Andreas Gladik, BSc

Applying Remote Attestation for Assuring Runtime Integrity in an Industrial Environment

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Computer Science

submitted to

Graz University of Technology

Supervisor

Dipl.-Ing. Dr.techn. Christian Kreiner

Institute for Technical Informatics

Graz, November 2017

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Graz, November 3rd, 2017 _____

Date Signature

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz am, 3. November 2017 _____

Datum Unterschrift

Abstract

Embedded systems have become widely available during the last years and are now integrated in several devices, such as cars, industrial machines or electric power plants. Although there is a lot of ongoing research to secure embedded systems, there have been various successful attacks on these systems in the past [FMC11; MR12].

Nevertheless, the security of embedded Control Devices is still not taken into account very often, since they are mostly located in a private network and therefore meant to be secure. But such a misinterpretation can be very dangerous, especially if we think of Cyber Physical Systems like for example in hydro-electric power plants, where an attack could cause a black out or cost human lives.

Since it is nearly impossible to prevent every potential attack either caused by vulnerabilities in software or human failure, the goal of this work is to detect attacks in an early stage. This way counter measures can be applied to minimize the impact of an attack. In our approach we use an improved Remote Attestation protocol to ensure the binary integrity of the software running on an embedded Control Devices. This protocol is not only able to statically verify the system state, but it also recognises attacks at runtime. To obtain continuous information about the state of a Control Device we extended the Linux kernel. The improved Remote Attestation protocol also integrates well into the Scari framework [IRK+17]. Whenever malicious behaviour is detected, the improved Remote Attestation protocol hands over the system control to the Scari framework. Scari may then isolate the attacked device and moves the control tasks to a backup Control Device.

Kurzfassung

Eingebettete Systeme wurden in den letzten Jahren weit verbreitet eingesetzt und sind mittlerweile in vielen Geräten, wie Autos, Industriemaschinen oder Kraftwerken integriert. Obwohl es viele laufende Forschungsarbeiten zur Sicherung von eingebetteten Systemen gibt, waren in der Vergangenheit verschiedene Angriffe auf diese Systeme erfolgreich [FMC11; MR12].

Dennoch wird die Sicherheit von eingebetteten Steuergeräten noch nicht sehr oft berücksichtigt, da sie meist in einem privaten Netzwerk angesiedelt sind und somit als sicher erachtet werden. Eine solche Fehlinterpretation kann aber sehr gefährlich sein, besonders wenn wir an Cyber-Physikalische Systeme denken, wie sie zum Beispiel in Wasserkraftwerken zu finden sind. Dort könnte ein Angriff einen Stromausfall verursachen oder Menschenleben kosten.

Da es nahezu unmöglich ist, jeden denkbaren Angriff zu verhindern, der entweder durch eine Schwachstelle in der Software oder durch menschliches Fehlverhalten verursacht wird, ist es das Ziel dieser Arbeit, Angriffe frühzeitig zu erkennen. Auf diese Weise können Gegenmaßnahmen ergriffen werden, um die Auswirkungen eines Angriffs zu minimieren. In unserem Ansatz verwenden wir ein verbessertes Remote Attestation-Protokoll, um die binäre Integrität der Software, die auf einem eingebetteten Steuergerät ausgeführt wird, sicherzustellen. Dieses Protokoll ist nicht nur in der Lage den Systemzustand einmalig zu verifizieren, sondern erkennt auch Angriffe während der Laufzeit. Um kontinuierliche Informationen über den Zustand eines Steuergerätes zu erhalten, wurde der Linux-Kernel von uns erweitert. Das verbesserte Remote Attestation-Protokoll integriert sich auch gut in das Scari-Framework [IRK+17]. Wenn ein Angriff erkannt wurde, übergibt das verbesserte Remote Attestation-Protokoll die Kontrolle über das System an das Scari-Framework. Scari kann dann das angegriffene Gerät isolieren und die Kontrollaufgaben an ein Backup-Steuergerät weiterleiten.

Acknowledgements

This master thesis was carried out at the Institute for Technical Informatics, Graz University of Technology.

First, I want to thank my supervisor Dipl.-Ing. Dr.techn. Christian Kreiner for his advices and for introducing me to Dipl.-Ing. Dr.techn. Tobias Rauter, BSc who came up with this interesting topic. Many thanks to my advisors Dipl.-Ing. Dr.techn. Tobias Rauter, BSc as well as Dipl.-Ing. Johannes Iber, BSc for the great mentoring and advices. I also want to thank all the other people at the institute, who always tried to help me if I had questions.

Thanks to my girlfriend Isabella Plappart, for all her love and support. I thank my family, especially my parents, Horst Gladik and Marianne Gladik for their back up and their believe in me. Without them it would not have been possible to reach this point.

Graz, November 2017 Andreas Gladik

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this thesis we introduce a modified Remote Attestation (RA) protocol, that simplifies the deployment process of distributed systems. This is achieved by the use of certificates and signatures for the trusted applications. Additionally we want to detect and mitigate possible malicious changes in the system before any damage can happen. The improved RA protocol, we introduce, allows to continuously monitor the system state of an embedded Control Device (CD). Detected malicious changes can be used by the Scari framework [IRK+17] to adopt the system, which should mitigate the attack.

1.1 Motivation

In the past industrial machines were controlled by Control Systems (CSs) that were built for exactly one specific task. Nowadays embedded CDs are usually built with Commercial-Off-The-Shelf hardware and require an Operating System (OS) and software to work. On the one hand this comes with the advantage that they are cheaper than custom systems and may also provide more functionality in many cases. On the other hand more functionality may result in more attack vectors on the system. Moreover these systems are often prone to common attacks [DAD+11]. Usually they also have a network connection, since this makes it much easier to maintain them. But by connecting these Cyber Physical Systems (CPSs) it is also much easier to attack them. This is why security has become more important than it ever was. Especially as an attack on CPSs can potentially have a big impact: e.g. an attack on a hydro-electric power plant could result in a blackout or physical damage.

Attacks to CPSs, which have already happened in the past, show that it is nearly impossible to prevent them. If an attacker really wants to get into a specific system, there may always be an exploit. For example Stuxnet used several zero-day-exploits and was designed to destroy the CDs controlling centrifuges used for the enrichment of uranium [FMC11; Lan11]. The attack on the Ukrainian power grid in 2015 caused an outage for about 225,000 people [LAC16]. Obviously there are much more attacks targeting critical infrastructure and Supervisory Control and Data Acquisition (SCADA) systems as summarized by [MR12]. These attacks show us that it is nearly impossible to build a system that does not have a single vulnerability. A more detailed description of these attacks can be found in Section 3.1.

One approach to make CPSs as part of critical infrastructure more secure against

different types of attacks, is to detect a potential attack in an early stage. This way countermeasure can be applied before it comes to an impact or the infection can spread in the network. To do so, many different aspects of the system have to be monitored, like network traffic, application sandboxing or binary system integrity.

1.2 Goal

The goal of this thesis is to provide one building block to increase the security of CPSs by detecting malicious software changes within a system. This is basically done by an adopted RA protocol. In contrast to default RA our approach also detects changes at runtime. Furthermore, malicious changes in the system are reported to initiate countermeasures. One possible countermeasure would be to transfer the control tasks of the machine to another device and isolate it from the rest of the network. This would prevent the infection to spread to other devices in the CPS.

1.3 Outline

In Chapter 2 we give a short introduction to the basics, needed to understand the rest of our work. Next, in Chapter 3 we discuss work related to ours. After that Chapter 4 contains an overview over our domain. We discuss the problems and our solutions including a detailed protocol description. Chapter 5 contains a description of the implementation of our prototype of the RA protocol for embedded control devices used in hydro-electric power plants. This implementation can be used to simply verify the state of a remote host, that is also running the protocol, to build up a secure and trusted network connection or for a monitor connection. We also describe the requirements of the implementation to the system where it is supposed to run on. In Chapter 6 we evaluate our work concerning different aspects like performance, maintenance effort and data traffic. Finally, we give a conclusion and discuss open problems and what is needed to be done to use our protocol in a real system in Chapter 7.

Chapter 2

Background

This chapter briefly introduces the technologies our work is based on. Section **??** gives a short overview over the concept of Trusted Computing (TC), including all technologies and terms that are important for this work. TC defines a set of technologies to establish trust within a system. It is developed and supported by the Trusted Computing Group (TCG).

2.1 Trusted Platform Module

The Trusted Platform Module (TPM) is a hardware module that implements a set of cryptographic algorithms in hardware, like Rivest, Shamir und Adleman (RSA), Secure Hash Algorithm (SHA) 1 and Keyed-Hash Message Authentication Code (HMAC) [Tru11a]. It provides a non-volatile memory that is used as secure store for keys, as well as a volatile memory that consists basically out of Platform Configuration Registers (PCRs). What is special about these registers is that all of them can only be changed by the extend operation shown in Figure 2.1. By only allowing to change the values by the extend operation, the new value always depends on the old one. This can be used for building up a chain of trust like described in Section 2.3. Usually a TPMs has 24 registers, where the PCRs 0-15 can only be reset to zero by performing a system reboot. In addition, there is also a hardware Random number generator (RNG). There exist different versions of TPMs specified by the TCG. The latest TPM version is version 2.0, that adds support for SHA-2 256, Elliptic Curve Cryptography (ECC) and the Advanced Encryption Standard (AES) [Tru16].

PCR_Extend(i, extentvalue)

1 : $PCRi_{new} := hash(PCRi_{old}||extentvalue)$

Figure 2.1: TPM_Extend operation [Tru11a, p. 29] [Tru11b, pp. 160-161]

The extent operation takes the index i of the PCR and the value that will be extended *extentvalue* to the current value of this register. The new value of the register is the hash of the old value and the *extentvalue*.

2.2 Trusted Software Stack

The Trusted Software Stack (TSS) is an interface for accessing the low level TPM. There are several implementations for different platforms and programming languages.

TrouSerS: TrouSerS is an implementation of the TSS written in ANSI C and maintained by IBM [Lai+16].

uTSS: μTSS is a lightweight C++ implementation for embedded devices, Linux and Windows by Sirrix [SZ10]. The goal of μTSS is to simplify the TPM usage by removing overhead. Since it is proprietary software there is no public implementation available.

jTSS: jTSS is an TSS implementation in Java, developed and maintained at the Institute for Applied Information Processing and Communication (IAIK) at Graz University of Technology [IAI13]. It comes with a dual license, that means for Free and Open-Source Software (FOSS) projects the GNU GPLv2 license is applied. For all other types of projects, the "Stiftung SIC Java (tm) Crypto-Software Development Kit Licence Agreement" applies.

TSS.MSR: With TSS.MSR Microsoft actually provides two TSS implementations. One for C# (TSS.Net) and one for C++ (TSS.CPP) [MB17]. While the C++ version only supports Windows 8+, the .NET version is actually cross platform and can run on any device which is supported by the .NET Framework, .NET Core or Mono.

A detailed comparison of these TSS implementations can be found in Table 2.1 on the next page.

2.3 Integrity Measurement

When we are talking about measurement in the context of TC, we do not mean that we are measuring a physical unit. Instead of that we are talking about calculating a cryptographic hash of the complete static byte-representation of a file or memory region. Such a measurement can then be compared with other measurements. Either both measurements are the same, meaning that the measurement input was the same, or the measurements are not the same, which implies that the input was different. Besides that, no additional information can be derived from comparing two measurements.

To measure the integrity of a system everything that could probably influence the behaviour of a system must be measured. This starts at boot time with the Basic Input Output System (BIOS). Since the chain of trust starts with this measurement it is also called static Core Root of Trust Measurement (CRTM). This chain continues with the bootloader and the OS kernel. All these measurements during system boot up are security critical, since they are the base of the system. Only if this Trusted Computing Base (TCB) is trusted, it is possible to extend the trust to the whole system. After system boot, every executable, shared library, configuration file and so on have to be measured and stored in the Stored Measurement Log (SML). With the help of the SML one can build up a chain of trust. This is possible since every measurement is extended to the PCR on the TPM, which serves as root of trust. To be able to establish trust into a system, every entry in the

	TrouSerS	***μ*TSS**	**jTSS**	**TSS.Net**	**TSS.CPP**
OS	Linux	Windows XP+ Linux	Windows Vista+ Linux	Windows 8+ Linux macOS 10.11+	Windows 8+
TPM version	1.2	1.2	1.2	1.2, 2.0	1.2, 2.0
programming language	ANSI C	C++	Java	C#	C++
maintainer	IBM	Sirrix	IAIK	Microsoft	Microsoft
license	BSD	proprietary	Dual: GNU GPLv2 (FOSS)	MIT	MIT

Table 2.1: Comparison of TSS implementations

SML must by verified and trusted. In addition to that the value in the PCR must match the calculated value from the SML.

In Linux, this is done by the Integrity Measurement Architecture (IMA) which was originally implemented by Sailer et al. in [SZJ+04] and officially introduced in the Linux kernel 2.6.30 in 2009. The IMA creates measurements with so called template-hashes. Template-hashes are built by calculating the hash of a file. This hash is then extended by the file name and hashed again. The exact calculation and the used hash algorithm can be configured during the Linux kernel configuration. If a TPM is connected to the machine, the template-hash will also be extended to the PCR 10 of the TPM.

All measurements are stored in an IMA measurements list. To verify the authenticity of this list, several conditions have to be fulfilled. First, the template-hash for each entry must be recalculated and match the value in the list. Second, the template-hash of each entry is added with the extend operation to the previous result and the final result must match the value in the PCR number 10. The first measurement that is made by an IMA is always the boot aggregate which is a SHA-1 over the PCRs 0-7. These registers contain the measurements made during the system boot, such as the measurements of the bootloader or the BIOS. If no TPM is present during the boot time, the boot aggregate is zero. In addition to the measurement calculation, it is also possible to define the kind of files, which should be measured by the IMA. [Km15]

2.4 Remote Attestation

The basic RA protocol involves two parties. The remote host, which is going to be attested, is called Prover. The party, which wants to attest the Prover, is called Challenger. The goal of the protocol is that the Prover convinces the Challenger, that the Prover is in a trustworthy system state. This is basically done by measuring every binary that is loaded during the system boot, starting from the BIOS to the OS Kernel modules, applications and libraries and their configurations. Figure 2.2 on the following page shows the execution of a basic RA protocol.

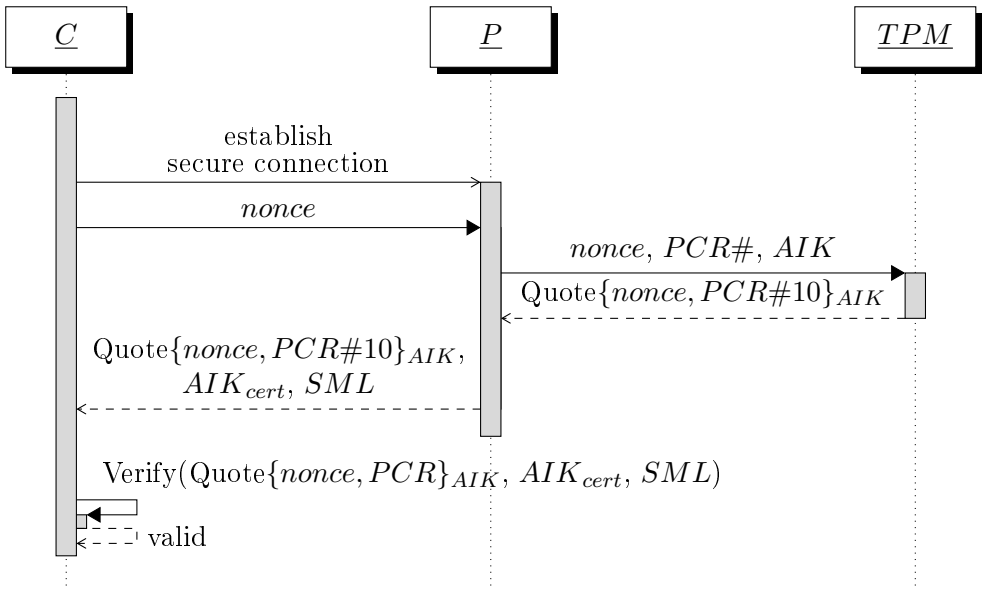

Figure 2.2: Basic Remote Attestation

The Challenger starts the RA protocol by building up a secure connection to the Prover. After the connection is established, a fresh unpredictable random number (nonce) is created by the Challenger and sent to the other party. The Prover loads the Attestation Identity Key (AIK) into the TPM and creates a quote of the PCR containing the system state and the nonce. A quote is a TPM operation that signs a structure containing the value of a single or multiple PCRs and an additional data payload with the AIK.
The quote structure, the signature of this structure, the AIK certificate as well as the SML is then sent back to the Challenger. The Challenger has to validate the signature of the quote. If the signature is valid, the Challenger has to compare the nonce of the structure with the previous created nonce to make sure that the quote is fresh. Only if both conditions are fulfilled the Challenger can be sure that the Prover was not able to cheat and the quote contains the current system state. Now the Challenger has to calculate the expected system state of the remote host with the help of the SML. If the calculated system state matches the value of the quote, the Challenger knows that the SML was not modified and is authentic. Now every entry in the SML must be checked in order to build up the chain of trust. This is usually done by comparing every entry with a database of trusted values. If all entries are trustworthy the Prover is in a trusted state. If only one entry is untrusted, the integrity of the remote host is probably violated.

Chapter 3

Related Work

In this chapter, we describe and discuss work related to our RA protocol. In Section 3.1 we describe previous attacks on CPSs, that were already mentioned in the motivation section of this work. In Section 3.2 we discuss work trying to mitigate masquerading attacks on RA. Section 3.3 covers possible ways for increasing the privacy for the Prover during RA. Section 3.4 describes a way how to not only attest the static load time but also the dynamic runtime integrity of the attested system. The approaches in Section 3.5 try to reduce the amount of trusted measurements by only taking measurements of components that are able to influence the behaviour of the system. An open standard that uses RA for controlling network access is described in Section 3.6. The problems and solutions of TPMs in combination with Virtual Machines (VMs) are discussed in Section 3.7. Section 3.8 covers two possible ways of how to isolate the integrity measurement component from the measured system in a virtualised environment with the help of a Hypervisor. Finally, Section 3.9 gives a short overview of how remote attestation can be done with the Trusted Execution Environment (TEE) of Central Processing Units (CPUs).

3.1 Previous Attacks on Cyber Physical Systems

In this section we describe some of the attacks already mentioned in the motivation of this work in more detail. A better understanding of previous attacks increases the understanding of the security problems we are facing today.

Stuxnet is probably one of the best known attacks on CPSs. In contrast to typical cyber attacks it did not try to steal or destroy information like in industrial espionage. It did not try to overtake the system to build up a botnet that can be used for other attacks. Instead it was designed to physically destroy specific machines. In this particular case centrifuges used for enrichment of uranium were attacked and only centrifuges in the Iran were destroyed. So, it was designed in a way that Stuxnet infected devices around the world, but only activated its harmful part after checking model number and program code of the infected CS to identify its target.

The goal of Stuxnet is to modify the behaviour of the infected Industrial Control System (ICS) in a way that they are slightly operating outside of its destined boundaries in order to let them fail after some time. This is done by injecting additional harmful code to the control loop, written in a way that the original control task can not notice any irregularities.

Another important fact we learn from Stuxnet is that the attacked CPS is not required to have a direct connection to the Internet. But how could the attacker infiltrate a facility with multiple faces and guarded towers? In fact this was very easy, they just threw a couple of infected Universal Serial Bus (USB) sticks on the parking lot of the facility. When the employees came to work, they found them, picked them up and took them to work. There they probably wanted to know what was on the sticks and plugged them into a PC. Once Stuxnet infected one device it could spread into the network and infect other CSs that where operating the centrifuges. [FMC11; Lan11]

There are two facts making Stuxnet to one of the first known Cyberwarfare weapons. First it was designed to destroy centrifuges used for enrichment of uranium that could be seen as a military target. Secondly the effort and quality put into it, makes it very likely that it was done by a big organisation that was financed by a government.

Another example for the impact of an attack on critical infrastructure is the cyber attack on the Ukrainian power grid in 2015. This attack was launched on three power distribution companies at the same time, causing an outage for about $225,000$ customers for several hours. This attack was initially started with stolen credentials that were stolen with the help of phishing mails. [LAC16]

And even if we assume that a system is secure and without vulnerability there is still man as a factor of uncertainty. Attacker use social engineering to create phishing mails that are hard to recognise as such. Even if everyone knows about phishing mails, these attacks are very successful, because most people believe that this does not happen to themselves. In fact [Sol15] found out that an attacker sends 100 phishing mails, 33 of them are opened and even more appalling, 11 recipients open the attachments.

To sum up, securing a system is important. However not every attack can be prevented. So, in order to secure a system, it is also important to detect successful attacks in an early stage to mitigate them or at least to know that there is a problem.

3.2 Masquerading Attack on Remote Attestation

One problem of the RA defined by the TCG is, that the Challenger is only able to verify if the quote was done by a trusted TPM. The Challenger is not able to be sure that it was done by the attested system. As shown in Figure 3.1 on the next page a malicious attested system (P_M) could simply forward all requests from the Challenger (C) to another system in a valid state (P_V). The good Prover has no possibility to distinguish a normal RA request from a forward one, since it basically only consists of a set of nonce. So the Prover quotes the nonce and sends everything that is needed to verify the system state back to the malicious Prover. Now P_M forwards all these data to the Challenger. Since P_V was in a valid system state, the Challenger will decide to trust the system. The problem is that Challenger can not distinguish between P_V and P_M. As a result, the Challenger will establish trust into a malicious system. This is caused by the fact that there is no technique present that allows to detect the forwarding of RA requests.

An obvious approach to solve this vulnerability would be to use the AIK to establish a secure connection and authenticate the involved parties. But this is not possible since in the TPM specification [Tru11a] AIKs are defined to be only allowed to sign data that is internally generated by the TPM. AIKs must not be used to encrypt or sign arbitrary

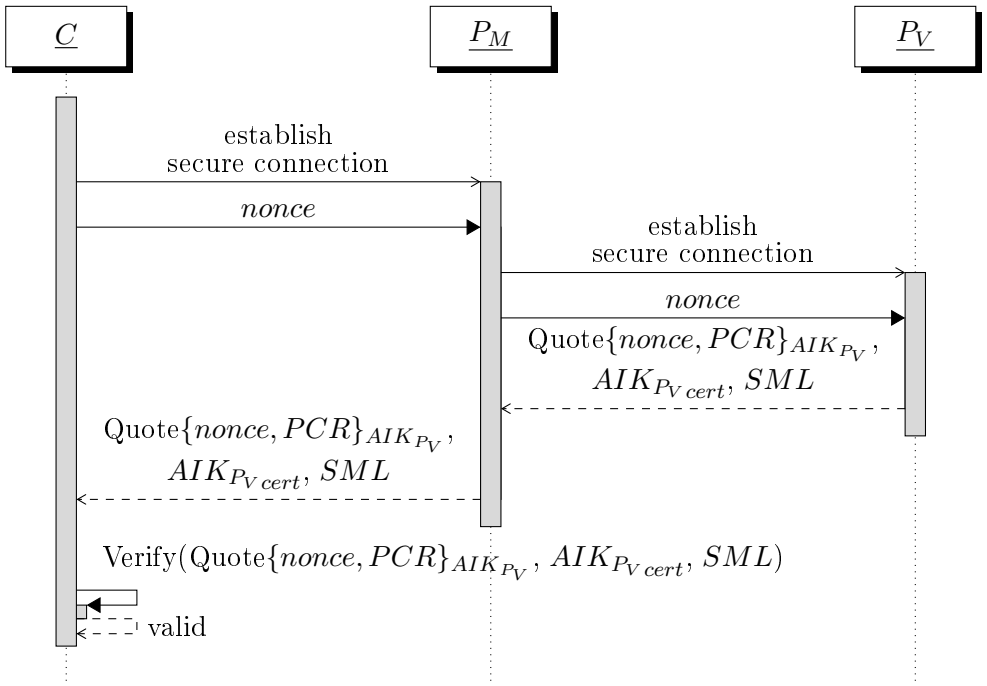

Figure 3.1: Masquerading Attack on RA

data. Since the private part of an AIK cannot leave the TPM it is not possible to abuse it for these operations.

There are several works available that try to solve this issue. The approach described in [GPS06] tries to establish a link between the RA and the secure tunnel endpoints. This is done by linking the Secure Sockets Layer (SSL) certificate to the AIK certificate. Since the SSL endpoint private key could become compromised this is not done directly. Instead an additional Platform Property Certificate is used for this task. This certificate contains the AIK public key and information from the SSL certificate like the domain name. It does not include the public key of the SSL certificate. This way there is no need to revoke the Platform Certificate in case of a compromised private SSL key. This approach has the disadvantage that the Challenger has to verify three certificate chains instead of only two. However, this can be changed by making the SSL certificate to a self-signed certificate since it does not provide any additional trust, when used together with the Platform Property Certificate and the AIK certificate.

The solution provided by Stumpf et al. in [STR+06] uses a different approach. This work extended the RA with an adopted Deffie-Hellman (DH) key exchange. As shown in Figure 3.2 on page 12 the Challenger starts with generating a fresh 160 bit nonce, like in the default RA protocol. In addition, the Challenger creates a new key-pair (K^C_{priv}, K^C_{pub}) with a generator g and group m on which both parties agreed before. Then a challenge with the nonce and the public part K^C_{pub} of the key-pair is sent to the Prover. The Prover generates the key-pair (K^P_{priv}, K^P_{pub}) and computes the session key K^{CP} from K^C_{pub} and K^P_{priv}. Now the Prover can generate a quote, but in difference to the default RA protocol the Prover does not simply use the nonce from the Challenger as payload. Instead the hash of the nonce and K^P_{pub} is calculated and used for it. After that the resulting quote,

K^P_{pub}, the SML and the AIK_{cert} are sent back to the Challenger. Since the Challenger knows the nonce and also has K^P_{pub}, the quote can be verified. In addition, the Challenger can be sure that K^P_{pub} really belongs to the Prover under inspection. If all entries from the SML are trusted and the PCR value from the quote and the value computed from the SML matches, the Challenger can be sure that the other party runs a trusted OS. Since a trusted system would not give away or leak private information, the Challenger can also rely on the fact that only the Prover is holding the private key K^P_{priv}. This means that only the Challenger and the Prover, that has generated the quote, are able to generate the session key. After the Challenger has computed the session key K^{CP} from K^C_{priv} and K^P_{pub}, the Challenger generates an additional nonce and sends it to the Prover. The Prover has to encrypt this nonce with the session key K^{CP} and sends the result back to Challenger. The Challenger encrypts the nonce with this session key. By comparing the result with the response from the Prover the Challenger can be sure that the Prover is also holding the correct session key. From this point on all further messages are encrypted with the session key. This way no malicious entity can read or modify any message.

In comparison to the solution from [GPS06] this protocol does not build a link between the SSL connection and the machine that has created the quote. This means if it is only used to attest a remote host without any further communication, a malicious remote host could still forward all messages from the protocol to a valid remote host. This way, the Challenger would still think that the malicious Prover is trusted. To prevent this, the protocol has to be extended by an additional step where the Prover has to send some information that can be linked to the SSL connection. If the use case just requires that only trusted hosts can read the data, which is sent to them the protocol can be used as specified by [STR+06]. For our use case this is not sufficient, since we want to identify the device that is challenged.

3.3 Property-based Attestation

As pointed out by Sadeghi and Stüble in [SS04] the Challenger is usually not interested in the exact configuration of the Prover. In fact, it is sufficient to know if the Prover satisfies specific properties or not. Therefore, Sadeghi and Stüble came up with the idea of a property-based attestation. This means that a Challenger only wants to verify that the other party offers certain properties in order to check if all required security requirements are satisfied. These properties are attested and certified by an additional Trusted Third Party (TTP). They describe several ways how this new TTP is integrated into the architecture and how a host can prove that his current configuration has specific properties. In general, these solutions can be categorised into three classes. The first two categories require hardware changes of the TPM, or software changes of the TSS. The last category does not require any changes in the existing TC architecture. All of these methods have their own advantages and disadvantages. But since changes to the TCG specifications are not an option for us, we only discuss the last category.

There are two proposed solutions for property-based attestation that are realizable with the current TCG specifications. Both add the TTP as a service that has to be called either on system state changes or during the RA process. This service certifies the properties of the current system configuration. This is done with a conventional (binary) RA. The

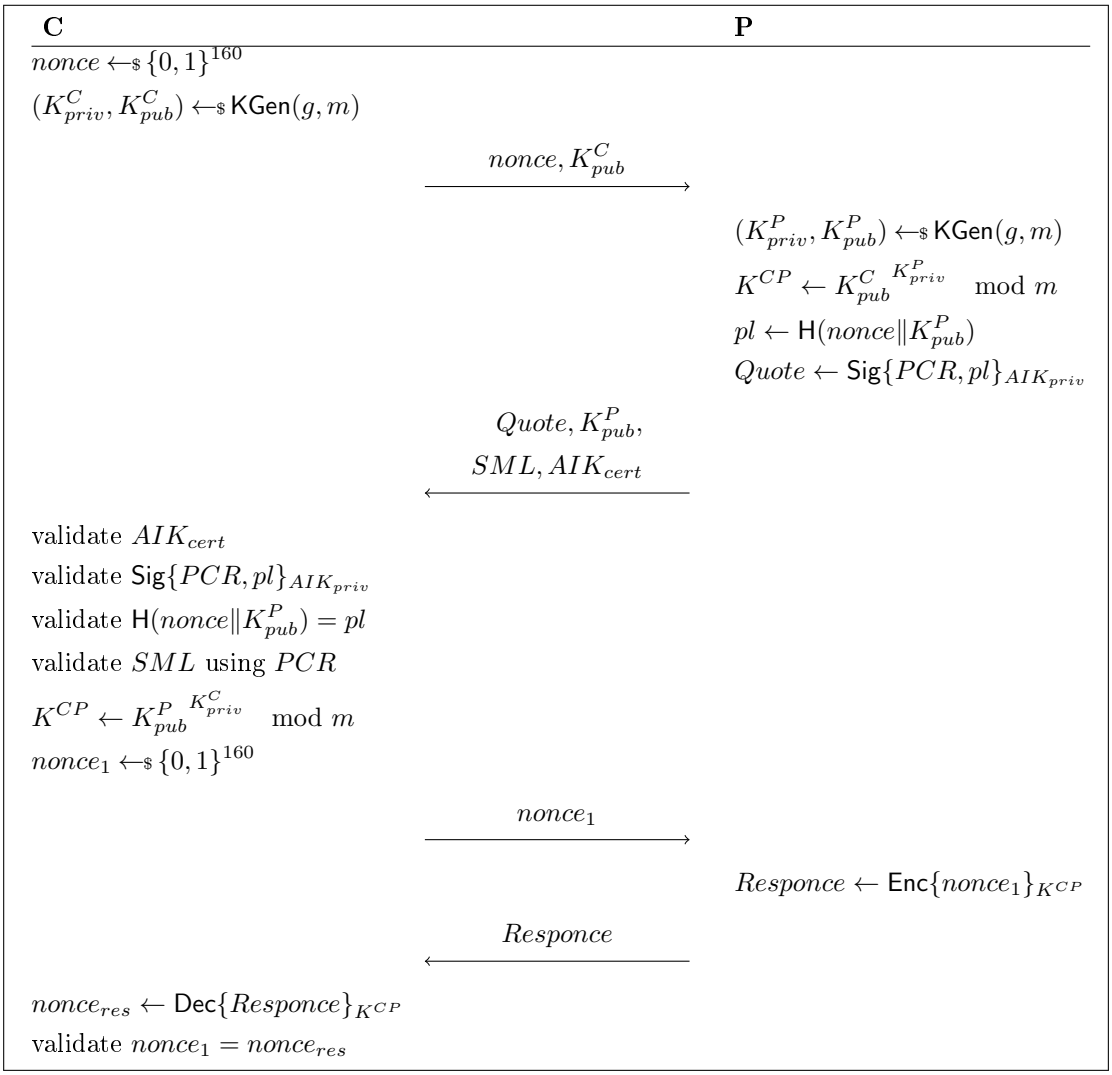

Figure 3.2: Robust Remote Attestation Protocol introduce by [STR+06]

TTP issues a property-certificate, which includes the system state (PCR value) for which it is valid and the properties which are fulfilled by this configuration. A Challenger can now attest a remote host by requesting a quote for the fresh nonce. But in contrast to the default RA the Prover does not return the SML. The SML is replaced by the property-certificate. In order to establish trust into the Prover, the Challenger has to verify several things: The signature of the quote as well as the property-certificate must be valid. The properties of property-certificate must match the required properties. The system state of the quote must match the system state from the property-certificate. Moreover, the nonce from the quote have to match the nonce sent to the Prover. Since the Challenger does not need to validate the SML, it is much easier to check if the Challenger can trust the Prover or not. But the downside of this approach is, that this complexity is only moved to the new service, which has to be trusted by the Challenger and the Prover.

The work by Chen et al. describes a property-based attestation scheme avoiding the need for a TTP [CLM+08]. This is done by using ring signatures. Before the actual protocol starts the Prover proposes a set of possible system configurations to the Challenger. The Challenger can now select a subset of trusted configurations $CS = \{cs_1, ..., cs_n\}$. In addition, both agree on a prime modulus P, a prime order Q and the generator g and h of a subgroup of $\mathbb{Z}_p^*$ of order P. To prevent the Prover from cheating, the discrete logarithm $\log_g(h) \mod P$ must not be known to the Prover. Assuming that the size of P and Q where chosen big enough, the discrete logarithm problem ensures the security of the protocol. During the protocol execution the TPM of the Prover creates and signs a commitment of the current system configuration $cs_{\mathcal{P}}$. This commitment is then used to create a ring signature. By verifying the commitment, the signature of the commitment and the ring signature the Challenger can be sure that the committed configuration is in CS, without revealing which one it was. For creating and signing of the commitment a new TPM command would be needed. But this could be achieved by some firmware changes. But even with this approach it might be possible that a TTP is needed in some cases. The main problem is that the Prover and the Challenger have to agree on trustworthy system states. Deciding which state is trustworthy and which is not, might be difficult or even impossible. In these cases, both parties must rely on a TTP for this decision. This method has the advantage that the Prover does not reveal its exact system state, increasing his privacy, while the Challenger is still be able to establish trust. But in order to achieve this privacy there are a few important things needed. First of all, the size n of CS has to be large enough to prevent the Challenger from guessing the configuration $cs_{\mathcal{P}}$. But this can be easily ensured by the Prover itself by simply not proceeding with the protocol if the selected set of system states CS is too small. The second thing is, that the Prover also has to ensure that the Challenger cannot learn anything about its actual system state by simply executing the protocol several times, while selecting a different set of system states CS on every run. The detailed steps of the protocol can be found in Figure 3.3 on the next page.

3.4 Dynamic Property Attestation

The default RA, as well as most modifications of it, like the property-based attestation from Section 3.3, only use static system properties like program binaries. They are called

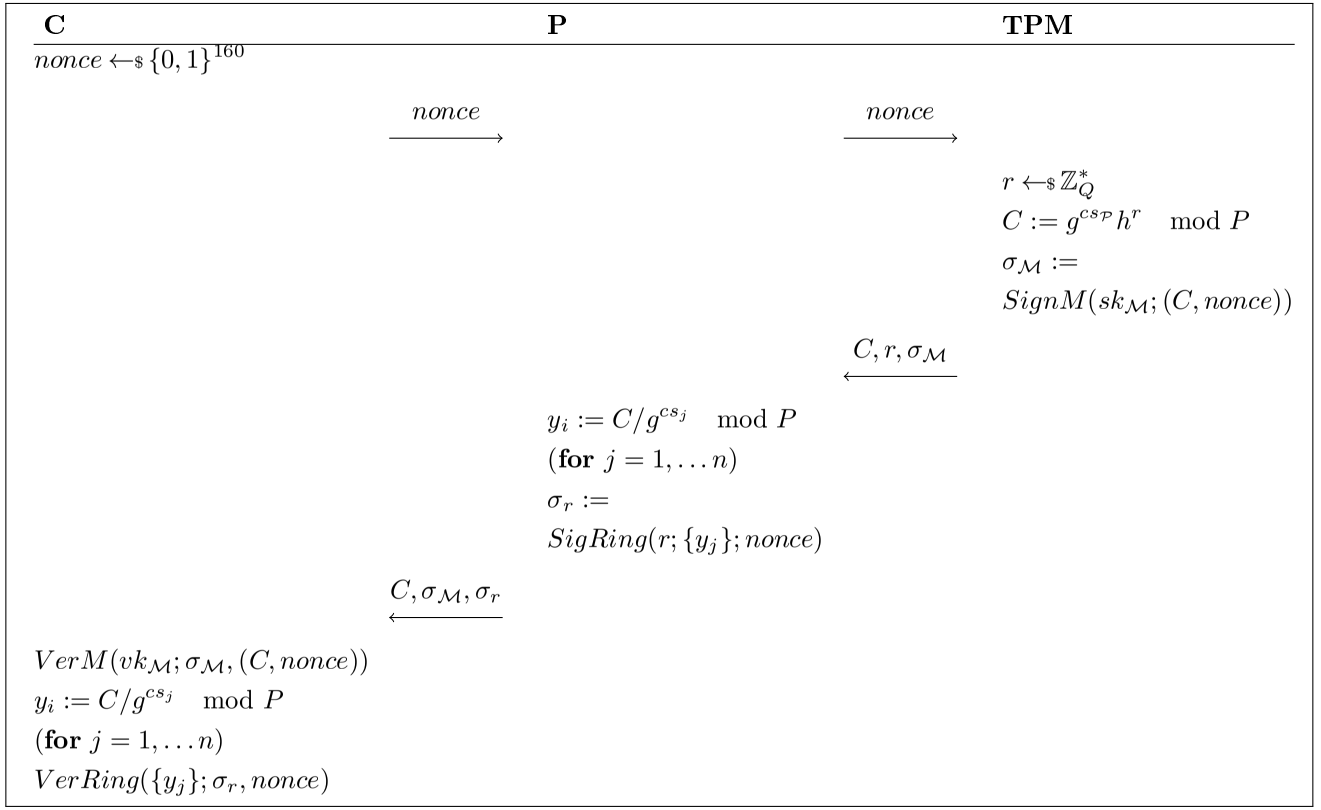

Figure 3.3: Property-based attestation without TTP from [CLM+08].

static, because as long as the binary is not changed the measurement will always be the same static result. While an application is running, the memory is not static anymore, since the stack as well as the heap are changing dynamically. Attacks that change the runtime behaviour of a process by modifying return pointers can therefore not be detected by these techniques. Thus, a RA protocol that uses dynamic runtime properties was introduced by Kil et al. in [KSA+09] to attest the runtime integrity of a remote host. Only if all dynamic properties of a process are fulfilled during the whole execution time, the process is trustworthy. In order to establish trust into a system, this must hold for all processes that are running on it. There are many possibilities for dynamic properties. But since these properties have to be monitored in a way that they are evaluated before they are changed again, not all of them can be used in an efficient way. Since finding a way to monitor such a dynamic property in a way that does not add too much overhead to the system execution is not a trivial task. The work of Kil et al. focused only on the following properties:

Structural Integrity: This dynamic property describes structural characteristics that must be fulfilled by the memory objects of a process. Such structural properties are for example that the function return address on the stack must always point to the address of the original CALL instruction. Another example is the frame pointer acting as the base address of the stack frame. Like the return pointer it is not allowed to change during the function execution. But there are also many constraints for the heap that must be fulfilled. One example would be that allocated memory chunks must always be linked together by meta data with a linked list. To find structural constraints for a specific application a static analysis was applied to the binaries. This is done by a dynamic property collector for every application.

Global Data Integrity: This category concerns global variables in a process. Even variables must satisfy certain properties during runtime, these properties concern their value or relations to other variables. So, the value of a constant variable is not allowed to change during execution. Or two variables must have the same value at a particular execution point. The problem with these constraints is that they are very difficult to obtain from an application in an automated way, since they may depend on the program input and the exact execution path. To solve this problem each application runs multiple times with different input during a training phase. The generated data traces from this training phase are then used to obtain the dynamic properties. In order to not obtain incorrect constraints from this training phase, which would result in false positives during the system runtime, it is important to use a good training set. The idea of Kil et al. was that if the program source code is fully covered during the execution of the training set, it should be good.

During the system runtime the obtained properties are then monitored by an integrity measurement component that is implemented as Linux Kernel Module (LKM). Whenever a system call is fired the measurement component interrupts and monitors the application properties. In addition, whenever a function that effects the heap is executed this is also interrupted to check its properties. Doing so guarantees that an attack is detected before it can take effect. If a violation has been detected the evidence is stored and extended to the PCR 8 of the TPM. This PCR acts as a kind of violation flag, whenever it does not

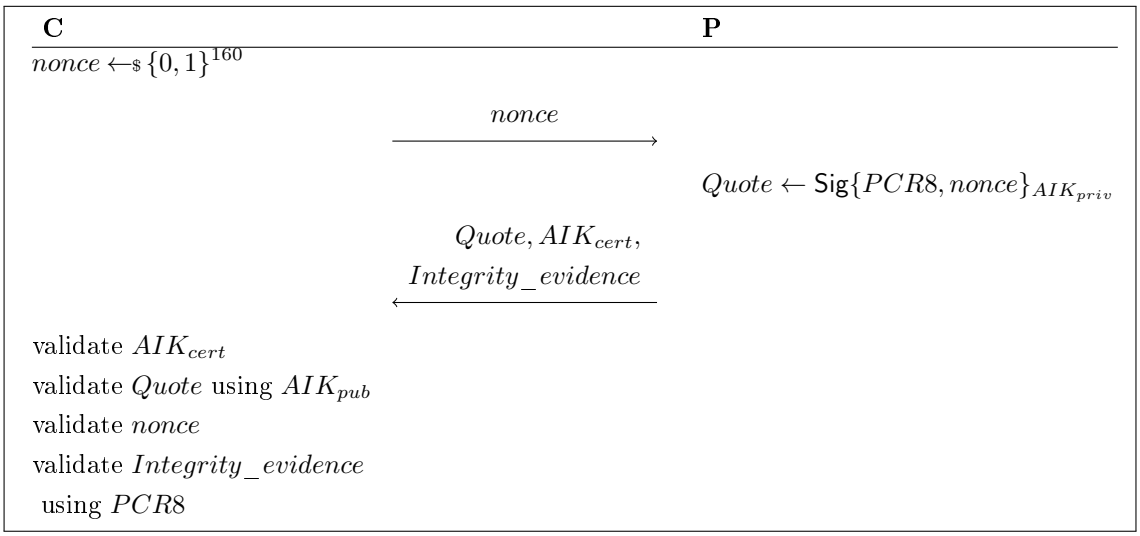

Figure 3.4: Dynamic-property-based RA by [KSA+09].

have the default value, the system is not trustworthy anymore. Since the PCR 8 is one of the registers that cannot be reset and its value is obtained through the quote command, an attacker cannot hide the violation during RA. Like shown in Figure 3.4 The dynamic RA process works similar to the default RA protocol. The Challenger starts by sending a fresh set of nonce to the Prover. The TPM of the Prover creates a quote on the PCR 8 and the nonce from the Challenger. After that the Prover sends the integrity evidence list, the quote and the signature back to the Challenger. The Challenger checks the signature and the nonce. If both are ok, the integrity evidence list is validated with the help of the value of the PCR 8. By validating the quote, the Challenger can be sure that the Prover was not able to modify the value of the PCR. In order to establish trust into the other system the Challenger must also be sure that the Prover runs an unmodified integrity measurement component. To ensure this the Challenger has to run the default RA protocol to attest the static boot and load-time integrity. In other words the dynamic property attestation does not replace the static binary attestation, in fact it is an extension to it. One problem of [KSA+09] is that in order to obtain all dynamic constrains for an application, not only its binary is needed, but also the source code must be available.

3.5 Policy-based Attestation

Another problem of most RA solutions is that the amount of measurements that have to be verified and trusted is very high. The system is either completely trusted or if there is only a single unknown measurement, the entire system is untrusted.

To reduce the amount of measurements that have to be verified and allow untrusted objects on a system, Jaeger et al. came up with a solution that used an information flow integrity approach named Policy-Reduced Integrity Measurement Architecture (PRIMA) [JSS06]. The basic idea is that the Challenger only has to check the measurements of trusted objects and their dependencies. As long as there is no read to an untrusted object the system can still be trustworthy. This is done by using Security-enhanced Linux

(SELinux) that enforces Mandatory Access Control (MAC). The MAC policies are then used to get the information flow of applications. This information flow data does not only contain dependencies between processes and static data, that can be measured by the IMA, but also for example untrusted dynamic network input. By using the Biba integrity model [Bib77] this would lead to an integrity violation. Since for certain applications such inputs should not have an impact on their trustworthiness, a modified version of the Clark-Wilson integrity model [SJS06] was used. This allows to add filters between specific applications and untrusted inputs that allow this dependencies without an integrity violation.

A comparison of load-time, Biba and Clark-Wilson-Lite integrity is shown by Figure 3.5 on the next page. The load-time integrity model does not take relations between processes into account. If there is a low integrity process the integrity is always violated. Moreover it does not handle network data that is usually also untrusted. In contrast to that the Biba integrity model uses data flows between processes. A high integrity process is only allowed to use data from other high integrity processes. If there is a data flow from a low integrity process or from a network to a high integrity process, the integrity is violated. Since the use of network data or data from low integrity processes should be allowed in some cases, the Clark-Wislon-Lite integrity model adds the possibility of filters. These filters are used to filter untrusted data from a specific source to allow this dependency without integrity violation.

To allow the Challenger to get and verify all this information the IMA measurement list is extended to measure the following things:

Trusted Subjects: These subjects that must be trusted by the Challenger to establish trust in the remote system. Trusted subjects are extracted from the MAC policies and build the TCB of the system.

Trusted Code/Data: Code and data used by the trusted subjects. The entry in the measurement list contains not only the data itself, but also the subject. This mapping information is needed to verify the information flow graph.

Information Flow: Since the information flow is constructed from the MAC policies they have to be measured during the boot process. This allows the Challenger to verify the correctness of the information flow model.

To establish trust, the Challenger needs to verify the measurement list. In addition, the measurement list can be used to extract the trusted subjects and the MAC policies. With this information the information flow graph can be constructed and then used to identify trusted and untrusted subjects.

To establish trust into a remote host the Challenger has to verify that the Prover runs the expected MAC policies. After that the extracted information flow is analysed by checking its dependencies of every trusted subject in it. To establish trust into a system all trusted subjects are only allowed to use other trusted subjects or trusted code and data. Furthermore all the corresponding measurements must be verified and valid. If there is a dependence to an untrusted application or data there must be a filter interface in the trusted subject and a filter implementation in the code of the subject to establish trust.

The solution of [JSS06] has two limitations. First it does not handle the possibility of MAC policies updates during the system runtime. And second like most RA solution

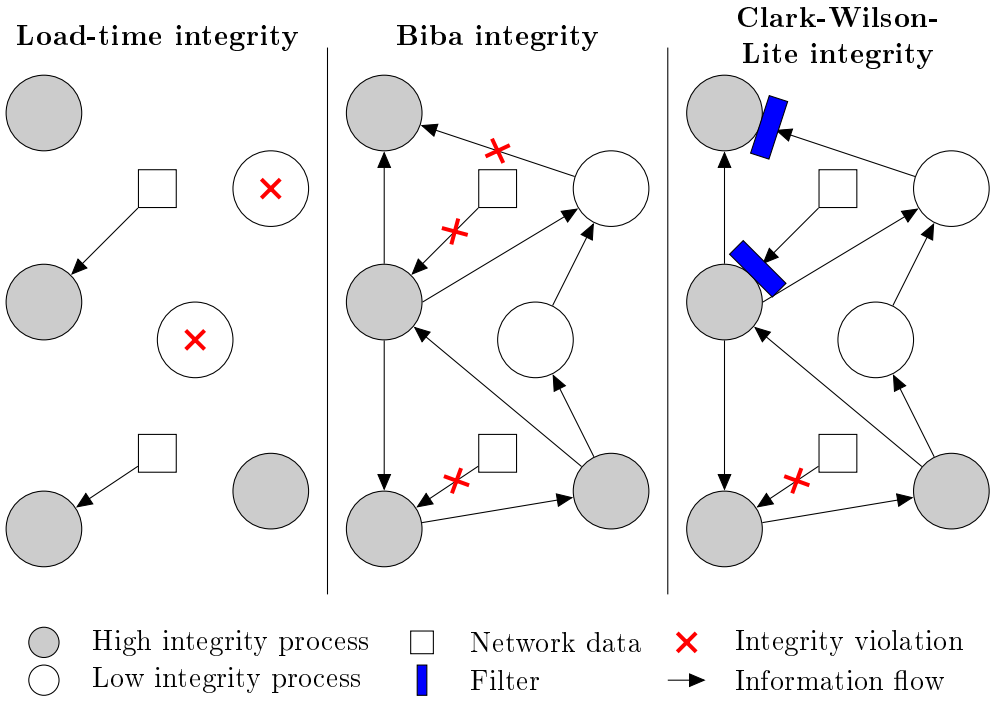

Figure 3.5: Comparison of integrity models. Figure adapted from [JSS06, page 22]

it only allows a binary trust decision. To overcome this limitations Xu et al. extended the information flow approach with a domain-based integrity model in [XZH+12]. Like in PRIMA the MAC policies of SELinux were used to obtain the information flows. They also used a Clark-Wilson integrity model with filters, to allow high integrity processes to access processes with lower integrity. Trusted objects are divided into two categories. The system domain (TCB_s) contains everything that must always be trusted, for example the kernel of the OS, policy enforcement or the IMA. The second category is the application domain (TCB_d) that contains every subject that can possibly have an impact on the trustworthiness of a specific application running on the attested host. All subjects that are not in TCB_s or TCB_d are called NON-TCB. The TCB_s domain is protected if there is no information flow from NON-TCB or TCB_d without a filter. For an application domain TCB_d all information flows must be from TCB_s or TCB_d. If there is an information flow from NON-TCB there must be a filter for this to protect the domain. To reduce the amount of system states measured by the TPM, not every binary measurement is directly extended to the PCR. Instead of that the measurement architecture maintains separate lists for trusted subjects, code, policies, filters and processes. To allow the Challenger to verify these lists, only these lists are measured and extended to the PCR or the TPM. Whenever there is a change in one of the subjects of TCB_s the system has to be rebooted and remeasured, since TCB_s also contains the measurement architecture itself. After that TCB_s is measured as the initial system state T_0. If there is a change in TCB_d during runtime, it will be measured and added to a new system state T_i. The system also introduces a way to rank integrity violation and calculate a risk level of it. This risk level should reflect the trustworthiness of the Prover. If the risk level is zero there was

no integrity violation and the system can be completely trusted. But since they did not clarify how values greater than zero should be treated by the Challenger to decide if trust can be establish or not, we are not going into detail here.

Both methods discussed so far have one disadvantage, they rely on already well predefined MAC policies in order to work correctly. In addition, the work of Xu et al. does not clearly point out if it is possible to have different application domains that have a different trustworthiness running on an attested system at the same time. To come around this limitations Rauter et al. proposed a privilege-based RA solution in [RHK+15]. Since this method does not need predefined MAC policies, the privileges of an application have to be obtained from another source. The work proposes two different solutions for doing so. Both use the application binary to retrieve this information. The first solution uses the program call graph to search calls to libraries and resource accesses in it. In case of a resource access it figures out if this is a read or write operation and which file is used. If it can not figure out this information, the highest possible privilege is used. This means that for example undefined file accesses are handled like a write operation to the whole file system. The second proposal tries to find known symbols to system libraries directly in the application binaries. This method has the disadvantage that it does not provide the parameters that are used for these calls. But execution time measurements that were taken by Rauter et al. show that it is much faster than the first method. Since the resource access information has to be extracted and measured during runtime, the performance of the used method is significant. In addition to this, the correctness and completeness of the obtained access information is critical for the solution, to allow a Challenger to do correct reasoning about the trustworthiness of a remote system. To check the integrity of the information flow the Biba model is used. Since this would not allow network access at all, like seen in Figure 3.5 on the preceding page, network access for a specific module can be permitted by adding a policy for it.

To build up a TCB a set of privileged modules has to be defined by the administrator of the system. This set contains all modules that have to be trusted in any case. Usually this contains the whole OS since every application is running on top of it. For all binaries in this set of privileged modules privileges are not measured any more. This means if there are no privilege measurements for a module, this automatically means that this is a privileged module that belongs to the TCB.

The systems extents the IMA to measure binaries and privileges. Both measurements are stored in one list, that contains the name of the measured module, the measurement type and some attributes. There are two types of measurements. The first type is the binary measurement that is already known from the IMA. In the case of a binary measurement the attributes field of the resulting entry in the measurement list contains only the measurement hash. The second type is the privilege measurement. Like already mentioned this type of measurements are only done for non-privileged modules. The attributes field is here used to store the Resource Access Descriptions (RADs) that are obtained from the binary. Basically, the systems supports two different types of RADs. The first is a network access RAD and does not contain any further information. The second is a file access RAD that contains the file or folder that is accessed and the type of operation (read or write). Like in the original IMA a PCR of the TPM is used to store the verification hash of the measurement list.

Like in the other RA protocols the Challenger starts by sending a fresh nonce to the

Prover and receiving the quote for the PCR and the nonce and the measurement list. If the measurement list was not manipulated it can be used to verify the trustworthiness of a system or a specific service of the Prover. To do so the Challenger has a list of so called communication policies that has one entry for every module of interest. This policies contain varied information about one module. For example, if a module is privileged or not. In addition, these policies contain file dependencies with the allowed mode of operation. Dependencies to other modules that are able to influence the behaviour of this module, are also part of the policies. Finally these policies also define if network access is allowed for a module or not.

These policies are used to obtain a list of modules that must be fully trusted. This list contains all modules that are marked as privileged and all services that are used by the Challenger. Then all their dependencies are added. For all modules in the resulting list a binary integrity check has to be performed. This is done by comparing the binary measurement of the module in the measurement list with a set of trusted measurements.

In addition, the resource access rules for all modules have to be checked. If there is a rule violation of a privileged process, it is added to the dependence list. If the process belongs to the non-privileged processes the verification fails. Like before this is done with the help of the measurement list, but this time the privilege measurements of the modules are used.

3.6 Trusted Network Connection

The Trusted Network Connection (TNC) is an open standard for controlling network access and is specified by the TCG in [Tru12]. The idea of TNC is that only endpoints that are trusted should be able to gain full access to a network. This is done by checking the integrity and the identity of this entity, which is basically done by RA. If an endpoint is not trustworthy, it can be isolated from the network. Since this isolation is done on layer-2 and layer-3 of the Open Systems Interconnection model (OSI model), switches and routers with support for the TNC standard are needed. The TNC specification defines multiple roles with different sets of functions that must be provided by these entities and can be found in Figure 3.6 on page 22:

Access Requestor: The entity wanting to have access to the network. In order to be able to gain access, several functions must be provided. A Network Access Requestor (NAR) initialises the process of getting access to a network on the Network Access Layer. On the Integrity Evaluation Layer a TNC Client is needed to provide integrity information. And on the Integrity Measurement Layer there have to be Integrity Measurement Collectors (IMCs) providing integrity measurements.

Policy Enforcement Point: For example, switches and routers that have to ensure that the security policies are realized. They are also called Network Access Enforcer (NAE) and operate only on the Network Access Layer.

Policy Decision Point: Decides if an Access Requestor should gain access to the network and what action should be taken. To be able to do so, the Policy Decision Point has corresponding functions to the Access Requestor on every layer. On the Network

Access Layer a Network Access Authority (NAA) is needed. On the Integrity Evaluation Layer a TNC Server collects integrity information and makes decisions based on this information. On the Integrity Measurement Layer Integrity Measurement Verifiers (IMVs) are needed to process the integrity measurements from the Access Requestor by comparing them with stored valid measurements.

Metadata Access Point: Gathers and shares information about the state of TNC elements.

Metadata Access Point Client: Can provide or consume information about Access Requestors from the Metadata Access Point. For example, this can be information about network activities of Access Requestors.

For each of these roles there exists hardware or software that implements the needed functionality from several different companies. Hence, one big advantage of TNC is that one is not depending on a single manufacturer. On the other hand, the TNC architecture is very complex and involves several roles and entities.

The process for a TNC Client to get access to a TNC network is shown in Figure 3.7 on page 23 and works as follows. Before the actual process can start, the client loads each Integrity Measurement Collector (IMC) that is relevant (0). The same holds for the TNC Server, that has to initialize each Integrity Measurement Verifier (IMV) (0). The protocol starts with the Network Access Requestor (NAR) sending a connection request to the Network Access Enforcer (NAE) (1). At this point a network access decision request is generated and sent to the Network Access Authority (NAA) (2). Receiving this request, the NAA checks the user authentication, platform credentials and the integrity of the requester. If one of these checks fails the connection is refused. While the user authentication check is done by the NAA itself, it sends a request to the TNC Server to perform the other checks (3). The TNC Server starts with the platform credentials authentication by checking the validity of the AIK certificate (4). To perform the integrity handshake the TNC Server informs the IMVs about this (5). The same thing is done by the TNC Client and the IMCs (5). The IMCs return some messages containing measurement data to the client. The integrity measurement information is then exchanged between the TNC Client and the TNC Server (6A). The client uses the IMCs to retrieve the requested information, while the TNC Server uses the corresponding IMV to analyse it. If a IMV needs more data from an IMC, a request is sent over the TNC Server to the TNC Client (6B). The client uses the IMCs to retrieve this information and sends it back. Once the IMV has enough information, the TNC Server is informed about the evaluation result (6C). As soon as the TNC Server has completely finished the integrity handshake, a recommendation is sent back to the NAA (7). Based on this recommendation the NAA can now decide if the network access will be granted to the Access Requestor or not. The final decision is sent to the NAE and the TNC Server to implement it (8).

3.7 Virtual TPM

Concerning virtualisation, it is not possible to simply pass through the TPM, since usually only one TPM is present on a machine, while every VM would need its own. An additional

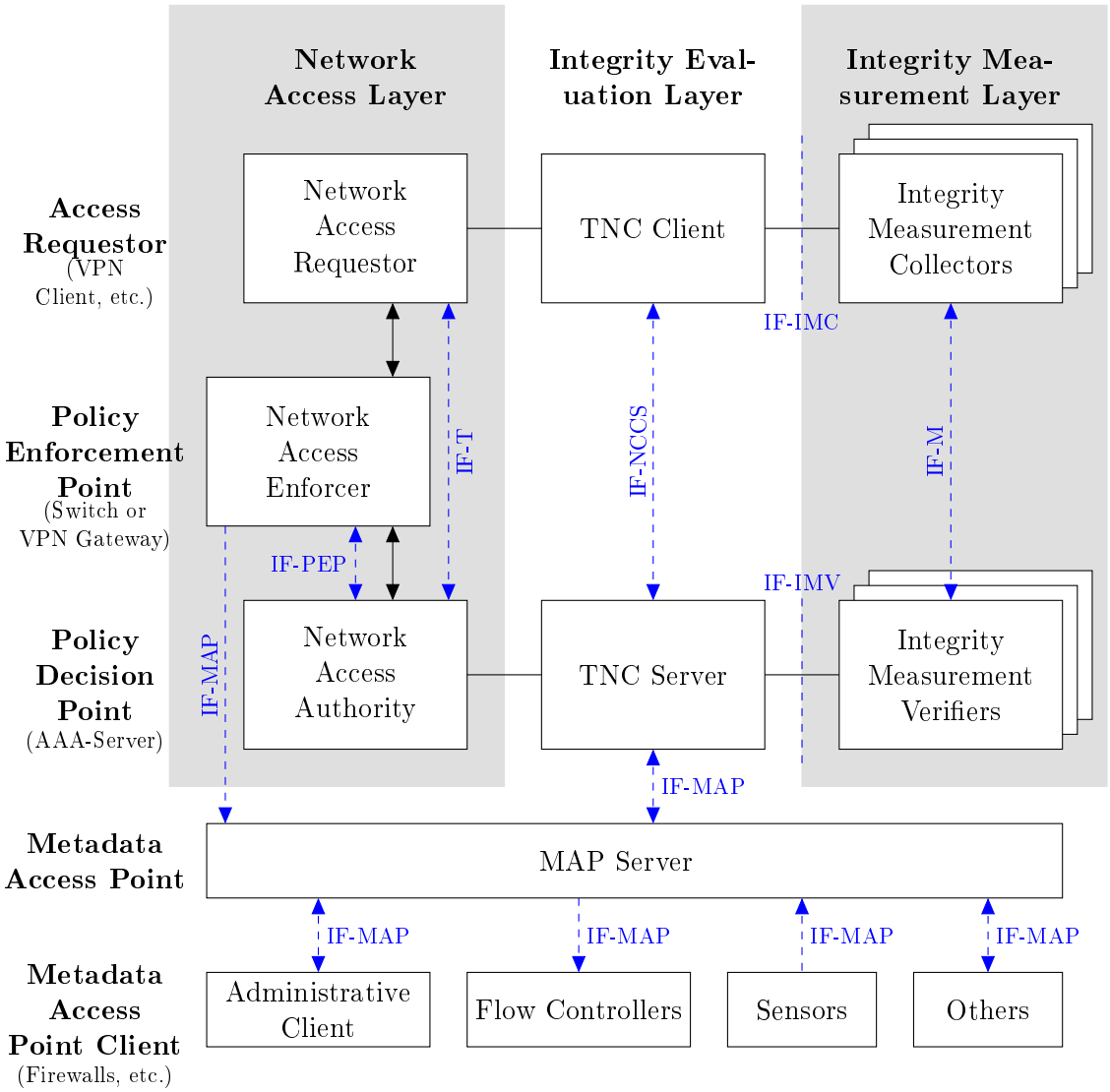

Figure 3.6: TNC Architecture. Figure adapted from [Tru12, page 18]

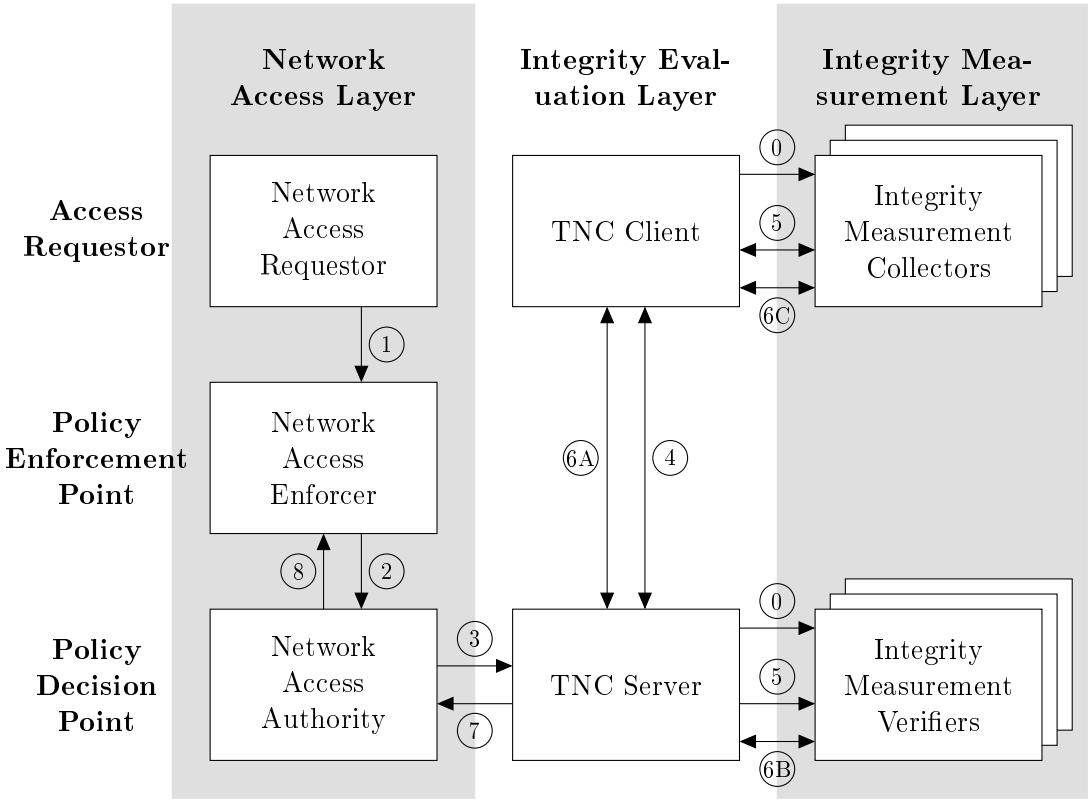

Figure 3.7: TNC granting network access process information flow. Figure adapted from [Tru12, page 19]

problem is that one of the big advantages of virtualisation is that VMs can be easily migrated to other hardware. By using a TPM this would not be possible any more.

To solve these problems a virtual TPM (vTPM) has been introduced by Perez et al. in [PSD+06]. The solution consists of a vTPM that implements the TCG TPM 1.2 specification and a vTPM Manager that is able to handle multiple vTPM instances. They provided two solutions how the vTPM can be used. The first one is a pure software solution where the vTPM Manager and the vTPM instances are running in the userspace of a Management VM. Every guest systems has its own vTPM instance and can communicate with it over a client-side TPM driver. The Hypervisor forwards this communication to the Management VM where a server-side TPM driver is needed to work with the vTPM Manager. The vTPM Manager works as a multiplexer that forwards the commands to the corresponding vTPM instance of the guest VM. The Management VM itself uses the hardware TPM of the system for its own integrity measurement. This way the integrity of the Management VM executing the vTPM instance can be verified. The architecture of this solution can be found in Figure 3.8 on the next page. The second solution uses a secure co-processor to run the vTPM Manager and the vTPM instances. The secure co-processor is tamper resistance and therefore adds again this property of a TPM that has been lost by the pure software solution. In comparison to the first solution, by using the secure co-processor there is no need for a hardware TPM in the system, since the Management VM owns the hardware and simply reserves the first vTPM instance for its own integrity measurement. Like in the first solution, guest VMs can communicate with their vTPM instance over the Management VM where a proxy forwards the communication to the vTPM Manager in the secure co-processor. The vTPM Manager and the vTPM instances work the same way as before. In Figure 3.9 on the following page the changed architecture with the secure co-processor can be found.

Both solutions allow to have multiple VMs, where every guest system has its own TPM. The vTPM acts like a real TPM, thus it is possible to use applications that use a TPM without any changes. But at least for RA it should be possible to distinguish a vTPM from a real TPM since they don't have the same security properties. To establish trust in a system a Challenger must also check the measurements of the Management VM providing the vTPM, the Hypervisor and its boot process. To allow this the vTPM instance of a guest VM has its PCR divided into two sections. Register 0 to 8 are mapped from the hardware TPM (or the first vTPM instance in case of the secure co-processor) to the vTPM instance. These registers are read only and contain the measurement of the boot process and the integrity measurement of the Management VM OS. All registers starting from PCR 9 can be used like on a hardware TPM and contain for example the integrity measurement of the guest VM itself. If the Challenger decides that verifying the trustiness of the VM alone is not enough, the vTPM allows the Prover to provide additional measurements of the underlining platform.

To solve the TPM migration problem Perez et al. also designed a secure protocol. With this protocol it is possible to migrate the VM and its vTPM with minimal downtime to another host. The migration process is started with the creation of a vTPM instance at the target host. This new instance then creates a nonce and sends it to the source vTPM. The source vTPM is then locked to this nonce and they are added to the TPM state and is validated by the destination TPM that was import before. This prevents the vTPM to be migrated to multiple instances. All the data of the vTPM, like PCRs, keys, counters and

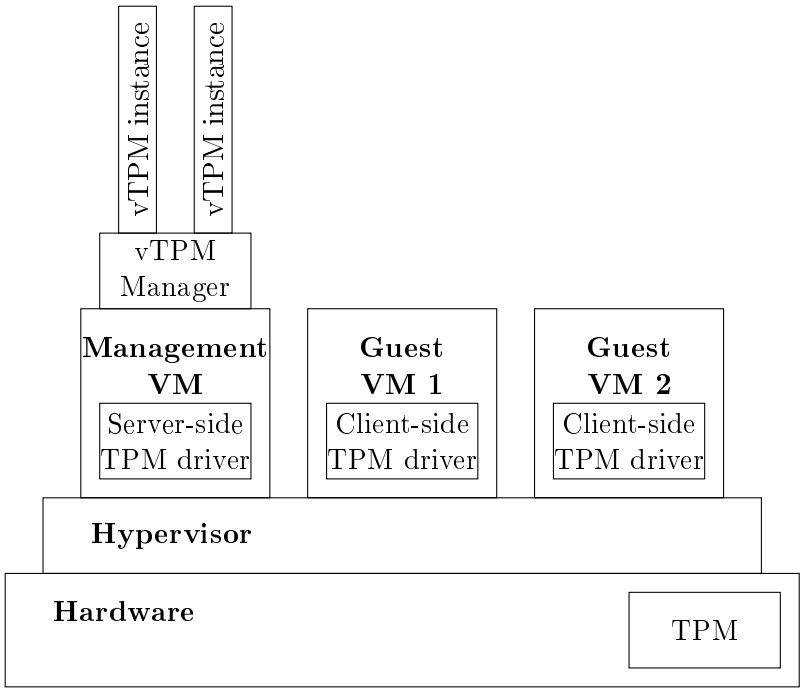

Figure 3.8: vTPM running in a Management VM. Figure adapted from [PSD+06, page 309]

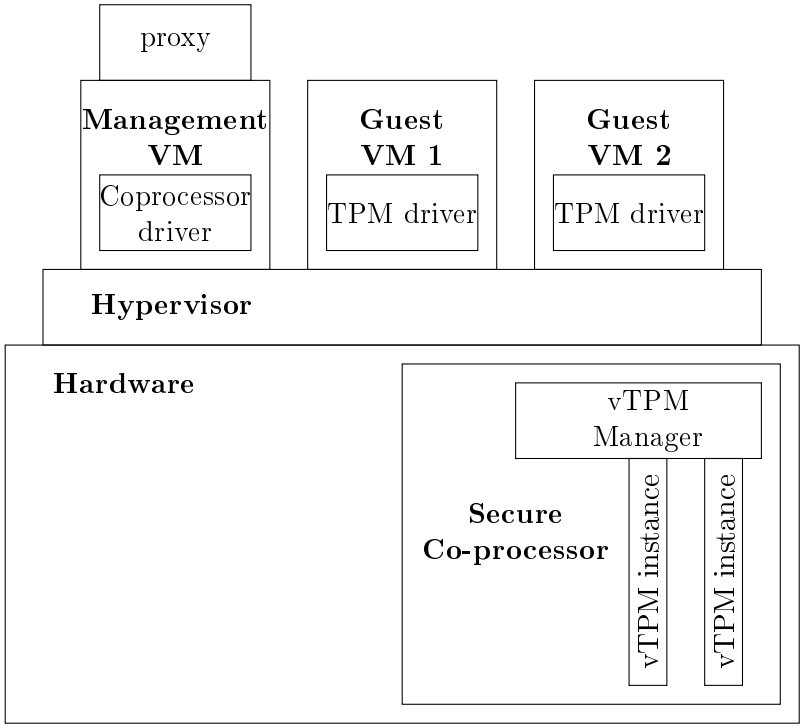

Figure 3.9: vTPM running in a secure co-processor. Figure adapted from [PSD+06, page 309]

so on are collected and encrypted with a symmetric key. During the serialisation of the TPM state the vTPM is locked and cannot be used by the system. At the end a migration digest is updated and included into the TPM state. The symmetric key itself is encrypted with a parent TPM storage key. To encrypt the symmetric key at the destination vTPM the storage key has to be migrated first. After that the TPM state can be decrypted and restored. Before the migrated vTPM instance resumes, the calculated migration digest is compared to the included one, to ensure that no malicious entity was able to modify the data.

Since 2006 when Perez et al. introduced the vTPM, it is still not supported by in many Hypervisors. A comparison of Hypervisors and their support for vTPMs is shown in Table 3.1 on the next page.

The only Hypervisors with build-in support for vTPMs are Xen and Hyper-V, both support TPM 1.2 and 2.0 for the host as well for the vTPM instances. But there are at least two Hypervisors where a community patch exists to add limited support. For example, Berger one of the authors of [PSD+06] released a software TPM implementation [Ber14b] and a patch for Qemu [Ber14a] to add vTPM support. This software TPM does not implement the migration feature of [PSD+06] but allows to have multiple vTPM instances running on the host system. It does not need a hardware TPM in the host system. Another TPM emulator is available from [Str04]. As far as we found out it is only possible to use one TPM instance at a time. There is also a vTPM extension for VirtualBox [WH12].

3.8 Hypervisor-based Attestation

All solutions so far have one big disadvantage, namely the lack of a strong isolation of the measurement architecture to the system that is measured. Of course, the IMA runs in the kernel space and is therefore protected, but if an attacker could get access to the kernel it might be possible to modify executables between measurement and execution. This problem is called Time of Check To Time of Use (TOCTTOU). To come around this problem Azab et al. used a different approach in [ANS+09]. They introduced a solution named HIMA for virtual systems, where the integrity measurement is done by the Hypervisor and therefore strongly isolated from the system that is measured. Since the integrity measurement is done by the Hypervisor, an attacker is not able to manipulate it, even if the attacker has full access to the guest and manages to modify the kernel. To achieve this the HIMA actively monitors guest events and the guest memory. This way the HIMA always gets track of the guest memory layout. So even when a process is modified in the guest VM this is detected and measured by the HIMA. This is done by interrupting the guest events whenever a user process or kernel module is created, terminated or modified. All measurements are done before the process is executed. By doing so the TOCTTOU consistency for the integrity of the measured programmes can be guaranteed. The measurements done by the HIMA are stored in a dedicated Management VM guest. In comparison to the IMA, that has to be configured for every system, the HIMA works as an "out-of-the-box" solution for every guest without any configuration or software needed. Figure 3.10 on page 28 shows the architecture of the HIMA. The Hypervisor and the management VM are assumed to be trustworthy at any time in this solution. This brings up the question: What happens if one of these components gets corrupted in any form? Since the HIMA

	Xen	**Hyper-V**	**Qemu**	**VirtualBox**	**VmWare**
vTPM support	yes	yes	not out of the box	not out of the box	no
minimum Version	4.3 4.6 (TPM 2.0)	Windows Server 2016	2.9 with [Ber14b] and [Ber14a]	4.0.2 with [Str04] and [WH12]	-
TPM needed	yes	yes	no	no	-
vTPM instances	n	n	n	1	0
TPM version	1.2, 2.0	1.2, 2.0	1.2, 2.0	1.2 (limited)	-

Table 3.1: Hypervisor vTPM support

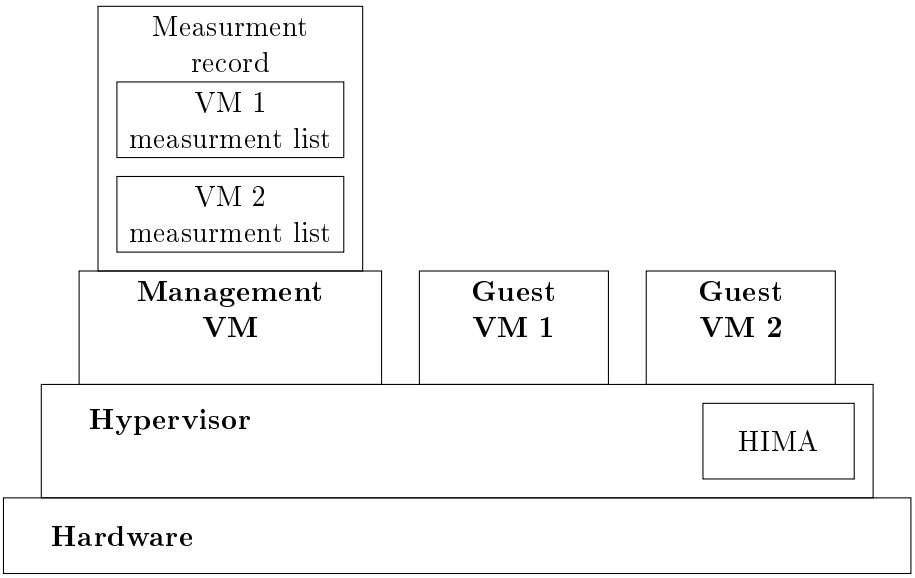

Figure 3.10: Architecture of the HIMA. Figure adapted from [ANS+09, page 463]

does not use a TPM it can only be used as a local solution, hence it is not able to provide evident prove that the measurements are not modified to another remote host.

Another solution that used the Hypervisor-based attestation approach is the Sensory Integrity Measurement Architecture (SIMA)[SKU10]. Like by HIMA [ANS+09] the guest systems are monitored, but how this is done differs. Furthermore, not only VMs, but also the host system itself is monitored. Many different specialist sensors monitor various aspects of the guest systems as well as the Hypervisor. The results of this sensors are then sent to a monitor that evaluates them and decides what actions should be performed. This way even attacks on the Hypervisor can be detected by the SIMA. The sensors are placed in guest VMs in the kernel space in order to monitor the guest system itself. And in die Hypervisor to monitor physical and virtual memory of the whole system including the Hypervisor and its guest VMs. As already mentioned every sensor is specialised to one specific aspect to keep the complexity low and therefore make them easier to develop and hold the resource consumption low. Sensors can run in different modes of operation. In the autonomous mode they just send detected irregularities to the monitor. In the monitor driven mode the monitor is able to send commands to the sensor. In the autonomous mode not all sensors are active or not all its functionality is activated to save system resources and do not slow down the system too much. In the monitor driven mode, the monitor can trigger further investigation into one specific aspect of the system or even a complete system check. The communication of the sensors and the monitor is done through the blackboard, a shared memory region, where every VM has some mapped memory. This should keep possible attack vectors to the monitor and the Hypervisor as small as possible. The monitor itself operates as a state machine with the following states. In the normal operation state, everything works as expected and no irregularity has been detected. In the error state, abnormal system behaviour has been reported. The monitor can switch sensors into monitor mode to get additional information about the problem in order to isolate it. The critical error state is entered if the isolation fails. The monitor induces a

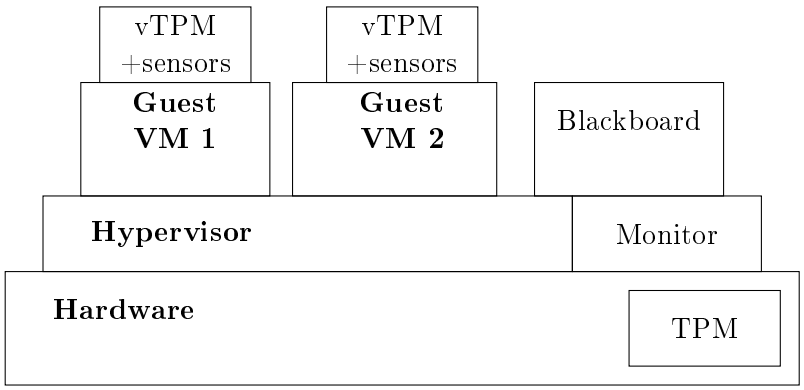

Figure 3.11: Architecture of the SIMA. Figure adapted from [SKU10, page 108]

complete in deep system check. If this check indicates an attack the SIMA switches into the alert state and completely isolates the affected VM. The last state is the panic state, it is entered if everything to confine the problem has failed. In this state the monitor indicates a core dump and restarts the guest or freezes the VM. Since there are some sensors placed directly in the guest VM it might be possible that an attacker detects them and tries to manipulate their results, but doing so would also produce a row of system events that make it possible to detect the attack by other sensors. To ensure that the Hypervisor, the monitor and sensors start in a trusted state they are measured by a hardware TPM in a secure boot process. For guest VMs this is done with a vTPM. An overview over the architecture of the SIMA can be found in Figure 3.11.

3.9 Trusted Execution Environment based Attestation

Another technology that has become widely available is the TEE. TEE provided a way to execute code in an isolated secure environment. TEE functionality is directly implemented by some CPUs in hardware. The memory and registers of this environment are separated in a way that it is not possible for normal applications to access or modify them. How this is exactly done differs between the manufacturers.

ARM: TrustZone is the name of the TEE of ARM. The processor can switch between two worlds, called normal world and secure world. The secure world runs a secure OS that allows the execution of signed programs that are called Trusted Applications (TA). The CPU is able to boot directly into the secure world to initialise it first and eventually start a rich OS in the normal world afterwards. The CPU has its own set of registers for the normal and the secure world. The memory mapping to normal or secure world is done with an additional bit, this means in a 32 bit CPU an additional 33^{th} bit is used for this.

Intel: The Software Guard Extensions (SGX) provide a TEE for Intel CPUs. In SGX the equivalent to the secure world is called secure enclave. This secure enclave can be started by normal applications and is a protected area in the address space of the application. To secure this area, it is encrypted. Since secure enclaves are created

by normal programs the source is loaded from unprotected memory. Whenever code is loaded into a secure enclave, it is first measured. [Cor14]

AMD: AMD uses a dedicated Platform Security Processor that is basically an ARM processor with TrustZone directly integrated into some of their CPUs.

There exist different approaches how the TEE can be used for RA. The first one tries to implement a subset of TPM functions in software that can run in ARM TrustZone [KR15]. This is useful because usually devices with an ARM CPU do not have a dedicated TPM available. In their approach they only implemented functionalities needed by the default RA protocol. The software TPM was implemented in a way that it is loaded by the kernel before the first measurement is done by the IMA. But this also means, that since it is loaded by the OS it cannot measure the boot process itself. One thing that is completely missing in this work is how they solved the problem of securely storing the Endorsement Key (EK) of the TPM, since the TEE is missing a secure non-volatile memory. They only generate an AIK that is securely stored in volatile memory. But this means that a generated AIK gets lost with a reboot. If there is no EK there is no way for a private Certification authority (CA) to identify the TPM during the certificate creation.

The work of Raj et al. tries to implement the full functionality of the TPM specification in software and calls this firmware-TPM (fTPM) [RSW+15]. Like in the work of Kylänpää and Rantala they implemented the software TPM for the ARM TrustZone. But they actually tried to fulfill the whole TPM specifications. To implement a TPM with the same probabilities as a hardware TPM there are some limitations of the ARM TrustZone as well as the SGX that have to be solved in order to achieve this goal.

No trusted non-volatile memory: Neither the ARM TrustZone nor the SGX of Intel provide a trusted non-volatile memory. This is a problem since the TPM needs at least an EK and a Storage Root Key (SRK) that can only be read and used by the TPM itself. And even for the other data like PCR values encryption alone is not enough, since there is still no place to store the key in a secure way. In addition, encryption alone does not prevent roll-back attacks. Since the state of the TPM is encrypted an attack can not change it to a arbitrary value. But an attacker could use an old encrypted state and overwrite the current state with it. This way an attacker can simply roll-back changes made to the PCR.

No entropy source: Like before both environments leak the existence of an entropy source that is only available for the TEE. But this is needed for the RNG functionality of the TPM. If the TPM uses a shared entropy source it is possible that the random values are foreseen or influenced by the normal world.

No secure clock: Many security functions need a secure clock. But since TrustZone has no protection for peripherals, they could be manipulated by the normal world and therefore be insecure.

Lack of access to firmware: Another problem is that the firmware of the most System-on-a-Chip (SoC) manufacturers is not open. This makes it very difficult to deploy software to the TrustZone.

To solve the problem with the trusted storage they add additional requirements to the hardware needed to run the fTPM. Since many mobile devices already have an embedded Multi-Media Controller that has a replay-protected memory block this memory can be used to protect from roll-back attacks. In conjunction with encryption it can also be used for storing the fTPM state. In addition to that a secure hardware fuse is needed that is exclusive to the secure world. The hardware fuse contains a unique key for the device and a secure entropy source. As already mentioned, there are some commands that need a secure clock in order to be secure. But the TrustZone has only a secure timer, that ticks in a predefined rate. This timer can be used to implement some of this commands in a secure way. But there are still a few commands that cannot be supported. In contrast of the work of Kylänpää and Rantala the fTPM is loaded before the rich OS. Like the name already lets assume, it is loaded with the firmware of the device. This way it could measure the whole system starting from boot time.

Chapter 4

Concept and Architecture

In this chapter we aim to give an overview of our solution and which problems we attempt to solve with it. Although our proposal contains a very generic solution that can be used in many ways, with only a few restrictions to the system, we start with a description of our system domain. This is meant to give an understanding about the constraints that are linked to it and why certain problems are solved in the way we did. We discuss the problems and possible solutions and in some cases why some of these solutions are not suitable for our system domain. In addition, this chapter contains a detailed description of the proposed protocol.

4.1 System Overview

The proposed RA system is designed to run on a customized and light weight Linux distribution on embedded CDs used in hydro-electric power plants. This embedded CD consists of a communication controller with a x86 CPU. This controller is called Communication CPU (CCPU) and is running a Linux OS. The CCPU has a network interface connected to the network of the company. Furthermore an application controller with a real time OS is connected to the CCPU. This so-called Application CPU (ACPU), is using sensors and actuators to perform critical control tasks that are defined by loadable software modules. The software modules are loaded by the CCPU. Usually one device handles multiple control tasks. Figure 4.1 on the following page shows a simplified overview of such a CD used in hydro-electric power plants. Although these CDs are connected to each other and are remotely maintainable, they do not have access to the internet. The software running on these devices is also strongly controlled by a central authority. This means that there is a well defined set of applications that is allowed to run on such a CD.

Although the protocol we are going to introduce was designed with the presented in mind, it could actually run on any device with Linux and a TPM. Even though maintaining the system for big Linux distributions might be difficult, because there are more measurements that must be checked and trusted.

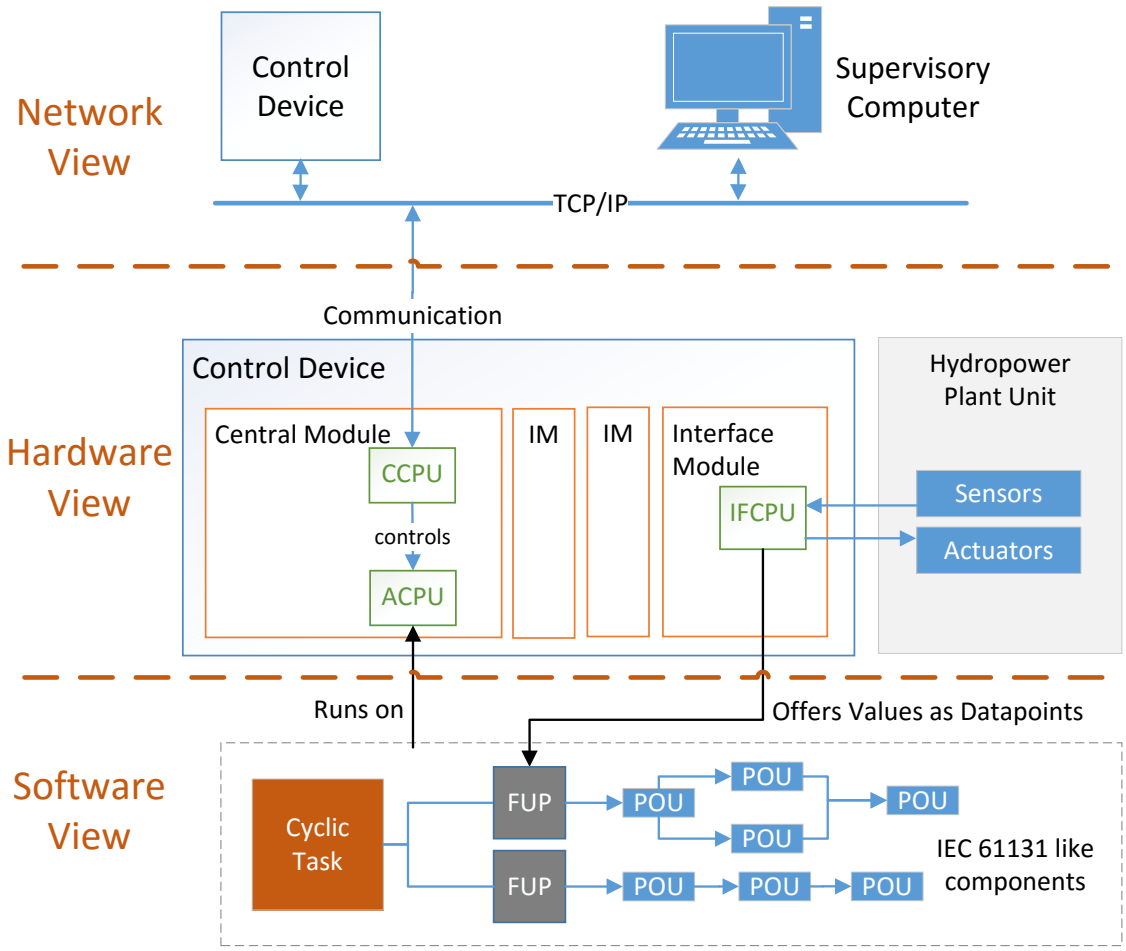

Figure 4.1: Overview of a CD of our system domain. Reprinted from [IRK+17, page 69] with permission

4.2 Attacker Model

Like defined in the TPM specification [Tru11a] we assume that an attacker does not have physical access to the attested CD (or at least not to the TPM). This is important since the TPM only has basic protection against tempering through timing changes. This means that the TPM is maintaining a tick count for every session. If the tick count differs from the expected count this is treated as an attack and the TPM shuts down. But of course, this cannot protect against all possible physical attacks like TPM reset attacks or bus manipulations like described by Winter and Dietrich in [WD13]. Furthermore, it cannot protect against side channel attacks or even (semi-)invasive attacks. By not having physical access it is not possible for an attacker to read the private part of any of the keys (like the AIK) protected by the TPM. This is essential since if there would be a way for the private AIK protecting the quote, to leave the TPM, the whole security concept would be broken.

4.3 Problem: Trusted Measurements Database

The ACPUs are running critical control tasks which are defined by software. We want to be sure that attacks on the CCPU loading this software module or any other application that could affect the stability of the CD, are detected in order to take appropriate steps against it.

One possible solution to detect software changes is the use of RA. But the default RA comes with one big disadvantage. Namely how to decide if the corresponding binary to a measurement is trustworthy or not. So there need to be a database with measurements of trusted applications for comparison. There are two possibilities. Either each device running the RA protocol has an own database or there is a central service. The first one comes with the disadvantage that whenever a new application is deployed to any of the devices running the protocol, the database of all devices in the distributed system has to be updated. That makes the deployment of new software very complex and error prone. If there is a central service where protocol participants can ask if a measurement is trustworthy or not it reduces the complexity of deployments. So only the central database has to be updated. But it also rises new problems: How can we trust the central service? What happens if the central service is not available for some reason?

4.4 Solution: Trusted Measurements Database

The main idea of this work is that we want a RA protocol without a local database of trusted measurements, making deployments to a difficult and error prone task. Furthermore, we do not want a TTP that has to be online and reachable during the protocol execution.

We solved this problem by using certificates and signatures. Every binary that gets measured by the IMA has a signature generated during the deployment or production process. Therefore, we introduced a new CA controlled by us. This private CA is used as the root of trust for these signatures. During the RA protocol the Challenger has to verify the signature of every measurement. If the Challenger does not have the signature of a measurement, the protocol allows to request it from the Prover. This way signatures

only have to be deployed to the host where the corresponding binary has been deployed to, since the signatures will be distributed during the protocol execution. Lets assume a new version of a application and the corresponding signature is only deployed to one Prover. After that a Challenger connects to this Prover and starts the RA protocol. When the Challenger receive the IMA measurements, there is no signature for the new measurement entry present. Now the Challenger simply requests this signature form the Prover. After the Challenger receives the signature for this measurement from the Prover, everything needed to verify the state of the Prover is available.

The use of signatures for measurements results in a new problem. How can we revoke the trustworthiness of a specific binary, for example if a vulnerability was found in it? Signatures can be invalidated by revoking the certificate used to create it. But if we want to revoke only a specific signature it is necessary to use a new certificate for every binary and every version of it. We propose to use 2 additional layers of certificates in addition to the root certificate. The first Layer contains a certificate for every application that is only allowed to create new certificates. By revoking this certificate all versions of a binary could be revoked. This can be useful if an application should no longer be used by any device. The certificates issued by this application level certificate are only allowed to create signatures and are mapped to a specific version of the application. This application version level certificates are used to revoke only a specific version of a binary. Another possibility would be to add a fourth level that represents the context of the certificate hierarchy under it. Possible contexts are software and hardware. An example for this certificate hierarchy can be found in Figure 4.2 on the next page.

Technically the use of the Online Certificate Status Protocol (OCSP) would be the best solution for the certificate revocation, since this would be very simple and always up to date. But since one of our goals was that we do not want to call a TTP during the protocol execution, we propose to use an offline Certificate Revocation List (CRL). This CRL could be deployed to all devices whenever a certificate has been revoked. Another possibility would be that the devices fetch a new version of the CRL from a central service from time to time. This way the deployment process stays as simple as possible, while there is still no need for a TTP that is reachable during the protocol execution.

4.5 Improved Remote Attestation Protocol

The RA process was designed in a way that we do not need any information about the Prover before starting it. Moreover, during the protocol execution there is no need to contact a TTP to check the correctness of the measurements. This means that everything that is needed to verify the state of the Prover is available through the protocol and can be requested by the Challenger. This is done with the help of Public Key Infrastructure (PKI). All measurements in the IMA must have a valid signature from a CA we trust. In our case, we used certificate pinning to our own private CA root-certificate. Since we control this CA, we can be sure that only software we signed during deployment will have a valid signature. By using signatures for the measurements, we were also able to reduce the deployment complexity of new or updated software. This is caused by the fact that we only have to deploy the signature of the measurement to the machine where we deploy the software. We do not have to care about all the other machines running the protocol,

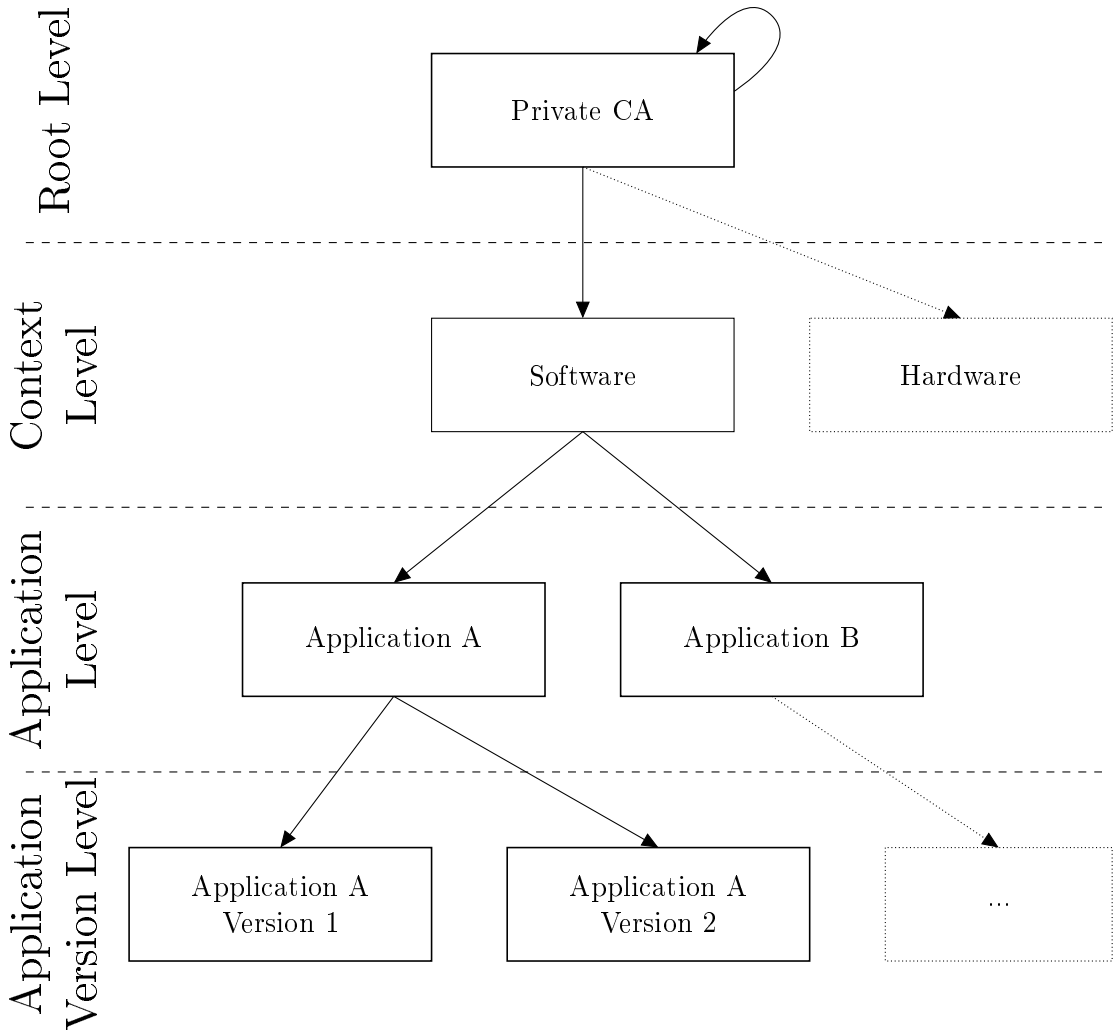

Figure 4.2: Certificate hierarchy

since they can simply request the new signature during the protocol execution. In contrast to the default RA protocol, every machine running the server part has exactly one AIK. This AIK is generated during production. In addition, a certificate for it is generated by our private CA. This certificate contains all the information needed to uniquely identify the owner, since we want to know exactly to whom we are talking.

To attest a remote host, the protocol from Figure 4.3 on the following page is used. The Challenger starts the process by building up a secure connection to the Prover. If the Challenger does not have the AIK certificate of the remote host, an AIKCertificateRequest is sent to the Prover. The received certificate is stored in the local certificate store in order to reuse it the next time it connects to the same Prover.

After the correct AIK certificate is present at the Challenger, a new set of nonce is generated and sent to the Prover as part of a QuoteRequest. Now the Prover has to execute the quote Command of the TPM with the nonce from the Challenger on the PCR 10. The resulting signature and the quote structure containing the nonce and the indexes of the PCRs that have been quoted as well as the composite hash are sent back to the Challenger. The Challenger verifies the quote by comparing the nonce, created in the step before and the nonce from the TPM_QUOTE_INFO2 structure, received from the Prover. If they are the same, the signature is checked.

If signature and the AIK certificate are valid we can be sure that the Prover was not able to manipulate the result of the quote. Now the Challenger has to check, if there is a stored composite hash for this specific Prover. If this is the case and the stored composite hash equals the composite hash from the quote structure we can trust the Prover. If there is no stored composite hash or it differs from the received hash, the Challenger sends an IMARequest. This way the Challenger can retrieve the IMA measurements from the Prover in order to recalculate the composite hash. Before the Challenger can calculate the composite hash, a signature for every entry in the IMA measurement list must be available at the Challenger. If this is not the case, the Challenger has to request the missing signatures from the Prover with a SignatureRequest.

When the IMA measurements and their signatures are available at the Challenger, we check the signature for every entry in the list. If all entries have a valid signature, the Challenger can use the entries to recalculate the composite hash. If this calculated composite hash equals the composite hash from the quote, the Prover is trustworthy. Only if this is the case we store the calculated composite hash for future connections with this Prover.

If the AIK certificate, at least one signature, the nonce from the quote or the composite hash is not valid, we cannot establish trust in the Prover, since this indicates a manipulation by the Prover.

4.6 Problem: Notification on State Changes

The protocol above allows us to establish trust into the state of another device at the time we execute it. But if the state of the device changes after we have already established trust, we would not notice this. Thinking of a use case where we want to transfer very sensitive data only to another device in a trusted state, another solution is needed. So, we would need something like a TNC. The problem with the TNC as defined by the TCG

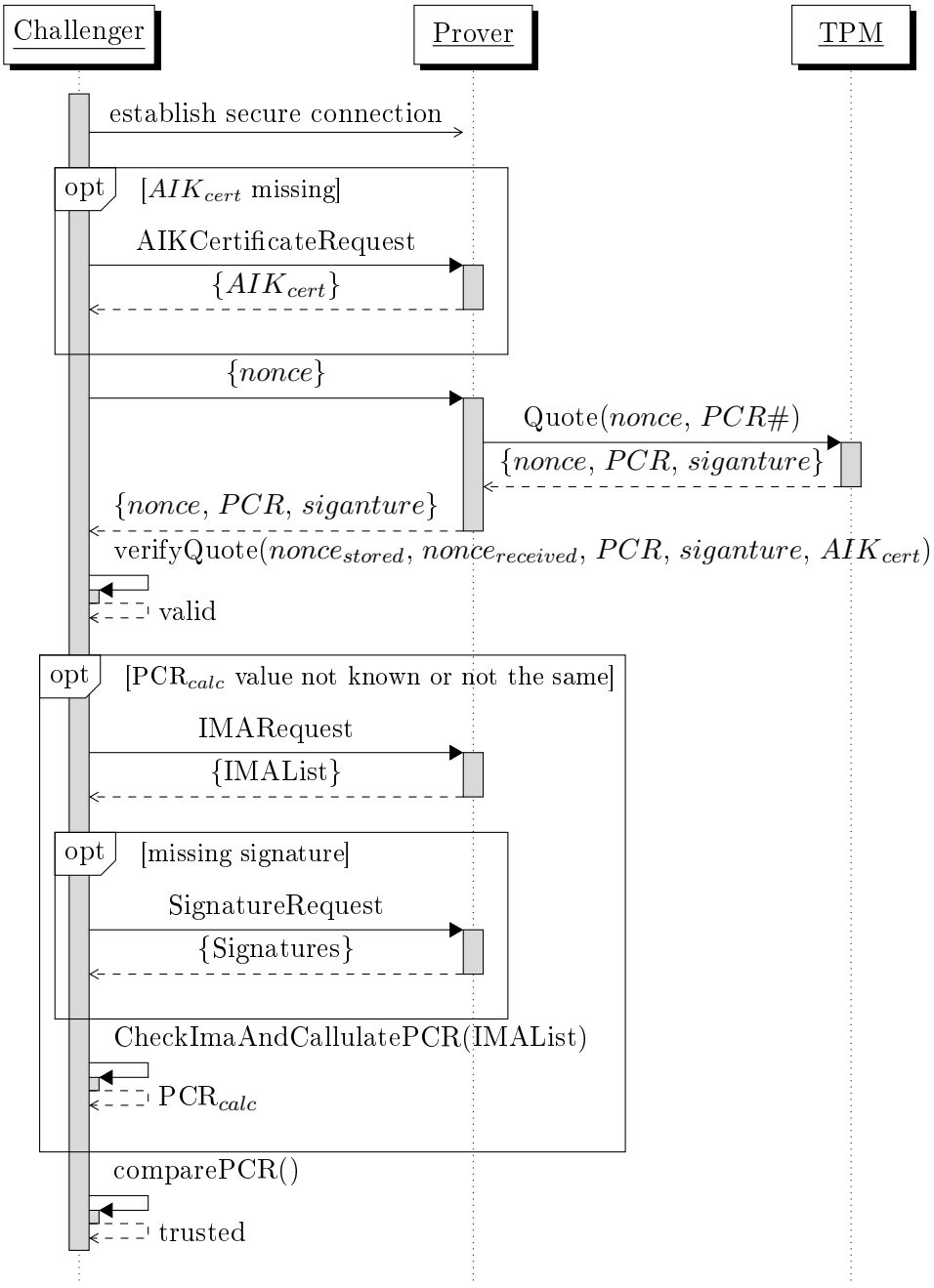

Figure 4.3: Sequence diagram of our RA protocol

is that it does not guarantee that we will be informed about a state change immediately. This is caused by the fact that TNC only checks the trustworthiness of a device when it tries to get access to a network. Of course, there are mechanisms to detect changes and revoke network access, after access to the network has been granted to the device. But this mechanism cannot ensure that the device is in a trustworthy state at any time after the first connection. But even when we are not interested in something like a TNC, an instant notification about a system state can be helpful. The faster an integrity violation is detected the more likely it is to take appropriate steps to prevent a possible infection from spreading to other devices.

4.7 Solution: Notification on State Changes

Our proposed idea to solve this problem is that every time we want to build a TNC to another device, we establish a secure connection to it. After that the protocol we introduce is used to establish trust. To get immediately informed about changes in the IMA after the protocol has been executed successfully, we propose some changes to the Linux kernel of the Prover. These changes ensure that after the IMA detects a change of a binary or configuration all active Transmission Control Protocol (TCP) sockets are closed. This way the Challenger gets informed that the system state of the Prover has been changed. To check if the Prover is still in a trusted state the Challenger reconnects to the Prover and executes the protocol again. There are two important things when using this. First of all, the Challenger must ensure that trust is only established if the Prover is running the modified kernel, since otherwise there would be no notification about changes. This can be done since the kernel is also measured during boot time. Secondly to get perfect confidentiality, the same connection as used for the protocol has to be used for further communication, since as long as this connection is up there has been no state change at the Prover device. For use cases where availability is more important than confidentiality and where for example very strict requirements to response times are in place it might be not possible to cut this communication. In such cases this solution can still be used for a separate monitor connection that is established and kept alive as long as we want to be informed about the state of the Prover. This way the Challenger still gets informed about changes immediately. This information can then be used to isolate the compromised device and move its tasks to another device. To only disconnect such monitor connections on IMA changes, the changes in the Linux Kernel must respect that by using a configurable white or black list. This list contains all applications that use a monitor connection. We would suggest using a white list, since most connections would not gain any information by an unexpected disconnection and this way it is much easier to configure. In our case there is only one application that is running the Prover on this list. Another possibility would be to disconnect all connections using the port that is used by the improved RA protocol. This way there would be no need to configure a list of applications.

Chapter 5

Design and Implementation

In the previous chapter we discussed all problems that we had to solve and how we solved them. This chapter explains how we actually used the concepts from before to implement a prototype of our protocol. It also explains how we integrated it into an existing framework to report integrity violations. In addition, we also discuss technical problems related to the use of certificates and signatures and how we solved them.

5.1 Modules

Our implementation of the system consists of several modules, all with their own purpose. There are 3 applications, 2 libraries and one Linux Kernel Module (LKM). In Figure 5.1 on the next page we can see the direct dependencies of these modules. Depending on which parts of the protocol should be executed, a different set of those modules are needed. If only the Challenger part of the protocol should be executed on a host, one only needs the RemoteAttestation module and both libraries.

Figure 5.2 on page 42 shows a simplified class dependency digram of our implementation. It does not contain all classes from our implementation, but it should be sufficient to get an idea how of our solution.

There are several constraints to the Linux distribution we are running on, for example the IMA and the Grand Unified Bootloader (GRUB) have to be active and configured to measure the system. To guarantee the detection of a compromisation, a physical TPM must be connected. The LKM needs at least a Linux kernel 4.5 with an additional change in the IMA, that calls the LKM, to work. If the system is used without the Terminator LKM it can still be used for RA, but not for trusted network connections or monitor connections.

Since the devices that will run the protocol are equipped with a version 1.2 TPM, our implementation is based on this version. In contrast to the TPM version 2.0, version 1.2 only supports SHA-1. This can become a problem because SHA-1 should not be used any more, since collisions have been found [SBK+17; SKP16]. Nevertheless it should be very straightforward to change the implementation in order to use version 2.0.

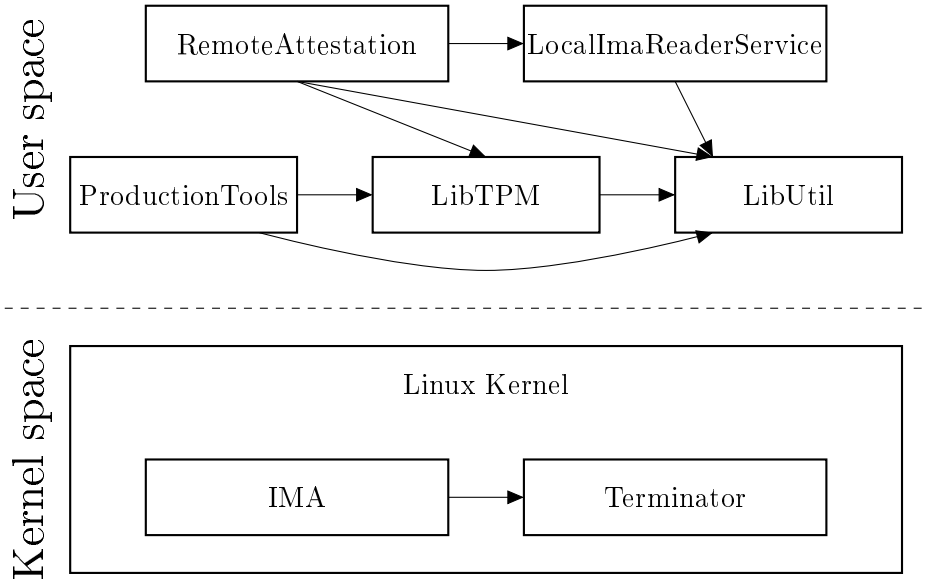

Figure 5.1: System Architecture

5.1.1 RemoteAttestation

This module implements the RA protocol. It provides a client (Challenger) and a server (ProverServer). Theoretically it can be used only for a single RA or for building up a trusted network connection. Since the current implementation is only a prototype, some changes are necessary for productive use. The RemoteAttestation Module depends on LibTpm and LibUtil. If the server is started it also needs the LocalImaReaderService running and optionally the Terminator LKM. Basically, the server starts a QTcpServer that listens on a predefined port for incoming connections. If there is a new connection, a new instance of a ProverConnection is created that handles all incoming requests from the Challenger. The ProverConnection is implemented in a stateless way. This means that the Challenger could actually send the request in a different order than defined or execute a step several times. While this made the implementation of our prototype very straightforward, it could perhaps also be used for some kind of Denial-of-Service attack. In order to prevent this in productive usage, some kind of counter measures should be implemented, like limiting the allowed request count for each Challenger. In contrast to the Prover, the Challenger part is implemented as state machine. Figure 5.3 on page 43 shows the state machine used by the Challenger. Basically there is a state for every step from the improved RA shown by Figure 4.3 on page 38. This way we ensure that for every Request only the correct response is accepted.

Since it is possible that a host only needs the client or the server part of the protocol, this module could be split into 3 new modules:

RemoteAttestationClient: This module would only contain the client part of the protocol and would only depend on the LibUtil and the new LibRemoteAttestation module.

RemoteAttestationServer: Here we would locate the server part of the protocol. This

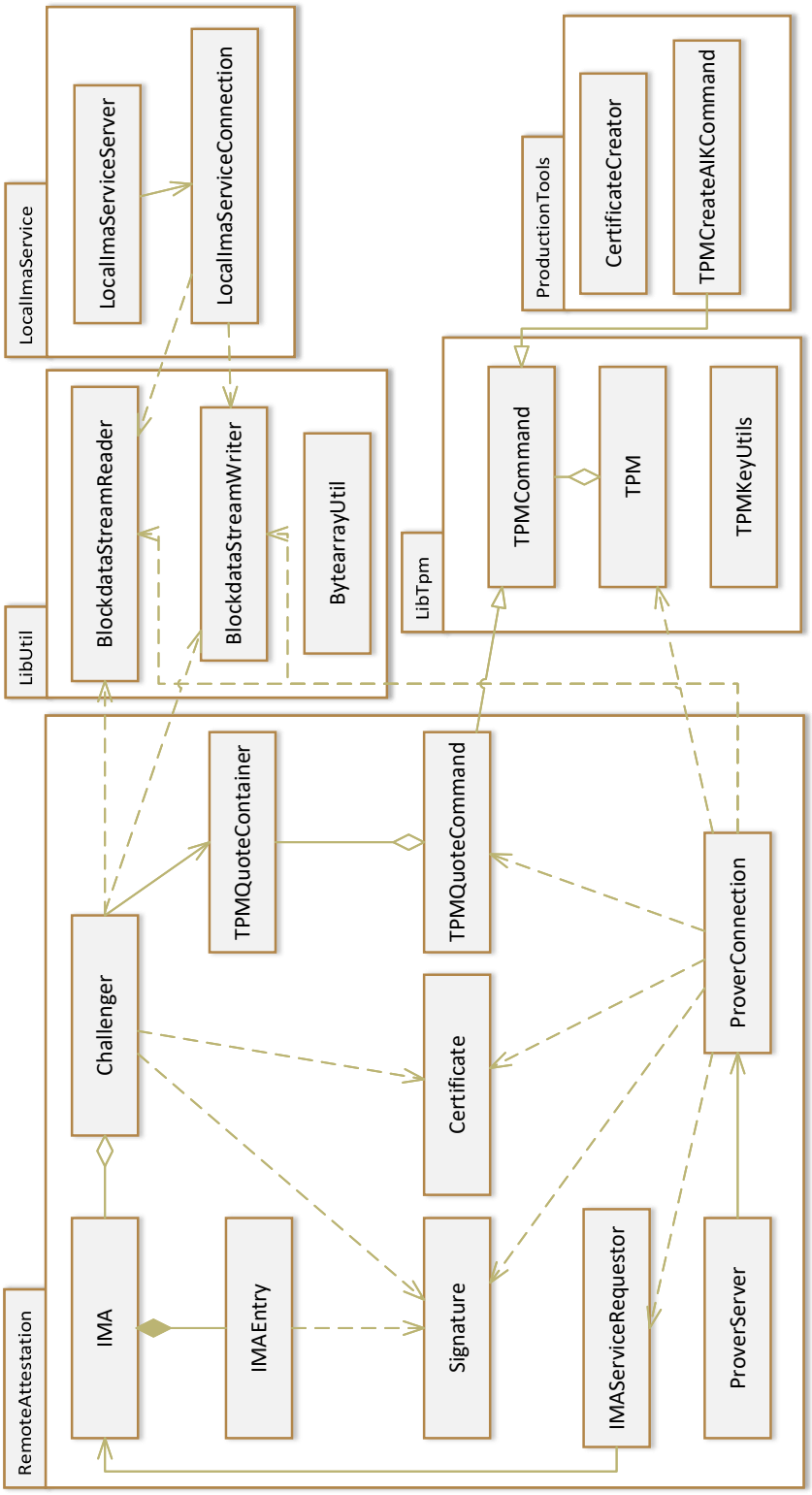

Figure 5.2: Class dependency diagram of our implementation

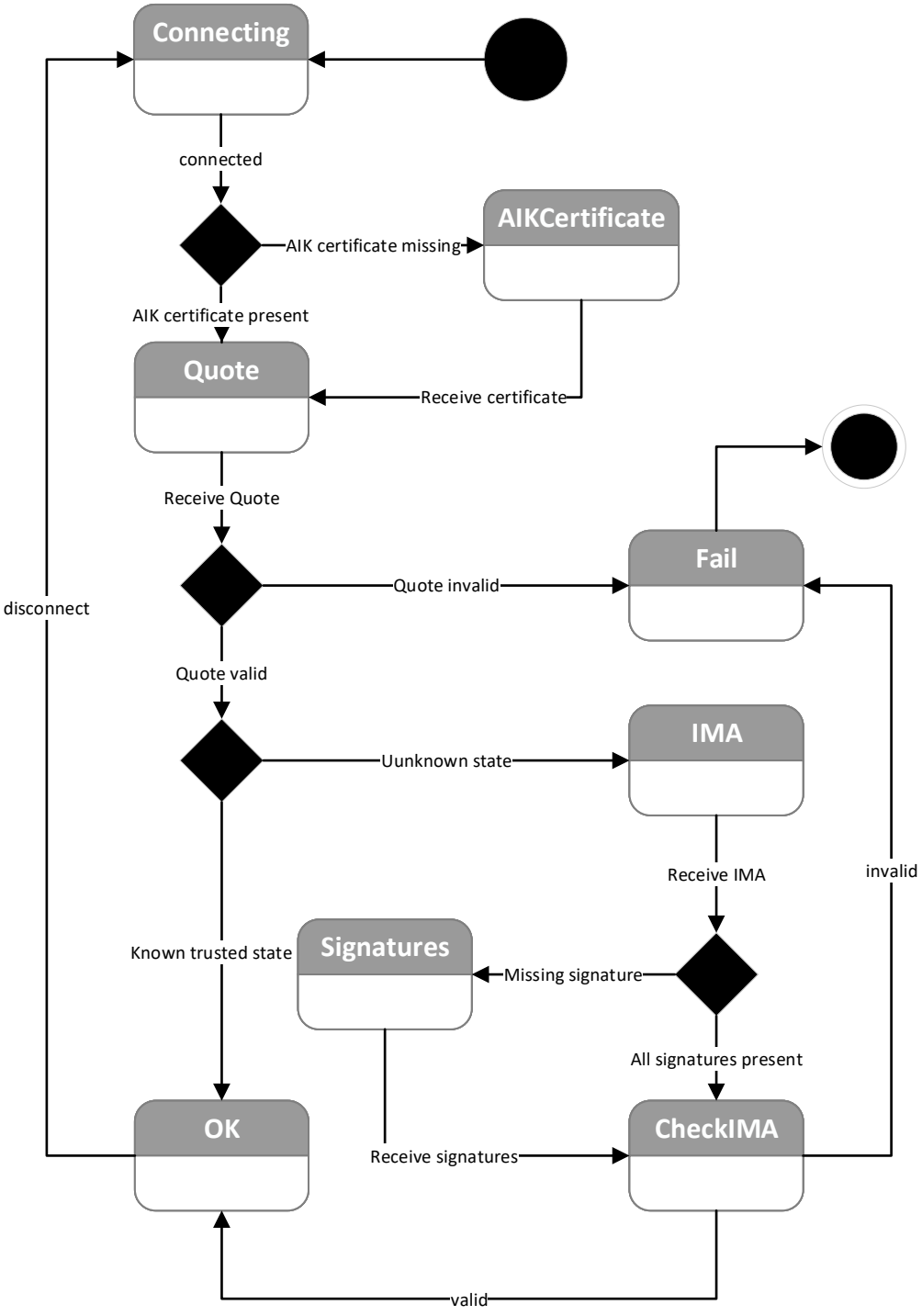

Figure 5.3: State machine of the Challenger

module would have the same dependencies as the original RemoteAttestation module. In addition to this it would also need the new LibRemoteAttestation module.

LibRemoteAttestation: This module would contain all the shared code between the RemoteAttestationClient and the RemoteAttestationServer modules. It would also depend on the LibUtil module.

5.1.2 LocalImaReaderService

On a Linux system, the measurements of the IMA are available through the ascii_runtime_measurements file under /sys/kernel/security/ on the file system. This file can only be opened by root. Since we need to read the ascii_runtime_measurements during the protocol execution and we do not want to run the application which runs the ProverServer as root, we introduce the LocalImaReaderService. This is a service which provides the ascii_runtime_measurements to other processes on the same machine. It must run as root to read the ascii_runtime_measurements file. It simply provides a local socket where other local applications can connect to and read the current ascii_runtime_measurements without needing root privileges. This service only has a dependency to the LibUtil module.

5.1.3 Terminator

This LKM closes (terminates) all open TCP sockets by sending a TCP RESET after the Linux IMA detects a change and has already measured it. To do so there is a small change in the IMA source code required that is needed to call the terminator. This changes only affect two files namely the "LinuxKernel/security/integrity/ima/ima.h" and "LinuxKernel/security/integrity/ima/ima_main.c". The changes to this files can be seen in Listing 5.1 for "ima.h" file and Listing 5.2 contains the changes to the "ima_main.c" file. Basically there are only two declarations, two symbol exports and one if statement that checks if the terminator module is loaded. In total there are only 7 new lines of code. The terminator LKM itself needs at least a Linux kernel 4.5 to be able to close open sockets in a proper way. Closing open sockets guarantees that the Challenger recognizes, a change on the remote host during the execution of the RA protocol. As already mentioned the sockets are closed by sending a TCP RESET. This guaranties that the Challenger is instantly notified about the closed socket and does not have to wait for a time out. If the RA protocol is used to build up trusted network connections this is even more important because this is the only reliable way a client gets informed that the RA protocol has to be executed again to be sure that the remote host is still in a trusted state. Since not all applications use the RA protocol, the terminator module has a white list of applications whose sockets should be closed on IMA changes. We also thought about using a black list of applications that should not be affected, but since the number of applications that actually run our protocol and therefore gain information from a closed network socket should be very small, we decided to use the white list approach. Doing so makes the correct configuration of the LKM much simpler and reduces the danger of unexpected problems in other applications regarding their active connections.

Listing 5.1: New declarations in ima.h of the IMA LKM.

```
static struct module *terminator_mod;
static void (*terminator_killAll_ptr)(void);
```

Listing 5.2: Changes to the IMA Kernal Module in ima_main.c.

```
EXPORT_SYMBOL(terminator_mod);
EXPORT_SYMBOL(terminator_killAll_ptr);
...
static int process_measurement(struct file *file, char *buf,
        loff_t size, int mask, enum ima_hooks func, int opened)
{
...
        if (action & IMA_MEASURE) {
                ima_store_measurement(iint, file, pathname,
                        xattr_value, xattr_len, pcr);
                if(terminator_mod){
                        (*terminator_killAll_ptr)();
                }
        }
...
}
...
```

5.1.4 ProductionTools

The production tools are meant to run during the production on the new machine, that should later run the server part of the RA protocol. It creates a new AIK and a certificate for it. Currently the certificate is also created on the new machine. This part should be extracted and run on the production machine or another trusted authority in order to not have to copy the private key of the root certificate to the new machine. For example the production tool would only create the new AIK. After that it would create a certificate signing request and send it to our own PKI. But since providing a tool to set up new devices was out of scope of this work, we only implemented a simple tool in order to set up our own environment. In order to run this application the LibTpm and LibUtil modules are needed.

5.1.5 LibTpm

The LibTpm library provides classes and functionality to work with the TPM. Right now, it only contains those parts which are actually needed by more than one other module. This library wraps the communication with the TPM to make it easy to use. Internally it uses the TrouSerS library in order to communicate with the TPM. Further it hides the complexity with all the special structs and memory allocation and deallocation when working with TrouSerS. This is done by the TPM class that uses the command pattern to

execute TPM commands represented by the TPMCommand class. Since the logic, how a Quote or AIK is created, is encapsulated into own commands, the TPM class is kept very simple. In addition it is very easy to add new commands in the future if needed.

5.1.6 LibUtil

This is a very simple library that only provides helper functions. It contains functionality that is needed by nearly all the other applications and libraries, like an easy way to work with byte arrays. In addition, it contains the functionality to read and write a QDataStream from a socket. For example the BlockDataStreamReader ensures that the whole message from the TcpSocket is available for read and provides it through a QDataStream.

5.2 Scari Integration

Scari (Secure and reliable infrastructure) is the name of an existing self-adaptive software system by Iber et al. introduced in [IRK+17], that was designed with the same industrial setting in mind as in our work. As shown in Figure 5.4 on the next page, Scari consists of five blocks. The knowledge base holds a hierarchical world model. Each device running the Sari framework only knows its parent node and subgraph. The informations of this model can be used by the other parts of the Scari framework during the adoption loop. In the observe phase a monitor detects something suspicious and creates a notification for it. There exist several different monitors, where every monitor is specialist for a specific monitoring task. Notifications generated by these monitors are then analysed in the orienting phase by Syndrome Processors. Like the monitors there are multiple specialist Syndrome Processors. Each Syndrome Processor filters the incoming notifications for those he can handle. If a problem is detected the Syndrome Processor generates one or multiple recommendations. In the case where the knowledge base does not contain enough information to generate a recommendation, the notification is forwarded to the parent entity in the hierarchical representation. Recommendations generated in the orienting phase are consumed in the deciding phase by a Recommendation Decision Maker, who selects the best recommendation and forwards it to a Plan Maker. The Plan Maker generates one or multiple plans how the recommendation can be implemented and sends it to the Plan Decision Maker. In this step the best plan is picked and forwarded to the acting phase of the adaptive loop. In this phase the actions of the plan are executed to adapt the system and changes are fed back to the knowledge base.

The integration of our RA protocol into Scari allows use to report detected integrity violations and react on it in an automated way. We added an integrity monitoring service which is informed about integrity violations of remote hosts by the process that is running the Challenger part. The monitor then creates an integrity violation event. This event is passed by the framework to a higher layer until a node decides that it can handle the event, by creating a plan what has to be done with the affected system. In our case, it checks if there is a hot standby machine available that is able to take over the task of the violated system. If this is the case, all resources are transferred to the new system and the old system is isolated from the network. If there is no hot standby machine available it would also be possible to find devices with available capacities and split the tasks between them.

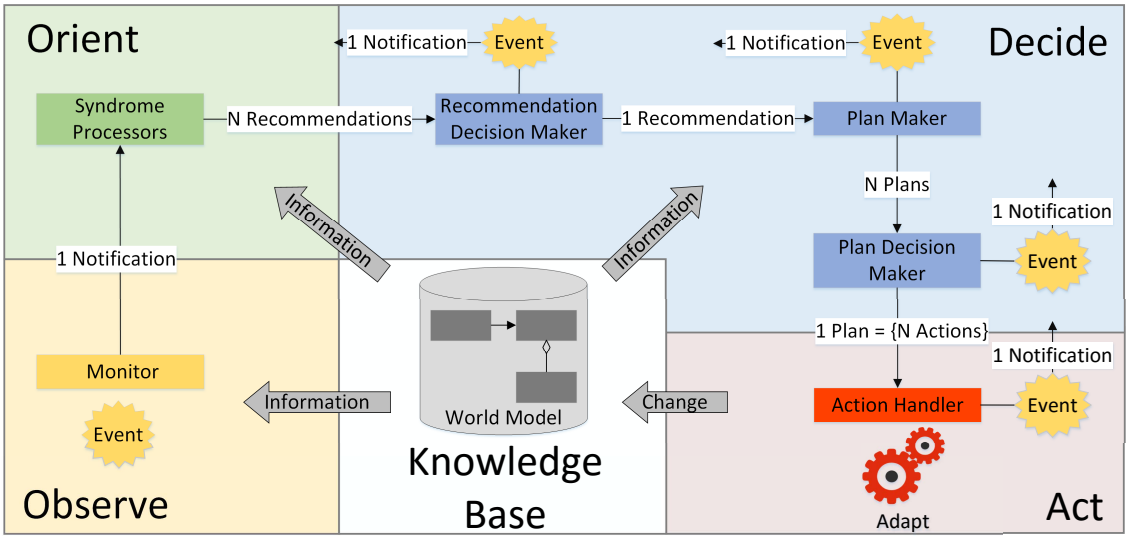

Figure 5.4: Scari adaptive loop. Reprinted from [IRK+17, page 72] with permission

5.3 Signatures and Certificates

As already mentioned we use signatures in order to detect manipulations in the measurement list. To store and transfer signatures we decided to use the Cryptographic Message Syntax (CMS) format which is based on Public Key Cryptography Standards (PKCS) #7 and defined by [Hou99]. There are several reasons for this decision. It is a well-known standard that is also supported by many libraries like OpenSSL, that we use in our implementation. It gives us the ability to include a certificate or a whole certificate chain that is needed to verify the signature. This simplifies the verification process of the signature. The disadvantage of including certificates is the size of the signature files. In our current implementation, all signatures are signed by the same certificate, so including this certificate in every signature might not seem like a brilliant idea. Actually, we foresee to use multiple layers of certificates. Therefore, it would be possible that every application has its own certificate or even that there is a new certificate for every version of the application. This would give us the ability to revoke the trustworthiness of the whole applications or of specific versions with known vulnerabilities. Since this implementation should only show that our concept actually works, we simply signed every signature with the root certificate. Since we already used the CMS format for signatures we decided to use it for storing and transferring the AIK certificates too. But there is no further reason to prefer the CMS format to one of the other possible formats.

Beside of the file format for signatures there was also the problem of how to link them to the application that is signed. Since the only information about an application are the attributes that are included in the IMA measurement list, we used the template-hash of the measurement to link it to the signature. This way the Challenger only has to calculate the template-hash for an entry in the measurement list. In addition the Challenger has to check if there has a signature for this hash in the signature store. If this is the case the Challenger can use the signature to verify the measurement, otherwise the signature is requested from the Prover first. Linking the AIK certificates is an even bigger problem. As

already discussed in Section 3.2 this is also important to prevent masquerading attacks on the RA process. In the first place, we planned to include the serial number of the device into the AIK certificate as well as into the SSL certificates. Since this would be a strong link of both certificates to a specific hardware. But also, the solution form [GPS06] which was already discussed previously, would be sufficient. Although we planned to establish a strong link, our current implementation simply used the IP address of a machine. We are aware that this is actually not a good idea since IP addresses can change and we strongly recommend to not use this for productive operation!

One thing that should be mentioned when working with OpenSSL and TrouSerS, the format of the public AIK key from the X509 certificate and the format that is needed to verify the signature of the quote, are not the same. In addition, there is no functionality in one of both libraries to convert between this blob formats. This means that we had to implement a manual conversion. Actually, this is not a hard task, since most values in the structure are fixed, but since it took us some time to realize why the validation of the quote fails, we felt like we should mention this.

Chapter 6

Evaluation

This chapter contains the evaluation of our introduced protocol and modules. In Section 6.1 we describe the test environment we used to evaluate our implementation. In the next Section 6.2 we present our results regarding the execution time of the protocol. After that we discuss the maintenance efforts for our system in Section 6.3, followed by the evaluation of the amount of data that has to be transferred during the protocol execution in Section 6.4. In the end we describe possible applications for our protocol in Section 6.5.

6.1 Test Environment

To test and evaluate our implementation we used VM running on an Ubuntu 16.04 distribution with a version 4.4 Linux kernel as host OS on a Personal Computer (PC) with an Intel® Core™i5-2500 at 3.3GHz and 8GB of RAM. QEMU was used for virtualisation.

The first problem we had to solve was that our host system did not have a hardware TPM. But since Xen and Hyper-V need a hardware TPM in order to enable their build-in vTPM support we were not able to use them. In addition, we had some problems with the TPM emulator [Str04] as well as the VirtualBox extension for it. This problem was solved by using the software TPM [Ber14b] created by Berger. With this software TPM it is possible to have multiple TPMs running on the host system. The second problem was that QEMU has no build-in support for TPMs, neither virtual TPMs nor passed through hardware TPMs. Fortunately, Berger created a patch for QEMU [Ber14a] that adds support for vTPMs. This allowed us to create a software TPM on the host system for each VM and pass it through to each guest. By doing so the virtual TPM is already present during boot time of the VM allowing to create and store measurements of the boot process in it. The IMA can then calculate a boot aggregate from it. This way the IMA measurement list of our VMs does not differ from machines with a real TPM regarding to what measurements are in there.

Like the host system, also the VMs are running on Ubuntu 16.04. But in difference to the host system a customized version 4.8 Linux Kernel was used, to be able to use the Terminator LKM. Figure 6.1 on the following page illustrates the architecture of our test environment. While the guest VM 1 is used as Challenger, the guest VM 2 runs the Prover part and the IMARequestorService. In our tests we used the Challenger of guest VM 1 to check the integrity of the guest VM 2.

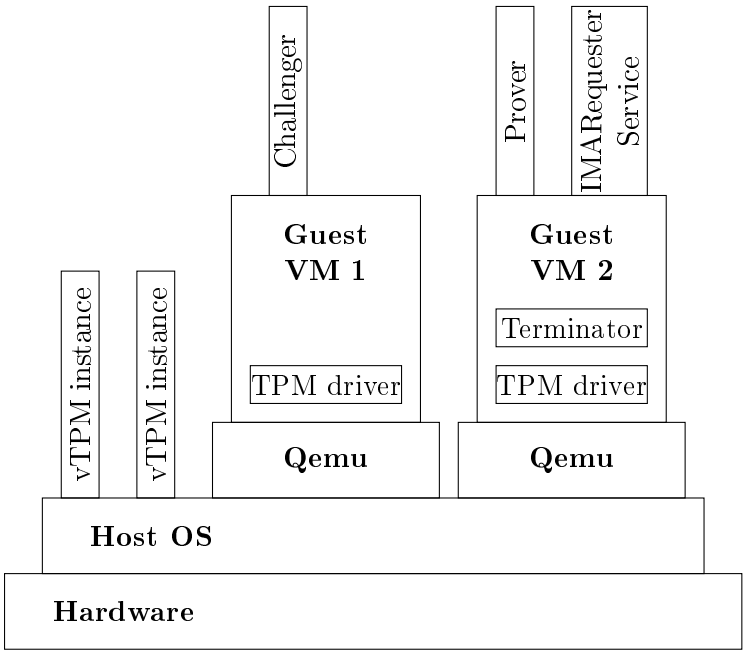

Figure 6.1: Test setup used for the evaluation of our implementation

6.2 Performance

Since there was no budget to test our implementation with multiple real CDs, we had to use virtualization to test our implementation. Nevertheless we decided to take performance measurements, to at least get some approximated values of the runtime of our improved RA protocol.

To take time measurements, we modify the code of the Challenger and add a "QElapsedTimer". This timer is started just before the protocol is started and stopped when it has finished. To take timings of single steps of the protocol we save the current elapsed time to a preallocated array. By holding this timings in memory during the protocol execution we try to minimize the impact of our measurements to the execution times of the protocol. The timings in this array are saved to a file on the hard-disk after the protocol has finished. Every data sample consists of 50 measurements to get a significant result.

In Figure 6.2 on page 52 you see the overall execution times of the protocol under certain preconditions. The first sample shows the measurements of a protocol execution where the Challenger connects to a certain Prover for the first time. So, the Challenger cannot take any of the shortcuts implemented in the protocol and has to execute every step. In average this takes 153 ms. The next boxplot shows a protocol execution where the Challenger already has the AIK certificate of the Prover, but there are some new signatures that have to be requested. This leads to an average execution time of 150 ms. The third sample is nearly the same with the difference that the Challenger has all signatures needed. The last plot shows the measurements of the case where the Challenger has a calculated composite hash of this Prover that was trustworthy and the state of the Prover is still the same. In this case the Challenger needs only one quote request to be sure that the Prover is trustworthy. As we can see this still takes an average time of 125 ms, although

in this case every shortcut in the protocol can be used. The reason for this can be seen in Figure 6.3 on page 53 where the execution times of the single protocol steps are mapped. It can easily be seen that the step with the biggest impact on the execution time is the quote request that has to be done on every protocol execution. Since the amount of data is very small during this step, the only operation that can consume that much time is the quote command of the TPM. At this point we also want to mention that we are using a vTPM so the Execution time of the quote command on a real TPM could be faster or even much slower. [Par10] shows that different TPM hardware implementations optimize different operations. Since the work only includes a plot of their results we had to approximate the timings. In their set-up Execution times for the quote command from about 330 ms to 888 ms were measured, depending on the chip that was used. An even higher value was measured by [RHI+17], where the quote command of the TPM on their embedded CD took 1.9 s.

This leads us to the result that the overall performance of our protocol highly depends on the performance of the quote command of the used TPM. Since we measured the single protocol steps, it is no problem to estimate the performance for another chip, as long as the execution time for the quote command is known. In our case the quote took 115 ms on average, all other steps together for the first time connection are 38 ms on average. In the case where all protocol shotcuts can be used the execution time without the quote is only 10 ms. So, the protocol execution time can be estimated by simply adding these values to the quote execution time. Of course, there are other factors that affect the execution time, like network speed, used hardware and so on. But this was not taken into account since it is very hard to consider this without knowing a specific scenario.

6.3 Maintenance Effort

The effort for maintaining the system highly depends on the number of measurements done by the IMA. Like described earlier there has to be a valid signature for every measurement. In our case the IMA had about 600 entries. This would mean that we need at least 600 signatures. But since we also want to revoke every single signature, every signature needs its own certificate. Resulting in the same amount of certificates. If we use the three-layer certificate architecture we proposed, this would add at least another certificate for every application. While this additional layer increases the amount of certificates, it simplifies the revocation of the trustworthiness of an application. Since we do not know how many versions of a binary or configuration are trusted, it is not possible to name a number of total needed certificates, but if we also count certificates for no longer trusted version it can easily reach a few thousand. But since all devices and used software are under strict control, we think that maintaining the certificates for them should be feasible.

6.4 Data Traffic

In this section we try to evaluate the amount of data that is sent over the network by our protocol. Table 6.1 on page 54 shows the data traffic of the RA protocol for a Prover with 600 entries in the IMA. Before the protocol execution the Challenger had no data about the Prover so the AIK certificate as well as all signatures have been requested. As mentioned

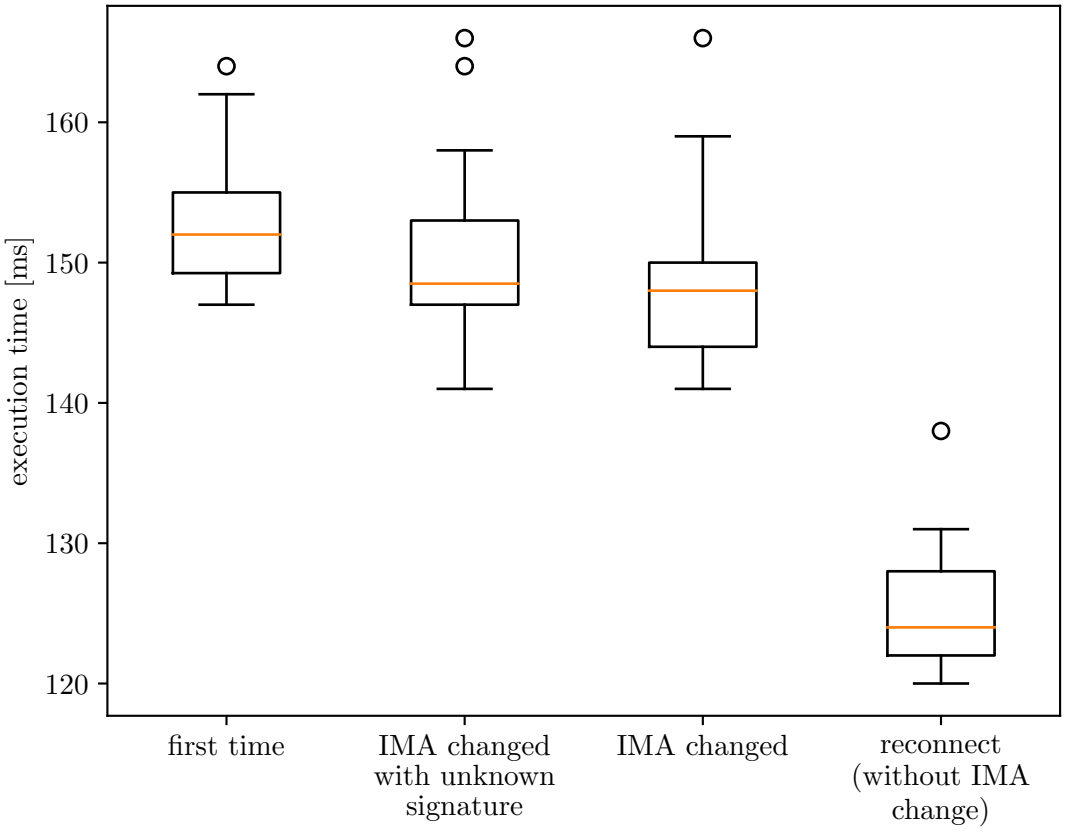

Figure 6.2: Overall protocol execution times under different preconditions

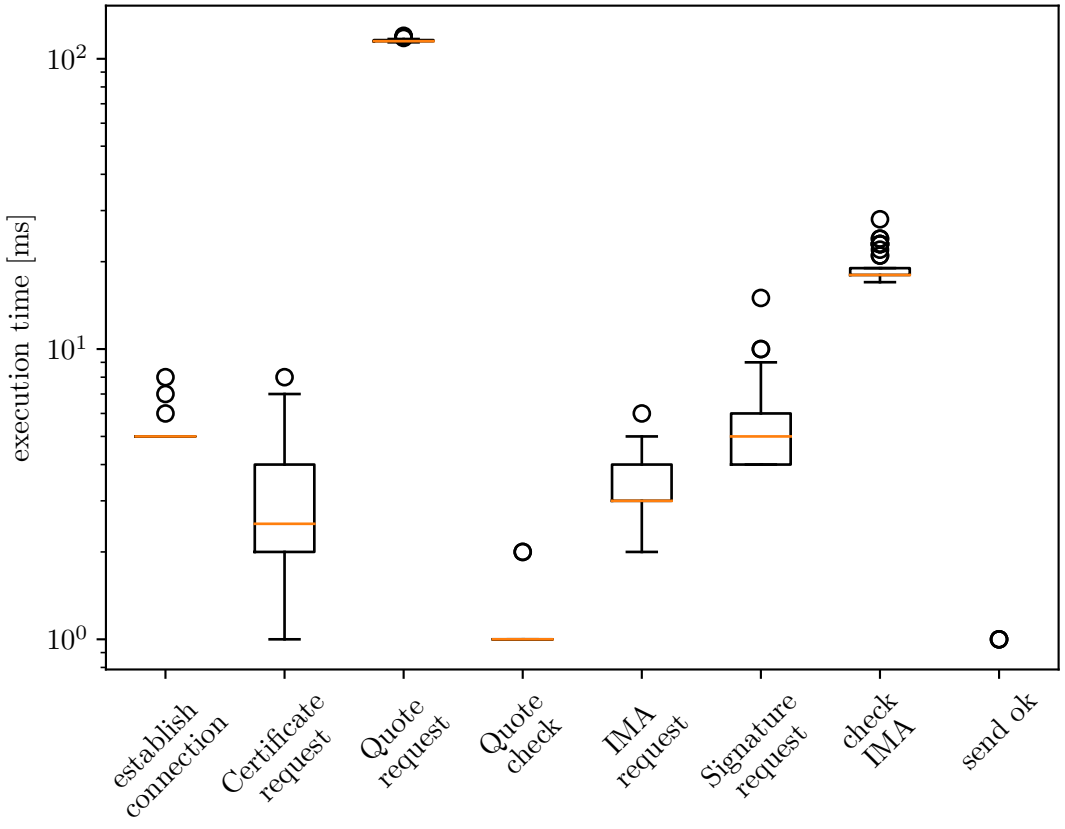

Figure 6.3: Execution times of protocol steps

	Quote	Certificate	IMA	Signature
request	36 B	12 B	12 B	14.41 kB
response	316 B	1.37 kB	32.20 kB	2.22 MB
sum	352 B	1.38 kB	32.21 kB	2.23 MB

Table 6.1: Data traffic by protocol step, request and response for a Prover with 600 entries in the IMA

in Section 6.3 on page 51 there are potentially a few thousand certificates. Nevertheless, during the RA of a host there are never more signatures needed than entries in the IMA. This means that in the worst case we have to request 1200 certificates and 600 signatures.

For certificates we used 4096 bit RSA with SHA256, which results in a file size of about 14.41 kB per certificate with Distinguished Encoding Rules (DER) encoding. Our signature also used SHA256 and and contained the certificate chain to simplify the verification process. For our three-layer certificate structure this results in a file size of about 3.7 kB per signatures in PKCS #7 CMS DER encoding. Since the signatures already include the certificate chain it is not necessary to request them. This means that in the worst case only 600 signatures have to be requested with a total size of about 2.2 MB. If we sum up all protocol steps only about 2.26 MB are transferred over the network.

6.5 Possible Applications

We identified three possible use cases where our protocol could be used:

Single check: Like the default RA protocol, our improved version can be used for a single integrity check of a remote host. When used in this mode our improved RA protocol only simplifies the deployment process of new software in the system. It is only necessary to deploy the new software and the corresponding signature to devices that are running this software. All the other devices will simply request the new signature during the protocol execution.

Real trusted network connection: In conjunction with the Terminator LKM our protocol can be used for our own version of TNC, with the difference that the actual network access is not managed by us. Rather we tested a way of establishing a secure connection between devices and executing our RA protocol on them. If the Prover is in a trusted state, we use the same connection for example to transfer data. As long as the connection is open the Challenger can be sure that the system state of the remote host has not been changed. This means that as long as the connection is open all data is forwarded to a trusted device. As soon as the system state changes the connection is automatically closed and the Challenger has to reconnect and check the integrity again before sending data can be resumed. Of course it can be dangerous to interrupt a network connection that is used to transmit important information, if there is no mechanism in place that ensures that no package gets lost. Problem could arise if the system requires a too short response time for our protocol execution. This is why we did not put more effort into this possible application of our protocol.

Monitor connection: The last use case we identified is called monitor connection by us. This means we have some kind of monitor process on the Challenger device that opens a connection to a remote host and executes our protocol on it. Like in the previous use case the Terminator LKM must run on the system of the Prover. If the Prover is in a trusted state, the connection is kept open. In case of a system state change at the remote host it is closed by the Terminator and the monitor process is informed about this change. If this happens, the monitor reopens the connection and executes the protocol again. Whenever the monitor detects an untrusted system state at the monitored remote device, this is reported in order to take appropriate steps. In our example the monitor is part of the Scari framework and reports the integrity violation to the other modules of the framework that actually plan and execute actions that should be taken to mitigate the possible attack.

Since the goal of this work was to detect attacks as soon as possible, we think that the monitor connection is the best way to achieve this. Of course, the protocol can also be used in the single check mode with polling. But this way we have a lot of overhead. In addition to the overhead there is still the problem that we do not know the state of the remote host between these single checks. Both problems are solved by the monitor connection. It cuts down the overhead by reducing the amount of protocol executions by only running on system state changes. In addition the timespan where the system state of the Prover is not known by the Challenger is reduced to the execution time of the protocol. This is possible since the Challenger is notified by the Prover on system state changes, but before this states actually is active.

Chapter 7

Conclusion and Future Work

In our implementation of the RA protocol, we split the default process up into several steps. This has the advantage that some of these steps become optional after the first RA execution, since we can simply cache some information like the AIK certificates, signatures and also trusted system states. By doing so, we are able to reduce the amount of data that has to be sent from the Prover to the Challenger. In the case of no system state change at the Prover, there are also less calculations necessary for the Challenger. By the use of signatures, we reduce the complexity of deploying new program versions. This is caused by the fact that no other system than the system where we deploy to, has to be updated. But using signatures has also several disadvantages, namely the added network traffic for exchanging the signatures during the protocol execution and the computation time needed for checking the validity of the signatures. The first disadvantage we tried to minimize by storing signatures and only requesting unknown signatures during the RA protocol. One additional disadvantage could be seen in the effort of managing all these signatures and certificates. Depending on the number of layers of certificates used for the signatures, the size of the CRL, which is needed to revoke old versions of a program, could become a problem. But using too little layers reduces the control over revoking trusted software. This is why we would recommend using two layers of certificates per application. The first layer would be a application-certificate that is only able to create new application-version-certificates which are able to sign the actual program versions. This way every application version would have its own certificate, allowing us to revoke single versions with known vulnerabilities. The application-certificate would also give us the ability to revoke all versions of an application by revoking a single certificate. This way we have very high control over what application a version is trusted and which not. But there is still one problem that has not been solved by us so far. It is not possible to deploy a new version and revoke the trustworthiness of the old version without a system reboot.

This is caused by the way the TPM extent command works. When a new version of an application is deployed, this new version is measured by the IMA and extent to previous value of the PCR. The old measurement is still be part of the chain of trust that was extended to this PCR. The problem is that the whole chain must be trusted in order to establish trust into the device. If we would revoke the trustworthiness of the old version while its measurement is still in the chain of trust, the system would be in an untrusted state. As mentioned above, the only solution so far is to reboot the system to clear the IMA measurement list. This is not a limitation of our protocol, rather a problem of RA in

general.

As mentioned in Section 2.1 it should be considered to switch to TPM version 2.0 since it is no longer recommended to use SHA-1. If this is done, the IMA should be configured to measure with SHA256. Continuing using SHA-1 could open up a possible attack vector on the RA process, since it is not any longer infeasible that an attack with sufficient computing power or budget could construct a hash collision for a malicious executable.

For productive usage, it is essential to link AIK and SSL certificates, to prevent masquerading attacks on the RA process.

Another possible enhancement of our protocol could be the combination with a privilege-based attestation approach from Section 3.5. This could cut down the number of trusted measurements and therefore reduce the amount of certificates and signatures needed. Besides the reduced maintenance effort for the certificates it would also reduce the amount of measurements that have to be checked during the protocol execution.

Glossary

Challenger The entity that wants to find out if a remote host is trustworthy during the Remote Attestation process.

nonce A number used only once.

Prover The entity that is going to prove that it is trustworthy during the Remote Attestation process.

Acronyms

ACPU Application CPU

AES Advanced Encryption Standard

AIK Attestation Identity Key

BIOS Basic Input Output System

CA Certification authority

CCPU Communication CPU

CD Control Device

CMS Cryptographic Message Syntax

CPS Cyber Physical System

CPU Central Processing Unit

CRL Certificate Revocation List

CRTM Core Root of Trust Measurement

CS Control System

DER Distinguished Encoding Rules

DH Deffie-Hellman

ECC Elliptic Curve Cryptography

EK Endorsement Key

FOSS Free and Open-Source Software

fTPM firmware-TPM

GRUB Grand Unified Bootloader

HMAC Keyed-Hash Message Authentication Code

ICS Industrial Control System

IMA Integrity Measurement Architecture

IMC Integrity Measurement Collector

IMV Integrity Measurement Verifier

IOT Internet of Things

LKM Linux Kernel Module

MAC Mandatory Access Control

NAA Network Access Authority

NAE Network Access Enforcer

NAR Network Access Requestor

OCSP Online Certificate Status Protocol

OS Operating System

OSI model Open Systems Interconnection model

PC Personal Computer

PCR Platform Configuration Register

PKCS Public Key Cryptography Standards

PKI Public Key Infrastructure

PRIMA Policy-Reduced Integrity Measurement Architecture

RA Remote Attestation

RAD Resource Access Description

RNG Random number generator

RSA Rivest, Shamir und Adleman

SCADA Supervisory Control and Data Acquisition

SELinux Security-enhanced Linux

SGX Software Guard Extensions

SHA Secure Hash Algorithm

SIMA Sensory Integrity Measurement Architecture

SML Stored Measurement Log

SoC System-on-a-Chip

SRK Storage Root Key

SSL Secure Sockets Layer

TA Trusted Applications

TC Trusted Computing

TCB Trusted Computing Base

TCG Trusted Computing Group

TCP Transmission Control Protocol

TEE Trusted Execution Environment

TNC Trusted Network Connection

TOCTTOU Time of Check To Time of Use

TPM Trusted Platform Module

TSS Trusted Software Stack

TTP Trusted Third Party

USB Universal Serial Bus

VM Virtual Machine

vTPM virtual TPM

Bibliography

[ANS+09] Ahmed M Azab, Peng Ning, Emre C Sezer, and Xiaolan Zhang. HIMA: A Hypervisor-Based Integrity Measurement Agent. In *Computer Security Applications Conference, 2009. ACSAC'09. Annual*, pages 461–470. IEEE, 2009 (cited on pages 26, 28).

[Ber14a] Stefan Berger. QEMU with CUSE vTPM support, project site, 2014. URL: `https://github.com/stefanberger/qemu-tpm` (visited on 07/13/2017) (cited on pages 26, 27, 49).

[Ber14b] Stefan Berger. SWTPM - Software TPM Emulator, project site, 2014. URL: `https://github.com/stefanberger/swtpm` (visited on 07/13/2017) (cited on pages 26, 27, 49).

[Bib77] Kenneth J Biba. Integrity Considerations for Secure Computer Systems. Technical report, MITRE CORP BEDFORD MA, 1977 (cited on page 17).

[CLM+08] Liqun Chen, Hans Löhr, Mark Manulis, and Ahmad-Reza Sadeghi. Property-Based Attestation without a Trusted Third Party. *Information Security*:31–46, 2008 (cited on pages 13, 14).

[Cor14] Intel Corporation. Intel Software Guard Extensions Programming Reference. online, 2014. URL: `https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf` (visited on 10/12/2017) (cited on page 30).

[DAD+11] Edita Djambazova, Magnus Almgren, Kiril Dimitrov, and Erland Jonsson. Emerging and Future Cyber Threats to Critical Systems. *Open Research Problems in Network Security*:29–46, 2011 (cited on page 1).

[FMC11] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32.Stuxnet Dossier. *White paper, Symantec Corp., Security Response*, 5(6), 2011 (cited on pages iii, iv, 1, 9).

[GPS06] Kenneth Goldman, Ronald Perez, and Reiner Sailer. Linking Remote Attestation to Secure Tunnel Endpoints. In *Proceedings of the First ACM Workshop on Scalable Trusted Computing*, STC '06, pages 21–24, Alexandria, Virginia, USA. ACM, 2006 (cited on pages 10, 11, 48).

[Hou99] Russell Housley. Cryptographic Message Syntax, whitepaper, 1999. URL: `https://tools.ietf.org/html/rfc5652` (visited on 06/02/2017) (cited on page 47).

[IAI13] IAIK. jTSS, project site, 2013. URL: `http://trustedjava.sourceforge.net/` (visited on 06/27/2017) (cited on page 4).

[IRK+17] Johannes Iber, Tobias Rauter, Michael Krisper, and Christian Kreiner. An Integrated Approach for Resilience in Industrial Control Systems. In *Dependable Systems and Networks Workshop (DSN-W), 2017 47th Annual IEEE/IFIP International Conference on*, pages 67–74. IEEE, 2017 (cited on pages iii, iv, 1, 33, 46, 47).

[JSS06] Trent Jaeger, Reiner Sailer, and Umesh Shankar. PRIMA: Policy-Reduced Integrity Measurement Architecture. In *Proceedings of the eleventh ACM symposium on Access control models and technologies*, pages 19–28. ACM, 2006 (cited on pages 16–18).

[Km15] Dmitry Kasatkin and mzohar. Integrity Measurement Architecture (IMA), wiki, version 23, 2015. URL: `https://sourceforge.net/p/linux-ima/wiki/Home/` (visited on 01/17/2017) (cited on page 6).

[KR15] Markku Kylänpää and Aarne Rantala. Remote Attestation for Embedded Systems. In *Conference on Cybersecurity of Industrial Control Systems*, pages 79–92. Springer, 2015 (cited on pages 30, 31).

[KSA+09] Chongkyung Kil, Emre C Sezer, Ahmed M Azab, Peng Ning, and Xiaolan Zhang. Remote Attestation to Dynamic System Properties: Towards Providing Complete System Integrity Evidence. In *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*, pages 115–124. IEEE, 2009 (cited on pages 15, 16).

[LAC16] Robert M Lee, Michael J Assante, and Tim Conway. Analysis of the Cyber Attack on the Ukrainian Power Grid. *SANS Industrial Control Systems*, 2016 (cited on pages 1, 9).

[Lai+16] Ashley Lai et al. TrouSerS, project site, 2016. URL: `http://trousers.sourceforge.net/` (visited on 06/27/2017) (cited on page 4).

[Lan11] Ralph Langner. Stuxnet: Dissecting a Cyberwarfare Weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011 (cited on pages 1, 9).

[MB17] Dennis Mattoon and Erich Barnstedt. TSS.MSR, project site, 2017. URL: `https://github.com/Microsoft/TSS.MSR` (visited on 06/27/2017) (cited on page 4).

[MR12] Bill Miller and Dale Rowe. A Survey of SCADA and Critical Infrastructure Incidents. In *Proceedings of the 1st Annual conference on Research in information technology*, pages 51–56. ACM, 2012 (cited on pages iii, iv, 1).

[Par10] Bryan Jeffrey Parno. Trust Extension as a Mechanism for Secure Code Execution on Commodity Computers, 2010 (cited on page 51).

[PSD+06] Ronald Perez, Reiner Sailer, Leendert van Doorn, et al. vTPM: Virtualizing the Trusted Platform Module. In *Proc. 15th Conf. on USENIX Security Symposium*, pages 305–320, 2006 (cited on pages 24–26).

[RHI+17] Tobias Rauter, Andrea Höller, Johannes Iber, Michael Krisper, and Christian Kreiner. Integration of Integrity Enforcing Technologies into Embedded Control Devices: Experiences and Evaluation. In *Dependable Computing (PRDC), 2017 IEEE 22nd Pacific Rim International Symposium on*, pages 155–164. IEEE, 2017 (cited on page 51).

[RHK+15] Tobias Rauter, Andrea Höller, Nermin Kajtazovic, and Christian Kreiner. Privilege-Based Remote Attestation: Towards Integrity Assurance for Lightweight Clients. In *Proceedings of the 1st ACM Workshop on IoT Privacy, Trust, and Security*, pages 3–9. ACM, 2015 (cited on page 19).

[RSW+15] Himanshu Raj, Stefan Saroiu, Alec Wolman, Ronald Aigner, Jeremiah Cox, Paul England, Chris Fenner, Kinshuman Kinshumann, Jork Loeser, Dennis Mattoon, et al. fTPM: A Firmware-based TPM 2.0 Implementation. *Microsoft Research*, 2015 (cited on page 30).

[SBK+17] Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, and Yarik Markov. The first collision for full SHA-1. Technical report, Cryptology ePrint Archive, Report 2017/190, 2017 (cited on page 40).

[SJS06] Umesh Shankar, Trent Jaeger, and Reiner Sailer. Toward Automated Information-Flow Integrity Verification for Security-Critical Applications. In *NDSS*, 2006 (cited on page 17).

[SKP16] Marc Stevens, Pierre Karpman, and Thomas Peyrin. *Freestart Collision for Full SHA-1.* In *Advances in Cryptology – EUROCRYPT 2016: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part I.* Marc Fischlin and Jean-Sébastien Coron, editors. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016, pages 459–483 (cited on page 40).

[SKU10] Björn Stelte, Robert Koch, and Markus Ullmann. Towards Integrity Measurement in Virtualized Environments – A Hypervisor based Sensory Integrity Measurement Architecture (SIMA). In *Technologies for Homeland Security (HST), 2010 IEEE International Conference on*, pages 106–112. IEEE, 2010 (cited on pages 28, 29).

[Sol15] Verizon Enterprise Solutions. Data Breach Investigations Report. *Verizon, Report*, 2015 (cited on page 9).

[SS04] Ahmad-Reza Sadeghi and Christian Stüble. Property-based Attestation for Computing Platforms: Caring about properties, not mechanisms. In *Proceedings of the 2004 workshop on New security paradigms*, pages 67–77. ACM, 2004 (cited on page 11).

[STR+06] Frederic Stumpf, Omid Tafreschi, Patrick Röder, Claudia Eckert, et al. A Robust Integrity Reporting Protocol for Remote Attestation. In *Second Workshop on Advances in Trusted Computing (WATC'06 Fall)*, pages 25–36, 2006 (cited on pages 10–12).

[Str04] Mario Strasser. A Software-based TPM and MTM Emulator, project site, 2004. URL: `https://github.com/PeterHuewe/tpm-emulator` (visited on 07/13/2017) (cited on pages 26, 27, 49).

[SZ10] Christian Stüble and Anoosheh Zaerin. μTSS: A Simplified Trusted Software Stack. In *Proceedings of the 3rd International Conference on Trust and Trustworthy Computing*, TRUST'10, pages 124–140, Berlin, Germany. Springer-Verlag, 2010 (cited on page 4).

[SZJ+04] Reiner Sailer, Xiaolan Zhang, Trent Jaeger, and Leendert Van Doorn. Design and Implementation of a TCG-based Integrity Measurement Architecture. In *USENIX Security Symposium*, volume 13, pages 223–238, 2004 (cited on page 6).

[Tru11a] Trusted Computing Group. TPM Main Specification - Part 1 Design Principles. online, whitepaper, version 1.2 revision 116, 2011. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-1-Design-Principles_v1.2_rev116_01032011.pdf` (visited on 03/24/2017) (cited on pages 3, 9, 34).

[Tru11b] Trusted Computing Group. TPM Main Specification - Part 3 Commands. online, whitepaper, version 1.2 revision 116, 2011. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Main-Part-3-Commands_v1.2_rev116_01032011.pdf` (visited on 03/24/2017) (cited on page 3).

[Tru12] Trusted Computing Group. TCG Trusted Network Communications TNC Architecture for Interoperability. online, whitepaper, version 1.5 revision 4, 2012. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TNC_Architecture_v1_5_r4.pdf` (visited on 04/27/2017) (cited on pages 20, 22, 23).

[Tru16] Trusted Computing Group. Trusted Platform Module Library - Part 1: Architecture. online, whitepaper, version 2.0 revision 01.38, 2016. URL: `https://trustedcomputinggroup.org/wp-content/uploads/TPM-Rev-2.0-Part-1-Architecture-01.38.pdf` (visited on 03/24/2017) (cited on page 3).

[WD13] Johannes Winter and Kurt Dietrich. A hijacker's guide to communication interfaces of the trusted platform module. *Computers & Mathematics with Applications*, 65(5):748–761, 2013 (cited on page 34).

[WH12] Marcel Winandy and Sören Heisrath. Virtual TPM for Virtualbox, project site, 2012. URL: `http://www.trust.rub.de/projects/vTPM/` (visited on 07/13/2017) (cited on pages 26, 27).

[XZH+12] Wenjuan Xu, Xinwen Zhang, Hongxin Hu, Gail-Joon Ahn, and Jean-Pierre Seifert. Remote Attestation with Domain-Based Integrity Model and Policy Analysis. *IEEE Transactions on Dependable and Secure Computing*, 9(3):429–442, 2012 (cited on pages 18, 19).

Index