Politecnico di Torino

Master's Degree in Computer Engineering

Master's Degree Thesis

Remote Attestation and Integrity Verification Solution in a Cloud Based Environment

Supervisors

Prof. Fulvio Risso
Prof. Thanassis Giannetsos

Candidate

Paride D'Angelo

December 2020

Acknowledgements

A big thank you goes to both my supervisors that allowed me to make this amazing experience in Denmark working at this Thesis project. This experience has been a big breakthrough for me because it permitted me to meet incredible and amazing people along the way that changed myself as a person and contributed to enriching the final outcome of this work. And so also a thank you to all the friends that I met in Denmark.

A special gratitude to my family and my relatives that, throughout all of these academic years, has supported me in all my choices and decisions allowing me to arrive where I am now.

An important thanks goes to all my colleagues of Politecnico di Torino with which I have spent all the days, of both the Bachelor's and the Master's programs, following courses and projects together. I think that many goals would have not been possible without your support and friendship.

A thank you to all my close friends and flat mates with which I have shared a lot of time and which pushed me in doing better every day.

I think that behind each final result, like for this Thesis work, there is a mix of experiences and persons that make it possible, some of them directly some others indirectly; people and events which are not visible but which are all part of the basement of the big iceberg whose peak is represented by the big reached goal.

Contents

Chapter 1

Introduction

1.1 Motivations

Nowadays, Cloud Computing technologies have become increasingly used in many different fields and applications. Companies are moving their resources and applications to a Cloud based environment finding it beneficial in terms of lower cost, higher performance, accessibility, and scalability. Also previous solutions and architectures, which before where relying just on the hardware capabilities of a device, now are moving towards a Cloud virtualized environment. At the same time technology presence has become more and more pervasive generating a big growth in the number of devices connected over the internet. Therefore networks and the devices connected to it have changed in the last years bringing to the use of new types of network schemes and of new kinds of related technologies. The current trend and direction networks are following is towards the increase of deployment of edge devices and fog computing nodes in order to move the computing infrastructure as close as possible to the source of data and to the users.

In this kind of context the capabilities of edge devices is increasing, allowing them to have more computational power and to deliver low latency and responsive services. The two current technologies that are central in this scenario and that make this shift possible are the Mobile Edge Computing (MEC) and the Cloud Computing. The new trend is to use both of these two technologies together in new kinds of solutions. This new tendency can be identified in Network Function Virtualization (NFV). The increase of Cloud Computing deployment and use generates a process of "softwarization" and virtualization of the remote edge devices: the remote devices become systems running in a Virtualized environment, allowing

more flexibility and customization of their applications and functionalities.

The increase of the number of edge nodes, like the IoT devices, connected to the network and the new kinds of software-cloud solutions and virtualizations used in the new application solutions make the surface for potential attacks and the number of potential vulnerabilities grow exponentially. In particular, the use of Virtualization and Cloud Computing techniques opens new kinds of security problems related to the verification of the correctness and integrity of remote edge device systems. For these reasons, new solutions are needed to generate a verifiable evidence of the correctness and integrity of the software running on a remote Virtual Machine.

A set of solutions and concepts related to the integrity verification process have been provided by the Trusted Computing Group[14] under the name of *trusted computing*: it refers to technologies, proposals and concepts for resolving computer security problems through hardware and software enhancements. The two most important concepts used in this context are *boot attestation* and *remote attestation*. All the solutions proposed by the TCG rely on the use of a so called *trusted anchor* which acts as a root of trust to secure and provide trustworthiness of all the operations during the attestation integrity verification process. The module proposed as a trust anchor by the TCG, that has also been used in this work, is the Trusted Platform Module (TPM).

The recent solutions proposed up to now are not able to address all the new security issues related to containers and Cloud Computing environment in a thorough and complete way. The focus of the main solutions that have been proposed is on providing integrity verification at boot-time or at most at load-time without considering possible attacks that might tamper the software program during run-time, i.e. during its execution. Load-time integrity techniques perform software measurements computation just before the software program is loaded into main memory not considering attacks that could happen during run-time. Among the solutions proposed by the trusted computing group the IMA [3] helps in performing boot-time and load-time attestation verification.

Although the IMA has many limitations, it has been one of the starting point for this work providing the basic useful concepts when it comes to remote attestation. The solution proposed here wants to build on top of the IMA capabilities extending the remote attestation protocol to be executed during run-time and to be integrated with a Cloud based Environment, also providing a stronger verifiable evidence of the the Prover's state.

The other solutions that have been proposed in the past in the remote attestation branch do not consider the possibility to perform the integrity check in a Cloud

based Environment. This work, instead, has as central objective to suggest a kind of attestation solution to be used for containers running on remote devices. The Cloud based environment is the central context and focus of this remote attestation work.

In the context described here, comprising multiple edge devices exploiting Cloud Computing capabilities, there is not only the need of a solution that is able to execute a successful remote attestation of the VMs but also of a solution that is scalable in the number of devices that can be attested. This comes from the fact that the Verifier needs to potentially manage and attest a large number of remote edge Virtual Machines at the same time. The aim of this work is to provide a very good solution in terms of scalability in the number of remote device the Verifier can attest.

The classical remote attestation pattern is the one where the Verifier attests the state of the edge devices' system. In fact all the previous proposals focused their attention on this type of scheme. In this work instead new solution paradigms have been designed permitting not only Verifier to VM attestation but also the attestation between two Virtual Machines.

As part of the whole protocol solution proposed, also two different execution tracer techniques have been used and integrated to enhance the level of security and tracking of the operations performed during the integrity verification process. The most crucial and critical phases of the whole integrity verification process are the run-time extraction of the binary data to be attested and all the communication process with the TPM. This is the first solution that also proposes a way to secure and verify these two phases by using two kinds of tracers: the eBPF tracer and the Intel Processor Tracer. They provide different levels of granularity and details of logged information performing different levels of execution tracing, enhancing the security and trustworthiness of the integrity verification steps.

1.2 Project objectives

The scope of this work is to provide a trustworthy and reliable solution to perform remote attestation integrity verification in order to provide a strong and verifiable evidence of the Prover's system running in a virtualized Cloud environment. Each remote system, that has to be attested, is therefore considered running on a Virtual Machine. And the whole solution has been designed with this context in mind.

The trust anchor used to guarantee security properties in the protocol solution

proposed here is the Trusted Platform Module (TPM), following in this way the guidelines provided by the TCG for performing reliable remote attestation. All the TCG concepts and directives provided by the TCG group, for what concerns remote attestation, have been taken and extended to enhance security and to adapt this kind of solutions to work in a Cloud Computing environment. In fact the solution in this document makes use of a software TPM running in the Virtual Machine of the remote device to be attested.

Two different protocols to perform the integrity verification process have been introduced. Although they follow different approaches and steps, they both rely on the use of the TPM and both of them are specifically designed based on TPM functionalities. These two protocols are defined into two different kinds of attestation schemes:

- Attestation by Quote: which is based on the production of a quote attestation data in order to allow the Prover to be attested. It follows more closely the guidelines provided by the TCG, being part of the classic scheme of remote attestation where the Verifier receives from the Prover the measurements, which reflects Prover's system state, to attest.
- Attestation by Proof: which is based on the use of TPM policy digest authentication mechanism, which combined to an Attestation Key can provide a verifiable evidence of Prover's system state. This protocol makes use of TPM specific properties to perform remote attestation in a new way.

This proposed solutions focus on providing remote attestation integrity verification during run-time, i.e. attestation during the execution of the software running on the Prover's system. Therefore providing the possibility for the Verifier to perform remote attestation at any time and a multiple number of times without interrupting the attested software's execution, and so without needing any reboot of the Prover's container. In addition to this the Verifier will be able to manage and attest an arbitrary, potentially high, number N of remote devices at the same time, making this solution highly scalable.

One of the main objectives here is to also enhance the level of security of the operations performed during the remote attestation protocol. These include the steps performed to extract the properties and the binary data to be attested as well as the communication and the commands exchanged with the TPM. For this reasons two tracing techniques have been used: the eBPF hooks and the Intel PT. Their main objective is to trace and verify the correctness of the overall state of the Virtual Function. The eBPF hooks and the Intel PT are two different technologies for doing tracing, permitting to reach two different levels of granularity of

logged information. This aspect is essential in the effectiveness that they, combined together, can reach in tracing Virtual Machine system.

The objective of this work is to present these two different types of tracers as techniques to be used combined into a remote attestation solution. The main objective of these two tracing techniques is to secure and verify the extraction of the binary data to be attested and to check the correctness of the messages and commands exchanged with the TPM. In this way the level of security and trustworthiness of the single steps and phases of the remote attestation process has been enhanced through these tracers capabilities. The Intel PT in particular is evaluated and proposed as a technique, not only to trace the execution of a "normal" program, like the binary extractor, but also to log the execution of a tracing program like the eBPF hooks. In this way the more high level capabilities of the eBPF are combined with the Intel PT, which performs the additional low level check guaranteeing a higher level of security in the overall attestation process.

An additional analysis of their capabilities and differences have been conducted to show their strengthens and limitations. They have been used also as a way to conduct a study on Control Flow Integrity and Control Flow Attestation techniques. This kind of techniques allow to detect attacks (like the Return Oriented Programming attack) which try to divert the execution flow of a software program without changing its binaries. Specifically the Intel PT has been used for the purpose of reconstructing the exact execution flow of a program in order to attest its control flow integrity. A specific analysis has been conducted with the aim of evaluating the highest level of program tracing complexity that can be managed efficiently by the Intel PT especially in terms of performance and scalability.

1.3 Structure of the report

Across this document, the solution proposed to deploy a remote attestation integrity verification protocol is presented. The document introduces, first of all, the technologies and background that have been used and then the actual architecture and implementation of the scheme. The explanation of the solution proposed follows a top-down approach: at the beginning the high level view and the big picture of the architecture, as well as the overall system model, are presented and then, in the following chapters, the single components of the solution are explained more in detail to better understand how they work and interact to each other.

In **chapter 2** the related works are presented, showing the history of all the

remote attestation solutions designed in the past that has been taken as reference of this work. Moreover this chapter highlights the improvements and additions introduced and carried out by this solution in comparison to the others.

In **chapter 3** describes the system and the threat model. The system model section provide a high level overview of the scheme and the actors present in this work. The threat model shows the attacker perspective and so what an adversary can and can not do in such a system.

In **chapter 4** all the technologies used and the theoretical backgrounds associated with them are presented. It reports all the information needed to understand how the architecture and the implementation work. Moreover the specific tools used in the implementation, for each technology, have been specified here. In this way whoever wants to repeat the experiments can know which tools to use.

Chapter 5 presents the overall architecture of the solution stating all the actors involved as well as their role and why they have been used. This description provides also the steps and the main phases that comprise the whole remote attestation protocol.

In the **chapter 6** all the step of the two designed integrity verification protocols are presented in detail. It shows the measurements and results obtained as well as all the commands used to implement this solution and all the evaluations and analysis performed on it. It describes in detail how the solution works showing the more practical part of work. This chapter also shows in detail how the eBPF and Intel PT work and all the conducted experiment with them.

Eventually **chapter 7** describes the main limitations and problem related to the solution deployment. Pointing out what can be done next and what can be improved to extend such a solution.

Eventually **chapter 8** contains the final conclusion comment to this document and the future works section. The future works section suggests what can be done next and the possible enhancements and problems to address to improve the current proposed solution, summarizing what was already stated in the chapter 7.

Chapter 2

Related works

Before talking about the actual solution and how it has been built it is important to mention all the previously related works, that have been done in the past, related to attestation. So lets examine the history of remote attestation solutions developed and theorized up to now to better understand which were the initial assumptions of this work and all the improvements and enhancements that have been brought.

The history of attestation starts from the 2001 when the Trusted Computing Group released the first Trusted Platform Module documentation in the Trusted Computing Platform Alliance document [1]. In that document the TPM attestation was introduced for the first time.

After that, in 2004, some relative researches were published focusing on property based attestations mechanisms [2]: attestation not just of a binary data but directed to attest real properties of the system.

Still in 2004 an other research papers proposed a software-based attestation solution[4]. This proposal aimed to avoiding hardware changes in order to implement an attestation solution providing a software alternative based on a series of checksums operations under determined assumptions.

In 2005 the dynamic root of trust was introduced. One example can be the Intel TXT [18] which is an hardware technology relying on TPM whose main goal is to perform attestation of the authenticity of a platform dynamically loading it into a trusted execution environment protecting it from software-based attacks.

Between 2010 and 2012 the new trend focused on adapting and applying attestation methodologies to embedded systems through minimal trust anchors. The

TPM was not suitable to be integrated into embedded systems due to cost reasons. One of the researches focused in this direction was SMART [5].

Between 2011 and 2014 some papers like PUFatt [6] focus on authentication enforcement of the Prover to the Verifier based on a physical and unclonable function.

Finally the last trend in around 2015 and 2016 was to focus the research on the attestation of a network of devices [9].

A big contribution towards the integrity verification direction has been given in 2004 with the standardization and implementation of the TCG-based Integrity Measurement Architecture (IMA) [3]. It permits to collect system files measurements either at boot-time, or before they are modified or also at load-time (in case the data to be attested is a software program), so just before the binaries are loaded into main memory. If available it makes use of a TPM to provide attestation of the system measurement at boot time or at load time.

Lets now understand the position of this work with respect to the others. The remote attestation and integrity verification solution proposed here starts and follows the initial assumptions and guidelines defined by the Trusted Computing Group initially in 2001 and also later, like for example with the IMA solution. However the aim of this work is to extend that concepts to provide a more scalable and secure solution broadening the working context even more.

Some of the cited paper tried to perform remote attestation by experimenting among different hardware or software solutions [4, 6]. Some of them make use of additional hardware features, like the Intel TXT [18], to perform the attestation while some others remove completely hardware components suggesting a solution that relies completely on software [4]. The solution proposed here instead makes just use of the TPM as trust anchor module to perform the whole attestation process. It does not make use of additional hardware to provide verifiable evidence of the remote Virtual Function but just exploits the security properties of the TPM as defined by the TCG [14]. Moreover, a software TPM implementation has been used resulting in not even having the TPM as a hardware component. However, even though the actual remote attestation procedure has been performed only by means of the TPM, the Intel Processor Tracing hardware feature [17] has been used in the final overall solution to enhance the correctness, trustworthiness and security of the general attestation protocol by tracing the communication with the TPM and the execution of the eBPF hooks. The assumption that has been done here is on the requirement of the presence of an Intel processor on the remote device in order to use its Intel PT feature to perform the Control Flow Attestation procedure. However the Intel PT is a pseudo-hardware based technique

representing an additional feature of the Intel processor, which nowadays is very widespread and already present built-in into many devices. Therefore the use of the Intel PT into the solution proposed here does not imply the use of additional hardware but it just makes use of an already existing Intel processor's feature.

All the solutions related to attestation make use of a *root of trust* which is a component trusted by the OS providing security properties, operations and correctness in the attestation process. The two main *trust anchors* used in previous projects are the Trusted Execution Environment (TEE) [16] and the TPM. The TPM provides a set of cryptographic functionalities, like the secure key management, in the form of APIs and allows a system to provide an evidence of its integrity. The TEE is a secure area that allows the execution of arbitrary code within a confined environment providing tamper-resistant execution to its applications [16]. Even though they are both valid alternatives the *root of trust* selected in this work is the TPM because nowadays it is widely used, there is a growth in the number of modern devices that already integrates it and it seems to be an increasingly prominent technology.

As mentioned, in 2004 a first property-based attestation approach was proposed [2]. Also in this work an integrity verification solution is proposed permitting to attest some specific properties of the system. In this way it is attested whether the system fulfills a set of properties without revealing its specific software or hardware configurations. However the solutions proposed in the past allowed just to use a specific and unique property configuration to be attested. The solution proposed here extends this mechanisms providing the possibility to attest multiple different properties at the same time for different devices. The Proof integrity verification protocol does that allowing the Verifier to define totally different customized properties to be attested for each different remote device. This provides a complete personalization of the attestation process depending on the specific case and application generating different levels of attestation granularity. The policy digest based advanced authentication mechanism of the TPM is used for this purpose.

Some past papers were specifically focused on IoT devices to provide low cost and efficient attestation solutions [5], and so avoiding to use the TPM. This work not focuses so much on IoT devices but more on the providing a solution that works in a Cloud based Environment.

In fact the point where this work wants to stand out, with respect of the other past research papers, is in the fact of providing a solution that brings the attestation paradigm into a Cloud based Environment. Since Cloud Computing has become more and more popular in the recent years this work wants to create a solution to implement a remote attestation scheme thought and working in a Cloud

Computing environment. This has been the main focus and context of this work. The main goal here is to provide a strong verifiable evidence of the Prover's system state to the Verifier in a Cloud based Environment. Through the use of the TPM and its strong security properties the designed protocol assures the Verifier on the trustworthiness and authenticity of the attestation data produced by the Prover, which reflects its system state.

Moreover all of these solutions permit to attest Prover's system only at boot-time or at most at load-time. They do not make the additional step of providing integrity verification during run-time. The solution proposed in this paper aims to fill this security hole in the attestation integrity verification domain. This work wants to provide a solution, always relying on a TPM module, to permit the Verifier to attest the Prover also during run-time in a trustworthy and verifiable way. This is also a consequence of the fact of working in a virtualized context. A container cannot be restarted every time a new integrity verification is performed, therefore the only way to do that is by performing attestation during run-time. The main advantage of that is that the Verifier can perform the attestation procedure at any time. The TCG IMA [3] implementation documentation, although implementing and addressing only attestation at load or boot time, had already theorized and the proposed the possibility and possible enhancement related to run-time tracing and the Cloud Computing integration. This work builds on top of that and proposes a solution to fill that left gap.

A deliverable document, belonging to the Future TPM project, called "Future Proofing the Connected World" [11], has been used as initial reference for the design of the quote attestation protocol. It has been the starting point that gave the idea for repeating a similar scheme, under the name of attestation by Quote protocol, but working in a virtualized environment and by making use of the Intel TSS commands. Then in this work that scheme has been additionally extended by providing a new one called attestation by Proof, but it will be described better afterwards in this document.

This solution also follows the trend started in 2015 of the attestation of network of devices. The aim here is to provide a scalable and lightweight solution that allows the Verifier to attest the system of a multitude of remote edge devices connected to the network.

Other works that have been used as reference are the C-FLAT [10], the CFPA [13] and the DIAT [12]. These three recent papers show another type of remote attestation called Control Flow Attestation consisting in the verification of Control Flow Integrity of the software running on the Prover side. They make use of sampling and tracing techniques to trace the execution of the program execution

in order to generate a control flow graph of the instructions executed during runtime. While those research works make use of software tracing and instrumentation techniques to record the execution flow, in the work of this document an hardware feature of the Intel processor, called Intel PT, has been used to log and extract the control flow graph during program execution. The Intel PT use is proposed here in the context of Control Flow Integrity verification. Its strengthens and limitations are showed, after a detailed analysis, in order to understand its properties. The main goal is to propose it as a valid alternative to other software tracing techniques used in Control Flow Attestation.

Taking as reference the CFPA [13] an eBPF hook tracer has been used and adapted to trace and collect information about the TSS commands exchanged with the TPM. The eBPF has been proposed as a valid software tracer to verify the correctness of the messages and data incoming and outgoing of the TPM. In this way a new type of eBPF possible application is showed allowing us to make an investigation on the capabilities of this technology in a different context.

This two tracers have been combined together in the solution proposed in order to join their characteristics and properties and to enhance the level of security and correctness of the integrity verification protocol. For this reason the Intel PT has been used to trace and verify the eBPF tracer itself guaranteeing its exactness.

Chapter 3

Problem statement

3.1 System model

Lets have a look at the basic model and concepts of our system. The remote attestation is a method used by a system (Prover) to authenticate its hardware or its software to a remote host (Verifier). This means that the remote device that acts as a Prover needs to provide a verifiable evidence about the software running on it to the Verifier. Therefore, the approach used to design a remote attestation protocol is to first understand how can the Verifier trust the attestation data sent by the Prover in a reliable way before deciding how the actual data exchanged should be. The Verifier must be able to identify the remote device in a unique way and must have a reliable warranty that the attestation data has been produced by that remote device and that it really reflects its system state.

Remote attestation is meant to verify the integrity of a software running on the system of the remote device. The security requirement of authenticity of the attestation data sent by the remote Prover is ensured by means of a so called *trust anchor*, which is a trusted component placed on the Prover system. In this work the trust anchor used is the Trusted Platform Module (TPM). It is a cryptographic coprocessor that, placed on the remote Prover system, allows it to execute a series of cryptographic and secure operations useful in the attestation procedure. Moreover the TPM has been designed and built to ease the remote attestation process providing all the required attestation functionalities.

In the last years, however, Cloud Computing demand and use has grown a lot and also its number of applications has broadened. For these reasons the remote attestation solution proposed has been designed and extended to be deployed in

a Cloud environment. The Cloud Computing has been used as central context of work. That has been realized by making each remote device's system, that needs to be attested, running on a Virtual Machine. In fact, the remote devices that need to be attested will often referred as "Virtual Machine" or VM in its short form across the other sections of this document.

Having in mind the Cloud environment as main context of work means finding a solution that can work and is deployable in a Virtualized environment; and so also trying a series of tests and analysis, on the technologies and modules intended to be used, inside a Virtual Machine. For example it has been chosen to use a software TPM in order to make it working in a Virtual Machine. The alternative could have been the use of a virtual TPM but at the end the software version was selected making the container having its own TPM, not shared with other OSs.

The architecture proposed allow the Verifier to attest multiple remote devices. It is a highly scalable solution, allowing the Verifier to manage, potentially, a high number of remote devices. The amount of data needed to be stored in the database of the Verifier is not so much for each remote device. The Verifier can keep a log of all remote devices' state to have a reference history of all the past performed remote attestations for each Prover.

The just described scheme reflects the basic general ideas and context behind this work. In our specific context a different nomenclature has been used to refer to the different actors of the remote attestation architecture. Here are the names used for some of the main actors described up to now:

- The Verifier is referred as **Orchestrator** in this work's context
- The Prover is directly referred as **Virtual Machine (or VM)**, due to the fact that the Prover's system runs in a Virtualized environment, or in some cases also just as remote device

Figure 3.1 exhibits the high level view of the model described up to now, showing its basic actors, i.e. the Orchestrator and the remote Provers devices (Virtual Machines).

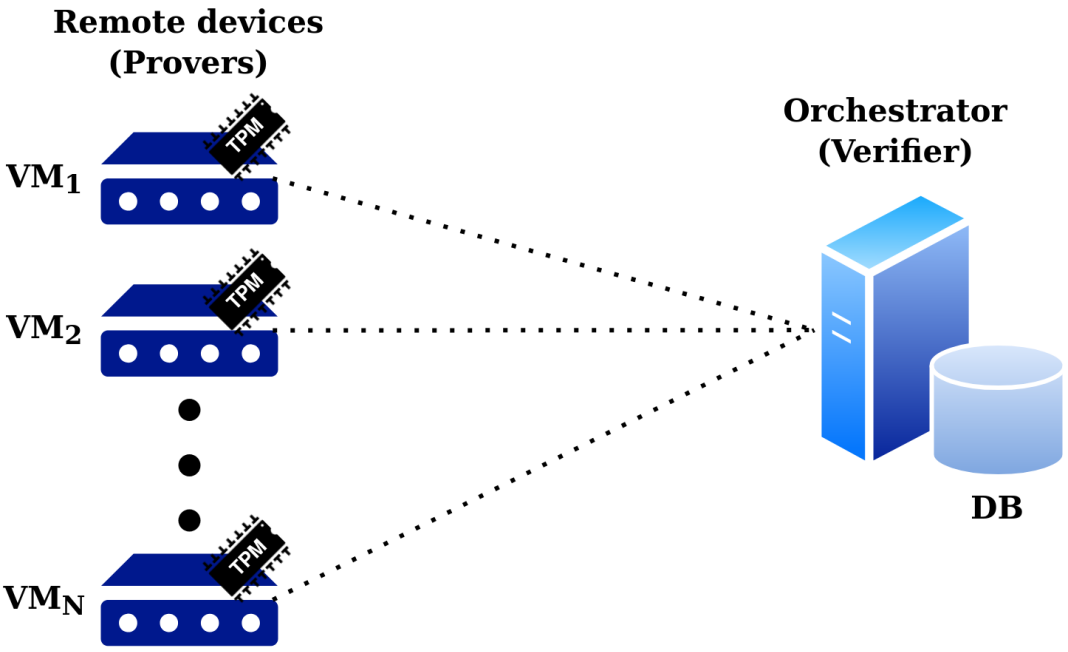


Figure 3.1. Remote attestation high level system model

The figure 3.1 shows that there is an Orchestrator connected through the network to N different remote devices, each of them having its own TPM. The Orchestrator has also a DB to store the reference values and the attestation information of all the remote devices.

Based on this scheme the Orchestrator can perform the remote attestation of a i-th (where i $\in$ {1, ...N}, with N equal to the number of remote devices) device in order to assess its system state. The key aspect of the solution proposed here is that the Orchestrator can start a remote attestation integrity verification process of a target Virtual Machine at any time t and multiple consequent times during run-time without the need of system reboot. This procedure will be clear when talking about the integrity verification protocol. In such a scheme it is always the Orchestrator that starts the remote attestation protocol sending a command to the remote device to trigger the whole process.

After receiving the integrity verification initiation message the VM has to extract the loaded binaries, compute the needed measurements on them and then produce the attestation data in order to provide a verifiable evidence of the system's state. The binary extraction is a potentially critical operation and it is secured by means of the Intel PT. This tracing mechanism allows to generate the exact control flow of the execution of the binary extractor in order to verify its correctness and also

the exactness of the data extracted.

The measurement and the production of a verifiable evidence includes the use of the PCRs of the TPM. They are used to store the measurement value that represents the system state and its extension operation is used to update its content according to the integrity verification protocol.

The methodology in which the VM provides the verifiable evidence of its state and the definition of the sequence of operations that must be performed by the VM and by the Orchestrator is defined by the integrity verification protocol. The overall protocol and the messages exchanged with the TPM are secured by means of the eBPF hooks program. Its role is to ensure correctness in the communication with the TPM.

After attesting the remote Virtual Machine the Orchestrator obtains as result the status indication of the integrity of the remote device's system, which may turns out to be *Trusted* or *Untrusted*. Based on the final outcome of the attestation process the Orchestrator can behave accordingly, taking or not some actions in response. If the outcome is positive, and so the system has not been changed and in a good state, the Orchestrator can just let the remote device continue its normal working; if the outcome is negative, therefore the attested system might have been tampered by a malicious actor, the Orchestrator should intervene by taking some decisions and some actions in order to tackle the problem. Based on the specific application and context in which the remote attestation has been deployed the actions that can be taken are different. For example if the remote devices are routers of a network the action that the Orchestrator could take is to shout down the tampered router and update the routing table in order to divert and redistribute the network traffic.

When talking about attestation of remote device's system we refer to the attestation of the integrity of a specific application or software program running on the VM of that remote device. The integrity and state of that specific critical software program of interest represents the state of the entire system it belongs to. Therefore, if that specific running software process is not considered trusted anymore because has been tampered by an attacker, the whole VM is considered untrusted by the Orchestrator. The goal of remote attestation is to attest a specific critical target section, which again is representative of the whole system in our purposes, instead of attesting the integrity of the entire OS or file system. The attestation process has to be narrowed to the specific data or software program of interest.

This solution scheme provides a series of security properties to the remote attestation protocol. The security properties ensured are: of course *integrity* of the VM's binaries, *authentication* of the remote VM and of the data that it produces,

non repudiation of the attestation data, since it is signed with the Endorsement Key unique for each TPM, *confidentiality* and *availability* of data, *accountability* of the data and information about the remote devices.

3.2 Threat model

After talking about the overall model of the system lets have a look at what an attacker can or can not do in this kind of scheme.

What an adversary would like to do is to tamper the software running on the remote device without letting the Orchestrator noticing it. Lets analyse how he might try to do that.

- MITM attack between the Orchestrator and the VM: the attacker might do a MITM attack between the Orchestrator and the VM. In this way he could try to manipulate the traffic and messages exchanged. However, apart from the public nonce and the public policy digest, the attestation and critical data are always sent through the network signed by the AK belonging to the TPM and associated to its Endorsement Key, making the falsification of the data impossible for an attacker.
- Tampering directly the software running on the VM: the attacker may try to directly tamper the binaries running on the VM without considering the integrity verification protocol. In this case it is important to examine the length of time window between two subsequent integrity verification checks requested, sent by the Orchestrator. After that the attacker takes over he remote device system the system will still be considered in a trusted state by the Orchestrator until it will request another integrity verification check. Therefore, in this kind of situation an attacker has a certain time window during which he can control the VM system without the Orchestrator knowing it. The length of this time window depends on the time intervals at which the Orchestrator performs the remote attestation requests and so its value can be decided during implementation. In the best case, i.e. in case the Orchestrator performs subsequent integrity verification of the remote device in a row one right after the other, the time window will be very short (in the order of 0,2-0,3 seconds by considering the TSS_Execute interface measurement) as it can be better seen from the tables 6.2, 6.3. In this way the malicious manipulation can be detected very fast.
- Manipulating the loaded binaries extraction: the attacker could try to fake

the binaries extraction process and tamper the loaded binaries. This possibility has been prevented by instantiating the Intel PT to trace and attest at low level the execution flow of the binary extractor.

- Manipulation of the communication with the TPM: the attacker could try to manipulate commands and parameters exchanged with the TPM. The eBPF hooks program is deployed specifically with the purpose of preventing this by tracing and verifying the correctness of the data sent towards and from the TPM.

- The eBPF tracer tampering: an attacker could try to manipulate the data extracted and logged by the eBPF tracer. The Intel PT is used to prevent this by tracing and recording the sequence of action performed during TSS commands extraction.

- Corruption of the Intel PT decoding process: if the attacker corrupts the post-processing program used to reconstruct the control flow graph produced by the Intel PT it may be a problem in terms of security because he could tamper the extraction of the loaded binary without letting the Orchestrator noticing it. The assumption that is made here is that the Intel PT control flow tracing is considered trusted.

- Reply attack: reply attacks are avoided by means of the use of nonces in the protocol.

Chapter 4

Background and technologies

4.1 Trusted Platform Module 2.0 (TPM 2.0)

The Trusted Platform Module (TPM), also known as the International Standard ISO/IEC 11889, is a cryptographic coprocessor. It is meant to secure hardware through its cryptographic capabilities. TPM specifications have been published by the Trusting Computing Group. Initially, as a first version, the TPM 1.2 was published and subsequently the new version TPM 2.0 was released adding and updating functionalities of the TPM. In this work we are going to make use of a TPM 2.0 and we are going to interact with it through the TSS 2.0 library.

Nowadays the TPM is almost always present as a coprocessor on our PCs or on server machines. Lately its use and demand is growing very fast thanks to its standardization and to its certification from government. It also proves to be a very efficient module capable of supporting security operations effectively.

The TPM 2.0 enables a series of cryptographic functionalities like:

- Keys management
- Hash computation
- Encryption/decryption functionalities
- Sign/verify functionalities
- Hardware random number generator
- Sealing and binding functionalities

Some specific functionalities may be understood better analyzing the single modules of the TPM architecture. The way the TPM is designed makes it a perfect candidate to be used for remote attestation and system integrity verification purposes.

It has been designed by TCG having some main goals in mind:

- Identification of a device: before the TPM device were identified through their MAC or IP address. Some of its unique characteristics like the Endorsement Key allow a device to be uniquely identified by a TPM
- Secure generation and storage of keys: management of keys left to the system may be vulnerable to software attacks and not so secure compared to TPM key management which relies on an hardware random number generator and a reliable way of key storing.
- Device remote attestation: the TPM is been though as a way to ensure system attestation and integrity as a more secure alternative to software techniques.

Based on the context of deployment the TPM may be implemented either as hardware device or as a software TPM. Furthermore, it could be either local on a device or remote communicating through the network to provide its functionalities to the device. Lets analyse the different possible contexts of use.

In general the TPM is sold as an hardware device by the TPM vendor. It contains a unique Endorsement Primary Seed (EPS) and an Endorsement Key and it can be integrated to a machine like a PC or a server as a cryptographic coprocessor.
When it is needed to use the TPM in a Virtual Machine, or more in general in a Cloud Environment, it cannot be accessed directly by the guest OS. The access to the TPM of the platform must be enabled by the Virtual Machine: therefore it is necessary that the Virtual Machine supports the virtualization of the hardware TPM module in order to make it also visible to the guest OS. When the hardware TPM is virtualized by the guest operating system it is called virtual TPM. An implementation example that follows this approach is the QEMU [22], which is a TPM TIS hardware interface following the Trusted Computing Group's specification.

The other approach that can be used in a Virtualized Environment is to use a software TPM. The software TPM is implemented following the documentation of the TPM definition provided by the TCG and so has the same properties and

functionalities of the hardware TPM. It works as a server process listening on a specific IP address and a specific port number. By default the IP address is 127.0.0.1 (localhost) and the port number is 2321. Once the software TPM is running it can receive TSS commands and send a response back. It is a flexible solution and works well in all the platform both virtualized and not. An example of software TPM is the ibmswtpm [15]

If the TPM is local on the machine that needs to execute secure cryptographic operations the communication between the system and the TPM happens through the bus of the machine. However, when the TPM is remote the network is the means of communication with the TPM. In both case must be considered that data sent to and received from the TPM is vulnerable to possible changes performed by an attacker acting like a MITM. A way to secure the communication with the TPM must be found for sensitive application, ensuring the integrity of the messages exchanges with the TPM.

In this work a local software TPM running on the Virtual Machine of the remote device has been used to perform the attestation. This choices were made because we consider as context of work the Cloud Computing environment. The software and virtual TPM solutions are both valid alternatives. At the end the software TPM have been chosen because it is simple to use, because it runs inside the container, resulting in a software TPM for each container scheme. However, in case the virtual TPM is used in a multiple containers per virtual function scheme many problems need to be addressed because the TPM will be shared among all of the Virtual Machines.

The IBM TPM software implementation has been used (downloaded from https://sourceforge.net/projects/ibmswtpm2/files/latest/download) and all its initial configurations, like the IP address and port number on which the server listen, have been kept unchanged. The thing to remember when using a software TPM is that it is necessary to start it up before being able to communicate with it.

4.1.1 Architecture

In figure 4.1 TPM architecture and its main components are reported. As mentioned also in the documentation it is necessary that all the components of the TPM specification are present in an implementation in order to be a trustworthy and complete standard[25]. There are different parts but they are all connected to each other to provide TPM functionalities. Lets analyse TPM architecture's components in brief and their role:

- I/O buffer: it is the means by which the system can interact and send and receive data with the TPM. It is basically the link between the system and the TPM components.
- Hash engine: it performs hashing operations and could be used both directly by the system and as part of other TPM operations (like as part of the key derivation function). The algorithm supported are the ones belonging to the SHA family.
- Asymmetric and symmetric engines: they are used for all the operations that involve asymmetric and symmetric key like signing, verifying, encrypting and decrypting.
- Key generation: it allows the generation of a new key. It generates either an ordinary key by using the Random Number Generator or a primary key starting its computation from a hierarchy seed. These techniques make use of a Key Derivation Function which generates a new key starting from an initial seed.
- RNG: it creates a random number and it could be useful in many situation like the creation of a key or like the creation of the nonce when needed.
- Authorization Subsystem: it is in charge of performing authorization checks before allowing the execution of a command.
- Volatile memory (RAM): it holds TPM transient data. So all the data stored on it will be lost if the power of the TPM is removed. It includes PCR, keys loaded, current sessions, etc.
- Non-volatile memory: it stores persistent objects and data associated to the TPM. So it retains data even if the power is removed. This memory contains Shielded Locations which can be accessed only through Protected Capabilities. It holds also data inserted by the TPM vendor like seed values or possible Endorsement Keys. Moreover it has some free memory useful to make some transient object permanent.
- Power Detection: it handles power states.

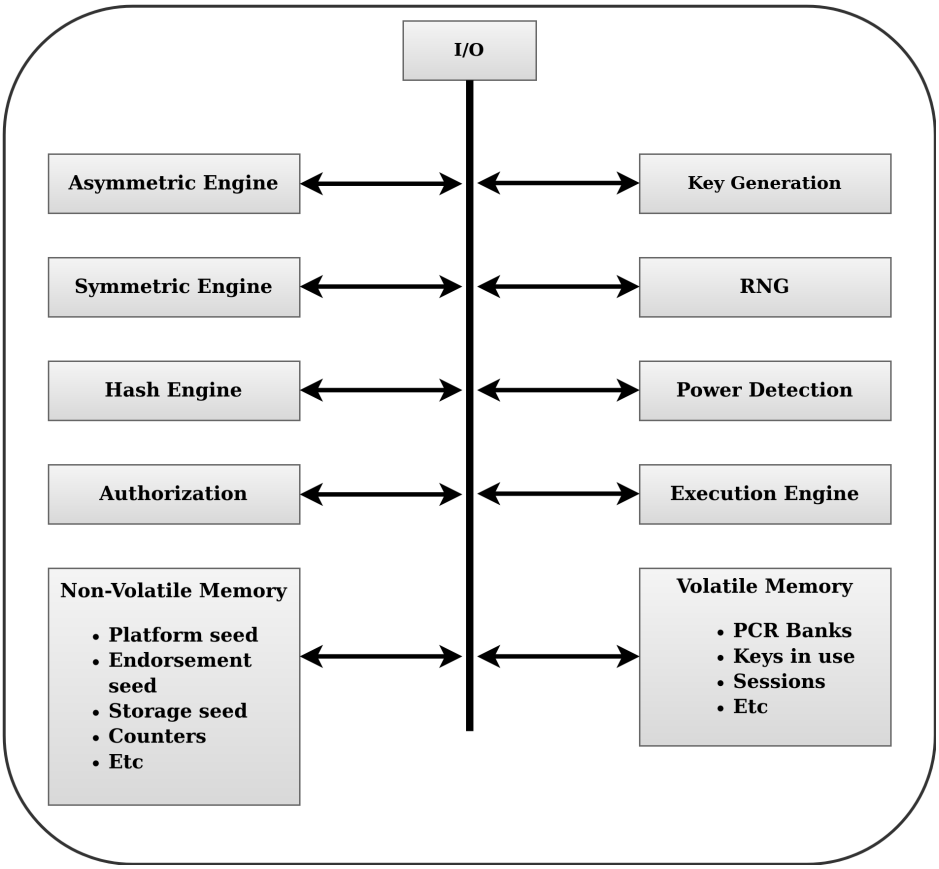


Figure 4.1. TPM Architecture

4.1.2 Hierarchies

According to the documentation of the TPM a hierarchy is a collection of entities that are related and managed as a group[8]. Those entities include permanent objects (the hierarchy handles), primary objects at the root of a tree, and other objects such as keys in the tree[8]. It is the way the TPM manages and organizes its own entities (which in most of the cases they are symmetric or asymmetric key).

According to the manual the cryptographic root of each hierarchy is called *seed*: it is a large random number that is generated inside the TPM at manufacturing time and it is never exposed outside[8]. Moreover each hierarchy has a *proof value* which is derived from the *seed* and which is used to ensure that a value has been generated from the TPM itself[8]. It is like a fingerprint of the TPM that it applies

on all the data that has been generated by itself in order to later check the data authenticity.

There are different types of hierarchies in the TPM, each of them meant for a different purpose. They can be either be persistent or volatile: the persistent hierarchies are retained after a reboot while the volatile ones loose their information after a reboot.

There are three persistent hierarchies:

- Platform Hierarchy
- Storage Hierarchy
- Endorsement Hierarchy

In addition to them there is only one volatile hierarchy, which is the NULL hierarchy.

The common properties are that each of them has an authorization value and a policy, each of them has a persistent *seed* which is used to derive keys and objects and that each of them can have a primary key from which all the descendant keys and objects can be created. The primary keys are keys that can be directly created starting from the *seed*, which is the cryptographic root of the hierarchy; and then other keys or objects can be created as a child of the primary key.

- The Platform hierarchy is meant to be used and controlled by the platform manufacturer, represented by the early boot code inserted in the platform by the manufacturer (BIOS)[8].
- The Storage hierarchy is meant to be used by the owner of the platform.
- The Endorsement hierarchy is the privacy tree and is meant to be used when a certain level of privacy must be ensured.
- The NULL hierarchy volatile, its *seed* isn't persistent and the *proof* is regenerated with different values on each reboot, the authorization is a password of length equal to zero and the policy is empty.

Since during the remote attestation integrity verification protocol it is necessary to ensure privacy and security during the whole process, the hierarchy used in this context is the Endorsement hierarchy. The Endorsement hierarchy has been

designed for remote attestation and for high security solutions and its properties make it suitable for our purposes.

The most important aspect of the Endorsement hierarchy is that its tree root is generated by the TPM manufacturer. The TPM vendor at manufacturing time select a unique primary seed (which form this time on will characterize the specific TPM) for the Endorsement hierarchy, generates one or more primary keys obtained from the seed and eventually generates a certificate for each of them (in general it is an x.509 key certificate). This property of the Endorsement hierarchy is necessary when a remote actor has to certify that a key is resident and associated uniquely to a specific TPM. The whole remote attestation protocol is based on this Endorsement hierarchy property: the Verifier, when receiving data from the remote device it is trying to attest, can rely on the Endorsement key and its certificate (issued by the TPM vendor) to know if that data was really sent by the target remote device or by another (malicious) actor.

Moreover the Endorsement hierarchy ensures privacy. In the context of the TPM the privacy property assumes a specific meaning. Privacy in this domain means the inability of remote parties (potentially malicious) receiving several TPM signatures to correlate them. Correlate signatures means to cryptographically prove that they come from the same TPM. The user making use of the TPM would like to use different keys for different purposes and applications; the attacker could try to correlate them, i.e. trace these keys back to a single user.

4.1.3 Endorsement Key

The definition of Endorsement Key (EK) according to the manual is: symmetric key pair consisting of a public and private key stored in a Shielded Location on the TPM[7]. It is defined as an RSA 2048 bit key. It is a primary key belonging to the Endorsement hierarchy, therefore derived from the Endorsement seed. Its private part MUST be kept secret, so it is never exposed outside of the TPM, while its public part can be read from outside.

As mentioned for the Endorsement hierarchy it is the TPM vendor who generated and preinstalled the Endorsement seed (Endorsement Primary Seed (EPS)) and one or more Endorsement Keys, as well as a public key certificate associated to each Endorsement Key (which in general is an x.509 certificate) at manufacturing time. After that the Endorsement Key has been created it is stored inside the TPM as an object and, any subsequent time, it is referenced by its object handle. The object handle is returned by the TPM every time the Endorsement Key is loaded. The key handle is initially transient and could be made persistent by

using the command TPM2_EvictControl. The range assigned to the Endorsement Primary Keys is 81 01 00 00 – 81 01 00 FF [23]. And the existence of a persistent object could be checked by invoking the tpm2_getcap command.

From manufacturing time on the TPM is uniquely identified by the Endorsement Primary Seed (EPS), which is created ad installed in the TPM by the TPM vendor. There can not be two TPM sharing the same Endorsement seed. In security and cryptographic operations we can rely on the the Endorsement key uniqueness in order to ensure authentication in critical operations and in communication between devices.

4.1.4 Attestation Key

The Attestation Key (AK) is a key with the following properties: it is a signing key and it is restricted. Moreover, also as a consequence, the Attestation Key must be non migratable. The restricted property means that the key can only sign a digest created by the TPM. The scope of this is to prevent forgery, i.e the signing of external data disguised as a valid and genuine attestation data. This mechanism ensures that the AK does not sign an arbitrary external data because in general the AK is used to sign data which reflect the TPM and the system state. In our specific case in remote attestation the AK is used to sign and to certify the measurements that represent the state of the system that the Verifier wants to attest.

In the documentation[25] it is explained how the process of signing works for a restricted signing key like the Attestation Key. Basically the TPM adds a special header to every message to be signed it produces: this value is a fixed value called TPM_GENERATED_VALUE. When the TPM has to sign an external data it checks the header of the data verifying that TPM_GENERATED_VALUE is not present. The digest operation also produces a ticket when the message that was digested did not start with the TPM_GENERATED_VALUE. The TPM_GENERATED_VALUE is equal to 0xff544347, and it differentiates data generated by the TPM from non-TPM data. At signing time the ticket associated to the message to be signed is provided in order to prove that the message was not an attempt of forgery.

While the Endorsement Key certification is simpler because it is the TPM vendor who provides the the Endorsement Key certificate, the Attestation Key certification requires some steps. Its certification is needed in situations, like in remote attestation, where there is a third party, that act as a Certificate Authority (CA), that has to provide a certificate for it. The TPM, in order to certify its Attestation

Key must provide an evidence that the key is a TPM-resident key. This kind of evidence may be provided by a previously created and already certified key. In our case we are almost forced to used the Endorsement Key to prove the evidence of key residence.

There are two distinctive ways to achieve AK certification. If the Endorsement Key is also a signing key it can directly certify that another object like the Attestation Key is resident on the TPM. The second way is used when the EK is not a signing key and it is called *activation of credential*. It involves a number of steps where the CA challenges the TPM sending to it a credential blob with a secret inside. The TPM will be able to unseal the secret only if it hosts both the EK and the AK for which the certification has been requested to the CA. The TPM can prove that the Attestation Key is a TPM-resident by using the TPM2_ActivateCredential() command. All the steps will be reported in another chapter more in detail.

4.1.5 Platform Configuration Registers

The Platform Configuration Registers (PCRs) are a set of specific memory register banks present in the TPM. They have some unique properties and represent a key component of the TPM. The PCRs are used to store hash values and their length depends on the hash algorithm used. The TCG PC Client Platform TPM Profile Specification defines the inclusion of at least one PCR bank with 24 registers. A complete TPM may include more banks and compliance with more hash algorithms: the hash algorithms allowed are SHA1, SHA256, SHA384 and SHA512. For each hash algorithm type a bank of 24 registers is provided inside the TPM.

The two operations allowed on PCRs are the *reset* and the *extend* operations. The *reset* operation simply set the target PCR register to zero. The *extend* operation is a updating of the target PCR by concatenating a hash value with the one already present in the PCR register. According to the documentation [25] the *extend* is defined as follow:

$$\mathrm{PCR}_{\mathrm{new}} := \mathrm{H}_{\mathrm{alg}} \left(\mathrm{PCR}_{\mathrm{old}} \parallel \mathit{digest}\right)$$

The *extend* operation may be used many consequent times on the same PCR register resulting in the computation of a cumulative hash, where the final value is the result of the consecutive extensions of all the preceding ones.

In the context of remote attestation and of integrity verification the hash value present in a PCR register may be a measurement representing the state of the system or the state of some data that has to be attested. Moreover, the TPM

allows the generation of policies, which act as a means of authentication for TPM operations, that are dependent to a specific PCR register value.

The way the PCRs are designed is very effective and secure against manipulations performed by an attacker. Lets consider a scenario in which the PCR value represents the state of the system to be attested. After the corruption of the system the PCR value would result storing a different untrusted value. At this point an attacker would like to force the the "system trusted value" into the PCR register in order to succeed the attestation process. To do this, the attacker should find another value whose hash with the current state of the PCR results in the trusted measurement. According to the properties of a secure hash this is unfeasible.

4.1.6 Policy Authorization

Policies are the way authorizations are managed inside the TPM. These policies also known as Extended Authorizations (EA) can be used to authorize actions of a TPM entity. With EA any kind of authorization is possible and may be used to enforce many different controls on the TPM before making the authorization succeed. Control that could be done, in order to manage the authorization of an action inside the TPM, could be:

- Requiring a certain value in a PCR register
- Requiring a certain password
- Requiring a specific value in a NV index

And there are many more other controls that could be done. They can be combined together in a logical statement by means of OR and AND operators resulting in a concatenation of controls and in a custom authorization policy. The authorization of an action is enabled by the TPM only if all the conditions specified by the policy associated to that action are fulfilled.

The authentication policy, when created, is represented by a unique value called *policy digest* or *authPolicy* and it may be associated to a TPM entity. Associating a policy digest to an entity means that from now on all the conditions and controls of the authentication policy must be satisfied in order to perform some action with that entity.

The policy authentication lifecycle reported on the documentation[8] is composed of these steps:

1. Creation of the policy digest
2. Creation of an entity using and associating the policy digest created before
3. Starting of a policy session
4. Fulfilling of the authentication controls required by the policy digest
5. If the previous step was successful, perform the intended actions

Lets analyse the steps of the authentication policy usage for the case in which the entity is a key object. In the step 2 the policy digest is used to for the key creation. In this phase a template structure is constructed and the policy digest is incorporated in it. Then this template is used for the actual key creation in order to be included into the key and so also the policy digest. At the key usage time the TPM is given a sequence of policy commands, like TPM2_PolicyPCR for policy PCR, that modify the digest of the policy session. After the execution of these commands the policy session used as an authorization session: if the digest accumulated in the policy session matches the policy digest of the entity then the command over the key is authorized [25].

The Extended Authorizations (EA) is an indispensable feature of the TPM in the remote attestation protocol. The authentication used is the policy PCR: which performs authentication checks on the value of a specific PCR to allow the execution of some actions over an entity, which in our case is a signing key. As will be also explained in the protocol steps specification the value stored in a PCR will represent, in remote attestation, the state of the system or application we are trying to attest. Starting from this consideration we may use the value of a PCR (state of the system) as an authentication parameter in the use of the Attestation Key. These concepts will be then explored more in depth afterwards during remote attestation by Proof explanation.

4.1.7 Activation of credentials

The activation of credential is a phase being part of the both the integrity verification protocol defined. This phase permits to ensure that the Attestation Key (for which a certificate is requested) and the Endorsement Key (for which the certificate has been provided by TPM manufacturer) are both resident in the TPM and that the Attestation Key is the child of the Endorsement Key.

In this phase the CA constructs a structure with a secret inside and encrypts it with the primary key public key (in our case is the public part of the Endorsement

Key). Only the TPM with the corresponding EK private key can recover the secret. If the secret is recovered successfully by the TPM it means that the AK (the one with the name provided as input of the tpm2_makecredential command) is resident inside the TPM and that it has been created by the TPM itself as a child of the EK. This is very important for the Orchestrator because it wants to be sure of the origin and trustworthiness of the Attestation Key.

Before the activation of credentials the Orchestrator could only know that the Endorsement Key is trusted and resident on the TPM of the VM thanks to the EK certificate issued by the TPM manufacturer. After this phase the Orchestrator can also know the origin and trustworthiness of the Attestation Key.

This section could be tricky and difficult. That is why all the steps that need to be performed are reported. All these following steps and messages are taken from the manual [8] and they are helpful to better understand how they allow the Orchestrator to be sure, from a security point of view, that the AK was created by the TPM of the target VM as a child of its EK. The steps taken from the manual represent the manual and reported here are the guidelines to a secure implementation of the credential activation.

All these steps are then implemented by means of the two commands tpm2_makecredential (called by the Orchestrator) and tpm2_activatecredential (called by the VM). These two command used together permits to execute this phase called activation of credential.

The activation is a way by means a credential provider can be assured of the key attributes it's certifying.

All the following steps are taken from the book called "A Practical Guide to TPM 2.0" [8] and are reported here.

The following happens at the credential provider:

1. The credential provider receives the Key's public area and a certificate for an Endorsement Key.

2. The credential provider walks the Endorsement Key certificate chain back to the issuer's root.

3. The credential provider examines the Key's public area and decides whether to issue a certificate, and what the certificate should say.

4. The requester may have tried to alter the Key's public area attributes. This attack won't be successful. See step 5 in the process that occurs at the TPM.

5. The provider generates a credential for the Key.

6. The provider generates a Secret that is used to protect the credential. Typically, this is a symmetric encryption key, but it can be a secret used to generate encryption and integrity keys. The format and use of this secret aren't mandated by the TCG.

7. The provider generates a 'Seed' to a key derivation function (KDF). If the Encryption Key is an RSA key, the Seed is simply a random number. If the Decryption Key is an elliptic curve cryptography (ECC) key, a more complex procedure using a Diffie-Hellman protocol is required.

8. This Seed is encrypted by the Encryption Key public key. It can later only be decrypted by the TPM.

9. The Seed is used in a TCG-specified KDF to generate a symmetric encryption key and an HMAC key. The symmetric key is used to encrypt the Secret, and the HMAC key provides integrity. Subtle but important is that the KDF also uses the key's Name.

10. The encrypted Secret and its integrity value are sent to the TPM in a credential blob. The encrypted Seed is sent as well.

At the end we obtain the following:

- A credential protected by a Secret
- A Secret encrypted by a key derived from a Seed and the key's Name
- A Seed encrypted by a TPM Encryption Key

The operations are summarized in the following picture:

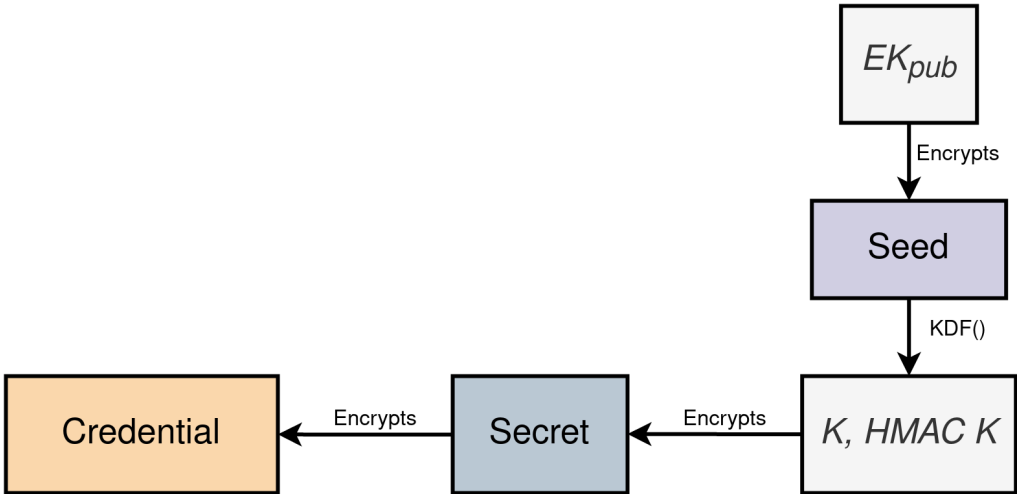


Figure 4.2. Verifier's steps to create the credential

Things that happen at the TPM:

1. The encrypted Seed is applied against the TPM Encryption Key, and the Seed is recovered. The Seed remains inside the TPM.

2. The TPM computes the loaded key's Name.

3. The Name and the Seed are combined using the same TCG KDF to produce a symmetric encryption key and an HMAC key.

4. The two keys are applied to the protected Secret, checking its integrity and decrypting it.

5. This is where an attack on the key's public area attributes is detected. If the attacker presents a key to the credential provider that is different from the key loaded in the TPM, the Name will differ, and thus the symmetric and HMAC keys will differ, and this step will fail.

6. The TPM returns the Secret.

The steps just listed are the ones that, according to the documentation, have to be performed to activate a credential.

Outside the TPM, the Secret is applied to the credential in some agreed upon way. This can be as simple as using the Secret as a symmetric decryption key to decrypt the credential[8]. This phase ensures that the credential provider that the credential can only be recovered if:

- The TPM has the private key associated with the Encryption Key certificate[8].
- The TPM has a key (the AK) identical to the one presented to the credential provider[8].

The steps executed by the TPM of the Prover are summarized in the following picture scheme:

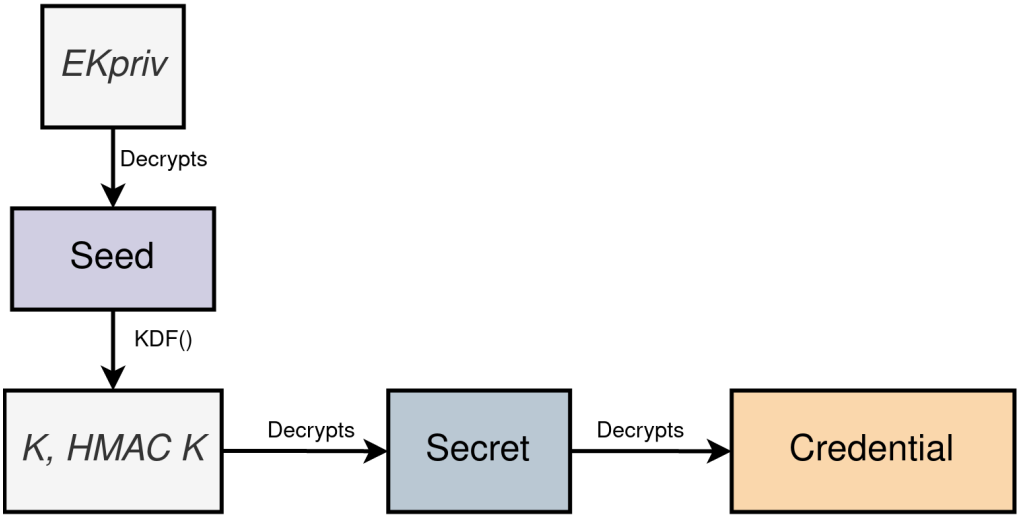


Figure 4.3. Prover's steps to disclose the secret from the credential blob

4.2 TPM Software Stack (TSS)

The TPM Software Stack (TSS) is a software stack used to drive and manage the TPM 2.0. It is a stack used to interact with the TPM and it is designed in order to hide the low level details of interfacing to the TPM to the application programmer. Designed as a software stack it has multiple layers and APIs at each layer: in this way allowing a programmer to intercept the stack at high level layer APIs or low layer APIs depending on the type of application. Any time an application needs

to interact with a TPM the TSS must be included so that a programmer can use its set of APIs to used the TPM functionalities.

In figure 4.4 TSS stack layers are showed. They go from the application layer to the TPM in order of abstraction: from the Feature API which is the most high level to the TPM driver which is the most low level.

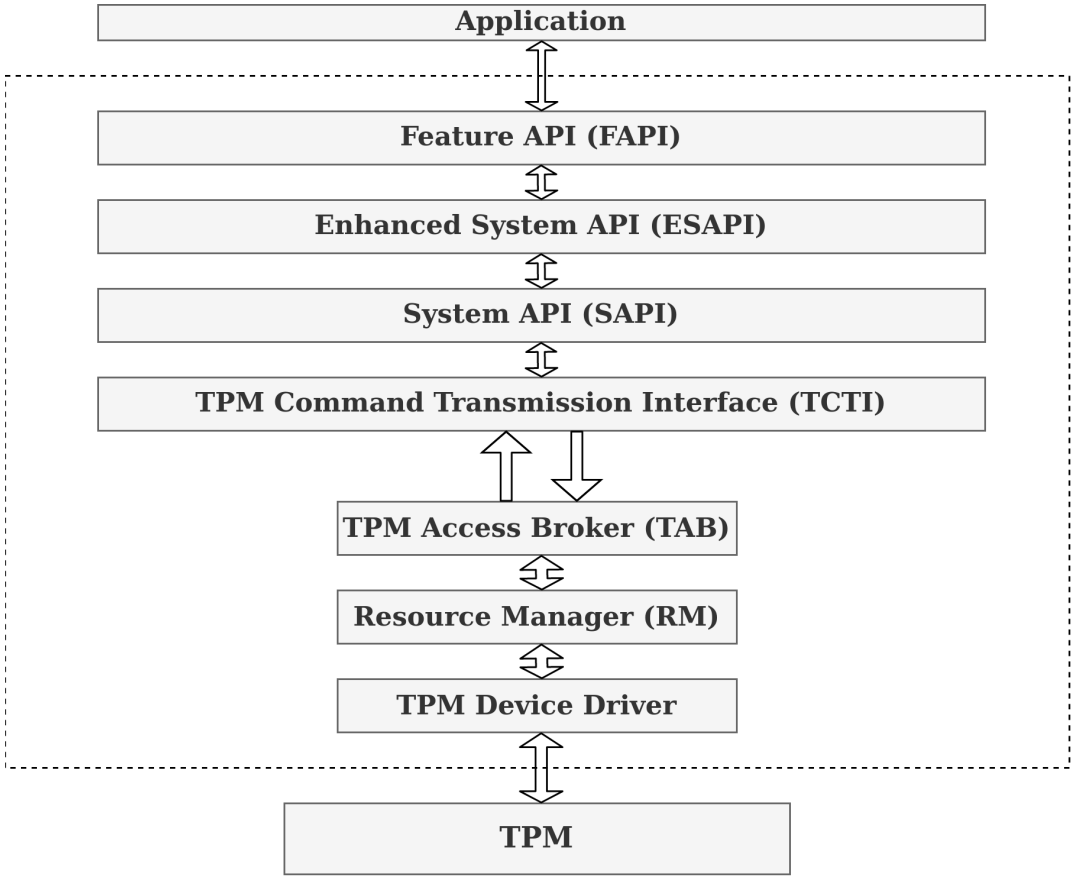


Figure 4.4. TSS stack

The TPM Device Driver is the OS driver that manages all the handshaking with the TPM and the reading and writing of data to the TPM.
The Resource Manager (RM) is responsible of managing resources, entities and object of the TPM swapping them in and out the TPM as needed, since the TPM has a limited number of resources. It is not a mandatory module but if not present it is responsibility of the upper layer to manage resources correctly and efficiently. The TAB manages multi-process access synchronization to the TPM.

The TPM Command Transmission Interface handles the communication with the lower layers of the stack. It permits to send and receive commands with the TPM. It is the central layer in the software stack and it is a communication bridge between the upper and the lower part.
The System API is the layer that provides a set of APIs to give access to all the functionalities of the TPM 2.0. It is meant for expert and more specific applications.
The Enhanced System API is the layer right above the System API and it is meant to add an additional abstraction in the access to the functionalities of the TPM. It provides support to various cryptographic operations in a simpler way with respect to SAPI.
The Feature API is the higher layer in therms of abstraction. It is used to allow applications to perform some TPM operations hiding completely the internal details. It is composed of a set of application APIs.

In addition to those layers there is also the MUAPI. The MUAPI performs the Marshaling of TPM commands into byte streams and Unmarshaling of the responses returned from the TPM.

All the aspects analysed up to now about the TSS have been defined in the documentation of the TSS by the Trusted Computing Group[14, 24]. TCG has defined all the layers, modules and the overall structure of the TPM software stack.

Following the TCG TSS specifications two different implementation of the TPM software stack have been proposed: the IBM TSS and the Intel TSS.

In this work it has been used the Intel TSS as software stack to interact with the TPM 2.0. Its implementation is different from the IBM TSS but they both rely on the same TCG TSS specifications. More specifically the commands and all the tests made to experiment the use of the TPM during the remote attestation protocol have been done by leveraging on tpm2-tools[20]. This is a set of command line tools, developed on top of the Intel TSS library, which permit to provide access to the TPM from the shell environment in linux. This is a simple way to interact with the TPM from the linux shell and it permits to concatenate the shell commands to test TPM functionalities.

The tpm2-tools code is open source and it can be inspected to understand how things work internally for each command. TPM commands implemented by tpm2-tools are almost all mapped one-to-one with TSS APIs of SAPI layer. By the way there is also a small percentage of tpm2-tools commands which don't have a direct pair in the list of TSS SAPI function but are the result of a combination of multiple TSS commands. The code may be extended and modified to implement some personalized functionalities upon the already present ones. However, unmodified

tpm2-tools commands have been used during this work: the commands that will be mentioned will refer to their original implementation.

The tpm2-tools commands also allow to explicitly specify the TCTI configuration to be used. As explained above the TCTI represent the Transmission Interface and so the mechanism to send and receive commands to and from the TPM. Through the `-T` option the TCTI can be specified for a tpm2-tools command. If the option is not specified, the default choice goes to the file `/dev/tpm0` which represent the hardware TPM present of the device. Otherwise we could explicitly specify the hardware TPM through `-T device:/dev/tpm0`.

In this work we are going to use a software implementation of the TPM. The software TPM works as a server active in localhost at a designed port. The TCTI must be configured, using the following option for each tpm2-tools command, to be able to send that command to the software TPM:

`-T mssim:host=localhost,port=`*`port_number`*

In general the default port number on which the virtual TPM is listening is the port number 2321 and the the default ip is 127.0.0.1, so localhost.

The other module that could be used to complete the TSS functionalities are the TPM2 access broker (TAB) and Resource Manager (RM). As mentioned from the documentation their use is not mandatory but they are very efficient and handy when deploying an application that interact with the TPM. For this purpose the tpm2-abrmd[19] could be used: it is a system daemon implementing the TPM2 access broker (TAB) and Resource Manager (RM). If tpm2-abrmd[19] is not used the management of TPM resources is delegated to the upper layers which are in charge of handling it explicitly.

4.3 Tracing and sampling techniques

4.3.1 Overview

This section introduces the two tracing techniques adopted in this work: the eBPF hook tracer and the Intel PT. They have been suggested into this document as valid and effective tracing techniques and have been used as a way to show two different types of tracing approaches. In fact they follow two totally different ways to trace the execution of a software program permitting to achieve two distinct levels of details and granularity of the logged information.

A separate analysis of their capabilities and characteristics have been conducted for each of them. In addition to that they have been proposed as part of the final remote attestation integrity verification solution, to show how effective they can be in guaranteeing correctness of protocol operations and phases and to show a practical use case in which they can be adopted.

Before talking about the way they work lets first of all introduce the concept of tracing. Software tracing means using logging techniques to record information about the execution of a program. The main reason why tracers are used is for debugging and monitoring purposes. They allow to debug and verify the operations taken by program by collecting data and information during its execution.

The main tracing techniques used nowadays can be roughly classified among software and hardware solutions. The software tracing implementations are generally based on *instrumentation*. Instrumentation means adding some additional code to an application program in order to monitor its behaviour. This kind of techniques may be performed either statically or dynamically. They basically add additional monitoring code in strategic points of the code and, every time these points are traversed, collect useful information on the program execution state. They do not require additional hardware and are lightweight. However, they add additional code and so increment the program execution time and require the recompilation of code any time that some additional logging instructions are added to the program.

On the other side there are the hardware tracing solutions. They do not add additional code to the software program to be traced but they make use of hardware features of the processor. In general this kind of approach is more lightweight with respect to software solutions, resulting in less impact on program execution performances. The software solutions in general are quite precise but permit to obtain relatively high level information. On the contrary the hardware solutions can potentially reach the lowest tracing level possible arriving at recording the single assembly instructions.

In this document the Intel PT and the eBPF hook are presented. The Intel PT is a hardware technique while the eBPF is a software one.

The eBPF software program exploits hooks to trace the execution of a target running module. Hooks are pieces of code able to intercept specific events and function calls in the code invoking a specific C code in response to it in order to collect execution information. Its implementation makes use again of the instrumentation mechanism: the function call, or point to be traced, is instrumented with the code in charge of logging execution information. It makes use of a Virtual Machine to inject code into the kernel without requiring kernel recompilation

at any time an event is attached.

These two techniques are proposed into this work because they allow to achieve different levels of granularity of logged data. In such a way they complete themselves and combined together they can achieve higher precision and reliability in the correctness of program executed.

Control Flow Attestation

Another important concept used into this work is Control Flow Attestation [10]. Control Flow Attestation is a scheme that allows a Verifier to attest the control flow path of a software program running on a remote device, without requiring the Verifier to have the source code. The goal of this solution is to attest the correctness of the execution of the software running on the remote device during run-time. A normal integrity attestation solution just detects attacks that tamper and modify the software code running on the remote machine by verifying its integrity. The Control Flow Attestation, instead, is able to detect attacks like buffer overflow attacks, or ROPs (Return Oriented Programming attacks) which try to divert the execution flow path of the program without actually modifying the binaries. However this kind of technique does not cover data-oriented attacks, attacks that just corrupt data variable executing a valid but unauthorized path in the code.

Lets analyse in brief how Control Flow Attestation works and which are the components involved. At th beginning there is an initial set up phase where the Verifier generates the static Control Flow Graph (CFG) of the software program to be attested through static analysis. Static analysis comprises a set of techniques used to statically and syntactically analyse the code of a program in order to generate its graph representation containing all the logical path that could be traversed and taken during its execution. Once the Verifier has generated the CFG of the remote device's software program it computes a measurement (hash value) for each of the valid execution path and stores them into a DB.

At this point, every time the software program to be attested is executed on the remote device, its execution is traced by means of a, relatively low level, tracer. The scope of the tracer, in this case, is to record the control flow graph of the program execution during run-time. Here a tracing technique to be used against the program execution must be selected between the ones available. The more the tracer is detailed and low level in the log output information it can produce and the more the Control Flow Attestation will be precise and effective in finding possible anomalies.

After the extraction of the execution flow path, its measurement is computed and sent to the Verifier to be checked against the list of valid reference values stored into the database.

In this work a Control Flow Attestation scheme has been used to verify the correctness of the binary extraction procedure and of the eBPF tracer program. In this work we make use of the Intel PT, which is an Intel hardware tracing feature able to reconstruct the execution flow graph at the assembly level reaching a high level of precision introducing only a low performance overhead.

An introduction on both eBPF and Intel PT technologies is reported in this chapter.

4.3.2 Extended Berkeley Packet Filter (eBPF)

The extended Berkeley Packet Filter (eBPF) is an evolution of the original Berkeley Packet Filter (BPF). Initially the BPF was designed for capturing and filtering packets from the network. The way it works leverages on a virtual machine and this makes it a very interesting tool. Basically the packet filtering program runs on a register-based virtual machine and this is a very interesting aspect that permits to run user-supplied programs inside of the kernel. BPF was a first good trial in this direction: the design and the idea were very interesting even though it was not using in an efficient way the potentials of the new architectures that, at that time, were moving towards 64-bit registers.

One step ahead was made with the development of the extended BPF, which was created following the footsteps and the design of the original BPF. The eBPF has been built having in mind the modern architectures and their hardware ISA. So now the eBPF is a virtual machine environment which uses 64 bit RISC instruction set capable of running just-in-time compiled eBPF programs inside the Linux kernel, reaching very good performances. While its initial purpose was the network packet filtering, nowadays eBPF is widely used to run user-space code inside the kernel by means of a virtual machine. It is a powerful tool to inspect and collect information from the kernel.

The eBPF works like a hook because it is triggered whenever some specific events in the kernel occur. The eBPF program can be "attached" to some specific code paths in the kernel, and whenever one of these kernel sections are traversed the eBPF program is executed. This specific feature has been used in this project allowing us to trace and inspect the execution of specific TSS commands from the kernel. Events that the eBPF program could be attached to could be system calls

or other functions of the kernel. The information obtained, gathered each time the target event occur and the hook is activated, might be additionally filtered in order to select only meaningful and useful ones.

Since the eBPF program is able to access kernel data structures it can also be used to debug and run performance analysis of the kernel. It is worth remarking that the eBPF allows a developer to execute the eBPF program and to change it many times without having to recompile the kernel.

In this work bcc has been used to perform all eBPF tests. bcc stands for BPF Compiler Collection and it is a toolkit useful for creating kernel tracing and manipulation programs, including several already built-in tools. It makes use of eBPF and is very useful to make the writing of BPF programs easier.

The structure of the eBPF programs written with bcc is composed of two parts: the C section and the front-end python section. The python part is the one that runs at the user level and it is useful to catch, manipulate and show the information that are gathered from the kernel side and also to manage the kernel functions to be hooked. On the other side the C section is the one composed of all the functions that are going to be called whenever their attached kernel section is executed and that are going to collect information from the kernel.

The bcc reference guide has been used to understand the C and python BPF functions that could be used to write eBPF programs.

4.3.3 Intel Processor Tracing (Intel PT)

The Intel Processor Trace (Intel PT) is a feature, or better an extension, added to the recent Intel processor architecture to collect information about software execution. It can trace the execution of a software program through dedicated hardware, added to the standard Intel architecture, in a very efficient way, causing only a minimal additional overhead on software execution. It belongs to the family of hardware tracing techniques.

The Intel PT can be used to execute *control flow tracing* of a software running on the system. During the tracing a variety of data packets are generated: this is the way Intel PT organizes the data it collects. There are different packet types and each of them if designed to store a different kind of information about the program execution and its control flow. Furthermore packets can record timing information making Intel PT also useful for performance debugging of applications. It has also filtering capabilities useful to trace only relevant and useful information and control capabilities to add timings and processor states for debugging purposes.

Lets have a look at the packets generated by the Intel PT during execution of the program tracing in order to understand what kind of information can be obtained. Packets may be divided in two categories: the ones which collect information about program execution and the ones which gather control flow information.

Packets information are taken directly from the Software Developer manual to precisely understand their purpose[17]. Packets about basic information on program execution are:

- Packet Stream Boundary (PSB): they are packets generated every 4K trace packets bytes and are useful as boundaries to facilitate decoder's work. It is the first packet the decoder should encounter.
- Paging Information Packet (PIP): it records modification on the CR3 register. This information, along with the CR3 value of the process taken from the OS, permits to associate linear addresses to the correct application.
- Time-Stamp Counter (TSC): it is used to keep track of timing information.
- Core Bus Ratio (CBR): it stores core-bus clock ratio.
- Overflow (OVF) packets: they are recorded whenever the processor experiences an internal buffer overflow.

On the other side there are the control flow packets; they are:

- Taken Not-Taken (TNT) packets:it is generated every time a conditional branch is encountered in the code. It records "T" for taken or "NT" for untaken depending on the direction of the branch during the execution of the program.
- Target IP (TIP): they store the target Instruction Pointer of some kind of event or instruction that causes a jump in the the code. It was noticed that this kind of packets not present when there is a jump to a deferred IP of the same section code (the same linear address space) but only when there are jumps that go to another address section. This happens when the program traced has some attached libraries: the software code is resident in an address section while the commands of the external library in another one. So when jumping between there two sections a TIP packet is generated.
- Flow Update Packets (FUP): they record the source IP for asynchronous events and all the cases where the source address can not be determined from the software binary.

- MODE packets: these packets are useful to help the decoder in understanding the trace log by providing some processor execution information.

We mentioned some times above about the decoder, it is an important element in Intel PT software execution tracing. The **decoder** is a software tool used to process all the data packets generated by the Intel PT during program execution. It is the real interpreter of all the binary data produced. Actually it works by processing both the Intel PT packets and the program binaries in order to reconstruct the exact execution trace. This whole process is also showed in figure 4.5. It has to be remarked that the decoder is a post-processing tool that can be used at a different time than the tracing operation.

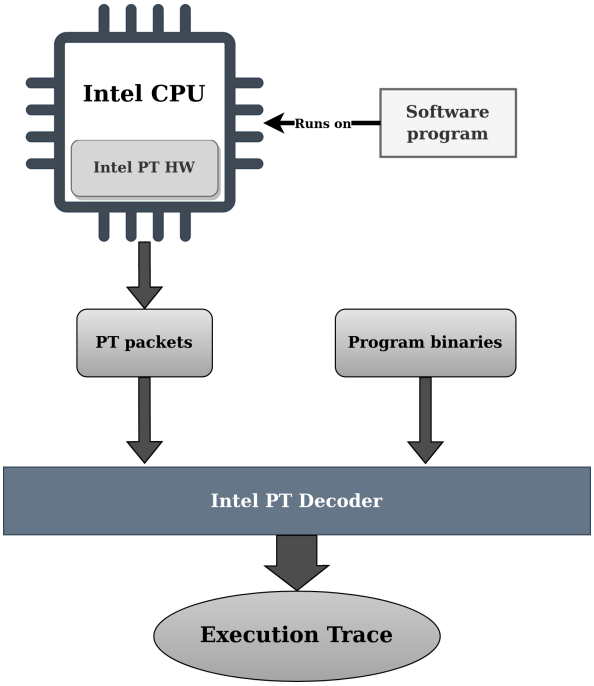


Figure 4.5. Intel PT decoding

In the context of this work the packets that we are going to focus our attention on are the ones that record control flow information. In particular the TNT packets will allow us to understand the path actually followed by the program in execution and furthermore to reconstruct the exact sequence of commands. It is a useful tool for remote attestation even though it has some limitations.

There are many implementations of software utilizing the Intel PT capabilities. The chosen one to conduct the experiments is **perf**: which is a performance

analysing tool already present in the linux kernel. It is a complete tool offering a rich set of commands and many functionalities. The most significant ones are:

- record: for executing and profile a program creating perf.data file
- report: read perf.data file and display the profile
- script: read perf.data file and display the trace output
- stat: for executing a program gathering performance counter statistics

Chapter 5

Remote attestation protocol architecture and components

5.1 Overview

In this chapter it is presented the overall remote attestation protocol architecture that has been designed and that has been used as reference for the entire work. With remote attestation protocol architecture we mean the high level description of the entire remote attestation process and of all its components, as well as the way they interact with each other and why. After defining each of the components and their role in the remote attestation architecture solution, in the next chapter, we will dive into each of them describing how they actually work and how they have been implemented. In this chapter some of the nomenclature that will be used in the following sections of this document is presented .

5.2 Phases and components

The system model described up to now shows which is the basic remote attestation integrity verification scheme and which are its main involved actors. Lets now have a closer look at the specific components involved in the entire process and their role. All the components and modules involved are interconnected to each other and depend from each other in composing the final architecture.

The whole procedure may be divided in two big process phases:

1. The **binary extraction process** phase
2. The **integrity verification process** phase

The components and modules involved in the first process are:

- The loaded binary extractor
- The Intel Processor Tracing to trace loaded binary extractor's execution

On the other hand, the components involved in the second integrity verification process are:

- The eBPF tracer to record TSS commands
- The software program implementing the integrity verification process
- The Intel Processor Tracing to trace eBPF tracer's execution

In both these two main section processes the actors involved are still the Virtual Machines and the Orchestrator, having as core trusted module the TPM resident on the remote Prover to be attested. The majority of the components just described are resident on the Virtual Machine's system.

5.3 Binary extraction process phase

Now lets analyse the two processes separately and more in depth to better understand the role of the mentioned components and the way they interact with each other.

The first phase of the remote attestation process regards the loaded binary extraction. It is used to extract the binary of the software to be attested when loaded into the main memory. It is loaded as soon as the software program is launched and its execution is started. The loaded binary extracted during this phase will be the data whose integrity will be attested through the integrity verification phase. Its integrity ensures the correctness and trustworthiness of the software running on the remote device.

The figure 5.1 shows the main components involved in the remote attestation architecture proposed and highlighting all the interactions that take place during the first binary extraction phase.

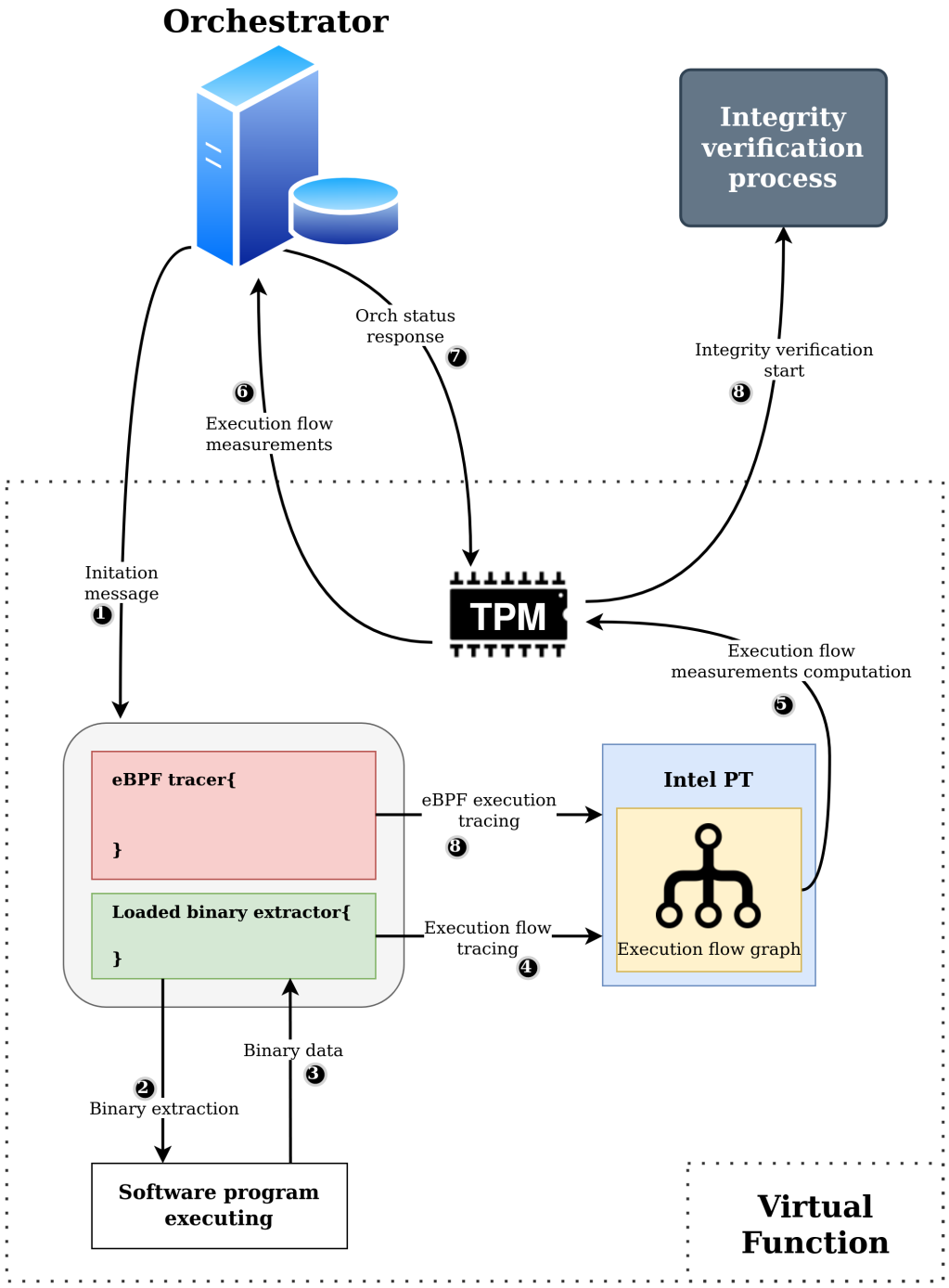


Figure 5.1. Binary extraction phase and main components of remote attestation architecture

As it can be seen by figure 5.1 the binary extraction process starts when the Orchestrator decides to attest the VM of the remote device by sending a message to it. This message is used to notify the Prover to start the whole attestation process. The Prover now knows that it has to start the loaded binary extraction phase. The loaded binary extractor is executed to extract the loaded binary data from the main memory of the operating system.

During the whole execution of the binary extractor the Intel PT is running in background with the purpose of tracing its execution. The Intel PT is used to extract and produce a thorough low level control flow graph of the binary extractor execution process. In this way the Intel PT produces an execution flow comprising the exact sequence of commands executed by the binary extractor. The tracing data obtained is useful to check if the loaded binary extraction operation was carried out in a correct and safe way following the exact expected flow of commands.

The execution flow graph obtained by the Intel PT represents the operations performed by the extractor during binaries reading. Its correctness will ensure to the Orchestrator that the reading of the binaries has been conducted in a secure and correct way without any deviation from its normal flow. Then the execution flow's measurement (i.e. hash function over the execution flow data) are computed through the TPM and are sent to the Orchestrator to be checked. The execution flow measurement is an hash string representing the summary of the entire control flow data to be checked.

A way to securely send the measurement data to the Orchestrator, ensuring its integrity and authentication, must be used. As already discussed in the introduction to the TPM, the Attestation Key associated to it can be used to sign the measurement data to guaranteeing its authenticity and integrity when sent from the VM to the Orchestrator. These kind of mechanisms will be then explained more in details during the Integrity verification process description in which the TPM of the VM will request a public key certificate to the Orchestrator in order to register its Attestation Key used to sign the produced attestation data ensuring authentication and integrity. In all this process the Orchestrator has to act also as a Certification Authority managing and issuing key certificates. This role may also be assumed by a separated relying party, but in this solution it has been considered as part of the Orchestrator.

Once the Orchestrator has received the measurement data of the execution flow of the binary extractor in a secure and reliable way from the VM it checks whether the measurement value is as expected or not. The assumption here is that the Orchestrator must already own the list of valid reference values, against which the

execution flow's measurement value is checked, prior to the remote attestation protocol execution. The Orchestrator, before starting the actual remote attestation protocol execution, has to perform a set up phase of offline pre-processing operations:

1. Generate the Control Flow Graph (CFG) of the software program through static analysis: this means using a series of static analysis techniques and tools in order to extract the Control Flow Graph of the software program to be attested. This CFG is different from the previously mentioned execution flow of the program. The execution flow graph is extracted by the Intel PT during run-time and represents the sequence of really executed instructions. On the other hand the CFG is a graph representing all the possible path and directions that the program could potentially take; it is the representation of the software program in a graph form and it is obtained in a static way through static analysis.

2. Determine all the valid paths, of the CFG of the software program, that could be taken. Then the Orchestrator computes the measurements of each of them through an hash function. All these obtained measurements must be kept by the Orchestrator into a DB. They are the set of all the valid reference values, each of them representing a correct execution flow of the binary extractor program.

After receiving the execution flow's hash measurement from the Prover, the Orchestrator checks whether it is preset in the set of the valid measurements or not. If the value is in the reference values set this means that the extraction of the loaded binary happened correctly and the Orchestrator sends a response to the Prover, which will then start the execution of the actual integrity verification process phase.

If the loaded binary extraction results to be incorrect the Orchestrator can decide whether to repeat again the attestation process sending a new start message to the remote device or to consider the remote system as in an untrusted state.

In this first binary extraction phase the assumption that has been made is that the loaded extractor is not trusted, or better that it could be potentially tampered by a malicious actor, and so the Intel PT tracing is used to verify and attest its execution correctness. This means just adding an additional level of security, moving security issues from the binary extractor to the program that implements the execution flow reconstruction and its subsequent measurement computation. The assumption here is that the part related to Intel PT and the execution flow graph extraction and measurements computation is considered trusted. However

considering the Intel PT capabilities it is able to obtain a very precise and thorough control flow graph at assembly level of the execution of the binary extractor, representing in this way a very reliable and highly secure technique.

5.4 Integrity verification process phase

After the loaded binary extraction phase the actual integrity verification process is performed. This second phase is used by the remote device to prove to the Orchestrator the evidence of its system state. As we know the Orchestrator may request the integrity verification at any time during the execution of the software program to be attested at Prover's side.

This section is the core section of the entire work because it is the central process by which the VM is able, not only to provide the software measurements to the Orchestrator but more importantly to do it in a reliable and secure way. The design of this part has been thought by considering the basic principle of remote attestation but also the specific characteristics and peculiar properties of the TPM module. Therefore, the proposed integrity verification solutions make sense only if considered and deployed with a TPM module because they are tightly bound to TPM's architecture and properties.

Two different solution have been proposed and tested to implement the remote attestation integrity verification protocol by means of the TPM:

- **Integrity verification by Quote**
- **Integrity verification by Proof**

Since these two solutions constitute the central part of whole the remote attestation process, characterizing its main mechanisms, they give the name to it.

The attestation by Quote reflects a more general and common scheme used in the context of remote attestation. Its can perhaps be deployed by means of other trust anchor rather than the TPM because it follows the guidelines of basic approach used as definition of remote attestation. On the contrary, attestation by Proof is a totally new remote attestation solution leveraging TPM specific properties and mechanisms. Although they both permit to reach the same goal offering the same properties from a security point of view, they have characteristics and mechanisms slightly different.

As an introduction to these two protocols lets describe their main basic principles and characteristics.

Attestation by Quote protocol: the basic principle behind the attestation by Quote is that the Prover has to produce a quote to be sent to the Orchestrator in order to provide an evidence of its system state and in order to be attested. A quote is basically a digest over a set of PCR values. The digest is computed as the hash of the concatenation of all of the digest values of the selected PCRs [27]. The quote is then signed with an Attestation Key used to generate a reliable evidence of the system's state. The Attestation Key is resident into the TPM of the Virtual Machine and its signature proves the authenticity of the measurement value. After receiving the quote from the VM the Orchestrator can check the Prover's state from it.

Attestation by Proof protocol: the attestation by Proof is based on the adoption of an Attestation Key on the TPM of the VM created by using a policy digest provided by the Orchestrator, used to then sign the attestation data (a nonce). The Orchestrator, by means of its TPM, can create a policy digest linked to a specific PCR value representing the correct state of the remote device. Forcing the Prover to create an Attestation key with the created policy digest means let it adopt a key that can only be used when the policy is satisfied, so only when the PCR value is correct and therefore only if the system is in the correct state. The attestation process starts when the Orchestrator sends a nonce to the remote device. Now the Prover has to sign that nonce by means of the Attestation Key previously created. The TPM will allow the AK to be used to sign correctly the nonce only if the PCR and the system state are correct. After sending the signed nonce back, the Orchestrator can check it verifying the remote devices state.

As it can be noticed there are some differences and peculiarities among the two protocols. In both of them, at the end of the whole process, the Orchestrator is able to know which is the current state of the VM. The integrity verification process terminates when the Orchestrator makes the required checks over the attestation data sent by the Prover and finds out which is the Prover's state, saving it into a DB to ensure accountability of attestation procedures. At this point the Orchestrator, based on the outcome of the integrity verification, may decide whether to not do anything, keeping the VM normally running, or to take some actions in order to address a possible problem if the remote device results to be in a malicious and untrusted state. The actions that the Orchestrator might take depend on the specific application.

Before talking about the technical details of the implementation of the two solutions it is important to understand their main differences and in which context their characteristics are more suitable.
The differences between them, divided by some properties, are reported here.

- Verification place and context of work (**local** vs **remote**):
  - Quote: the Quote protocol follows a *remote* scheme in the sense that the actual check of the quote produced by the Prover, and the consequent declaration of its state, happens inside the Orchestrator. It is in the Orchestrator the place where the actual integrity verification check occurs. This solution is suitable only for a Orchestrator to VM attestation scheme.
  - Proof: the Proof protocol follows a *local* scheme in the sense that the actual check of the policy digest authentication and the consequent nonce signing happens inside the VM. It is in the VM where the actual integrity verification check occur, i.e. the policy digest check at signing time, because if that step fails it means that the VM system is compromised and the whole attestation process aborts. This kind of solution also works in a VM to VM attestation scheme.
- Multiple run-time attestation process mechanisms:
  - Quote: the Quote protocol requires the PCR value to be extended each time a new integrity verification is performed under Orchestrator's request. This means that first of all the Orchestrator has to keep track of the number of times the PCR has been extended (which is equivalent to the number of times the integrity verification has been requested); we are going to discuss how it was addressed later in the implementation chapter. And secondly it means that the integrity verification can be requested a multiple number of times and at any moment by the Orchestrator during run-time, without needing to change the Attestation Key of the Virtual Machine each time. In fact the VM can use the same AK for producing the quotes for its entire lifetime.
  - Proof: in the Proof protocol the remote attestation can be performed a multiple number of subsequent times, as well as in the Quote protocol, but it can be done in two different approaches:
    - The first one is based on extending the PCR value each time the integrity verification is performed following the same approach of the Quote protocol. However, in the Proof protocol if the PCR

value to be checked changes a new AK must be created at each time. The new AK has to be created by using a new policy digest, bound to the new PCR value, constructed at each new attestation process by the Orchestrator. This scheme is a little bit worse in terms of performance because the creation of a new AK at each attestation time adds more overhead to the protocol execution. Taking into consideration the time of the whole process and the time added for the AK creation, and also depending on the specific application, a performance and feasibility analysis of this solution should be done before its deployment.

* The second one is based on resetting the PCR value at every time a new integrity verification is performed, before computing the measurements. Therefore, at each time the Orchestrator requests a new integrity check the VM can reset the value of the PCR, compute the measurements of the software binary again and then extend the PCR. In this way there are not subsequent extensions that make the PCR value changing at any time, but the extension is performed only once after the PCR reset. The PCR value should remain always the same and the AK can remain unchanged at any time.

During the whole integrity verification phase, independently from which one of the two protocols is used, an eBPF tracer is used to trace and record all the commands and data exchanged with the TPM. Some eBPF hooks are leveraged to intercept the transit of TSS commands in order to record and log them and their parameters. This is an additional level of security added to the integrity verification protocol. The program that implements the integrity verification process, which communicates with the TPM by means of a sequence of TSS commands, is considered untrusted. The role of the eBPF hooks is to verify that the sequence of TSS commands executed, as well as their parameters, are correct and in the right order. This additional security mechanism enhances the integrity verification process reliability and correctness.

The eBPF will allow us to gather TSS commands information like the binary data sent and received to the TPM, as well as the type of commands and their latency. However the eBPF tracer execution, used to verify the correctness of the TSS commands, could be diverted and tampered by an adversary in order to fake the sequence of TSS commands or their parameters. For this reason the Intel PT is used as a tracing technique to follow and attest the execution flow of the eBPF hooks program. It will be an additional highly secure mechanism to assure that the eBPF program execution is correct and so also the TSS commands that

implement the integrity verification phase.

The motivation behind the choice of using the Intel PT to trace and attest the eBPF tracer is that the Intel PT is a more precise tracing technique able to reach a lower level of granularity with respect of the eBPF. A eBPF program can extract more high level information about process execution wile the Intel PT can be used to obtain the exact assembly execution flow graph. These two kinds of techniques complete each other in this way, tracing at two different levels of granularity and producing a complete set of log of records. In this way the integrity verification process may be considered highly secure providing a verifiable evidence of its correctness in all its phases to the Orchestrator.

The assumption here is that the Intel PT monitoring and tracing technique as well as the control flow extraction is considered trusted.

Chapter 6

Remote attestation protocol components implementation

6.1 Overview

In this chapter we are going to discuss about the integrity verification protocols and all its steps in detail. After the presentation of all the messages that are need to be exchanged between the Orchestrator and the Prover the actual commands used to implement and test the effectiveness of the two protocols and their timings are reported.

This is the notation that will be used across the protocol specification and diagrams:

Symbol	Description
$\mathcal{O}\mathsf{rc}$	The Orchestrator
$\mathcal{VM}$	The Virtual Machine running on the remote device
$\mathcal{TPM}$	The software TPM of the remote Virtual Machine
AK	Attestation Key
EK	Endorsement Key
KH	Key Handle of a loaded key
$\mathcal{C}_{\text{blob}}$	Credential blob
s_{ref}	Reference secret generated from the Orchestrator
s_{cred}	Secret disclosed from the credential blob
h_{meas}	Measurement of the loaded binaries
h_{ref}	Measurement reference value computed by the Orchestrator
$\mathcal{I}$	PCR register indexes
$\mathcal{Q}_{\text{attest}}$	Quote over a specific set of PCRs
$\mathcal{Q}_{\text{attest}}$	Signature of the quote data
$\mathcal{P}_{\text{digest}}$	Policy digest
$\mathcal{I}_{\text{pcrs}}$	PCR register indexes
$\mathcal{T}_{\text{digest}}$	Public key template
T	Creation ticket
d_{templ}	Key template data
h_{creat}	Hash of the creation data
d_{ticket}	Ticket data: proof that the digest was generated by the TPM
$\mathcal{S}_{\text{nonce}}$	Signature of the nonce

Table 6.1. Notation used

It is also worth noticing that the certificate management operations in general are managed by a relying third party that acts as Certification Authority. The Certification Authority performs many actions like the issuing of a certificate for a public key or the storing of certificate data along with their expiration date. In these two protocols it has been considered, for simplicity, that the CA runs inside the Orchestrator. Depending on the implementation the CA could run on another machine different from the one of the Orchestrator. The same consideration can be done for the DB: the DBMS was considered to be on the same machine of the

Orchestrator, but depending on the implementation choice it could also be running on a different remote machine. The type of DBMS and the technology on which it relies on are not mentioned, it is up to the developer choosing the more suitable one.

Both attestation by Quote and by Proof protocols are composed of two main phases:

1. Attestation Key creation phase: it is the phase where a new AK, associated to the EK of the TPM, is created and then certified by the Orchestrator in order to be used in the integrity check phase.

2. Integrity check phase: this represents the the actual attestation phase in which the Prover produces the attestation data for the Orchestrator and in which the Orchestrator checks it to get to know the remote device's state. In this phase the AK previously created is used in the attestation process to produce the attestation data and to ensure its authentication.

Lets analyse all the single steps in detail in the following sections.

6.2 Attestation by Quote

6.2.1 Creation of AK

1. Loading of the Endorsement Key:

   In this step the Endorsement Key of the TPM, that was generated and injected by the TPM vendor at manufacturing time, is loaded. By loading the EK its key handle is obtained. The EK used is a 2048 RSA key.

2. Creation of the AK:

   The VM creates an Attestation Key as a child of the EK, previously loaded in the first step. The command tpm2_createak is used for the creation of the AK returning the context of the AK, the public part of AK and the name of the AK. In this solution a RSA 2048 key has been created.

In order to generate an Attestation Key as a non-migratable restricted signing key the attributes `fixedtpm`, `fixedparent`, `restricted` and `sign` have to be SET by the tpm2_createak command.

The AK has been created in PEM format. This choice is necessary because later on the tpm2_checkquote command will need a public key in PEM format as parameter.

3. Get of AK certificate:

   The VM asks the Orchestrator (CA) to issue a certificate for the AK it generated. The VM, along with the request, sends the EK certificate, the name of the AK and the public part of the AK created in the previous step. The EK certificate has been retrieved using the command tpm2_getekcertificate. It retrieves the Endorsement key Certificate for the TPM EK from the TPM manufacturer's endorsement certificate hosting server[21]. Its argument specifies the URL address for the EK certificate portal[21]. Moreover this command could also be used in an offline mode.

   The retrieved EK certificate is an X.509 certificate encoded in ASN.1 format containing the public part of the EK.

4. Check of the EK certificate:

   The Orchestrator, after receiving the EK certificate, the name and public part of the AK created by the VM, verifies the signature of the EK certificate and the one of all certificates in the chain up to the Root CA Certificate. In the case of the TPM, the EK certificate is issued by the TPM Manufacturer which ensures that the Endorsement Key is resident on a specific TPM. This process consists of checking whether the EK Certificate was issued by a trusted TPM Manufacturer. This operations are performed by the CA, and as it was mentioned before it's role is played by the Orchestrator.

5. Check if the EK of the VM is in the DB:

   The Orchestrator (CA) checks whether the EK certificate of the VM requesting an AK certificate is associated to a key that was previously added to the DB by the Network Administrator. In this way the Orchestrator (CA), after checking the validity of the EK certificate, also checks if the certificate is actually associated to the key of the remote device. This operation prevents any VM from requesting and obtaining an AK certificate.

6. Activation of a credential in order to check that the AK is resident on the TPM and that it is the child of a certified EK:

   The Orchestrator, using tpm2_makecredential command, generates a credential blob, which is created by using a TPM public key to protect a secret that is used to encrypt the attestation key certificate[21]. The Orchestrator has to select a secret to be used in the command. This credential blob is then sent to the VM.

   The VM will then use the credential blob as input for the tpm2_activatecredential command. This command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object. This guarantees the registrar that the attestation key belongs to the TPM with a qualified parent key in the TPM[21], and so ensuring that the Attestation Key, for which the VM has requested the certificate, is owned by the TPM and that it is also a child of the Endorsement Key whose certificate was previously provided by the VM.

   If the tpm2_activatecredential execution is successful it produces as output the secret that the Orchestrator used as parameter of the tpm2_makecredential command.

7. The VM returns back to the Orchestrator the disclosed secret from the credential blob:

   Outside the TPM, the secret could be applied to the credential in some agreed upon way [8]. This can be as simple as using the secret as a symmetric decryption key to decrypt the credential [8].

   In this solution the VM, after disclosing the secret from the credential blob sent by the Orchestrator, just returns the secret back to the Orchestrator. This solution is a simplified version of the one proposed in the documentation but provides the proof that the VM was able to disclose the secret and so that the AK and the EK are resident on the TPM of the VM, where the EK is the parent of the AK for which the certificate has been requested.

8. The Orchestrator checks the value of the secret:

   The Orchestrator, after receiving the secret value from the VM, checks if it is equal to the value of the secret that it had previously used as input for the tpm2_makecredential command. If the two values are the

same it means that the EK and the AK are both resident on the TPM of the VM.

Otherwise if this check fails the Orchestrator can stop all the certification process not issuing the requested AK certificate.

9. The Orchestrator(CA) issues the AK certificate and sends it to the VM:

   The AK certificate is created by the Orchestrator(CA) by simply signing the public part of the AK created by the VM with its asymmetric key. It then could create a data structure to put together the public portion of the AK and the signature just created.

   This is the basic way to create a certificate for the AK of the VM. Of course this could be subject to some extensions and addition of attributes and data about the Attestation Key to then be signed by the Orchestrator(CA) in order to create the final certificate structure. An example of additional data could be the key's name, which moreover can completely identify the key because its is a digest computed including the public key and also its attributes [8].

   After the creation of the AK certificate the Orchestrator sends it to the VM.

10. Store of the AK certificate into the DB:

    The Orchestrator stores the issued AK certificate in the DB. The list of the issued AK certificates is used to keep track of the Attestation Keys of the remote devices. It will then be useful during the actual integrity verification part.

6.2.2 Creation of AK sequence diagram in attestation by Quote

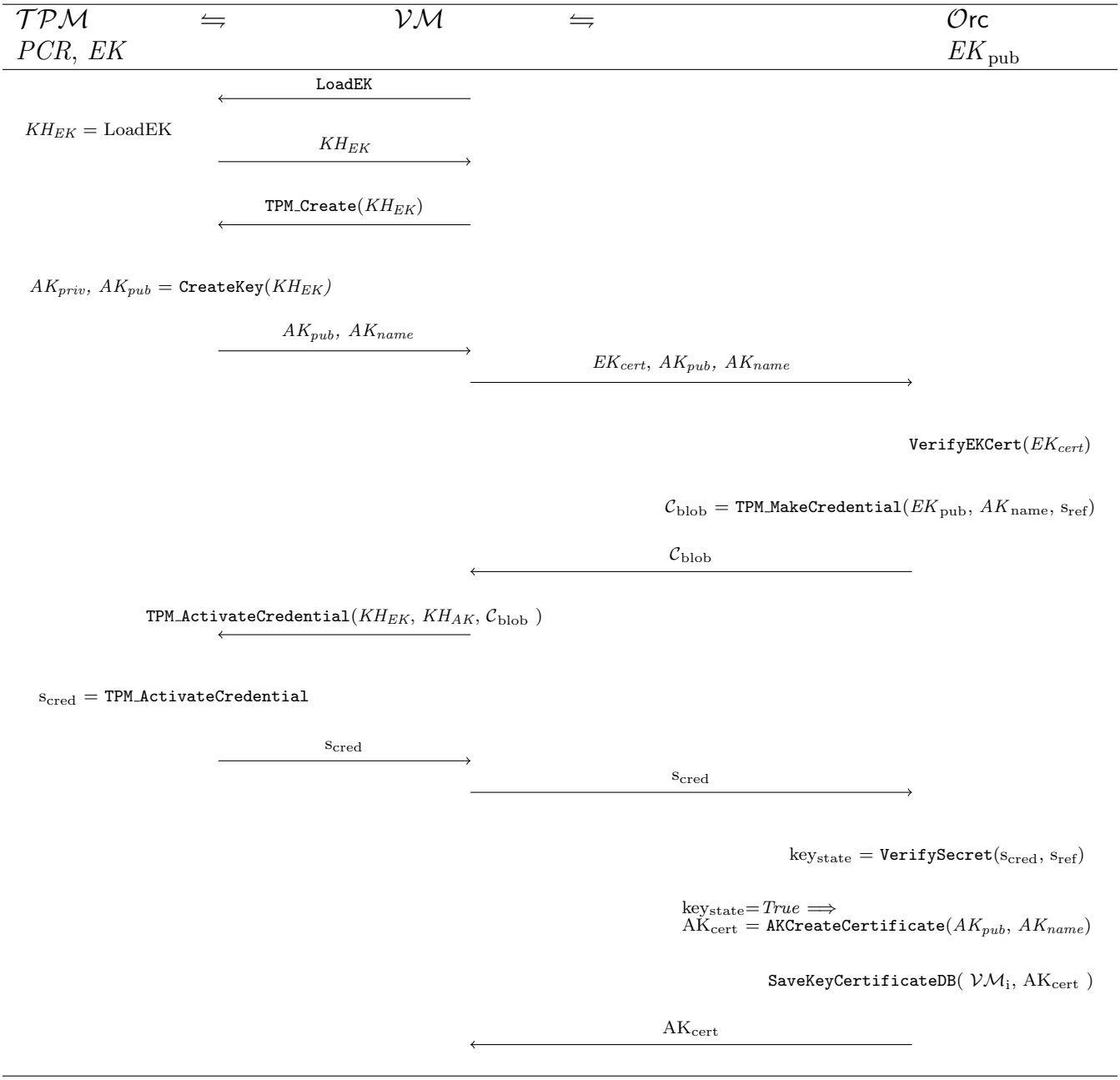


Figure 6.1. Attestation Key creation in Attestation by Quote

6.2.3 Integrity check

1. The Orchestrator sends the integrity verification request message to the VM:

   The Orchestrator simply sends a message to the VM as a request for an integrity verification. In this way the Orchestrator can trigger the starting of the actual integrity check process.

   This message could be of any type. In the actual implementation of the protocol it could be realized in many ways. It is up to the developer deciding how this message should be. It is just a trigger message with the purpose of making the integrity verification process start.

   Together with the message a random nonce is sent to the Prover to be used during the production of the quote. It is needed to avoid reply attacks and to give freshness to the attestation data produced.

2. The VM computes the necessary system measurements and extends the PCR bank:

   The VM computes all the necessary measurements of the loaded software binaries on the Virtual Machine. A measurement consists basically on computing a digest on data of which we want to attest the integrity (computation of the hash). These data could be like the firmware, the OS, or , as in our case, the loaded binary of an application running on the system. In our scope the data attested is the loaded binary of a software program running. The loaded binary extractor read the binaries and then its digest is computed.

   Once the VM has computed the final measurement it uses that value to extend a PCR register.

   Prior to this step the VM should have depicted a PCR bank to be used for the entire integrity verification process. At the beginning of the integrity verification process the selected PCR register must be reset.

3. The VM produces the TPM quote using the AK:

   The VM uses the tpm2_quote command which produces a quote and a signature for a given list of PCRs in a given algorithm/banks[21]. The command produces also a signature over the quote data. This signature must be created by using the AK previously certified and the random nonce received from the Orchestrator. Using the AK ensures that the

quote was produced by the TPM of the VM to be attested and the use of the nonce ensure freshness over the attestation data produced.

The quote represents the current state of the PCR, which reflects the current state of the system because it is extended with the computed measurements over the data to be attested.

After that the VM produces the quote and its signature they are sent to the Orchestrator.

4. Check if the AK certificate of the VM is present in the DB:

   The Orchestrator checks if there is a AK certificate for the VM sending the quote. If the AK certificate is present inside the DB the VM has already a valid Attestation Key and a valid certificate for it.

   If the AK certificate is not present into the DB it could mean that the VM has not performed the first phase for requesting a certificate for its AK, maybe that the AK certification phase was not successful or maybe that the AK certificate is expired. If the AK certificate is not present in the DB the Orchestrator should stop the integrity verification process.

5. Check if the quote has been signed with the AK associated to the VM:

   If the TPM of the VM has a valid AK associated to an AK certificate (previous step) the Orchestrator checks whether the quote was actually signed by that Attestation Key making use of the previously sent random nonce. In the verification process the command tpm2_checkquote is used. It uses the public portion of the provided key to validate a quote generated by a TPM; This will validate the signature against the quote message[21]. The provided key to validate the quote of course is the public part of the AK of the VM and it must be in PEM format.

   This is a crucial part in the integrity check phase because if the quote signature verification fails it means that the quote data was not produced by the TPM of the VM or maybe that it was modified by some attacker. In both cases the Orchestrator should take some actions: it could start again another integrity verification process to be sure about the VM state, before considering it untrusted. After some assessments, if the verification of the signature over the quote fails, the Orchestrator must take a decision over the VM state.

6. Verify the measurements and the TPM quote value:

The Orchestrator verifies the measurements and TPM quote sent by the VM. To validate an attestation quote, a remote Orchestrator can use a PCR to recalculate the digest value[25]. After the recomputation of the integrity measurements and after recalculating the quote the Orchestrator could compare it with the quote value provided by the VM: if they are equal the integrity verification succeeds and the VM is in a trusted and good state.

Another way to check the quote is by checking directly the PCRs value. This can be done because the tpm2_checkquote returns to the standard output the value of the PCRs on which it computed the quote. It is also returned the PCR bank number used and so also the length and type of hash algorithm used to compute the quote. The Orchestrator could read this PCR value and compare it with the expected one. If they match it means that the VM is in a trusted state.

This is the final and most important step of the integrity verification. It is the actual step where the integrity measurements are checked to verify VM's state

If this step fails it means that the VM is not in a trusted state and maybe its software has been tampered by an attacker. The Orchestrator has to perform some actions to address a possible VM system attack. The action of the Orchestrator depends on the context and on the specific application in which this protocol is applied.

7. The Orchestrator stores the integrity verification result:

   The result of the VM integrity verification process is stored by the Orchestrator in the DB. The Orchestrator should keep a log of all the integrity verification outcomes in order to maintain an the history the VMs states.

   The log is very important because it is a way by means the Orchestrator can keep track of the VMs state. The history data could also be useful, in the future, for forensics reasons.

An additional consideration must be done about the step 6 of the integrity check. This step is regarding the actual check of the measurement value provided by the VM. In this Quote scheme the integrity verification can be requested from the Orchestrator a multiple number of subsequent times. Each time a new integrity process is started a new measurement of the VM system is computed, and subsequently the PCR register is extended with the new measurement every time. So at

the end, after many subsequent integrity verification processes, the value stored in the PCR register is equal to the result of many subsequent extensions performed by using the values of the measurement computed at each different integrity check procedure. Therefore, the PCR value is changing each time a new attestation is performed.

The Orchestrator, in order to correctly check the quote, has to keep track of the number of times the extension operation has been performed over the PCR register and so the number of times it has started a new VM integrity verification. For this reason a counter is needed at the Orchestrator side to keep track of the number of past performed integrity checks. When the Orchestrator has to verify the quote value sent by the Prover it has to take into account the number of cumulative extensions to be executed before getting the right value to be matched with the quote.

6.2.4 Attestation by Quote sequence diagram

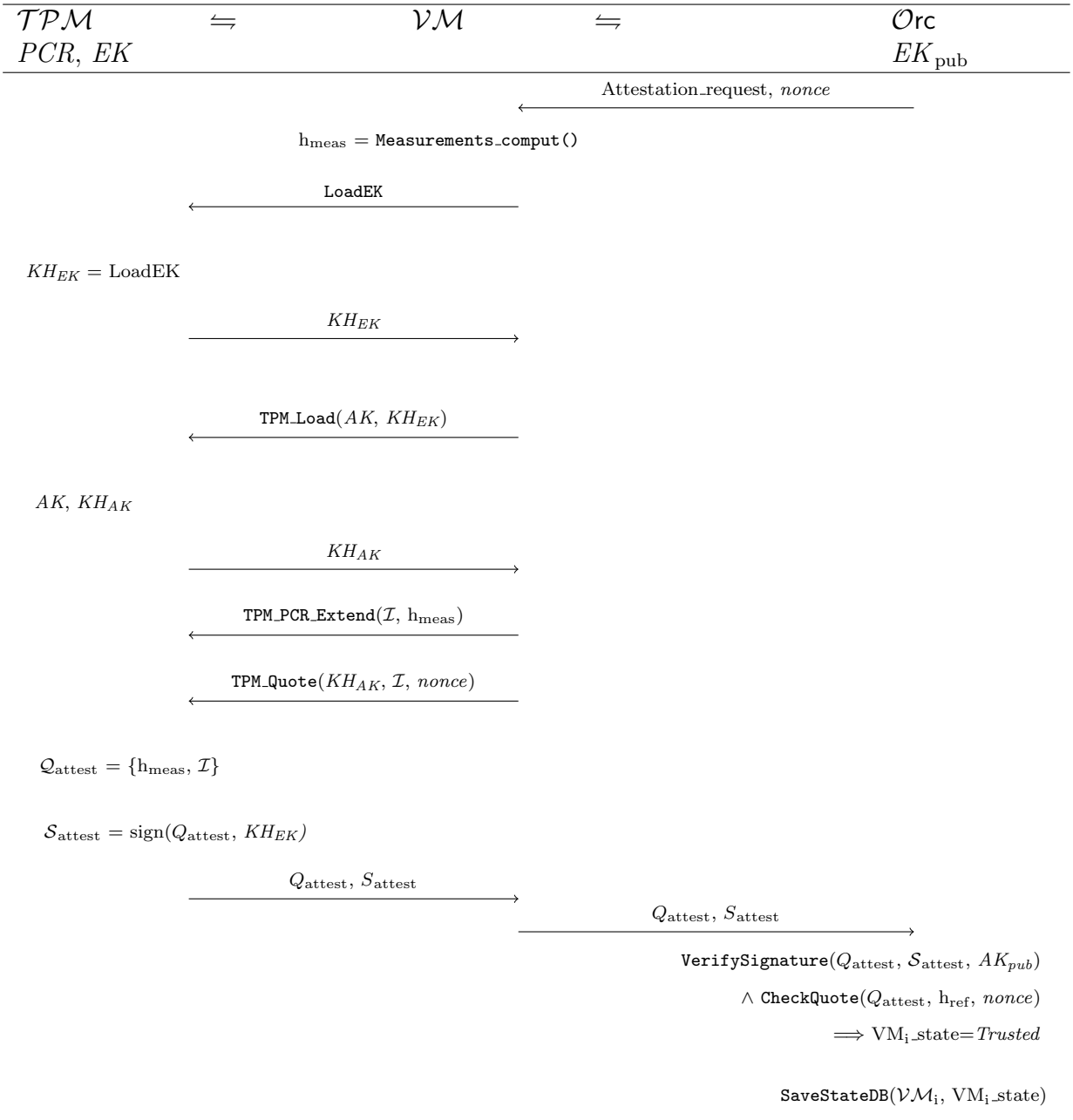


Figure 6.2. Attestation by Quote

6.3 Attestation by Proof

6.3.1 Creation of AK

1. The Orchestrator creates a policy digest associated to the correct expected value of the PCR:

   The Orchestrator first of all starts a new policy session by calling tpm2_startauthsession command. Then it calls tpm2_policypcr command declaring an expected PCR value in order to create a policy digest: it is called PCR policy. The PCR policy is a policy digest which is bound to specific PCRs value.

   The policy can then be used to create a key. A key created with the use of a PCR policy is bound to a certain PCR value. This means that the key can be used until the PCR remains in a state compliant with the policy digest. If the key is used for any operation while the PCR is not in the right state, so storing an incorrect value, the TPM will rise an error about the policy check.

2. The Orchestrator sends the policy digest to the VM asking the VM to create a new AK with it:

   The Orchestrator simply send the policy digest created in the previous step and sends it to the VM asking it to create a new Attestation Key by using the provided PCR policy. The implementation of the message used to convey the policy digest is not specified here and it is up to the developer choosing how this message should be.

3. The VM creates the AK using the policy digest received by the Orchestrator:

   The VM, after receiving the policy digest, uses it for the creation of an Attestation Key. However first of all the VM has to load the Endorsement Key generated by the TPM manufacturer in order to get its key handle. The EK used is a 2048 RSA key. After that the EK has been loaded on the TPM it can be used as a parent for the creation of the AK.

   The VM has to use the policy digest, received from the Orchestrator, for the creation of the AK. The policy digest is simply included into the key object at creation time [25]. The first step is to create a template structure for the object of type TPM2B_PUBLIC with the policy digest

inside. Then the caller provides this template at AK creation time for its association with the policy digest.

Different from the Quote protocol, where the tpm2_createak command was used for the AK creation, in the Proof protocol the command used is tpm2_create. This choice is necessary because the tpm2_createak does not allow the use of a policy digest during the key creation. On the other hand the tpm2_create command allows the use of more options and personalization over the creation key parameters and attributes.

In order to generate an Attestation Key that is non-migratable restricted signing key the attributes `fixedtpm`, `fixedparent`, `restricted` and `sign` have been SET through the options of the tpm2_create command. The AK created is an RSA 2048 key.

After that the key has been created it has to be loaded on the TPM. Another thing to remember is that, after loading the key on the TPM, it must be made persistent by means of the command tpm2_evictcontrol. If it is not made persistent on the TPM there will be problems afterwards during the next commands execution. The Attestation Key must be made persistent in order to be able to make all the protocol working.

4. Get of AK certificate:

   The VM asks the Orchestrator(CA) to issue a certificate for the AK it generated. The VM, along with the request, sends the EK certificate, which contains the public part of the EK, and the name and public part of the Attestation Key for which it wants to request the certificate. The EK certificate is retrieved by using the tpm2_getekcertificate which requires as argument the URL address for the EK certificate portal. It can also be used in an offline mode.

5. Check of the EK certificate:

   The Orchestrator, after receiving the EK certificate, the name and public part of the AK created by the VM, verifies the signature of the EK certificate and the one of all the certificates in the chain up to the Root CA Certificate. The EK certificate is issued by the TPM Manufacturer who ensures that the Endorsement Key is resident on the selected TPM. This step is used checking whether the EK Certificate was issued by a trusted TPM Manufacturer. This certificates operations are performed by the CA, which is a reliable third party whose role, as it was mentioned before, is also played by the Orchestrator in this solution.

6. Check if the EK of the VM is in the DB:

   The Orchestrator (CA) checks whether the EK certificate of the VM requesting an AK certificate is associated to a key that was previously added to the DB by the Network Administrator. In this way the Orchestrator (CA), after checking the validity of the EK certificate, also checks if the certificate is actually associated to the key of the remote device whose integrity we want to attest. This prevents any VM from requesting and obtaining an AK certificate.

7. Activation of a credential in order to check that the AK is resident on the TPM and that it is the child of a certified EK:

   This activation credential phase is the same as the Quote one.

   The Orchestrator, using tpm2_makecredential command, generates a credential blob, which is created by using a TPM public key to protect a secret that is used to encrypt the attestation key certificate[21]. The Orchestrator has to select a secret to be used in the command. This credential blob is then sent to the VM.

   The VM will then use the credential blob as input for the tpm2_activatecredential command. This command enables the association of a credential with an object in a way that ensures that the TPM has validated the parameters of the credentialed object. In an attestation scheme this guarantees the registrar that the attestation key belongs to the TPM with a qualified parent key in the TPM[21], and so that the Attestation Key, for which the VM has requested the certificate, is owned by the TPM and that it is also a child of the Endorsement Key whose certificate was previously provided by the VM.

   If the tpm2_activatecredential execution is successful it produces as output the secret that the Orchestrator used in the tpm2_makecredential command.

8. The VM returns back to the Orchestrator the secret disclosed from the credential blob:

   Outside of the TPM, the disclosed secret could be applied to the credential in some agreed upon way [8]. This can be as simple as using the secret as a symmetric decryption key to decrypt the credential [8].

   In this solution the VM, after disclosing the secret from the credential blob sent by the Orchestrator, just returns the secret back to the

Orchestrator. This solution is a simplified version of the one proposed in the documentation but provides the proof that the VM was able to disclose the secret and so that the AK and the EK are resident on the TPM of the VM, where the EK is the parent of the AK for which the certificate has been requested.

9. The Orchestrator checks the value of the secret:

   The Orchestrator, after receiving the secret value from the VM, checks if it is equal to the value of the secret that it had previously used as input for the tpm2_makecredential command. If the two values are the same it means that the EK and the AK are both resident on the TPM of the VM.

   Otherwise if this check fails the Orchestrator can stop all the certification process not issuing the requested AK certificate.

10. The VM creates the attestation data for the AK:

    The VM creates the attestation data of the AK using the tpm2_certifycreation command. This command attests the association between a loaded public area (in our case the AK) and the provided hash of the creation data[21]. The creation data and the creation ticket are needed to be used as input of the command and they are produced during the key object creation.

    As a result of tpm2_certifycreation execution we obtain the *attestation data* and its signature. Since the Endorsement Key is only a decryption key and not a signing key it can not be used to sign the attestation data. Therefore it has been decided to use the AK, signing in this way its own attestation data. This can be done because the Orchestrator, after the activation of credential phase, already knows that the AK is resident on the TPM of the VM and that it is the child of the EK for which it has a valid certificate.

    The attestation data contains the "magic" number, which is the proof that the AK was created by the TPM, and also contains the "attested" field containing the "objectName" and the "creationHash".

    The attestation data represent a proof that the AK was created by the TPM itself and a link between the AK and its creation data.

11. The VM sends the attestation data and the template of the AK to the Orchestrator:

The VM sends the *attestation data*, created in the previous step, and the *template data*, obtained during the AK creation, to the Orchestrator. Sending these two data to the Orchestrator is needed for the purpose of making the Orchestrator check if the AK was created in a correct way.

12. The Orchestrator verifies that the attestation data was signed by the AK of the VM:

    The Orchestrator verifies the signature of the attestation data with the public part of the AK. This step ensure that the attestation data was produced by the TPM of the target VM and not by any TPM.

13. The Orchestrator checks from the template data and attestation data that the AK was correctly created:

    The Orchestrator now checks whether the AK was created in a good way.

    First of all it checks the "magic" field in the attestation data to be sure that the AK was created from a TPM. The "magic" field should always be equal to TPM_GENERATED_VALUE (i.e. "magic" value equal to 0xff544347). Then it checks the "objectName" and the "creationHash" fields in the attestation data. The name of a key is computed as the digest over the public part of the key and its attributes so it completely identifies the key.

    Then the Orchestrator could also additionally check the template data, which is of type TPM2B_PUBLIC. It contains the attributes used for the key creation: this permits to check whether the key was created as a non-migratable restricted signing key. The template contains also the value "authPolicy" for the authentication policy only when the policy digest is used during key creation: this could be also another important check to do because it permits to know if the policy digest was really used for the AK creation.

    If all the checks in this step are successful the Orchestrator can be sure that the AK creation phase was correct and it can proceed to issue the AK certificate.

14. The Orchestrator (CA) issues the AK certificate and sends it to the VM:

    The AK certificate is created by the Orchestrator (CA) by simply signing the public part of the AK created by the VM with its asymmetric

key. It then could create a data structure to put together the public portion of the AK and the signature just created.

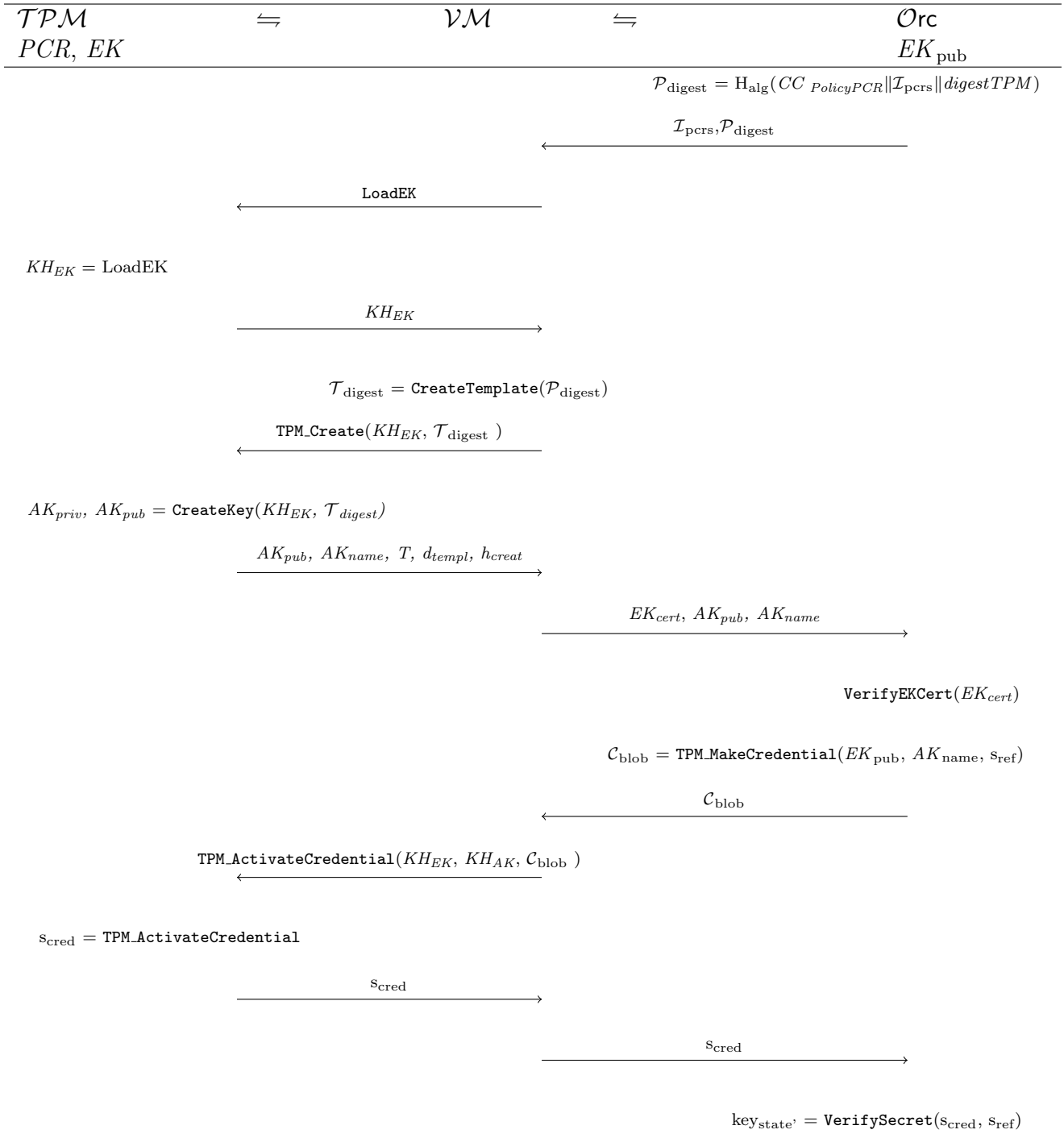
This is the basic way to create a certificate for the AK of the VM. Of course this could be subject to some extensions and addition of attributes and data about the Attestation Key to then be signed by the Orchestrator (CA) in order to create the final certificate structure. An example of additional data could be the key's name, which moreover can completely identify the key because its is a digest computed including the public key and also its attributes [8].

After the creation of the AK certificate the Orchestrator sends it to the VM.

15. Store of the AK certificate into the DB:

    The Orchestrator stores the issued AK certificate into the DB. The list of the issued AK certificates is used to keep track of the Attestation Keys of the remote devices. It will then be useful during the actual integrity verification part.

6.3.2 Creation of AK sequence diagram in attestation by Proof



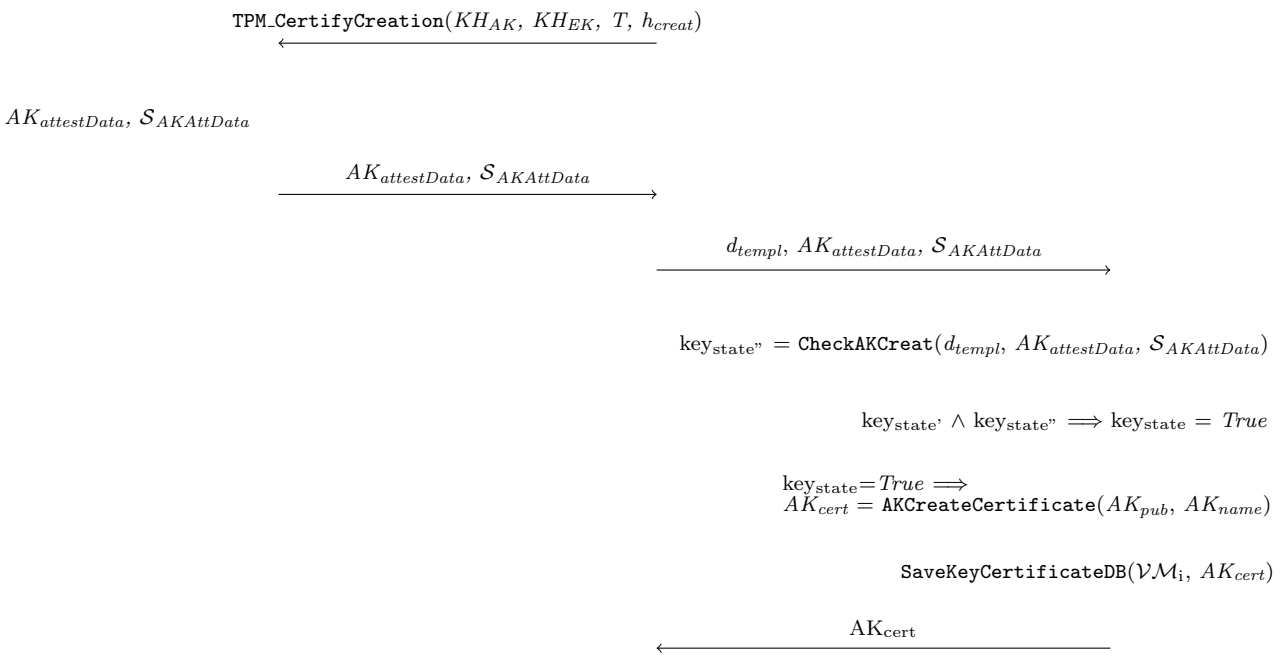


Figure 6.3. Attestation Key creation in Attestation by Proof

6.3.3 Integrity check

1. The Orchestrator sends a nonce to the VM:

   The Orchestrator creates a random nonce and sends it to the VM by means of a message. This is the first message that triggers the starting of the real integrity check phase.

2. The VM signs the nonce using the previously created AK:

   Now that the AK has been certified by the Orchestrator it can be used in the integrity verification phase. The VM uses the AK to sign the nonce received from the Orchestrator.

   This is the most important step of the Proof integrity verification protocol. The AK has been created by the VM using the policy digest provided by the Orchestrator. This means that the use of the Attestation Key is conditioned by the state of the PCR. If the PCR is in the correct state, the one declared in the PCR policy, the VM can use the key otherwise the use of the key will raise a TPM error.

   In our case the VM is able to sign the random nonce using the AK only if the PCR is in a good state. If the VM is not able to sign the nonce with the AK it means that the PCR is not in a good state and so the system has been modified or tampered.

   If the VM is not able to sign the nonce with the AK the integrity verification protocol stops and the VM system is considered in an untrusted state.

   The VM uses the tmp2_sign command to sign the nonce. Since the AK is a restricted key this command needs a ticket parameter to be able to work correctly. The ticket must be the hash value of the data that we want to sign with the AK: in this case the ticket is the hash value of the nonce. This ticket is produced by TPM2_Hash when the message that was digested did not start with TPM_GENERATED_VALUE[26]. This ticket is used to indicate that a digest of external data is safe to sign using a restricted signing key[25]. A restricted signing key may only sign a digest that was produced by the TPM[25]. If the digest was produced from externally provided data, there needs to be an indication that the data did not start with the same first octets as are used for data that is generated within the TPM[25].

3. The VM sends back the signed nonce to the Orchestrator:

The VM sends the signed nonce back to the Orchestrator. The signed nonce, if correctly signed, represent the proof of VM system integrity.

4. The Orchestrator verifies the signature:

   The Orchestrator verifies that the nonce was signed by the AK of the TPM. To do this first of all the tpm2_loadexternal command is used to load the public part of the AK into the TPM of the Orchestrator. And then the tpm2_verifysignature command is used to actually verify the signed nonce.

   If the signature succeeds it means that the nonce was correctly signed by the TPM of the VM and it proves that the VM system is in a trusted state.

5. The Orchestrator stores the result of the integrity verification check:

   The result of the VM integrity verification process is stored by the Orchestrator in the DB. The Orchestrator should keep a log of all the integrity verification outcomes in order to maintain the history the VMs state.

   The log is very important because it is a way the Orchestrator can keep track of the VMs state and it could also be useful in the future for forensics reasons.

6.3.4 Attestation by Proof sequence diagram

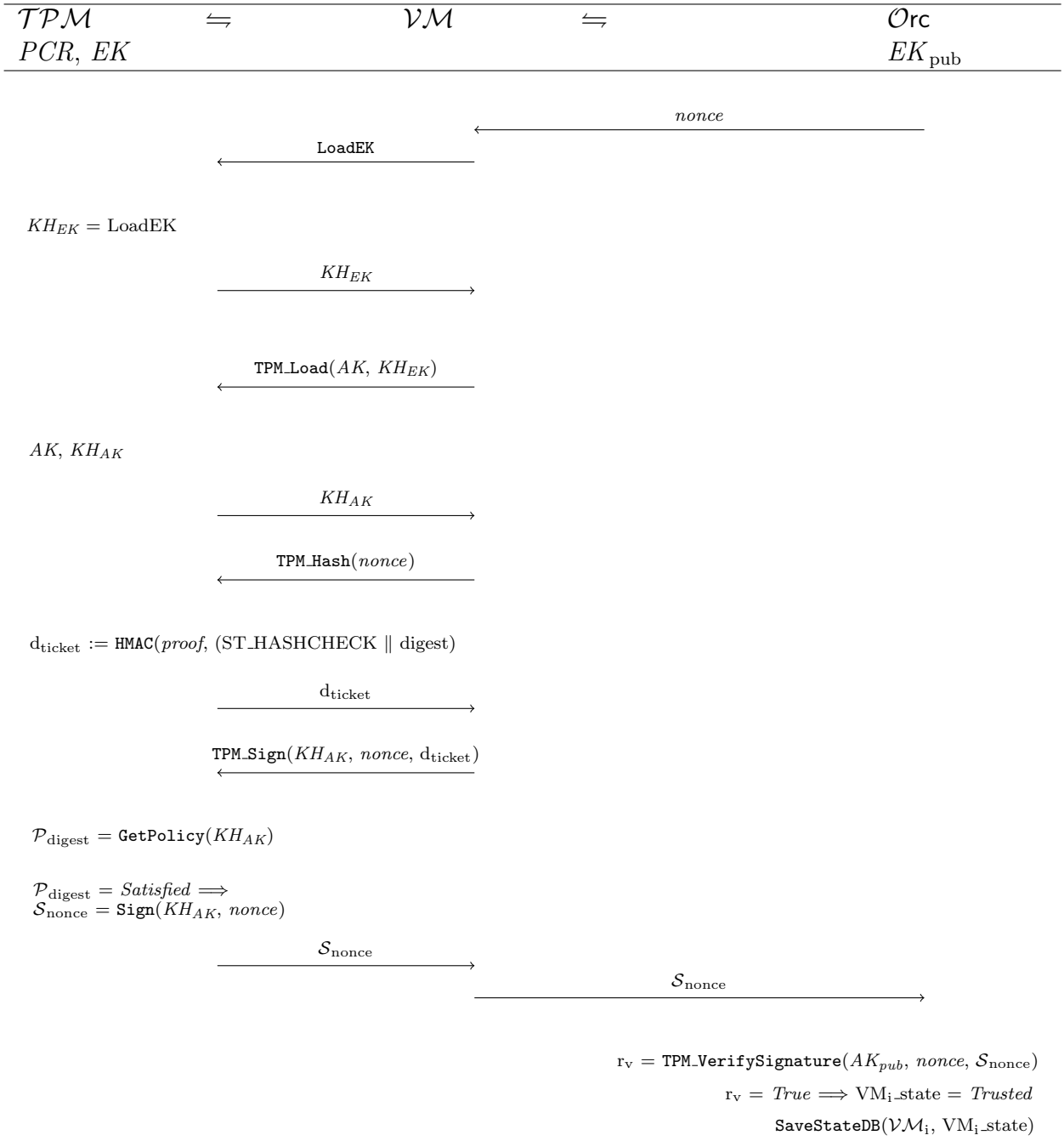


Figure 6.4. Attestation by Proof

6.4 Integrity verification commands timings

6.4.1 Timing measurements approaches

Part of the analysis conducted on the integrity verification protocol is regarding the time needed for commands execution. It allows us to understand how much time a single command takes to be executed and also to estimate the time required for the entire attestation process to be executed. Having a rough idea of the time necessary to attest the remote device can be necessary for the evaluation of the feasibility of such a solution and for the estimation of the overhead brought by the deploying of such attestation protocol. Knowing the total time required for the whole process is also necessary to make a threat modeling evaluation.

Different approaches have been used to measure the time required by the tpm2-tools commands to be executed. The tables 6.2, 6.3 show the timings that have been taken. As can be seen, timings have been measured both for Quote and for Proof integrity verification protocol processes.

Three different approaches have been used to measure the time required for each command's execution. The three approaches are:

- The `time` bash command: the time command is a common linux tool used to perform timing measurements of a bash command. It provides the real time, the user time and the system time measures of a command's execution. For each tpm2-tools command the execution time has been taken as the sum of the user time (CPU time spent in user-mode code) and the system time (CPU time spent in the kernel).

- The TSS_Execute interface: TSS_Execute is an interface belonging to the TSS library. It is contained in the file called Tss2_Sys_Execute.c. It is called every time a new TSS command is invoked and every time a TSS command terminates its execution. The function accessed at TSS command's calling is `Tss2_Sys_ExecuteAsync` while the function invoked at TSS command's termination is `Tss2_Sys_ExecuteFinish`. Taking the absolute value of the difference between the time at which the `Tss2_Sys_ExecuteAsync` is invoked and the time at which the `Tss2_Sys_ExecuteFinish` function is executed permits to obtain the time measurement of the command. It is worth noticing that the time measured include also the marshalling and unmarshalling of the executed command. The advantage of this kind of time computation is that it provides the exact time spent inside the TPM for the execution of the

TSS command. Discarding and not considering the execution time required by operations outside of the TPM.

- The eBPF tracing time: this time measurement has been taken by means of the eBPF tracer that has been used to log TSS commands and data exchanged with the TPM. Exploiting the eBPF program used capabilities to understand when a TSS commands is sent to the TPM and when its response is returned back from the TPM permits to measure TSS timings. Computing the absolute value of the difference between the time at which a TSS command is traced by the eBPF program and the time at which the response is caught make us obtain the time measurement. The time value obtained in this way represents the time measured from the eBPF tracer point of view. It is the time required by the eBPF hook program to trace the total execution of a TSS commands and with a good approximation can represent the time required by the command to be executed.

Down below in the tables 6.2, 6.3 all the timings obtained are reported both for Quote and Proof integrity verification protocols.

6.4.2 Attestation by Quote timings

Timings of Quote commands:

TPM 2.0 Command	TPM 2.0 Timings (sys+usr)	TPM 2.0 Timings (TSS_Execute)	TPM 2.0 Timings (tpm_hook)
Quote AK creation	**0.09s**	**0.2488s**	**0.0843s**
tpm2_createek	0.009s	0.0820s	0.0207s
tpm2_createak(rsa2048)	0.011s	0.1608s	0.0333s
tpm2_makecredential	0.046s	/	/
tpm2_startauthsession	0.006s	0.0006s	0.0029s
tpm2_policysecret	0.008s	0.0012s	0.0107s
tpm2_activatecredential	0.010s	0.0042s	0.0167s
Quote integrity check	**0.022s**	**0.00629s**	**0.01254s**
tpm2_pcrextend	0.006s	0.00025s	0.00165s
tpm2_quote	0.011s	0.00604s	0.01089s
tpm2_checkquote	0.005s	/	/

Table 6.2. Quote commands timings

6.4.3 Attestation by Proof timings

Timings of Proof commands:

TPM 2.0 Command	TPM 2.0 Timings (sys+usr)	TPM 2.0 Timings (TSS_Execute)	TPM 2.0 Timings (tpm_hook)
Proof AK creation	**0.174s**	**0.29687s**	**0.19915s**
tpm2_pcrread	0.008s	0.00053s	0.00290s
tpm2_startauthsession	0.006s	0.00046s	0.00358s
tpm2_policypcr	0.008s	0.00103s	0.00713s
tpm2_createek	0.010s	0.05027s	0.02089s
tpm2_policysecret	0.010s	0.00170s	0.01275s
tpm2_create	0.012s	0.18183s	0.05911s
tpm2_load	0.010s	0.04385s	0.02647s
tpm2_evictcontrol	0.009s	0.00096s	0.00980s
tpm2_makecredential	0.062s	/	/
tpm2_activatecredential	0.011s	0.00340s	0.02235s
tpm2_certifycreation	0.012s	0.01167s	0.02239s
tpm2_loadexternal	0.008s	0.00062s	0.00754s
tpm2_verifysignature	0.008s	0.00055s	0.00424s
Proof integrity check	**0.026s**	**0.00527s**	**0.02098s**
tpm2_hash	0.009s	0.00033s	0.00121s
tpm2_sign	0.009s	0.00391s	0.01392s
tpm2_verifysignature	0.008s	0.00103s	0.00585s

Table 6.3. Proof commands timings

6.4.4 Command timings evaluation

The command timings, both for the Quote and the Proof protocol, have been collected and are useful to perform an evaluation on the feasibility and additional performance overhead of the deployment of this kind of integrity verification solution. The tables 6.2, 6.3 show the overall sequence of commands for the two protocols and the corresponding recorded timings measured for each of them. The timings are measured in three different approaches to make a complete analysis of the required time for integrity verification protocol execution. In this way the different obtained measurements can be compared and taken as reference depending on the context.

In the tables 6.2, 6.3 the execution time required for each single command is reported as well as the overall final time spent to execute each complete protocol phase: for the key creation phase and for the actual integrity check phase. Those time measurements allow to make an evaluation on the impact, in terms of performance, that such an integrity verification solution has when used in a specific application.

The collected measurements show that the time overhead added by the deployment of one of these two integrity verification solutions is very low. Both the Quote and the Proof protocols are lightweight and do not impact so much in terms of performance.

The AK creation phase is the more time consuming even though still does not require so much time to execute. The AK creation duration for the Quote protocol is almost 0,25 seconds while for the Proof protocol is almost 0,3 seconds, according to the measurement taken through the TSS_Execute interface. This phase is of course longer in the attestation by Proof because it includes more steps.

On the other hand the actual integrity verification phase duration is incredibly fast and lightweight. For the Quote protocol it is around 0,006 seconds while for the Proof one it is around 0,005 seconds (taking as reference the measurements collected through the TSS_Execute interface). The low time required to execute a complete integrity verification allow the Orchestrator to attest many different devices at the same time in a very fast and light way making this kind of solution highly scalable.

The collected measurements give us also an idea about the length of the time window between two subsequent remote attestation processes. Considering the case in which an adversary launch an attack on one of the remote devices the time window between two subsequent integrity checks represent the maximum time

window in which that VM was considered trusted by an attacker even though its system was actually tampered. Having the execution time of the integrity verification process very short means that if the Orchestrator performs multiple subsequent attestations the remote machine's tampered state is detected immediately. The worst case scenario is represented by the use of the Proof protocol by creating a new AK at every time a new integrity verification is requested by the Orchestrator. In this case the overall attestation process lasts around 0,3 seconds. This time window is still very short and, in case the remote device was tampered, allows to detect a possible tampering attack right after it is launched.

6.5 Loaded binary extractor

In this remote attestation protocol, the attestation of the software program running on the Virtual Machine of the remove device is performed through the integrity check of the program's loaded binaries. Lets understand why the program binary should be extracted and all the ways it can be extracted from the linux environment.

When a software program is executed on a machine a new process starts. Then the program executable is loaded into the main memory in the address space assigned to the process, which enters the "waiting" state. As soon as a context switch loads the process into the CPU, the process enters the "running" state and the execution of the program actually starts. In general the loaded executable is in ELF format.

During load time, or even after that the executable has started running, an attacker could try to tamper and modify the loaded binary. These are attacks that happen during run-time, so during software execution. If an adversary is able to do that, the software running on the Virtual Machine would not be trusted anymore because it could divert from its normal expected execution flow or even change radically its behaviour. Therefore the loaded binary, which represents the software code that is actually running on the remote device, must be attested by the Orchestrator in order to check its integrity. It is important that the loaded binary code, running on the remote device, remains unchanged and safe: for this reason the remote attestation and integrity verification protocol is deployed to ensure integrity of the remote device.

The first thing to do is to understand how the loaded binary could be extracted from the linux operating system. Whenever a process starts in the linux environment a PID is assigned to it and a new folder, containing all the run-time data associated with that process, is created in the file system. That folder can be found at:

`/proc/`*`PID`*`/`

With *PID* equal the PID of the running process that has to be attested. Inside this folder we can find the loaded binary that we need. There are two places where the binary can be found inside this folder:

1. In the file named `exe`
2. In the file named `mem`, but it can only be accessed by reading first the location of the code section from `maps` file

With the first approach it is just necessary to write a program that reads the content of the `exe` file and sends it to the module that will compute the measures on it. This is the approach that has been chosen: it was written a C program that simply opens and reads the `exe` file saving its content, i.e. the loaded binary.

The second approach is also valid but require two subsequent steps: reading first the `maps` file and the `mem` file to read the actually loaded binary. Reading directly the `mem`, which contains the memory mapped the same way as in the process, will result in an error due to the fact that its first part is not mapped in the process. The way to read it is to read fist the `maps` file, which contains the addresses (offsets) of the various sections of the process, in order to know at which address the code section is. An then, with this information, a seek operation to the code section in the `mem` file can be performed to read the binary.

About the language, it was chosen the C language due to its better integration with the Intel PT. As will be described in the chapter about Intel PT analysis, it resulted easier and more intuitive the tracing of a C program rather than a python script. The lack of direct mapping between the python instructions and the Intel PT log make the control flow graph of a python program difficult to reconstruct, but this will be discussed in the Intel PT chapter. All of this because the Intel PT will be used to trace the execution of the loaded binary extractor. This is an additional secure control useful to verify if the extraction was performed correctly without any modification on the binary.

Once the binary has been extracted its measure is computed (in general by an hash operation) and it is used to extend a PCR register which will represent the state of the system. Than, as we already discussed, the PCR register value will be used in the integrity verification check to attest the remote device's state.

Another possible way to extract the loaded binaries of the software program is through an eBPF hooks program. By using "uprobe" capabilities an ad hoc eBPF program can be written to actually trace and extract the loaded binaries of a specific software program by reading the correct memory maps of the running process. Basically, through "uprobe", a user level function belonging to the program in execution can be used as hook to attach and execute a C eBPF function that will collect and record the memory maps allocated to that process during run-time.

This kind of eBPF program is a valid and efficient alternative to the previously described C program used to read process memory from the linux file system. However, in this work, the C loaded binary extractor approach has been preferred to the eBPF one due to integration reasons with the Intel PT. As will be explained in the Intel PT tracing chapter, there are some Intel PT limitations related to the

extraction of the control flow of the eBPF program execution. On the other hand the Intel PT tracing output of C programs results to be easier to interpret and understand.

6.6 Use of eBPF to trace TSS commands

In this section it is presented the use of eBPF hook and so its capabilities and the way it has been exploited in the remote attestation protocol.

The eBPF, as already described, permits to run some user-provided code in a virtual machine inside the linux kernel. Moreover it can act as hook of some specific kernel events: it can be "attached" to a kernel code path and whenever that section is executed the eBPF is triggered to collect useful information. Due to its lightweight capabilities it can run in the kernel gathering data without impacting too much in terms of performances. For this reason it can also be used for profiling and tracing of software execution.

In the defined remote attestation architecture, the process implementing the integrity verification protocol needs to interact directly with the TPM. In fact, as we saw, both the attestation by Proof and by Quote are composed of a sequence of TSS commands that needs to be sent to the TPM in order to perform the integrity verification. The operations of sending data to and receiving data from the TPM are potentially critical because an attacker could try to tamper the communication between the system and the TPM. An attacker could try to modify the commands and/or the parameters sent to the TPM or change the binary response received from the TPM.

Depending on the locality of the TPM there are different approaches that could be adopted. If the TPM is local a malicious actor could try to interfere in the operating system potentially in the system bus. While if the TPM is remote the attacker has a wider surface for the attack because it could perform a MITM attack on the network that connects the device with the TPM.

In this work the TPM is a software one resident on the Virtual Machine of the remote device and so it was only considered the case in which the TPM was local on the system. For this reason the eBPF has been used as a hook to log all the TSS commands that are sent to the software TPM and the response in order to check if they are as expected. Additionally the filtering capabilities of the eBPF were used to select only the useful data to produce a meaningful output.

Using the eBPF as hook means attaching the eBPF program to a specific set of

kernel functions. It is necessary to select a set of kernale functions that allow us to trace and log TSS commands and data exchanged with the TPM, which is our purpose. Therefore the two kernel system calls that have been selected as hook functions are:

- __vfs_read (or vfs_read for older versions of linux kernel)
- __vfs_write (or vfs_write for older versions of linux kernel)

The functions used to bind the eBPF program to the __vfs_read and the __vfs_write are the `attach_kprobe` and the `attach_kretprobe` called over a BPF object in the python section of the code.

Taking as example the __vfs_read hook,
`attach_kprobe(event="__vfs_read", fn_name="trace_read_entry")`
is used to instrument the __vfs_read kernel function, by using kernel dynamic tracing of the function entry, to attach it to the BPF defined trace_read_entry function written in C language. From now on, whenever __vfs_read is called the BPF function trace_read_entry is called.

On the other hand,
`attach_kretprobe(event="__vfs_read", fn_name="trace_read_return")`
is used to instrument the __vfs_read kernel function, by using kernel dynamic tracing of the function return, to attach it to the BPF function trace_read_return defined in C language. From now on, whenever __vfs_read returns the BPF function trace_read_entry is called.

Therefore these two functions will act like a hook for __vfs_read and __vfs_write triggering the BPF defined C function that will gather all the needed information and data. For our purpose of tracing and logging the TSS commands exchanged between the system and the TPM the following information have been collected:

- Absolute time
- Process name (command name)
- Thread identifier (TID)
- Type of operation: write (W) or read (R)
- Size of data exchanged in bytes
- Latency(ms): difference between return time and call time

- Device type: it is CHAR FILE for the hardware TPM and SOCK in the case of software TPM: it is`tpm0` for hardware TPM while it is TCP for software TPM
- Name of the file on which the operation is performed
- Data read or written

The __vfs_read and __vfs_write are the two basic functions called every time some data is written or read in the system. Since they are so general and since the most of the operations of the system can be reduced to reads and writes these two functions are called constantly in each instant a multiple number of times. This introduces the need for eBPF filtering capabilities. It is important to filter all the amount of data collected by the eBPF to keep and show only the meaningful ones.

The filtering parameters that has been used could be:

- The process name: that should correspond to the command sent to the TPM; it can be used to filter a single command to be analysed.
- The name of the file on which the operation is performed: the value to be filtered for this parameter depends on whether the TPM is hardware or software; if the TPM is hardware the OS sees it a file in the file system, which could be read or write; in linux the hardware TPM is represented by `/dev/tpm0`; if the TPM is software the file to be filtered is `TCP` because the commands are sent to the TPM as a tcp connection to the port 2321.
- The PID: it can be filtered based on a specific PID of the process running the commands.
- The operation type: it can be filtered selecting a specific operation, read or write, based on which of the two we want to verify.

Lets make a test to understand which could be like a possible output of the used eBPF program. The test has been performed by calling the command `tpm2_getrandom --hex 4` while the eBPF program was running in the meantime hooking vfs_read and vfs_write. The eBPF program was launched so that it could filter the command tpm2_getrandom. Down below it is shown the output obtained from this test:

```
TIME(s)  COMM            TID    MODE  BYTES          LAT(ms)   DEVICE TYPE    FILENAME     DATA
-------------------------------------------------------------------------------------------------------------------------------------
6.621    tpm2_getrandom  29037  W     22 of 22       0.02      SOCK           TCP          \x80\x01\x00\x00\x00\x16\x00\x00\x01
                                                                                          z\x00\x00\x00\x06\x00\x00\x01\x00\x0
```

```
                                                                                          0\x00\x00\x7f
[i] MANAGED TO PARSE COMMAND SEND TO TPM!!
{'Command Header': {u'command_code': {u'tpm_cc': u'TPM_CC_GetCapability'},
                    u'command_size': 22,
                    u'tpmi_st_command_tag': u'TPM_ST_NO_SESSIONS'},
 'Parsed Command': {u'cap_property': 256,
                    u'capability': {u'tpm_cap': u'TPM_CAP_TPM_PROPERTIES'},
                    u'property_count': 127}}
------------------------------------------------------------------------------------------------------------------------------------------
6.623    tpm2_getrandom   29037  R     4 of 4          5.62     SOCK          TCP          \x00\x00\x00\x00
------------------------------------------------------------------------------------------------------------------------------------------
6.624    tpm2_getrandom   29037  R     387 of 387      0.00     SOCK          TCP          \x00\x00\x00\x00\x01\x83\x00\x00\x00
                                                                                          \x00\x00\x00\x00\x00\x06\x00\x00\x00
                                                                                          .\x00\x00\x01\x002.0\x00\x00\x00\x01
                                                                                          \x01\x00\x00\x00\x00\x00\x00\x01\x02
                                                                                          \x00\x00\x00\x9f\x00\x00\x01\x03\x00
                                                                                          \x00\x018\x00\x00\x01\x04\x00\x00\x0
                                                                                          7\xe3\x00\x00\x01\x05IBM \x00\x00\x0
                                                                                          1\x06SW  \x00\x00\x01\x07 TPM\x00\x0
                                                                                          0\x01\x08\x00\x00\x00\x00\x00\x00\x0
                                                                                          1\t\x00\x00\x00\x00\x00\x00\x01\n\x0
                                                                                          0\x00\x00\x01\x00\x00\x01\x0b \x19\x
                                                                                          10#\x00\x00\x01\x0c\x00\x1666\x00\x0
                                                                                          0\x01\r\x00\x00\x04\x00\x00\x00\x01\
                                                                                          x0e\x00\x00\x00\x03\x00\x00\x01\x0f\
                                                                                          x00\x00\x00\x02\x00\x00\x01\x10\x00\
                                                                                          x00\x00\x03\x00\x00\x01\x11\x00\x00\
                                                                                          x00@\x00\x00\x01\x12\x00\x00\x00\x18
                                                                                          \x00\x00\x01\x13\x00\x00\x00\x03\x00
                                                                                          \x00\x01\x14\x00\x00\xff\xff\x00\x00
                                                                                          \x01\x16\x00\x00\x00\x00\x00\x00\x01
                                                                                          \x17\x00\x00\x08\x00\x00\x00\x01\x18
                                                                                          \x00\x00\x00\x06\x00\x00\x01\x19\x00
                                                                                          \x00\x10\x00\x00\x00\x01\x1a\x00\x00
                                                                                          \x00\r\x00\x00\x01\x1b\x00\x00\x00\x
                                                                                          06\x00\x00\x01\x1c\x00\x00\x01\x00\x
                                                                                          00\x00\x01\x1d\x00\x00\x00\xff\x00\x
                                                                                          00\x01\x1e\x00\x00\x10\x00\x00\x00\x
                                                                                          01\x1f\x00\x00\x10\x00\x00\x00\x01 \
                                                                                          x00\x00\x00@\x00\x00\x01!\x00\x00\x0
                                                                                          7\xc4\x00\x00\x01"\x00\x00\x01\x94\x
                                                                                          00\x00\x01#2.0\x00\x00\x00\x01$\x00\
                                                                                          x00\x00\x00\x00\x00\x01%\x00\x00\x00
                                                                                          \x9f\x00\x00\x01&\x00\x00\x018\x00\x
                                                                                          00\x01\'\x00\x00\x07\xe3\x00\x00\x01
                                                                                          (\x00\x00\x00\x80\x00\x00\x01)\x00\x
                                                                                          00\x00s\x00\x00\x01*\x00\x00\x00o\x0
                                                                                          0\x00\x01+\x00\x00\x00\x04\x00\x00\x
                                                                                          01,\x00\x00\x04\x00\x00\x00\x01-\x00
                                                                                          \x00\x00\x01\x00\x00\x01.\x00\x00\x0
                                                                                          4\x00
[i] MANAGED TO PARSE COMMAND RECEIVED FROM TPM!!
{'Command Header': {u'response_code': u'TPM_RC_SUCCESS',
                    u'response_size': 387,
                    u'tag': {u'tpm_st': u'0x0'}}}
Execution time of the command : 0.00311994552612
------------------------------------------------------------------------------------------------------------------------------------------
6.631    tpm2_getrandom   29037  R     4 of 4          0.00     SOCK          TCP          \x00\x00\x00\x00
------------------------------------------------------------------------------------------------------------------------------------------
6.635    tpm2_getrandom   29037  W     39 of 39        0.02     CHAR DEVICE   20           Time taken by command is : 0.005836427\n
------------------------------------------------------------------------------------------------------------------------------------------
6.636    tpm2_getrandom   29037  W     9 of 9          0.05     SOCK          TCP          \x00\x00\x00\x08\x00\x00\x00\x00\x0c
------------------------------------------------------------------------------------------------------------------------------------------
6.636    tpm2_getrandom   29037  W     12 of 12        0.07     SOCK          TCP          \x80\x01\x00\x00\x00\x0c\x00\x00\x01{
                                                                                          \x00\x04
[i] MANAGED TO PARSE COMMAND SEND TO TPM!!
{'Command Header': {u'command_code': {u'tpm_cc': u'TPM_CC_GetRandom'},
                    u'command_size': 12,
                    u'tpmi_st_command_tag': u'TPM_ST_NO_SESSIONS'},
 'Parsed Command': {u'bytes_requested': 4}}
------------------------------------------------------------------------------------------------------------------------------------------
6.637    tpm2_getrandom   29037  R     4 of 4          0.00     SOCK          TCP          \x00\x00\x00\x00
------------------------------------------------------------------------------------------------------------------------------------------
6.640    tpm2_getrandom   29037  R     16 of 16        0.00     SOCK          TCP          \x00\x00\x00\x00\x00\x10\x00\x00\x00\
                                                                                          x00\x00\x04T\x0b\'\xf2
[i] MANAGED TO PARSE COMMAND RECEIVED FROM TPM!!
{'Command Header': {u'response_code': u'TPM_RC_SUCCESS',
                    u'response_size': 16,
                    u'tag': {u'tpm_st': u'0x0'}}}
Execution time of the command : 0.00396203994751
------------------------------------------------------------------------------------------------------------------------------------------
6.641    tpm2_getrandom   29037  R     4 of 4          0.00     SOCK          TCP          \x00\x00\x00\x00
------------------------------------------------------------------------------------------------------------------------------------------
```

```
6.643    tpm2_getrandom   29037  W     39 of 39        0.01      CHAR DEVICE    20          Time taken by command is : 0.000215223\n
-------------------------------------------------------------------------------------------------------------------------------------------
6.643    tpm2_getrandom   29037  W     8 of 8          0.01      CHAR DEVICE    20          540b27f2
-------------------------------------------------------------------------------------------------------------------------------------------
```

This result obtained shows the single TSS commands sent and received. It can be noticed that the tpm2-tools command tpm2_getrandom makes two TSS command invocations: TPM_CC_GetCapability and TPM_CC_GetRandom TSS commands. Looking at the output, first of all it can be easily checked if the sequence of commands sent and received is as expected. The ordered sequence of TSS commands can be compared to a reference list in order to notice any kind of anomaly or deviation. In addition to this control there is the check of the parameters. For each command its parameters could be checked, again, to verify that they are as expected.

Specifically from that test we can see that the GetRandom parameter for the number of requested bytes is 4: it reflects the real parameter passed to the command. And as value returned by the TPM we obtain the random string "540b27f2".

The data logged for each message exchanged is binary data, in the example it is represented as hexadecimal data. A comfortable way to analyse and understand the data exchanged is by using a custom parser which interpret the data exchanged and represents it in a human readable form. Therefore a python parser has been used to understand which commands where sent to the TPM. From the showed output the parser is able to detect and report the commands exchanged and their parameter in an organized way.

Moreover, since that eBPF program is able to understand when a command is sent to the TPM and when the response is received from it, it has also been used to compute the time required for the execution of each TSS command. The execution time has been computed as the difference between the time at which the response is received minus the time at which the command is detected being sent to the TPM. For this purpose the parser has been used as a signaler of the start of stop of the time computation. Whenever the python parser detect a command sent to the TPM the start time is taken and then, when the parser catches its response, the end time is taken and the difference is computed. The time required by each TSS command is then reported as eBPF program output.

As already pointed out the remote attestation solution is meant to be working in a Cloud Computing environment. Therefore these tests have been performed on a Virtual Machine in order to assess that the eBPF works well in a virtualized environment. Based on the result obtained the eBPF program used behave correctly on a Virtual Machine logging and reporting all the TSS commands that has been invoked and all the responses sent back from the TPM. The important thing

to do is to filter in a proper way in order to obtain a good output containing only with the required information.

The one thing to keep in mind while deploying the eBPF tracer is the **maxactive** value defined in bcc as a field in the kretprobe struct. This value specifies the maximum number of instances of a specified function that can be probed simultaneously. This means that there is an upper threshold that limits the number of kernel functions that can be hooked at the same time. Therefore in TSS commands tracing it can limit the number of messages that can be traced if they are exchanged very close in time to each other. The maxactive value is defined until up to double the number of available cpus on the system. This is a well known limitation of bcc and must be taken into account. In a virtualized environment it is necessary to configure the number of cpus made available to the Virtual Machine, configuring in this way the maxactive value.

The tests conducted show that configuring the Virtual Machine with only one cpu results in problems in tracing TSS commands: not all the messages will be caught and showed by the eBPF tracer. Increasing the number of cpu made available to the Virtual Machine to four cpus ensures that all the exchanged TSS commands and data messages are traced.

6.7 Intel PT tracing analysis

6.7.1 Overview

The Intel Processor Tracing, as already mentioned, has been used to trace the execution of the loaded binary extractor, in order to verify and check the correctness of the binary read process, and to log the execution of the eBPF tracer while intercepting the TSS commands. Intel PT's capabilities allow to trace the execution of a software program producing a series of binary packets. These packets are then processed by a decoder to reconstruct the real execution flow of the program and so the actual sequence of commands performed by the program.

This tracing technique has been used as the loaded binary tracer in the context of this work in order to try to insert it into a specific practical use case. Additionally, a more in depth tests, analysis and evaluation on its capabilities have been performed to understand better its properties. This additional analysis highlights Intel PT strengths and weaknesses allowing to make an assessment of feasibility in deploying such a tracer in an application.

However, it must be kept in mind that the Intel PT is a tracing technique which is bound and tight to a specific hardware module. The use of such a tracing tool requires the presence and the use of a recent Intel processor with Intel PT hardware capability. This could be a limitation that must be kept into consideration when deploying it. However most of the common machines and devices nowadays have, already built on board, an Intel processor of late generation with the Intel PT feature. The Intel PT has been chosen in this work also for this reason: because Intel processors use is very common and widespread nowadays, making its requirement not a so strict assumption.

The following section will describe all the tests and analysis that have been conducted to evaluate Intel PT capabilities and properties.

6.7.2 Basic tests and analysis

The first part of the analysis that has been conducted focus on understanding the Intel PT capabilities: which kind of information can be obtained as output of the Intel PT and how the software decoder works.

The first test has been performed on a simple C "Hello World" program. The command used to launch the program tracing is:

```
perf record -e intel_pt//u ./helloworld
```

This is the basic command used to perform intel_pt sampling with perf tool. It makes the execution of the "helloworld" program start and then trace the program execution collecting all the information. The `//u` parameter is used to trace only userspace commands. As result of the `perf record` command execution the file `perf.data` is generated. It contains all the tracing information in a binary packets format.

The `perf.data` file, containing all the output information of the Intel PT tracing of the "helloworld" program, can be used as input to a post-processing tool to produce a more human readable output. There are different ways for extracting meaningful information from the `perf.data` file obtaining different levels of granularity in the output. The software program used for this purpose is the decoder which makes use of Intel PT packets and program binaries to reconstruct the control flow of the software execution.

Having a look at the Intel PT binary packets produced it can be noticed that they have no meaning just by themselves. They are just an ordered sequence of information with apparently no connection to any code. It is duty of the decoder

to parse the `perf.data` file and also the program binaries to connect the dots and merge them in order to generate meaningful information. Especially for reconstruction of control flow of the software execution, the binary packets have sense only if associated to the program binaries. The decoder pass across all the packets in order from the beginning to the end and at the same time pass through the program binaries instruction in sequence to understand the association packets-commands. For example when it encounters a TNT packet the decoder can link it with the corresponding instruction in the program binary showing which specific branch was taken or not.

Two main commands have been used to decode and extract meaningful information from the `perf.data` file:

- `perf script -D`: it displays a verbose dump of the trace data. From this dump the single packets information can be seen in a readable form and the virtual addresses assigned during execution time to the program file and its external libraries are shown. Still this representation is very rough and difficult to read in order to reconstruct the control flow of program execution but allow to know some basic information.

- `perf script --itrace=bi0`: this decoder, processing Intel PT packets and program's binaries, reconstructs the execution flow and the exact sequence of instructions for the software program traced. This command shows the sequence of all the instructions executed as well as the branches taken by the program. Moreover, the `--insn-trace --xed` options can be used to additionally disassemble the instructions: the result will be a sequence of instruction and branches with the corresponding assembly code associated.

By using `perf script --itrace=bi0` a little section of the whole output that is obtained can be something like this:

```
helloworld 10057 [000] 16538.852347:    1 instructions:u:      7f50298340a9 __libc_start_main+0xe9 (/usr/lib/x86_64-linux-gnu/libc-2.31.so)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      7f50298340ac __libc_start_main+0xec (/usr/lib/x86_64-linux-gnu/libc-2.31.so)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      7f50298340b1 __libc_start_main+0xf1 (/usr/lib/x86_64-linux-gnu/libc-2.31.so)
helloworld 10057 [000] 16538.852347:    1     branches:u:      7f50298340b1 __libc_start_main+0xf1 (/usr/lib/x86_64-linux-gnu/libc-2.31.so)
                                                                                                  =>     55d2b0f7a149 main+0x0 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a149 main+0x0 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a14c main+0x3 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a14d main+0x4 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a14e main+0x5 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a151 main+0x8 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a158 main+0xf (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1     branches:u:      55d2b0f7a158 main+0xf (/home/helloworld)
                                                                 =>     55d2b0f7a050 _init+0x50 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a050 _init+0x50 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a053 _init+0x53 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1 instructions:u:      55d2b0f7a054 _init+0x54 (/home/helloworld)
helloworld 10057 [000] 16538.852347:    1     branches:u:      55d2b0f7a054 _init+0x54 (/home/helloworld)
                                                            =>     7f50298945a0 _IO_puts+0x0 (/usr/lib/x86_64-linux-gnu/libc-2.31.so)
```

The output shows the data that the decoder permits to extract from the binary packets. The attributes shown are: the process name, the PID, the absolute time, the record type (instructions or branches), the virtual address and the symbolic name of the instruction. Moreover, the taken branches are represented by the addresses of the source and destination separated by a "=>" symbol. Additionally, if the xed (X86 Encoder Decoder) is enabled the assembly instructions are also decoded from raw bytes and shown at the output.

Every single line of the output obtained represent a single assembly instruction or branches of the traced program. This prove that the Intel PT is actually capable of tracing every single instruction in a very detailed way. This could led us to the possibility of generating the exact control flow of a program execution not only by logging the main operation but the exact sequence of assembly commands. It is a level of details that other techniques, like the ones based on software instrumentation or based on hooking are not able to achieve.

The next simple test that has been done is the tracing of a slightly more complex program composed of different path that could be taken during its execution. What we are trying to understand is if the Intel PT is capable of tracing the path/branch was actually taken during run-time. And so also, if the information obtained from the decoder are enough to detect the executed path and possible variation from the expected one.

This second test has been performed over this very simple C program:

```c
#include <stdio.h>
int main(){
  int a, b;
  scanf("%d %d", &a, &b);
  if(a == 1){
    if(b == 1)
      printf("Double 1\n");
    else
      printf("1 and 0\n");
  }
  else{
    if(b == 1)
      printf("0 and 1\n");
    else
      printf("Double 0\n");
  }
  return 0;
}
```

Figure 6.5. If-else with 4 possible paths basic C program for test

In the above program depending on the inserted input the walked through path changes. The experiment conducted show that the Intel PT is capable of tracing and detecting the actual path taken during run-time.

As we saw earlier the output of the decoder consists of an ordered sequence of assembly instruction and branches. It is very detailed and precise but could be a little bit disorienting for the amount of information and level of detail it can be obtained. A way to produce a personalized and more simple output is to build an information filter/parser to extract just the necessary and meaningful data for our purpose.

For these reasons a python parser program has been developed to produce a more clear output from the one generated by the decoder. This filter takes as input two files:

1. A file containing the assembly code of the C program to be traced
2. A file containing the output data produced by the Intel PT decoder

The first file can be generated by disassembling the program binary by means of GDB tool or in a more simple way by using the `objdump` tool. They produce the assembly sequence of commands starting from the program binary. Next to

each instruction it is also represented its offset with respect to the beginning of the main function (something like `<main+0x1c>`): this will simplify the work of our python parser. The second file is just obtained by saving the output produced by the `perf script --itrace=bi0` command execution.

The main objective of the python parser built is to merge the information contained in the two files. It first of all collects in a list all the assembly instructions from the file containing the disassembly of the program. And then it parses the whole decoder's output file marking, in an ordered way, each time one of the program commands was found and all the branches between them. At the end of this process we end up with a more high level control flow graph made by just by the user level assembly instructions that directly map the C commands used.

Tracing the execution of the simple C program reported in figure 6.5 providing as input "1 0" and then using the above mentioned python parser produces the following output result result:

```
0- (node n.0) -- main+0x0 endbr64 endbr64 (instruction)
1- (node n.1) -- main+0x4 push %rbp (instruction)
2- (node n.2) -- main+0x5 mov %rsp,%rbp (instruction)
3- (node n.3) -- main+0x8 sub $0x10,%rsp (instruction)
4- (node n.4) -- main+0x12 mov %fs:0x28,%rax (instruction)
5- (node n.5) -- main+0x21 mov %rax,-0x8(%rbp) (instruction)
6- (node n.6) -- main+0x25 xor %eax,%eax (instruction)
7- (node n.7) -- main+0x27 lea -0xc(%rbp),%rdx (instruction)
8- (node n.8) -- main+0x31 lea -0x10(%rbp),%rax (instruction)
9- (node n.9) -- main+0x35 mov %rax,%rsi (instruction)
10- (node n.10) -- main+0x38 lea 0x555555556004 (instruction)
11- (node n.11) -- main+0x45 mov $0x0,%eax (instruction)
12- (node n.12) -- main+0x50 callq <__isoc99_scanf@plt> (branch)
    13- (node n.13) -- main+0x55 mov -0x10(%rbp),%eax (instruction)
    14- (node n.14) -- main+0x58 cmp $0x1,%eax (instruction)
    15- (node n.15) -- main+0x61 jne <main+99> (instruction)
    16- (node n.16) -- main+0x63 mov -0xc(%rbp),%eax (instruction)
    17- (node n.17) -- main+0x66 cmp $0x1,%eax (instruction)
    18- (node n.18) -- main+0x69 jne <main+85> (branch)
|
|--->19- (node n.22) -- main+0x85 lea 0x555555556013 (instruction)
     20- (node n.23) -- main+0x92 callq <puts@plt> (branch)
      21- (node n.24) -- main+0x97 jmp <main+133> (branch)
|
|----->22- (node n.33) -- main+0x133 mov $0x0,%eax (instruction)
       23- (node n.34) -- main+0x138 mov -0x8(%rbp),%rcx (instruction)
       24- (node n.35) -- main+0x142 xor %fs:0x28,%rcx (instruction)
       25- (node n.36) -- main+0x151 je <main+158> (branch)
|
|------>26- (node n.38) -- main+0x158 leaveq leaveq (instruction)
        27- (node n.39) -- main+0x159 retq retq (branch)
-----------------------------------------------------------------
(node n. 0) - main+0x0      endbr64  endbr64                  (executed)
(node n. 1) - main+0x4      push     %rbp                     (executed)
(node n. 2) - main+0x5      mov      %rsp,%rbp                (executed)
(node n. 3) - main+0x8      sub      $0x10,%rsp               (executed)
(node n. 4) - main+0x12     mov      %fs:0x28,%rax            (executed)
(node n. 5) - main+0x21     mov      %rax,-0x8(%rbp)          (executed)
(node n. 6) - main+0x25     xor      %eax,%eax                (executed)
(node n. 7) - main+0x27     lea      -0xc(%rbp),%rdx          (executed)
(node n. 8) - main+0x31     lea      -0x10(%rbp),%rax         (executed)
(node n. 9) - main+0x35     mov      %rax,%rsi                (executed)
(node n. 10) - main+0x38     lea      0x555555556004           (executed)
(node n. 11) - main+0x45     mov      $0x0,%eax                (executed)
(node n. 12) - main+0x50     callq    <__isoc99_scanf@plt>     (executed)
(node n. 13) - main+0x55     mov      -0x10(%rbp),%eax         (executed)
(node n. 14) - main+0x58     cmp      $0x1,%eax                (executed)
(node n. 15) - main+0x61     jne      <main+99>                (executed)
(node n. 16) - main+0x63     mov      -0xc(%rbp),%eax          (executed)
(node n. 17) - main+0x66     cmp      $0x1,%eax                (executed)
(node n. 18) - main+0x69     jne      <main+85>                (executed)
(node n. 19) - main+0x71     lea      0x55555555600a
```

```
(node n. 20) - main+0x78     callq    <puts@plt>
(node n. 21) - main+0x83     jmp      <main+133>
(node n. 22) - main+0x85     lea      0x555555556013            (executed)
(node n. 23) - main+0x92     callq    <puts@plt>                (executed)
(node n. 24) - main+0x97     jmp      <main+133>                (executed)
(node n. 25) - main+0x99     mov      -0xc(%rbp),%eax
(node n. 26) - main+0x102    cmp      $0x1,%eax
(node n. 27) - main+0x105    jne      <main+121>
(node n. 28) - main+0x107    lea      0x55555555601b
(node n. 29) - main+0x114    callq    <puts@plt>
(node n. 30) - main+0x119    jmp      <main+133>
(node n. 31) - main+0x121    lea      0x555555556023
(node n. 32) - main+0x128    callq    <puts@plt>
(node n. 33) - main+0x133    mov      $0x0,%eax                 (executed)
(node n. 34) - main+0x138    mov      -0x8(%rbp),%rcx           (executed)
(node n. 35) - main+0x142    xor      %fs:0x28,%rcx             (executed)
(node n. 36) - main+0x151    je       <main+158>                (executed)
(node n. 37) - main+0x153    callq    <__stack_chk_fail@plt>
(node n. 38) - main+0x158    leaveq   leaveq                    (executed)
(node n. 39) - main+0x159    retq     retq                      (executed)
```

The above displayed output shows:

1. On the bottom section, the assembly instructions of the program. At each of them a node number is assigned and a label is reported for those of them that have been executed.

2. On the top, the control flow graph of the execution of the program. It reports the ordered sequences of node numbers, making reference to the bottom assigned enumeration, and the branches executed inside the code represented as arrows between two instructions.

As it can be seen it traces and reports correctly the branches taken pointing out the executed portion of code based on the input provided. The same correctness has been obtained by testing other types of test C codes: a program with a switch-case statement, a program with 2 for loops and a program with 2 for nested loops. In all the tests done the Intel PT has always traced all the commands and branches taken is a precise and correct way following the right path executed during run-time based on the input inserted. This basic tests can give us the certainty of the correctness of that the Intel PT in providing a precise control flow graph of program execution.

The experiments conducted show how the post-processing of tracing information is important and potentially costly in terms of performance in the context of Intel PT. After that the Intel PT tracing has produced the binary data packets the decoder (the first post processing tool) must be used to extract human readable information from them producing the entire control flow sequence of commands and/or branches. After that, an additional post processing tool or parser is required to filter and extract only important and useful information manipulating them based on the specific application of use and purposes.

6.7.3 Python program and eBPF tracing tests

Another part of the analysis regards how well the Intel PT can trace the execution of other types of program languages like python program and eventually of the eBPF hook program, described before for TSS command tracing.

As seen before, the basic experiments conducted on the Intel PT over simple C programs permitted to report and produce the exact execution flow graph of the program execution in a correct way. The key point of C program tracing by means of the Intel PT is the availability of a direct mapping between the C assembly commands and the logged traced sequence of executed instructions. By using disassembly tools like `objdump` or GDB the assembly sequence of commands of the C program can be obtained and can be stored as static reference. Subsequently, when inspecting the decoded packet information, the reference and direct mapping to that C program commands can be found in the run-time traced sequence of instructions. As we saw before this reference to the C program binary is clearly reported in the form `<main+offset>`. This mapping was also used by the coded python parser to reconstruct the C code level control flow graph of the program execution.

Some tests have also been performed over very simple python programs to understand which is the output tracing obtained through the Intel PT. The python program used was characterized by a basic if-else statement whose path direction can be decide through program input parameter. The goal of this test was to verify whether the decoded packets of the execution tracing could allow us to detect which control path was actually taken during run-time. After analysing the tracing output produced by the Intel PT decoder it has not been possible to reconstruct the real execution flow graph in a direct way.

Due to python language's nature, which exploits an interpreter to execute a program, its code is never compiled before execution. Thus it has not been found a way to extract the sequence of assembly commands of a python program in a static way. It is for this reason that the tracing output of the python program produced by Intel PT, after being parsed by the decoder, will not be able to provide a reference of the position and of the mapping of the instruction traced with the python program file executed. The Intel PT tracing of a python program results in an undefined sequence of assembly commands without any reference and so without a clear and direct clue of how the program was actually executed.

The same result obtained with the python program has been found tracing the execution of the eBPF program described in the previous sections. There is no direct mapping between the instructions traced by the Intel PT and an

hypothetical sequence of reference commands that comprise the eBPF program. Moreover the number of data packets produced is very huge also just after a couple of seconds of program execution making the execution flow graph analysis potentially expensive and difficult.

The unavailability of direct mapping of python program tracing with respect to C program make the analysis and understanding of the output obtained quite difficult. A way to obtain meaningful information about the program execution, in order to understand the actual sequence of commands that was executed in a clean and readable way when it comes to python and other interpreted language, is outside of the scope of this work and have not been inspected.

The positive result is that the Intel PT is able to trace the program execution and consequently producing the actual sequence of assembly commands. With these information, the way to extract meaningful information to reconstruct the control flow graph at a higher level, understanding which branch of the code was taken, is up to a post processing tool analysing the tracing data. The reconstruction of the execution flow graph should be the result of a reverse engineering operation and process starting from the assembly sequence obtained and the python code. A solution to this problem could be quite complex and expensive in terms of time required to reconstruct the control flow graph. Even though the execution flow could be reconstructed in a good way - some experiments must be carried out on different python program complexities - a tool to do reverse engineering of the traced assembly could be unfeasible in term of time overhead required when implemented in a solution.

6.7.4 Timings and memory overhead evaluation

Part of the analysis on Intel PT capabilities is focused on understanding which could be the memory and performance overhead added by it when deployed to trace the execution of a program. The goal here is to try to use the Intel PT to trace the execution of a set of several C programs of different levels of complexity. These C programs should be more and more complex, in an increasing way allowing to understand how the Intel PT behave with programs of different orders of complexity and which is the upper limit of complexity that the Intel PT can still manage in a feasible and efficient way.

The approach used here to test the Intel PT properties and to measure the additional time and performance overhead that it introduces is incremental: an Intel PT evaluation analysis has been performed by means of software programs of different orders of complexity in order to see how it behaves and responds. Therefore,

the tests executed have been done by trying to collect timing and memory measurements while using the Intel PT to trace different C programs of incremental orders of complexity. Starting from the very simple "Hello World" program, the complexity of the program traced has been increased by fist adding more instruction or "printf"s and then by adding a bigger number of cycle iterations in order to increment the number of instruction executed and the time spent for program execution.

The scope of this section is to find out and monitor how the Intel PT behave in tracing C programs of higher levels of complexity and eventually which is the maximum handled program complexity by the Intel PT without incurring in a too high time and space additional overhead.

First of all a set of C programs of increasing complexity has been picked to be used with the Intel PT in order to perform this performance and feasibility analysis. The idea here is to start from the very simple "Hello World" program, which represents the first lowest level of complexity, and then to add some additional complexity by trying to introduce and try other statements or loops. The second level of complexity is represented by a `switch-case` statement program or an `if-else` statement program where the value of a variable makes the execution select one of the multiple possible paths. The third level of complexity is represented by a program with two or three `for` loops. The forth level of complexity includes nested loops programs. A fifth level could potentially comprise recursive programs but they have not been examined in this analysis.

The Intel PT performance and feasibility analysis has been performed on all these kinds of programs with the goal of understanding how it behaves when tracing different programs of different orders of complexity.

The following table 6.4 shows and summarizes all the C programs that have been used as case studies to conduct this Intel PT analysis:

Name	Program description
helloworld	The basic Hello World program
switchcase	A switch-case statement with 8 choices
ifelse	A double if-else statement with four possible direction paths
500for-ifelse	A `for` loop of 500 cycles with a double if-else statement inside
1000forprintfs	A `for` loop of 1000 cycles with a `printf` in the loop
5000forprintfs	A `for` loop of 5000 cycles with a `printf` in the loop
for(8000000)	A `for` loop of 8000000 cycles with a sum in the loop
2*for(8000000)	Two `for` loops of 8000000 cycles with a sum inside each loop
3*for(8000000)	Three `for` loops of 8000000 cycles with a sum inside each loop
nested-0.15s	A nested `for` loop program with a sum in both loops that lasts 0,15 seconds
nested-0.3s	A nested `for` loop program with a sum in both loops that lasts 0,3 seconds
nested-0.5s	A nested `for` loop program with a sum in both loops that lasts 0,5 seconds
nested-1.5s	A nested `for` loop program with a sum in both loops that lasts 1,5 seconds
nested-2s	A nested `for` loop program with a sum in both loops that lasts 2 seconds
nested-3s	A nested `for` loop program with a sum in both loops that lasts 3 seconds
nested-4s	A nested `for` loop program with a sum in both loops that lasts 4 seconds
nested-6s	A nested `for` loop program with a sum in both loops that lasts 6 seconds
nested-10s	A nested `for` loop program with a sum in both loops that lasts 10 seconds
nested-20s	A nested f`for`or loop program with a sum in both loops that lasts 20 seconds
nested-30s	A nested `for` loop program with a sum in both loops that lasts 30 seconds

Table 6.4. Programs used for the Intel PT analysis

Lets now have a look at how these C programs have been used to collect the Intel PT timings measurements first and the memory measurements later and lets

see which are the obtained results.

Timings analysis and measurements

The set of C programs, belonging to different increasing levels of complexity, presented in the table 6.4 have been used to collect Intel PT tracing timing measurements. The approach that has been followed is to first measure the time required by a program for its normal execution (without being traced) and then taking the time necessitate for its execution while traced by the Intel PT. These two measurements are used to compute the timing performance overhead introduced by the use of the Intel PT to trace its execution.

The program execution time has been taken by the command:

```
time ./program
```

On the other hand, the execution time of the program while traced by the Intel PT has been measured by the following command:

```
time perf record -e intel_pt//u ./program
```

This second time measurement includes the execution time of the program along with the time added by the Intel PT to trace it and so also the subsequent creation and writing of the **perf.data** file with the binary data packets.

The timings obtained from the measurement computation and the calculated performance overhead are reported in the table 6.5 down below:

Program	**Execution time (s)**	**Intel PT tracing time (s)**	**Time overhead (%)**
helloworld	0,003	0,143	4666,67
switchcase	0,003	0,138	4500,00
ifelse	0,003	0,120	3900,00
500for-ifelse	0,003	0,130	4233,33
1000forprintfs	0,006	0,150	2400,00
5000forprintfs	0,014	0,162	1057,14
for(8000000)	0,036	0,148	311,11
2*for(8000000)	0,053	0,168	216,98
3*for(8000000)	0,067	0,192	186,57
nested-0.15s	0,146	0,286	95,89
nested-0.3s	0,318	0,522	64,15
nested-0.5s	0,510	0,620	21,57
nested-1.5s	1,585	1,860	17,35
nested-2s	2,005	2,282	13,82
nested-3s	3,027	3,417	12,88
nested-4s	4,073	4,679	14,88
nested-6s	6,088	6,925	13,75
nested-10s	10,059	11,459	13,92
nested-20s	20,074	23,386	16,50
nested-30s	29,982	33,731	12,50

Table 6.5. Intel PT timings

To have a more clear idea on the level of overhead and its correlation to the average program execution time it is better to represent the data of the table 6.5 in a graphical form. Figure 6.6 shows the relation between the execution time of the program (on the horizontal axis) and the overhead percentage (on the vertical axis). It exhibits the overall curve trend of the data also highlighting the single points corresponding to the single table 6.5 measurement entries.

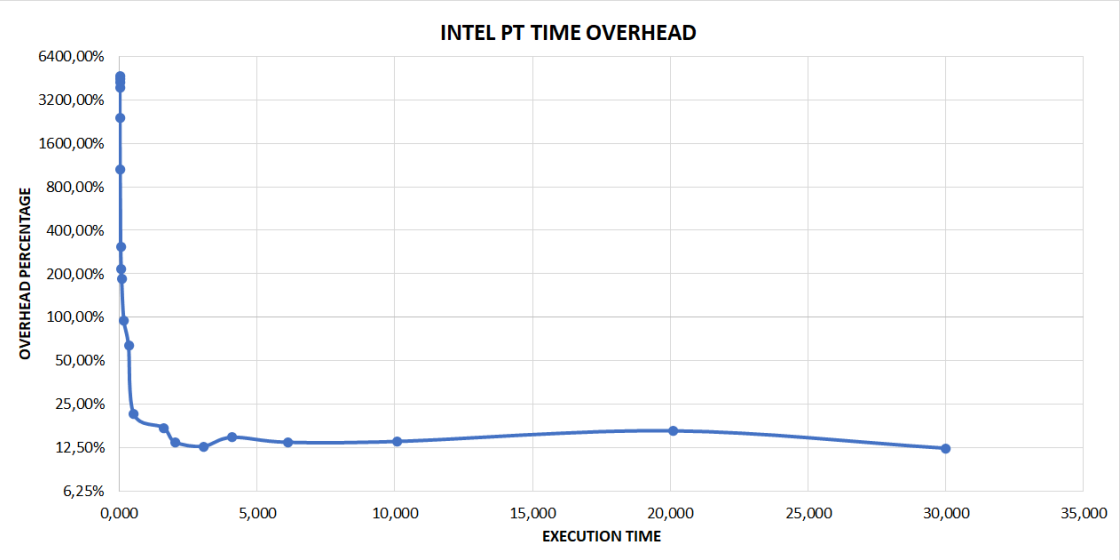


Figure 6.6. Relationship between program time and overhead percentage

Lets discuss the obtained results shown in the figure 6.6. The chart shows a huge level of time overhead for fast programs (programs having a very short execution time) and a stabilization of the curve for programs with a longer execution time. It seems that there is a point where this change of tendency happens and it can be found at around 0.5s. All the programs that have a shorter execution time suffer a higher time overhead when traced with the Intel PT while programs from 0,5 upwards tend to have an almost steady performance degradation resulting to a convergence to a specific value. At the end the curve converges to a value of performance overhead that is around the 13 %.

The convergence and stabilization of the curve to this value for all the tested programs having an execution time grater than 0,5 seconds (in our case they were all nested `for` loops) leads us to select the value of 13 % as the average obtained time overhead percentage introduced by the Intel PT when tracing the execution of a program with a comparable complexity of a nested `for` loop program. This overhead percentage value could be different for other types of programs. But according to the tests executed the tracing of the nested `for` loop programs return a value of about 13 % of performance degradation.

Considering the experiments conducted the time overhead obtained is relatively low and does not affect program's execution so much. A 13 % of performance degradation on a program execution is acceptable. However, when deploying the

Intel PT tracing in a specific application this additional introduced overhead percentage must be kept into consideration.

For programs having an execution time lower than 0,5 seconds the overhead seems very high in percentage with respect to the normal execution time. However, from the table 6.5 it can also be noticed that for all these fast programs their Intel PT tracing timings are steady, all around the range between 0.140 and 0.200 seconds more or less. This means that the Intel PT requires a fixed amount of time to dumb the processor's states and to write that data in packet format in `perf.data` file which seems to be around 0.140 and 0.200 seconds plus a variable delta time that depends on the program duration. When the program duration is very short this fixed writing and operational time is huge compared to the normal execution time and therefore it results to be much higher, resulting in a high time percentage overhead; while when the program execution requires more time than 0.140 and 0.200 seconds the fixed Intel PT time component results to be very low in percentage compared to the normal execution time and prevails the delta component which is low and almost stable in percentage to the program's duration. For the tested `for` loop programs the delta component of the additional time overhead added by the Intel PT resulted to be around 13 % of the program's duration.

The generation of the binary packets is just the first step towards the reconstruction of the final control flow graph. The next step is to use the decoder to extract information from the binary data in order to transform and represent them in a human readable and comprehensible form. In fact the complete reconstruction of the control flow graph of the program execution is the result of the two subsequent steps of tracing and than decoding. The goal of the conducted analysis is also to understand how much time the reconstruction of the whole execution flow takes in order to understand the performance and effectiveness impact that the Intel PT has when used in a control flow attestation solution.

Therefore lets measure also the time required by the decoder to generate the whole control flow graph starting from the binary tracer. The objective of these measurements is to understand how much time is required in the whole process from the tracing until having the final execution flow graph generated.

The table 6.6 reports all the obtained time measurements for the decoding phase. It contains also the dimension of the `perf.data` file for each program in order to try to reveal a possible relationship or pattern between the amount of binary packets generated and the required time to decode it.

The decode time measurements have been taken in two different ways:

1. By using `time perf script --itrace=bi0 > tmp`: it takes the time of the generation of the whole detailed control flow sequence containing both the single instructions and the taken branches. Including all the information it will end up being quite huge in terms of dimension.

2. By using `time perf script --itrace=b > tmp`: it measure the time required for generating a execution flow graph containing just the taken branches, without reporting all the instruction. This way of decoding produces less data but will result in being more lightweight.

This table 6.6 below shows the list of all the time measurements obtained during the decoding phase by using the two decoding commands, the dimensions of the `perf.data` file along with the ratio in percentage between the two different decoding timings:

Program	**perf.data dimension**	**itrace=bi0 time measure (s)**	**itrace=b time measure (s)**	**b/bi0 (%)**
helloworld	18876	0,485	0,142	29,27
switchcase	18732	0,484	0,183	37,81
ifelse	18772	0,535	0,133	24,86
500for-ifelse	27612	0,565	0,160	28,32
1000forprintfs	251668	4,842	0,812	16,76
5000forprintfs	1385388	25,938	3,448	13,29
for(8000000)	1489452	83,781	17,453	20,83
2*for(8000000)	2976620	174,069	39,548	22,71
3*for(8000000)	4438404	236,287	58,083	24,58
nested-0.15s	13084556	694,048	176,661	25,45
nested-0.3s	26727468	1611,069	407,650	25,30
nested-0.5s	44591252	/	676,186	/

Table 6.6. Intel PT decode timings

As it can be seen from the table 6.6 as the dimension of the binary data produced by the tracing increases the time to then decode it grows accordingly. The figure 6.7 helps to better understand and visualize the relationship between the `perf.data` dimension in bytes and the required time to decode it.

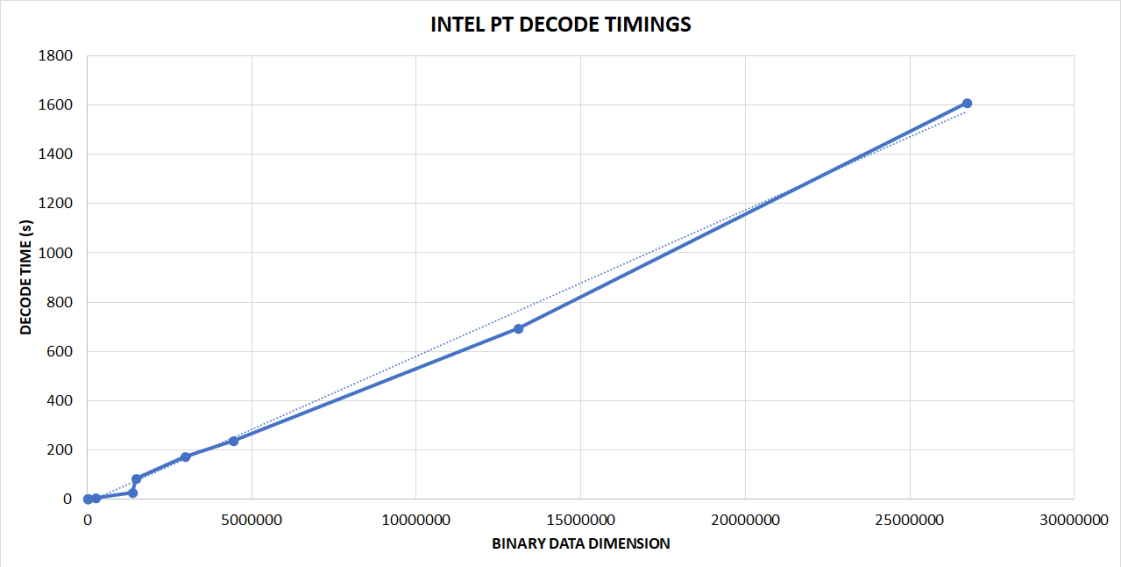


Figure 6.7. Relationship between binary data dimension and required decoding time

As it can be noticed from the image 6.7 there is a linear relationship between the `perf.data` dimension and the decode time required by using the `itrace=bi0` option in `perf script` command. The relationship with the decoding time by making use of the `itrace=b` is not reported to avoid redundancy but follows the exact same linear trend of the plot in figure 6.7.

This type of analysis let us understand how much time the decoding procedure lasts based on complexity of the software program. The decoding phase time is surprisingly high as it can be seen from the table 6.6. Putting in place the complete decoding technique used to obtain the exact sequence of both instruction and branches executed the time required to decode the binary packets results to be acceptable but still high for very simple and small programs while it increases very much, even though linearly, for more complex programs.

Lets take as example the case of the `for(8000000)` loop program. Its normal execution time is quite low, equal to 0,036 seconds, while its decoding time, used to obtain a complete instruction plus branches control flow graph, corresponds to 83,781 seconds, which seems to be too much high. This results highlight the weakness in terms of scalability of the Intel PT usage.

A way to reduce this decoding time is to decide to decode and filter just the branch instructions, reporting in this way just the control flow information by using the `itrace=b` option. This approach can perfectly fit the scopes presented

in this work of using the Intel PT as Control Flow tracing technique. So lets see how much the decoding time can be reduced in this way, still obtaining a complete and clear output that allows to reconstruct the execution flow of the program. As shown in the table 6.6 the time to decode just the control flow data is around 20-25% of the time required to produce a complete output, resulting in a gain of 75-80% in terms of performance over the decoding phase.

By decoding just the control flow information and so only the branch instructions the time overhead is lower and so the level of Intel PT's manageable complexity of the traced program can be increased without incurring in a too high performance degradation. However, despite this branch filter is beneficial it just moves the upper handled complexity threshold a little bit further, but it does not solve Intel PT's scalability problems completely. At the end the upper handled complexity limit is still present as well as the decoding limitations in terms of performance. Of course the additional time overhead evaluation depends on the specific application, but when the decoding time is in the order of minutes, for programs having a normal execution time equal to 0,067, the performance degradation seems too high.

The decoding time must be summed up with the additional time overhead required by the Intel PT for tracing the program's execution in order to obtain the entire duration time from program execution to the time at which its execution flow graph is ready and available. Furthermore, sometimes an additional parser or post-processing tool is needed to extract and manipulate the output information produced by the decoder to generate useful control path information, additionally increasing in this way the post-processing time duration. For example also in the solution proposed in this document a python parser has been used to extract just the essentially useful data from the decoder's output in order to attest the execution of the loaded binary extractor.

Memory analysis and measurements

The other analysis that has been conducted, in parallel to the time performance evaluation, is on the amount of data produced by the Intel PT. This evaluation helps to understand which should be the memory capabilities of a device implementing a solution making use of the Intel PT to trace the execution of a program.

The table 6.7 below shows, for each tested program the dimension of the `perf.data` produced, the amount of data produced as output of the full decoding and the amount of data produced by the decoding of just the control flow data.

Program	**perf.data dimension**	**itrace=bi0 data dimension (bytes)**	**itrace=b data dimension (bytes)**	**b/bi0 (%)**
helloworld	18 876	29 193 613	3 848 603	13,18
switchcase	18 732	29 074 077	3 836 260	13,19
ifelse	18 772	29 068 201	3 836 591	13,20
500for-ifelse	27 612	37 971 561	4 957 355	13,06
1000forprintfs	251 668	333 187 949	52 095 234	15,64
5000forprintfs	1 385 388	1 823 477 040	288 589 952	15,83
for(8000000)	1 489 452	6 178 863 666	1 634 481 078	26,45
2*for(8000000)	2 976 620	12 354 953 168	3 274 534 884	26,50
3*for(8000000)	4 438 404	18 242 916 458	4 914 508 946	26,94
nested-0.15s	13 084 556	56 307 629 920	15 347 738 764	27,26
nested-0.3s	26 727 468	1,14552E+11	31 173 195 158	27,21
nested-0.5s	44 591 252	/	53 488 551 359	/

Table 6.7. Intel PT decode data dimension

Lets focus our attention on the amount of data generated during the Intel PT tracing and decoding processes. In the first tracing phase the data are produced, in the more condensed form of binary packets. On the other hand the second decoding phase generates the execution flow graph of the program's execution, starting from the binary packets, in a human readable form in a more thorough and extensive way resulting in the generation of a larger amount of data. By looking at the table 6.7, the dimension of the `perf.data` file seems to remain quite small in the order or Megabytes, when increasing the complexity of the traced program.

On the contrary the dimension of the data produced as a result of the post-processing decoder results to be very high, even for relatively small programs. Taking as example the `for(8000000)` program, the amount of data produced as result of the entire and complete decoding is close to 6 Gigabytes; while the dimension of the output data obtained from the decoding by filtering only the control flow information is around 1,5 Gigabytes. This amount of data produced is also related and proportional to the time required to produce it. It increases in a linear way but it reaches values extremely high.

Another example, which shows how the amount of data produced by the decoder can grow as the complexity of the traced program increases just a little bit, is the `nested-0.5s` nested loop program. It has a normal execution time of 0.5 seconds and the amount of data produced from the branch filtered decoding phase is around 53 Gigabytes, extremely high. This results show how the post-processing

phase of the Intel PT could result to be a big cumbersome in terms of memory management and space and also in terms of time required to eventually additionally post-processing it.

For less complex programs, on the contrary, the dimension of the data produced seems to be in the order of some Megabytes. This amount of data produced by the decoder result to be manageable in terms of space required to store them. Taking as example the `500for-ifelse` program, the amount of data generated by the perf decoder when extracting and reporting only the branches information is just almost 5 Mega Bytes. It is still a manageable and small amount of data.

Time and memory overhead final considerations

After reporting all the measurement results obtained in terms of time and data dimension when trying to extract the control flow graph of the execution of a program by means of the Intel PT, some consideration must be done on them.

As seen from the results, the Intel PT sampling during program execution seems to not affect so much its overall performance. On the contrary it emerged that the post-processing decoding operation could represent a potential cumbersome both in terms of time and memory space required. Although they increase linearly as the complexity of the program grow their increase result to be too much higher than expected.

In fact despite the post-processing time seems to be very high for more complex programs, it must be said that the Intel PT time overhead during run-time is quite low and acceptable, allowing the application to execute without a too high performance degradation. We can conclude that, based on the obtained results, the Intel PT tracing during run-time just adds a small additional time overhead to the program execution. In the case of the nested `for` loop programs (the maximum level of complexity measured) the additional time overhead was about 13 % of the normal execution time. Taking as reference this value the Intel PT tracing phase is very efficient, generating a very low performance degradation. This is a good result because it means that the program can run without being slowed down too much during its execution. On the other hand the post-processing decoding phase represents the bottle neck of the execution flow extraction procedure due to its high processing time and amount of data generated. The decoding phase, even when filtering and extracting just the control flow packet information, requires a too much time for its execution, resulting in a big cumbersome when it comes to more complex programs.

Therefore, even though the Intel PT usage, in order to reconstruct the exact sequence of executed commands, is feasible in tracing the execution of a program, its decoding phase has big limitation and it can be unfeasible when it comes to more complex programs. In the experiments conducted the program that seems to represent the pivot of shift between the still manageable complexity and the non-manageable one is the `for(8000000)` loop program. For the programs less complex than it the time and memory space required seem to be feasible and manageable, while for the more complex programs the computational resources required start to become unsustainable.

This let us conclude that the Intel PT is practical and feasible just for programs of low complexity. It must be remarked that this Intel PT scalability issue does not refer to its tracing during program execution, which is very lightweight, but it refers to the subsequent decoding process of the binary packets. The results obtained from the analysis show that there is an upper limit of program complexity that can be decoded in a feasible and efficient way. However, despite these limitations the set of less complex programs with which the Intel PT can be used in an effective ways includes a high number of applications and cases.

In the specific context of the binary extractor of this work the Intel PT has resulted to be a perfect tracing technique to monitor its execution. The C binary extractor is a very simple program and the Intel PT does a good work in extracting its execution flow graph, in a highly detailed way, allowing to verify if the binary extraction process was executed correctly.

Time and space limitations of the Intel PT decoding must be kept into consideration when using it into a solution, evaluating its impact based on the specific application.

Chapter 7

Discussions and critics

In this work good results have been achieved in the design of the integrity verification protocol solution. However there are some main limitations and points of improvement associated with it. This section reports the main limitations and problems encountered during the solution design and during the analysis conducted.

Multiple containers management As said at the beginning, this solution is designed and meant to attest the integrity of a container running on a virtual function. It focuses on attesting a single container without considering the possibility to attest multiple containers or Virtual Machines running on the same device. A next possible improvement could be to work in this direction trying to find ways and adding adaptations to this solution to make it working also for the attestation of multiple containers per virtual function scheme.

There are many issues and limitation that must be still solved when it comes to attestation of multiple containers on the same machine, especially when a virtual TPM is used. The first problem is basically related to the fixed and limited number of CPR register banks which sets an upper limit to the number of measurements that the TPM can store at the same time and consequently the number to containers that it could potentially manage.
The second problem is related to the new kinds of security issues that this new type of scheme opens. Having a TPM, that could be hardware of software, associated to the machine but shared between all the virtual machines, opens new security issues. Since the TPM is shared, in this scheme the PCRs value measurements associated to a container could be potentially read and modified by another container having in this way access to sensible information of the other VMs and

interfering on the final protocol outcome.

Intel PT limitations The other limitation encountered in this work is related to the Intel PT capabilities. Although it results to be a very precise and detailed tracing technique being able to produce the exact control flow graph of program execution at the assembly level, it has a series of limitations related to scalability.

The use of the Intel PT for the purpose of reconstructing the execution flow is feasible only for less complex programs. The tracing of a very simple program with such a technique is very efficient and does not add to much overhead in terms of performance and data space required. On the other side the tracing of more complex programs resulted to be inefficient and unfeasible in terms of time required for control flow extraction and amount of data produced by the decoder. These kinds of problems put an upper limit to the maximum complexity manageable by an Intel PT tracing solution. It must be made clear that these kinds of scalability limitations do not refer to the Intel PT tracing phase during run-time, which resulted to be lightweight and efficient, but refer to the subsequent post-processing decoding phase which, according to the results obtained, is a potential cumbersome and is costly and unfeasible in terms of time and space when it comes to complex programs.

For the purpose of this work the Intel PT appears to be very effective in tracing the C loaded binary extractor program because it is very simple. However, the use of Intel PT in tracing eBPF programs resulted to be a cumbersome and the tracing of a binary extractor based on eBPF hook seems to be highly inefficient due to the lack of direct mapping of instructions and due to the big amount of data generated. This shows another limitation of such techniques in reconstructing the execution flow graph of interpreted, python-like, programs where the direct mapping between the program instructions and the ones recorded by the Intel PT is not possible. This could be solved by an additional process of reverse engineering of the Intel PT recorded log instructions; however this additional post-processing operation increases even more the time required to reconstruct the final execution flow path and so its use should be evaluated both in terms of feasibility, correctness and precision.

The other aspect to consider, related to the Intel PT tracing, is that it is a pseudo-hardware based technique. This means that the assumption that is made here is on the presence of an Intel processor of last generation on the remote device. This assumption must be kept into consideration. However Intel PT tracing has been chosen because nowadays Intel processors are highly common and widespread in all devices. From this perspective no other additional hardware is required.

Considering the Intel PT deployed in a Control Flow Attestation solution context a possible problem could be arisen by compiler optimizations. The Intel PT is able to follow and trace a program with and without compilation optimizations. However if some kind of code optimization are applied to the program whose execution has to be attested, like loop-unrolling, the control flow graph representation of the program changes and so also the final computed measurement. When using such a solution the program version at the Orchestrator and the one at the remote device to be attested must be exactly the same for the integrity verification to succeed; otherwise if some optimizations are applied, and maybe there is a little discrepancy between the two code optimization versions, the remote attestation procedure will fail, even though the path executed is correct.

The last critic that must be done about Intel PT is that, potentially, it can be used in a Virtual Machine, but in the current available implementation it is not supported. Maybe with some adjustments and specific options, in virtual environments "linux kvm", Intel PT virtualization can be enabled in order to make it visible to `perf` tool. An additional effort should be done by Virtual Machine / containers developers in this direction to enlarge and strengthen the integration of Intel PT feature in virtualized environments.

A discussion should be done about the Control Flow Attestation procedures that might be chosen in an attestation scheme and the one that has been picked in the solution proposed here. There are two possible Control Flow Attestation schemes:

1. Hash check based
2. Graph traversal based

The first one, the *hash check based*, is the one used in this document, as also described earlier in the previous chapters. It this type of scheme the Orchestrator keeps stored all the valid hash measurement in the database, each of them corresponding to a valid control flow path of the program running on the remote device that has to be attested. Whenever the VM's software program is attested the remote device provides a measurement of the executed flow to the Orchestrator in order to be checked against the list of valid measurements stored in the DB. This kind of solution allows the Orchestrator to attest the execution of the remote program without having to access or store the binaries of that program. However, as it can be imagined the number of valid and possible paths increases exponentially as the complexity of the program grows. This means that the number of valid hash values of a program tend to be very high even for small programs. Additionally, if the number of remote VMs to be attested is big the amount of data that the

Orchestrator has to store into the database is huge and potentially unfeasible to manage. This kind of hash check based scheme is not a scalable solution at all.

On the other hand the *graph traversal based* Control Flow Attestation could be a new kind of technique that may try to solve the scalability issues of the hash check based one. In fact it does not require the Orchestrator to store an hash value for each possible valid path of the program. In this type of scheme whenever the VM is asked to attest the execution of its software running program it does not provide the hash measurement of the flow path executed, but furnishes to the Orchestrator the complete control flow graph, and so the exact sequence of executed commands. The AK of the TPM could be used to authenticate the control flow data generated through the Intel PT. At the Orchestrator side the sequence of commands received from the VM are used to traverse a Control Flow Tree, a tree representation of the software program, in order to check if its execution has followed a correct and valid path or not. The kind of tree that can be used for this purpose is the Merkle Tree, a hash based binary tree data structure used for efficient data verification. Following the sequence of commands traced by the Intel PT against the Merkle Tree of the target software program the attestation is performed by the Orchestrator. Since the Merkle Tree is a binary tree its traversal is efficient because at each step only one of the two path is taken, according to the tracing information, and the other one is completely excluded and so also all its subtrees. This type of scheme requires the Orchestrator to keep stored the binaries of the software program to be attested in order to be able to extract its Merkle Tree. In this scheme the Orchestrator does not have to keep stored all the single valid hash measurements, whose number could be potentially huge and impossible to manage, but it has just to keep track of the binaries and of the Merkle Tree, or another graph data structure, for each remote VM. The Merkle Tree is used as reference for the Control Flow Integrity of the remote program. This scheme requires a very less amount of data to be stored with respect to the hash check based solution. A possible way to do this is to use the xed (X86 Encoder Decoder) on the VM during decoding phase to directly reconstruct the exact sequence of assembly instructions to be sent to the Orchestrator. Since the assembly instruction reconstruction during decoding is highly expensive, a possible alternative is to just send the binary packets or the decode output to the Orchestrator and then let the Verifier perform the xed reconstruction of the assembly instructions to be than checked against the Control Flow Tree. This way is more efficient because the Orchestrator enhanced computing capabilities are leveraged resulting in a performance gain.

This second type of techniques to perform the Control Flow Attestation based on the graph traversal has not been used in the solution of this document but it could represent a better and more scalable alternative to the hash check based

one. The design of a scheme that goes in this direction could represent a possible enhancement to this type of context and to this solution.

Chapter 8

Conclusions

8.1 Conclusion

In this report we have focused our attention on remote attestation and integrity verification in a Cloud based Environment. As nowadays network schemes and architectures have changed following new trends, like Network Function Virtualization, and moving towards the predominance of new kinds of technologies, like the Mobile Edge Computing and the Cloud Computing, new kinds security issues come out. It has been made clear the urgent need of new solutions that are capable of addressing the new kinds of security problems, that Cloud and virtualization technologies open, related to the integrity and correctness of the remote edge systems connected to the network. This has been the initial starting consideration for this work.

The remote attestation schemes were used in the past to allow a device, acting as Verifier, to check the integrity of multiple remote edge systems. However, the past proposed solutions had many limitations in terms of security and did not consider the possibility of attesting remote virtualized containers. The previous remote attestation proposals just considered integrity verification during boot-time, or at most during load-time, leaving a big security hole not considering all the possible attacks that could happen during run-time. The other limitation of those solution is that they were not meant and could not work in a Cloud based Environment. The remote attestation solution proposed here has been presented to solve all the new kinds of security problems related to the attestation of Cloud based Environments. It has been designed with the scope of not only providing an attestation scheme working in a virtualized environment but also of assuring

strong security and correctness properties during the whole process.

The solution proposed in this report is able to provide a strong and verifiable evidence of the state and integrity of the remote Virtual Machine to the Orchestrator during run-time. As demonstrated the Orchestrator is able to ask the VM to provide an evidence of its system state, in order to verify its integrity, at any time and a multiple number of times during run-time without requiring the reboot of the container. The protocol solution proposed made use of a TPM module as trust anchor to enforce and ensure reliability and trustworthiness of the whole integrity verification procedure.

Two different types of protocols to achieve integrity verification have been presented in this document: the attestation by Quote and the attestation by Proof. The attestation by Quote follows the more classic remote attestation scheme where the Prover has to generate a quote data, over the PCRs storing the measurements reflecting the system state, and then send it to the Verifier in order to be attested. On the other side the Proof protocol follows a new type of scheme leveraging the TPM specific properties. It make use of the TPM authentication policy digest mechanism which permitted to have a more flexible property based attestation: the Orchestrator was able to attest different sets of properties for each VM in a customizable way by means of different policy digests.

Apart from the TPM security properties, used to cryptographically secure the overall protocol, ensuring authentication of the attestation data and strong cryptographic evidence of the Prover state, also the correctness and reliability of all the other steps of the overall attestation process have been assured. For this purpose the eBPF hooks tracer and the Intel PT tracing techniques have been used. These two different tracing techniques have been used to log and verify the correctness of the loaded binary extraction phase during run-time and the whole communication between the process implementing the integrity verification protocol with the TPM. Good results have been obtained from the use of these two kinds of techniques. They permitted to efficiently trace information at different levels of granularity: the eBPF gathering more high level data while the Intel PT legging very low level ones (at the assembly level).

A more detailed study has been conducted for the Intel PT. It has been used both for tracing the execution of the loaded binary tracer and the eBPF hooks program itself. In this way the two tracers capabilities have been combined together adding an additional level of security to the verification of the communication with the TPM. A separate analysis has been conducted on the Intel PT capabilities in the context of Control Flow Attestation. The experiments on the Intel PT have

been performed by using it to trace the execution of multiple C programs of increasing level of complexity in order to see how the Intel PT behave in terms of performance and scalability and to understand which is the highest level of complexity still manageable by such a technique. The Intel PT tracing phase resulted to be very efficient because it just added a small percentage of performance degradation during program execution. On the other hand, however, the subsequent decoding phase, required to reconstruct the control flow graph and the exact sequence of executed instructions, resulted to be highly inefficient and unfeasible when it comes to more complex programs. The Intel PT allowed to put in place an hash check based Control Flow Attestation scheme to perform the Control Flow Integrity check of the execution of a remote running software.

Overall these two tracing techniques resulted to be very effective and precise in the specific context of this work.

8.2 Future works

There are still some problems or point of improvement that have remained opened in the proposed solution.

As also stated in the previous chapter this solution has been developed and designed to attest multiple devices (or VMs) with a single container per virtual function. It has not been adapted to work with multiple containers per virtual function and so multiple virtual machines per devices. In case a virtual TPM is used a number of issues must be addressed. The PCR registers in a TPM are limited and this set an upper limit to the number of containers that could possibly be attested on the same machine. Moreover if the TPM is shared among multiple containers there are many problems related to privacy and confidentiality due to the fact that they all have access to the same TPM and so also to information of the other Virtual Machines. Some work should be done to extend the solution proposed to be working efficiently and reliably into a multiple container per virtual function scheme.

The other problem that should be solved regards the Intel PT decoding phase. As it came out from the tests conducted over the Intel PT the decoding operation of the binary packets, used to reconstruct the execution flow graph and the exact sequence of instructions of the executed program, is a cumbersome when it comes to more complex programs both in terms of time performance and of space required. The Intel PT tracing operation per se is very efficient and do not add too much overhead, on the contrary the decoder produces too much data and its

time duration is too high. Some more developments should be done on the decoder in order to reduce the time required to reconstruct the control flow graph of the program execution. The decoder should be enhanced by maybe adding some more filters to it that allow to select and drain only the information strictly useful in the context of Control Flow Attestation. More efficient and lightweight ways to decode the binary packets produced by the Intel PT must be found.

Another consideration regards the difficulty to integrate the Intel PT into Virtual Machines. More effort should be put in this direction in order to make the Intel PT more easily integrated into modern well-known Virtual Machines allowing the linux guest OS to interact and exploit the Intel PT feature of the processor.

The last improvement that can be added to this work is related to the Control Flow Attestation scheme. In the solution of this report the hash check based Control Flow Attestation scheme has been selected. However as already stated before this kind of approach has many limitations in terms of scalability. The Orchestrator has to keep stored, for all the multiple VMs, all the hash value measurements corresponding to the all the valid flow paths; the number of valid flow paths is very huge and increases exponentially even for low complexity programs. Another type of scheme that could overcome the hash check based Control Flow Attestation is the graph traversal one. This new kind of scheme allow the Orchestrator to just store a single tree representation for each VM's software program reducing both the space required to be stored and the time needed for the Control Flow Integrity check. A good data structure candidate suitable for this kind of solution is the Merkle Tree. A way to enhance this scheme even more is to have the Prover sending to the Orchestrator just the binary packets or the decode output and letting the Orchestrator, which has more computational power, reconstruct the actual sequence of assembly instructions through xed tool. Working and developing in this direction could allow to improve the solution proposed in this report arriving at a more scalable remote Control Flow Attestation scheme.

Bibliography

[1] Trusted Computing Group. *Trusted Computing Platform Alliance (TCPA).* 2001.

[2] Ahmad-Reza Sadeghi and Christian Stüble. «Property-based attestation for computing platforms: caring about properties, not mechanisms». In: Jan. 2004.

[3] Sailer, R., Zhang, X., Jaeger, T., Van Doorn, L. *Design and Implementation of a TCG-basedIntegrity Measurement Architecture.* 2004.

[4] A. Seshadri et al. «SWATT: softWare-based attestation for embedded devices». In: *IEEE Symposium on Security and Privacy, 2004. Proceedings. 2004.* 2004.

[5] Karim Eldefrawy et al. «SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust». In: *NDSS 2012, 19th Annual Network and Distributed System Security Symposium, February 5-8, San Diego, USA.* San Diego, UNITED STATES, Feb. 2012. URL: http://www.eurecom.fr/publication/3536.

[6] Joonho Kong et al. «PUFatt: Embedded Platform Attestation Based on-Novel Processor-Based PUFs». In: 2014.

[7] Trusted Computing Group. *TCG EK Credential Profile For TPM Family 2.0; Level 0.* TCG, 2014. URL: https://trustedcomputinggroup.org/resource/tcg-ek-credential-profile-for-tpm-family-2-0/.

[8] Will Arthur and David Challener With Kenneth Goldman. *A Practical Guide to TPM 2.0.* Apress, 2015. URL: https://trustedcomputinggroup.org/resource/a-practical-guide-to-tpm-2-0/.

[9] Moreno Ambrosin et al. «SANA: Secure and Scalable Aggregate Network Attestation». In: 2016.

[10] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, Gene Tsudik. *C-FLAT: Control-Flow Attestation for Embedded Systems Software.* 2016.

[11] Sofianna Menesidou and Thanassis Giannetsos. «Future Proofing the Connected World: A Quantum-Resistant Trusted Platform Module». In: 2018.

[12] Tigist Abera et al. «DIAT: Data Integrity Attestationfor Resilient Collaboration of Autonomous Systems». In: 2019.

[13] Nikos Koutroumpouchos et al. «Secure Edge Computing with LightweightControl-Flow Property-based Attestation». In: 2019.

[14] URL: https://trustedcomputinggroup.org/.

[15] *IBM's Software Trusted Platform Module*. URL: http://ibmswtpm.sourceforge.net/.

[16] IEEE. *Trusted Execution Environment: What It is, and What It is Not*. URL: https://ieeexplore.ieee.org/document/7345265.

[17] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3*. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.html.

[18] Intel. «Intel Trusted Execution Technology (Intel TXT): Enabling Guide». In: URL: https://software.intel.com/content/www/us/en/develop/articles/intel-trusted-execution-technology-intel-txt-enabling-guide.html.

[19] Linux TPM2 & TSS2 Software. *tpm2-abrmd*. URL: https://github.com/tpm2-software/tpm2-abrmd. Open source code.

[20] Linux TPM2 & TSS2 Software. *tpm2-tools*. URL: https://github.com/tpm2-software/tpm2-tools. Open source code.

[21] Linux TPM2 & TSS2 Software - Developer community. *tpm2-software/tpm2-tools*. URL: https://github.com/tpm2-software/tpm2-tools/tree/master/man. Open source code.

[22] *QEMU TPM Device*. URL: https://www.qemu.org/docs/master/specs/tpm.html.

[23] *Registry of Reserved TPM 2.0 Handles and Localities*. URL: http://www.trustedcomputinggroup.org/resources/registry.

[24] Trusted Computing Group. *TCG TSS2.0Overview and Common Structures Specification*. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TSS_Overview_Common_Structures_v0.9_r03_published.pdf.

[25] Trusted Computing Group. *Trusted Platform Module Library Part 1: Architecture*. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part1_Architecture_pub.pdf.

[26] Trusted Computing Group. *Trusted Platform Module Library Part 2: Structures*. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part2_Structures_pub.pdf.

[27] Trusted Computing Group. *Trusted Platform Module Library Part 3: Commands*. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_TPM2_r1p59_Part3_Commands_pub.pdf.

Appendix A

Integrity verification protocols commands

In this appendix section all the actual tpm2-tools commands, used for the implementation of both the Quote and Proof integrity verification protocols, are reported. Each command is labeled with the actor in charge of its invocation. The list of reported commands here follow exactly the sequence of steps described into the chapter presenting the implementation of the two protocol solutions (chapter 6). They can be used to try, verify and test the integrity verification protocols presented in this document.
It is important to remark that the following option must be added to each command in order to be sent to the software TPM:

```
-T mssim:host=localhost,port=port_number
```

A.1 Attestation by Quote commands

This is the exact sequence of commands used to perform the integrity verification by Quote protocol:

```
#AK creation
VM: tpm2_createek -c ek.ctx -G rsa -u ek.pub

VM: tpm2_createak -C ek.ctx -c ak.ctx -u ak.pub \
    -n ak.name -G rsa -f pem
```

```
Orch: echo "secret" > secret.data

Orch: file_size=‘stat --printf="%s" ak.name‘

Orch: loaded_key_name=‘cat ak.name | xxd -p -c $file_size‘

Orch: tpm2_makecredential -e ek.pub  -s secret.data \
      -n $loaded_key_name -o mkcred.out -T none

VM: tpm2_startauthsession --policy-session -S session.ctx

VM: tpm2_policysecret -S session.ctx -c 0x4000000B

VM: tpm2_activatecredential -C ek.ctx -c ak.ctx \
    -i mkcred.out -o actcred.out -P"session:session.ctx"

#Integrity verification
Orch: echo "nonce" > nonce.data

VM: tpm2_hash -g sha1 --hex file -o file_out # measurement computation

VM: read hvar < file_out

VM: tpm2_pcrextend 16:sha1=${hvar}

VM: tpm2_quote -c ak.ctx -l sha1:16 -m quote.out \
-s sig.out -o pcrs.out -g sha256 -q nonce.data

Orch: tpm2_checkquote -u ak.pub -m quote.out -s sig.out \
-f pcrs.out -g sha256 -q nonce.data
```

A.2 Attestation by Proof commands

This is the exact sequence of commands that has been used to perform the integrity verification by Proof protocol:

```
#Policy digest creation
Orch: tpm2_pcrread -o pcr.dat "sha1:16"
```

```
Orch: tpm2_startauthsession --policy-session -S session.ctx

Orch: tpm2_policypcr -S session.ctx -l "sha1:16" -f pcr.dat \
      -L policy.dat

#AK creation
VM: tpm2_createek -c ek.ctx -G rsa -u ek.pub

VM: tpm2_startauthsession --policy-session -S session.ctx
VM: tpm2_policysecret -S session.ctx -c 0x4000000B

VM: tpm2_create -C ek.ctx -P "session:session.ctx" -Grsa:rsassa:null \
    -u key.pub -r key.priv -L policy.dat -t ticket.key -d createdig.key \
    --creation-data=creation.data --template-data=template.data \
    -a "fixedtpm|fixedparent|sensitivedataorigin|restricted|sign"

VM: tpm2_policysecret -S session.ctx -c 0x4000000B

VM: tpm2_load -C ek.ctx -P "session:session.ctx" -u key.pub \
    -r key.priv -c key.ctx -n key.name

VM: tpm2_evictcontrol -C o -c key.ctx 0x81010002 #If the object
                                                  is not permanent
                                                  cannot be used for
                                                  activate credentials

VM: echo "secret" > secret.data

VM: file_size=‘stat --printf="%s" key.name‘

VM: loaded_key_name=‘cat key.name | xxd -p -c $file_size‘

VM: tpm2_policysecret -S session.ctx -c 0x4000000B

Orch: tpm2_makecredential -e ek.pub  -s secret.data -n $loaded_key_name \
      -o mkcred.out -T none

VM: tpm2_startauthsession --policy-session -S session.ctx
VM: tpm2_policysecret -S session.ctx -c 0x4000000B
```

```
VM: tpm2_activatecredential -c 0x81010002 -C ek.ctx -i mkcred.out \
    -o actcred.out -P "session:session.ctx"

VM: tpm2_policysecret -S session.ctx -c 0x4000000B
VM: tpm2_certifycreation -C 0x81010002 -c 0x81010002 -d createdig.key \
    -t ticket.key -g sha256 -o sig.nature --attestation attestat.ion \
    -f plain -s rsassa -P "pcr:sha1:16" -q policy.dat

Orch: tpm2_loadexternal -C o -u key.pub -c vm.key.ctx -n extname

Orch: tpm2_verifysignature -c vm.key.ctx -g sha256 -m attestat.ion \
      -s sig.nature -t ticket.key -f rsassa

#Integrity verification
Orch: openssl rand -hex 6 > nonce.plain

VM: tpm2_hash -C e -g sha256 -o hash.bin -t ticket.bin nonce.plain
    #The ticket is required during the sign

VM: tpm2_sign -c 0x81010002 -g sha256 -o sig.rssa nonce.plain \
    -t ticket.bin -p "pcr:sha1:16"

Orch: tpm2_verifysignature -c vm.key.ctx -g sha256 \
      -m nonce.plain -s sig.rssa
```