:8080): Valid Quote"
time="2021-03-26T13:12:26Z" level=error msg="prover(10.42.0.245:8080): Illegitimate PCR state: PCRs don't match Parsed
Quote"
time="2021-03-26T13:12:41Z" level=info msg="Starting attestations"
time="2021-03-26T13:12:42Z" level=info msg="prover(10.42.0.245:8080): Valid Quote"
time="2021-03-26T13:12:42Z" level=error msg="prover(10.42.0.245:8080): Illegitimate PCR state: PCRs don't match Parsed
Quote"
```

Figure 4.14: The verifier re-sends an attestation request. The prover responds. The verifier detects the incorrect state and signals it.

4.4 Conclusion

We developed a containerized RA framework that uses TPM version 1.2. We wanted to validate our framework with two use cases. Hence, we defined a test protocol and applied it to the use cases. Our test protocol defines how our framework will be used in the use cases. The test is in two phases. In the first phase, we want to detect a legitimate state, while the second phase aims to detect a malicious state. To switch from a legitimate to a malicious state, we defined a standardized manipulation that adds one byte of zeroes to a measured file. During the second phase, we detected the malicious state in both of our use cases. These results successfully validated our framework.

Conclusion

Our work aimed to develop a Remote Attestation (RA) framework for detecting malware injection attacks in EDs. Our framework developed in GO uses TPM-based RA to remotely detect state changes in the Edge device (ED)'s configuration. We validated our work on two edge computing use cases: the trust noise use case and the Nuvla.io and NuvlaBox use case. In the former, ED collects noise data. Each collected sample is validated by a spatial correlation module that uses neighbors samples to quantify whether the sample is acceptable. The goal was to detect a malicious change in the spatial correlation source files. In the latter use case, Nuvla.io is a platform that allows deploying, managing, monitoring, and updating Edge-to-Cloud applications. NuvlaBox is an ED that runs Nuvla.io's software stack in Docker containers. The goal was to detect a malicious change in the Docker binaries.

Our results are promising. We were able to detect a malicious change in the prover state in both use cases. However, to detect this change, a new measure was required on the prover. The current state of the prover's RA software does not perform any measurement by itself. The main reason is that defining what is measured is a policy that must be carefully defined according to the use case. We developed a RA framework. A framework is a tool, not a solution; when used with a carefully defined measuring policy, it can remotely detect state changes.

As previously stated, defining a clear policy of what is measured and when it is measured is primordial to detect changes. Depending on what we measure, we might detect runtime malware injection attacks. At the same time, our validation only focused on static malware injection attacks in which the attacker wrote their malware in the persistent memory. It is possible to measure runtime software; however, the measuring software must know precisely when and what to measure to get clean and consistent results. Hence, it could be challenging to use our framework to detect runtime attacks, and other detection systems might be more appropriate.

The difficulty in defining the measure policy lies in the fact that the tree of possible legitimate states presented in the figure 2.4overgrows. The policy must be as efficient as possible, *i.e.*, measure each component representing a threat while keeping the tree as small as possible. An example of a measure that could help to reduce this tree size is defining an order of measurements. Since the tree is sensitive to the order of measures, allowing different orders might cause the tree to overgrow. Each permutation of legitimate measurements leads to a legitimate state that must be described to the verifier during the measuring phase. In our work, we defined proof of concept use cases that were quite simple. Indeed, there was only one legitimate state, and any change was considered illegitimate. Hence, the underlying policy definition was straightforward. Real use cases would be much more complicated.

Lausanne, the April 22, 2021

Ludovic Gindre

Bibliography

[1] (). "IoT market size worldwide 2017-2025", Statista, [Online]. Available: https://www.statista.com/statistics/976313/global-iot-market-size/ (visited on 02/28/2021).

[2] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and other botnets", *Computer*, vol. 50, no. 7, pp. 80–84, 2017, Conference Name: Computer, ISSN: 1558-0814. DOI: 10.1109/MC.2017.201.

[3] D. Bonderud. (Oct. 4, 2016). "Leaked mirai malware boosts IoT insecurity threat level", Security Intelligence, [Online]. Available: https://securityintelligence.com/news/leaked-mirai-malware-boosts-iot-insecurity-threat-level/ (visited on 03/01/2021).

[4] Y. Xiao, Y. Jia, C. Liu, X. Cheng, J. Yu, and W. Lv, "Edge computing security: State of the art and challenges", *Proceedings of the IEEE*, vol. 107, no. 8, pp. 1608–1631, Aug. 2019, Conference Name: Proceedings of the IEEE, ISSN: 1558-2256. DOI: 10.1109/JPROC.2019.2918437.

[5] A. Greenberg, "The reaper botnet has already infected a million networks", *Wired*, 2017, ISSN: 1059-1028. [Online]. Available: https://www.wired.com/story/reaper-iot-botnet-infected-million-networks/ (visited on 02/25/2021).

[6] A. Cui, M. Costello, and S. Stolfo, "When firmware modifications attack: A case study of embedded exploitation", 2013. DOI: 10.7916/D8P55NKB. [Online]. Available: https://doi.org/10.7916/D8P55NKB (visited on 02/25/2021).

[7] G. Hernandez, O. Arias, D. Buentello, and Y. Jin, "Smart nest thermostat: A smart spy in your home", p. 8, 2014.

[8] J. Maskiewicz, B. Ellis, J. Mouradian, and H. Shacham, "Mouse trap: Exploiting firmware updates in {USB} peripherals", presented at the 8th {USENIX} Workshop on Offensive Technologies ({WOOT} 14), 2014. [Online]. Available: https:

//www.usenix.org/conference/woot14/workshop-program/presentation/maskiewicz (visited on 02/25/2021).

[9] T. Li, X. Wang, M. Zha, K. Chen, X. Wang, L. Xing, X. Bai, N. Zhang, and X. Han, "Unleashing the walking dead: Understanding cross-app remote infections on mobile WebViews", in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, Dallas Texas USA: ACM, Oct. 30, 2017, pp. 829–844, ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134021. [Online]. Available: https://dl.acm.org/doi/10.1145/3133956.3134021 (visited on 02/25/2021).

[10] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: Software-based attestation for embedded devices", pp. 272–282, 2004. DOI: 10.1109/SECPRI.2004.1301329. [Online]. Available: http://ieeexplore.ieee.org/document/1301329/ (visited on 02/25/2021).

[11] C. Castelluccia, A. Francillon, D. Perito, and C. Soriente, "On the difficulty of software-based attestation of embedded devices", in *Proceedings of the 16th ACM conference on Computer and communications security - CCS '09*, Chicago, Illinois, USA: ACM Press, 2009, p. 400, ISBN: 978-1-60558-894-0. DOI: 10.1145/1653662.1653711. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1653662.1653711 (visited on 02/26/2021).

[12] V. Costan and S. Devadas, "Intel SGX explained", 086, 2016. [Online]. Available: http://eprint.iacr.org/2016/086 (visited on 02/26/2021).

[13] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin, "TrustZone explained: Architectural features and use cases", in *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*, Pittsburgh, PA, USA: IEEE, Nov. 2016, pp. 445–451, ISBN: 978-1-5090-4607-2. DOI: 10.1109/CIC.2016.065. [Online]. Available: http://ieeexplore.ieee.org/document/7809736/ (visited on 02/26/2021).

[14] TCG, *TPM main part 3 commands*. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf (visited on 02/02/2021).

[15] ——, *TPM main part 2 TPM structures*. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf (visited on 02/02/2021).

[16] ——, *TPM main part 1 design principles*. [Online]. Available: https://trustedcomputinggrou
org/wp-content/uploads/TPM-Main-Part-2-TPM-Structures_v1.2_rev116_01032011.pdf (visited on 02/02/2021).

[17] A. Tang, S. Sethumadhavan, and S. Stolfo, "{CLKSCREW}: Exposing the perils of security-oblivious energy management", presented at the 26th {USENIX} Security Symposium ({USENIX} Security 17), 2017, pp. 1057–1074, ISBN: 978-1-931971-40-9. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang (visited on 02/25/2021).

[18] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A high resolution, low noise, l3 cache side-channel attack", presented at the 23rd {USENIX} Security Symposium ({USENIX} Security 14), 2014, pp. 719–732, ISBN: 978-1-931971-15-7. [Online]. Available: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom (visited on 02/25/2021).

[19] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems", in *2015 IEEE Symposium on Security and Privacy*, ISSN: 2375-1207, May 2015, pp. 640–656. DOI: 10.1109/SP.2015.45.

[20] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks", *arXiv:1703.06986 [cs]*, Aug. 20, 2017. arXiv: 1703.06986. [Online]. Available: http://arxiv.org/abs/1703.06986 (visited on 02/25/2021).

[21] N. Zhang, H. Sun, K. Sun, W. Lou, and Y. T. Hou, "CacheKit: Evading memory introspection using cache incoherence", in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*, Saarbrucken: IEEE, Mar. 2016, pp. 337–352, ISBN: 978-1-5090-1751-5 978-1-5090-1752-2. DOI: 10.1109/EuroSP.2016.34. [Online]. Available: https://ieeexplore.ieee.org/document/7467364/ (visited on 02/25/2021).

[22] R. Keegan, *Microarchitectural attacks on trusted execution environments*, Dec. 27, 2017. [Online]. Available: /v/34c3-8950-microarchitectural_attacks_on_trusted_execution_environments (visited on 02/25/2021).

[23] I. Bente, B. Hellmann, T. Rossow, J. Vieweg, and J. von Helden, "On remote attestation for google chrome OS", in *2012 15th International Conference on Network-Based Information Systems*, Melbourne, Australia: IEEE, Sep. 2012, pp. 376–383, ISBN: 978-1-4673-2331-4 978-0-7695-4779-4. DOI: 10.1109/NBiS.2012.55. [Online]. Available: http://ieeexplore.ieee.org/document/6354852/ (visited on 03/24/2021).

[24] M. Garrett. (Oct. 30, 2020). "Attestation at enterprise scale - TPM.dev 2020 Mini-Conf - day 2", TPM.Dev, [Online]. Available: https://developers.tpm.dev/posts/9242884 (visited on 03/24/2021).

[25] TCG. (). "About TCG", Trusted Computing Group, [Online]. Available: https://trustedcomputinggroup.org/about/ (visited on 01/14/2021).

[26] Dansimp. (). "BitLocker (windows 10) - microsoft 365 security", [Online]. Available: https://docs.microsoft.com/en-us/windows/security/information-protection/bitlocker/bitlocker-overview (visited on 03/25/2021).

[27] C. Fruhwirth, *New methods in hard disk encryption*, Jul. 18, 2005. [Online]. Available: https://clemens.endorphin.org/nmihde/nmihde-A4-ds.pdf.

[28] "Intel® trusted execution technology hardware-based technology for enhancing server platform security", p. 8,

[29] cbedford. (). "Documentation de VMware vSphere", [Online]. Available: https://docs.vmware.com/fr/VMware-vSphere/index.html (visited on 03/25/2021).

[30] (). "Virtual trusted platform module (vTPM) - xen", [Online]. Available: https://wiki.xenproject.org/wiki/Virtual_Trusted_Platform_Module_%28vTPM%29 (visited on 03/25/2021).

[31] (). "ChangeLog/2.11 - QEMU", [Online]. Available: https://wiki.qemu.org/ChangeLog/2.11#TPM (visited on 03/25/2021).

[32] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov, "The first collision for full SHA-1", in *Advances in Cryptology – CRYPTO 2017*, J. Katz and H. Shacham, Eds., vol. 10401, Series Title: Lecture Notes in Computer Science, Cham: Springer International Publishing, 2017, pp. 570–596, ISBN: 978-3-319-63687-0 978-3-319-63688-7. DOI: 10.1007/978-3-319-63688-7_19. [Online]. Available: http://link.springer.com/10.1007/978-3-319-63688-7_19 (visited on 01/26/2021).

[33] TCG, "AIK certificate enrollment", p. 72, Mar. 24, 2011. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/IWG_CMC_Profile_Cert_Enrollment_v1_r7.pdf.

[34] ——, *TCG software stack (TSS) specification version 1.2 level 1*. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_Version_1.2_Level_1_FINAL.pdf.

[35] D. Gambetta, "Can we trust trust? diego gambetta", Aug. 9, 2000.

[36] R. N. Akram and R. K. L. Ko, "Digital trust - trusted computing and beyond a position paper",

[37] (). "Definition of TRUST", [Online]. Available: https://www.merriam-webster.com/dictionary/trust (visited on 01/14/2021).

[38] TCG, *TCG PC client specific implementation specification for conventional BIOS*, Feb. 2012. [Online]. Available: https://www.trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientImplementation_1-21_1_00.pdf (visited on 02/02/2021).

Abbreviations

ACA Attestation Certification Authority

ACS Access Control System

AIK Attestation Identity Key

API Application Programming Interface

CA Certification Authority

CC Cloud Computing

CLI Command line Interface

CPU Central Processing Unit

CRTM Core Root of Trust Measurement

DDoS Distributed Denial-of-Service

DoS Denial-of-Service

Edge-to-Cloud Edge to Cloud

ECC Eliptic-Curve Cryptography

EC Edge Computing

ED Edge device

EK Endorsement Key

FFI Foreign Function Interface

GPU Graphics Processing Unit

IoT Internet of Things

MBR Master Boot Record

MITM Man in the Middle

OS Operating System

PCR	Platform Configuration Register
PCR	Platform Configuration Register
RA	Remote Attestation
RSA	Rivest-Shamir-Adleman
RTM	Root of Trust for Measurement
RTR	Root of Trust for Reporting
RTS	Root of Trust for Storage
RTS	Root-of-trust-for-storage
SHA-1	Secure Hash Algorithm 1
SHA-256	Secure Hash Algorithm 256
SRK	Storage Root Key
SWATT	SoftWare-based ATTestation
TCG	Trusted Computing Group
TCO	Total Cost Ownership
TPM	Trusted Platform Module
TSPI	TCG Service Provider Interface
TSP	TCG Service Provider
TSS	TCG Software Stack
VM	Virtual Machines
VPN	Virtual Private Network

List of Figures

1.1 Percentage of attacks targeting Edge-computing [4] 9

1.2 General Remote Attestation scheme. 9

1.3 General Hardware-based Remote Attestation scheme. 11

2.1 Components of a Trusted Platform Module complying with the TPM version 1.2 standard, Credits: Eusebius (Guillaume Piolle), https://en.wikipedia.org/wiki/Trusted_Platform_Module#/media/File:TPM.svg 17

2.2 Example of a chain of trust with a root certificate, an intermediate certificate, and an end-entity certificate, Credits: Yuhkih (Wikipedia) Source: https://commons.wikimedia.org/wiki/File:Chain_Of_Trust.svg#/media/File:Chain_Of_Trust.svg 18

2.3 Example of key pairs hierarchy with SRK as the root key 20

2.4 Example of a hash tree with two branches, one legitimate and one malicious 23

2.5 Each measured boot process step is detailed as a sequence diagram. . . . 25

2.6 Chain of trust starting from CRTM to the OS and extending to applications. 26

2.7 Credential activation protocol's sequence diagram. 28

3.1 Nuvla.io's integration in its environment. Credits: SixSq© Source: https://nuvla.io 30

3.2 NuvlaBox's software stack. Credits: SixSq© Source: https://sixsq.com . . 31

4.1 Prover's architecture 37

4.2 Verifier's architecture 38

4.3 First measure stored in the 23rd PCR. 41

4.4 Verifier and Prover containers start. Verifier sends attestation request to prover. Prover receives it and processes it 41

4.5 The prover responded the request. The verifier verified it and declared it as valid. 41

4.6 On the prover, a malicious modification by an attacker is simulated with the dd command. 41

4.7 A periodic measure occurs and remeasures the same file and stores it in 23rd PCR. 42

4.8 The verifier re-sends an attestation request. The prover responds. The verifier detects the incorrect state and signals it. 42

4.9 First measure stored in the 23rd PCR. 43

4.10 Verifier and Prover containers start. Verifier sends attestation request to prover. Prover receives it and processes it 44

4.11 The prover responded the request. The verifier verified it and declared it as valid. 44

4.12 On the prover, a malicious modification by an attacker is simulated with the dd command. 44

4.13 A periodic measure occurs and remeasures the same file and stores it in 23rd PCR. 45

4.14 The verifier re-sends an attestation request. The prover responds. The verifier detects the incorrect state and signals it. 45

List of Tables

A.1 PCRs usages 61

Appendix A

PCRs usages

PCR Index	PCR Usage
0	S-CRTM, BIOS, Host Platform Extensions, and Embedded Option ROMs
1	Host Platform Configuration
2	Option ROM Code
3	Option ROM Configuration and Data
4	IPL Code (usually the MBR) and Boot Attempts
5	IPL Code Configuration and Data (for use by the IPL Code)
6	State Transitions and Wake Events
7	Host Platform Manufacturer Specific
8	Defined for use by the Static OS
9	Defined for use by the Static OS
10	Defined for use by the Static OS
11	Defined for use by the Static OS
12	Defined for use by the Static OS
13	Defined for use by the Static OS
14	Defined for use by the Static OS
15	Defined for use by the Static OS
16	Debug
17	Dynamic CRTM
18	Dynamic RTM
19	Dynamic RTM
20	Dynamic RTM
21	Dynamic RTM

22	Dynamic RTM
23	Application Specific

Table A.1: PCRs usages