JOHANNES KEPLER
UNIVERSITY LINZ

Author
Michael Preisach
1155264

Submission
Institute for Networks and Security

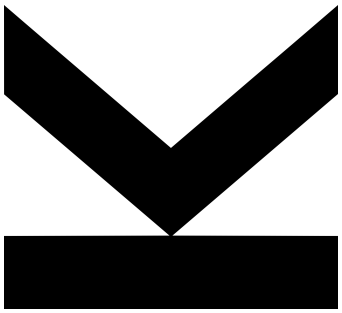
First Supervisor
Univ.-Prof. DI Dr. **René Mayrhofer**

Second Supervisor
DI **Tobias Höller**

Assistant Thesis Supervisor / Mitbetreuung
Dr. **Michael Roland**

July 2021

Project Digidow: Biometric Sensor



JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

JOHANNES KEPLER
UNIVERSITY LINZ

Statutory Declaration

I hereby declare that the thesis submitted is my own unaided work, that I have not used other than the sources indicated, and that all direct and indirect sources are acknowledged as references.

This printed thesis is identical with the electronic version submitted.

Linz, July 2021

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

What is it all about? Why is that interesting? What is new in this thesis? Where is the solution directing to?

Kurzfassung

Das am Institut für Netzwerke und Sicherheit entwickelte Projekt *Digital Shadow* benötigt in vielen Bereichen ein prüfbares Vertrauen um eine Erkennung von Nutzern anhand ihrer biometrischen Daten zu erkennen und Berechtigungen zuzuteilen. Das Vertrauen soll dem Nutzer die Möglichkeit geben, die Korrektheit des Systems schnell und einfach zu prüfen, bevor er/sie disesm System biometrische Daten zur Verfügung stellt Diese Masterarbeit beschäftigt sich nun mit den existierenden Werkzeugen, die ein solches Vertrauen schaffen können. Das implementierte System kombiniert diese Werkzeuge, um damit sensible Daten von Nutzern aufzunehmen und im Netzwerk von Digital Shadow zu identifizieren. Es soll dabei sicher gestellt sein, dass eine fälschliche Verwendung der sensiblen Nutzerdaten ausgeschlossen wird. Anhand dieses Systems werden die Eigenschaften einer vertrauenswürdigen Umgebung für Software diskutiert und notwendige Rahmenbedingungen erläutert.

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

JOHANNES KEPLER
UNIVERSITY LINZ

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World, funded by the Christian Doppler Forschungsgesellschaft, 3 Banken IT GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH, and Österreichische Staatsdruckerei GmbH.

1 Introduction

We all live in a world full of digital systems. They appear as PCs, notebooks, cellular phones or embedded devices. Especially the footprint of embedded computers became so small that they can be used in almost all electrical devices. These embedded systems form the so called *smart* devices.

All these new devices made life a lot easier in the past decades. Many of them automate services to the public like managing the bank account, public transportation or health services. The list of digital service is endless and will still grow in the future.

The downside of all these digital services is that using these services generate a lot of data. Besides the intended exchange of information, many of the services try to extract metadata as well. Metadata answers some of the following questions. Which IP address is sending or receiving? What kind of device is that? Is the software of the device up to date? Was this device here in the past? What else did the owner on the device? This set of questions is not complete.

Aggregating metadata is not required to fulfill the function of the requested service. However, aggregating and reselling the metadata brings the provider more margin on the product and hence more profit. Consequently, the market for metadata is growing and yet only partly regulated. Since metadata aggregation is one downside of using smart services, providers try to downplay or to hide these aggregation features where possible. Often a proprietary layer is used either on the client or the server side to hide those functions. The result is a piece of software which is provided as binary and the user cannot prove what this software is exactly doing besides the visible front end features.

There are of course other purposes for delivering software in a closed source manner. Firmware of hardware vendors is usually not disclosed. Instead, vendors provide an API where an *Operating System* (OS) can connect to. Some companies deliver complete closed source devices with internet connection. In such cases, the feature of closed source is

to protect the intellectual property of those companies. Any user of these closed source products must use them as black box and needs to *trust* the vendor that it is working correctly.

There is, however, a special need for users to keep sensitive data secret. Especially when providing confidential data like passwords or biometric data, a certain level of trust is required. This means that the user assumes that the provided sensitive data is handled properly for only the designated usage. One may argue that a password can easily be changed when revealed to the public. Unfortunately, this does not apply to a fingerprint since a human usually has only ten of them during lifetime.

1.1 Trust

When using a system with an authentication method, trust plays a key role. For black box systems this trust is cast to the vendor of the system or device. There is however no mathematical proof that the device is indeed executing the software as intended from the vendor.

This thesis will therefore use the term *trust* as a cryptographic chain of proofs, that a system is behaving in an intended way, a so called *Chain of Trust*. By providing a Chain of Trust, a user can ask the vendor for a certification of its devices and consequently comprehend the state of the system at hand. The Chain of Trust will be separated into two parts, namely the creation of trust on a certain system, and the transfer of trust over the network for verification purposes.

1.2 Project Digidow

The Institute for Networks and Security is heavily using the cryptographic form of trust in the project *Digital Shadow* (Digidow). DigiDow introduces an electronic authentication system, which aims to minimize any generation of metadata on system and network level and hence maximizes the level of privacy for their users. The project furthermore aims to specify a scalable solution for nationwide or even worldwide applications including provable trust and integrity to the user.

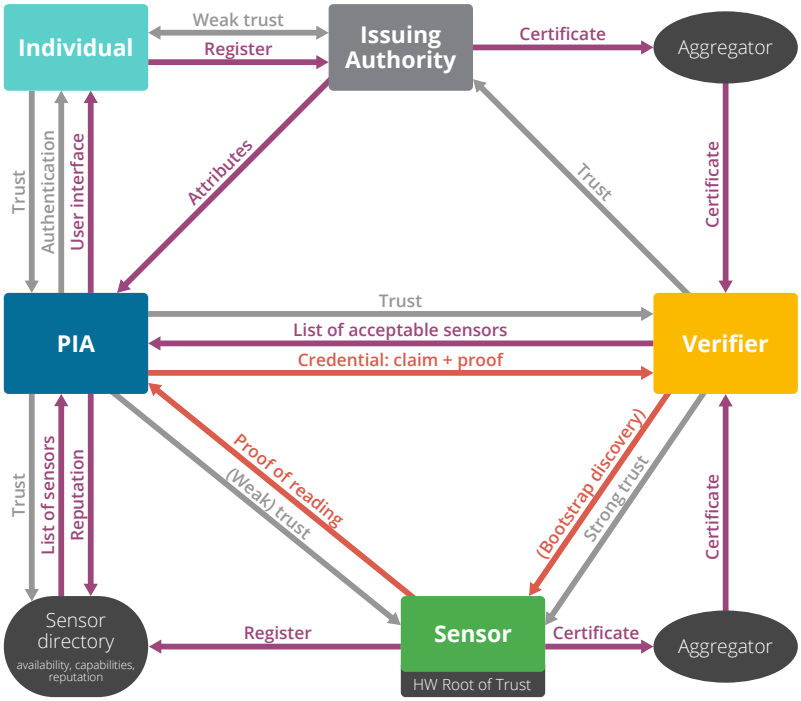


Figure 1.1: Overview of the Digidow network with its interactions

The picture in Figure 1.1 provides an overview of the authentication process within Digidow. At the time of this writing, the exact order and definition of every step is not yet finished and may change during the progress of the whole project. DigiDow introduces three main parties which are involved in a common authentication process.

- *Personal Identity Agent* (PIA): The PIA is the digital shadow of an individual who wants to be authenticated. This individual is also the owner of the PIA and should be able to manage sensitive data and software on it.
- *Verifier* (V): This is the party that verifies the whole authentication process and may finally trigger the desired action if all went well.
- *Biometric Sensor* (BS): For authentication, an individual has to be uniquely identified. Therefore, the BS records biometric data from the individual and passes it into the DidiDow network.

For scalability, we assume that there are large numbers of all parties. The illustration also shows a draft of how which steps need to be performed between above mentioned parties during an authentication process.

(1) All relevant parties need to find each other via the Digidow network. When this step is finished, it is assumed that for every step the individual hosts for communication are identifiable and ready for the authentication process.

(2)(3) Eventually an individual wants to authenticate itself and the BS records the biometric data. With this data and a corresponding unique ID, the BS knows which PIA to contact.

(4)(5)(6) The BS contacts the PIA and sends the recorded dataset as well as a cryptographic signature to proof that the sensor is valid and this is an honest authentication attempt.

(7) The PIA proves authenticity of the received signature and compares the data with its own saved biometric datasets. Assuming all is correct, the PIA certifies that the person standing in front of the BS is indeed the owner of the PIA. The verifier checks the certification and finally triggers the desired action for the asking individual.

The above illustration is an early draft of the whole setup and is under constant development. A more recent version of the whole system can be found at the Digidow project page[1]. This thesis will contribute a prototype setup the Biometric Sensor and discuss how to create trust into this system.

1.3 Our Contribution: Deriving Trust from the Biometric Sensor

The Digidow network is designed to preserve privacy and to build trust for any user. A key feature is to show the user that all involved parts of the system are working as intended. So we design a prototype based on the common x86 architecture and use the cryptographic features of *Trusted Platform Modules* (TPM). A TPM is a passive crypto coprocessor available on many modern PC platforms which has an independent storage for crypto variables and provides functions to support above mentioned features.

We define a solution for installing and booting a Linux kernel with TPM-backed integrity measurements in place. We use an attached camera as example for a biometric sensor hardware to create the dataset to continue with the authentication process. This dataset will be combined with the integrity measurements of the system and a signature from the TPM and finally sent to the PIA for verification and further computation.

By building a system with an integrated TPM, the system should be able to provide the following properties:

- *Sensor Monitoring.* The system should be able to monitor the hardware sensor (fingerprint sensor, camera, etc.) itself.
- *System Monitoring.* It should be possible to track the state of the system. Especially every modification of the system at hardware level should be detected.
- *Freshness of Sensor Data.* To prevent replay attacks, the system should prove that the provided biometric data is captured live.
- *Integrity of Sensor Data.* As it is possible for an adversary to modify the provided data during the identification process, integrity should guarantee that the data is unmodified until identification is done.

[1] https://digidow.eu

- *Confidentiality of Sensor Data.* It should not be possible to eavesdrop any sensitive data out of the system. Furthermore almost all kinds of metadata (e. g. information about the system or network information) should not be published.
- *Anonymity.* Given a message from a BS, an adversary should not be able to detect which BS created it.
- *Unforgeability.* Only honest BS should be able to be part of the Digidow network. Corrupt systems should not be able to send valid messages.

The thesis focuses on a working setup as basis for future research. Since the Digidow protocols are not yet finalized, some assumptions are defined for this work and the prototype implementation:

- Any network discovery (Step 1 in **??**) is omitted. BS and PIA are assumed to be reachable directly via TCP/IP.
- We look into a protocol which proves trustworthiness from BS to PIA. Any further proofs necessary for a Digidow Verifier are also not focused in this thesis.
- The sensitive datasets will be transmitted in cleartext between BS and PIA. It is considered easy to provide an additional layer of encryption for transportation. However this should be considered in the Digidow network protocol design. This thesis focuses only on the trust part between BS and PIA.
- Any built system is considered secure on a hardware level. Any threats which are attacking the system without changing any running software on the system may remain undetected. This includes USB wire tapping or debug interfaces within the system revealing sensitive information.

1.4 Description of structure

In chapter 2 we will outline a variety of projects which do not contribute to this thesis. There is, however, scientific work that is used as scientific background to this thesis as described in chapter 3. This includes especially the theoretical foundations of the network protocol. Together with that, we will introduce our theoretical solution for the previously stated problems in chapter 4. Chapter 5 introduces then a working implementation with all

necessary parts for a working prototype. Finally we will present the results and limitations in chapter 6 and give an overview of future work.

2 Related Work

There exist already a variety projects and implementations which touch the field of trusted computing. We will introduce some of these projects and discuss why these do not meet the purpose of this thesis.

Schear et al. [18] developed a full featured trusted computing environment for cloud computing. They show in their paper how a TPM of a hypervisor can be virtualized and used by the guest operating system. This includes trusted bootstrapping, integrity monitoring, virtualization, compatibility with existing tools for fleet management and scalability. The concept of a well known virtual environment does, however, not apply to our contribution. Furthermore, the system should be self contained as good as possible and it should be possible to get information about the system via anonymous attestation.

The *Fast IDentity Online* Alliance (FIDO) is an organization which standardizes online authentication algorithms. When the first generation of TPMs were available, the consortium defined a standard for Direct Anonymous Attestation with Elliptic Curve cryptography (ECDAA). When the newer standard, TPM 2.0, was published, FIDO decided to update their algorithm to be compatible with recent developments. This standard is still in development; a draft version from February 2018 is published on the FIDO website[1]

- What exists in the field?
- Keylime – DONE
- Xaptum ECDAA – part of concept
- FIDO 2 ECDAA – noteworthy in background?
- Strongswan Attestation –
- Linux IMA – mentioned in Background

- Secure Boot – in difference to trusted boot
- Intel TXT
- Trusted Execution Environment (TEE)
- nanovm (nanovms.com)

3 Background

In this chapter we describe four main concepts which will be combined in the concept of this thesis. The TPM standard is used to introduce trust into the used host platforms. *Trusted Boot* and the *Integrity Measurement Architecture* (IMA) are two approaches to extend trust from the TPM over the UEFI / BIOS up to the OS. The generated trust should then be provable by an external party—in our case the PIA—by using the protocol of *Direct Anonymous Attestation* (DAA).

3.1 Trusted Platform Module (TPM)

The *Trusted Platform Module* (TPM) is a small coprocessor that introduces a variety of cryptographic features to the platform. This module is part of a standard developed by the Trusted Computing Group (TCG), which released the current revision 2.0 in 2014[20].

The hardware itself is strongly defined by the standard and comes in the following flavors:

- *Dedicated device.* The TPM chip is mounted on a small board with a connector. The user can plug it into a compatible compute platform. This gives most control to the end user since it is easy to disable trusted computing or to switch to another TPM.
- *Mounted device.* The dedicated chip is directly mounted on the target mainboard. Therefore, removing or changing the TPM is imossible. All recent Intel and AMD platforms supporting TPM2.0 are able to manage a TPM within the BIOS, even as mounted device.
- *Firmware TPM (fTPM).* This variant was introduced with the TPM2.0 Revision. Instead of using a dedicated Coprocessor for the TPM features, this variant lives as

firmware extension within Intel's Management Engine or AMD's Platform Security Processor. Both Intel and AMD provide this extension for their platforms for several years now. When activating this feature on BIOS level, the user gets the same behavior as when using a mounted device.

- *TPM Simulator.* For testing reasons, it is possible to install a TPM simulator. It provides basically every feature of a TPM but cannot be used outside the OS. Features like Trusted Boot or in hardware persisted keys are not available.

Dedicated and mounted devices are small microcontrollers that run the TPM features in software giving the manufacturer the possibility to update their TPMs in the field. fTPMs will be updated with the platform updates of the CPU manufacturers.

The combination of well constrained hardware and features, an interface for updates and well defined software interfaces make TPMs trustworthy and reliable. When looking up the term *TPM* in the Common Vulnerabilities and Exposures database, it returns 23 entries[8]. Eight of them were filed before the new standard has been released. Another seven entries refer to vulnerabilities in custom TPM implementations. Six entries refer to the interaction between the TPM and the operating system, especially the TPM library and the shutdown/boot process. The last two entries describe vulnerabilities in dedicated TPM chips, which are mentioned in further detail:

- *CVE-2017-15361*: TPMs from Infineon used a weak algorithm for finding primes during the RSA key generation process. This weakness made brute force attacks against keys of up to 2048 bits length feasible. According to Nemec et al.[14], 1024 bit keys required in the worst case scenario 3 CPU months and 2048 bit keys needed 100 CPU years. Infineon was able to fix that vulnerability per firmware update for all affected TPMs.

- *CVE-2019-16863*: This vulnerability is also known as "*TPM fail*" [13] and shows how to get an elliptic curve private key via timing and lattice attacks. The authors found TPMs from STMicroelectronics vulnerable, as well as Intel's fTPM implementation. Infineon TPM show also some non-expected behaviour, but this could not be used for data exfiltration. STM provided an update like Infineon did for their TPMs. Intel's fTPM required a platform firmware update to solve the issue.

3.1.1 Using the TPM

On top of the cryptographic hardware, the TCG provides several software interfaces for application developers:

- *System API (SAPI).* The SAPI is a basic API where the developer has to handle the resources within the application. However, this API provides the full set of features.
- *Enhanced System API (ESAPI).* While still providing a complete feature set, the ESAPI makes some resources transparent to the application like session handling. Consequently, this API layer is built on top of the SAPI.
- *Feature API (FAPI).* This API layer is again built on top of the ESAPI. It provides a simple to use API but the feature set is also reduced to common use cases. Although the interface was formally published from the beginning, an implementation is available only since end of 2019.

The reference implementation of these APIs is published on Github[7] and is still under development. The repositories are maintained by members of TCG. At the point of writing stable interfaces are available for C and C++, but other languages like Rust, Java, C# will be served in the future. The repository additionally provides the tpm2-tools toolset which provides the FAPI features to the command line. Unfortunately, the command line parameters changed several times during the major releases of tpm2-tools[16].

3.1.2 The Hardware

With the previous mentioned software layers the TCG achieved independence of the underlying hardware. Hence, these layout made the different flavors of TPMs possible

With the TPM2.0 standard, TCG defined a highly constrained hardware with a small feature set. It is a passive device with some volatile and non-volatile memory, which provides hardware acceleration for a small number of crypto algorithms. The standard allows to add some extra functionality to the device. However, the TPMs used in this project provide just the minimal set of algorithms and also the minimal amount of memory.

Since TCG published its documents, several IT security teams investigated the concept and implementations of TPMs.

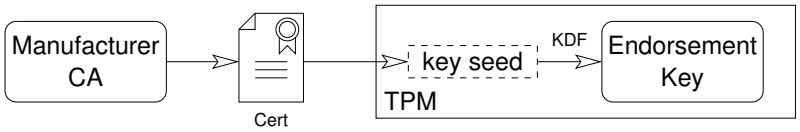


Figure 3.1: The manufacturer certifies every TPM it produces

3.1.3 TPM Key Hierarchies

A TPM comes with four different key hierarchies. These hierarchies fulfill different tasks and are used in different use cases on the whole platform. Will Arthur et al.[2] provide a more detailed description on how the hierarchies work together.

- *Platform Hierarchy*: This hierarchy is managed by the platform manufacturer. The firmware of the platform is interacting with this hierarchy during the boot process.
- *Storage Hierarchy*: The storage of a platform is controlled by either an IT department or the end user and so is the storage hierarchy of the TPM. It offers non-privacy related features to the platform although the user may disable the TPM for her own use.
- *Endorsement Hierarchy*: This is the privacy-related hierarchy which will also provide required functionality to this project. It is controlled by the user of the platform and provides the keys for attestation and group membership.
- *NULL Hierarchy*: The NULL hierarchy is the only non-persistent hierarchy when rebooting the platform. It provides many features of the other hierarchies for testing purposes.

Each of the persistent hiearchies represent its own tree of keys, beginning with a root key. Since TPM2.0 was published, these root keys are not hard coded anymore and can be changed if necessary. The process of key generation described below is similar to all three persistent hierarchies.

3.1.4 Endorsement Key

The *Endorsement Key* (EK) is the root key for the corresponding hierarchy. Figure 3.1 illustrates the certificate chain of building a new EK. Every TPM has, instead of the full

EK, a unique key seed to derive root keys from. This key seed comes with a corresponding certificate. This TPM certificate is signed by the TPM manufacturer by using its own root *Certificate Authority* (CA). When the platform user wants to create a new EK, a *Key Derivation Function* (KDF) generates this new EK such that the TPM certificate identifies it and the chain keeps intact. Since the platform supports root key generation, it is also possible to encrypt the key and store it on an external storage, e.g. on the platform disk. Consequently it is quite easy to have different EKs at once to address privacy features also between different functions of the endorsement hierarchy.

3.2 Trusted Boot

A boot process of modern platforms consists of several steps until the OS is taking over the platform. During these early steps, the hardware components of the platform are initialized and some self tests are performed. This is controlled by either the BIOS (for legacy platforms) or the UEFI firmware. There exists no source of trust and hence no check for integrity or intended execution in this common boot procedure.

3.2.1 Platform Configuration Register

The *Trusted Computing Group* (TCG) introduced their first standard for a new Trusted Computing Module (TPM) in 2004. As part in this standard, TCG defined a procedure where every step in the early boot process is measured and saved in a *Platform Configuration Register* (PCR). In this context, *Measuring* means a simple cryptographic extension function:

$$\text{new\_PCR} = \mathit{hash}(\text{old\_PCR} \,||\, \text{data}). \tag{3.1}$$

The function "||" represents a concatenation of two binary strings and the hash function is either SHA1 or SHA256. In recent TPM-platforms, both hashing algorithms can be performed for each measurement. Consequently, both hash results are available for further computations.

The formula shows that a new PCR value holds the information of the preceeding value as well. This *hash chain* enables the user to add an arbitrary number of hash computations.

One can clearly see that the resulting hash will also change when the order of computations changes. Therefore, the BIOS/UEFI has to provide a deterministic way to compute the hash chain if there is more than one operation necessary. The procedure of measurements is available since the first public standard TPM1.2. For TPM2.0, the process was only extended with the support with the newer SHA256 algorithm.

A PCR is now useful for a sequence of measurements with similar purpose. When, for example, a new bootloader is installed on the main disk, the user wants to detect this with a separate PCR value. The measured firmware blobs may still be the same. So the TPM standard defines 24 PCRs for the PC platform, each with a special role and slightly different feature set. The purpose of every PCR is well defined in Section 2.3.3 of the *TCG PC Client Platform Firmware Profile*[11] and shown in table 3.1. Especially those PCRs involved in the boot process must only be reset according to a platform reset. During booting and running the system these registers can only be *extended* with new measurements.

Table 3.1: Usage of PCRs during an UEFI trusted boot process

PCR	*Explanation*
0	SRTM, BIOS, host platform extensions, embedded option ROMs and PI drivers
1	Host platform configuration
2	UEFI driver and application code
3	UEFI driver and application configuration and data
4	UEFI Boot Manager code and boot attempts
5	Boot Manager code configuration and data and GPT / partition table
6	Host platform manufacturer specific
7	Secure Boot Policy
8–15	Defined for use by the static OS
16	Debug
17–23	Application

When TCG introduced Trusted Boot in 2004, UEFI was not yet available for the ordinary PC platform. Consequently, TCG standardized the roles of every PCR only for the BIOS platform. Later, when UEFI became popular, the PCR descriptions got adopted for the new platform.

3.2.2 Static Root of Trust for Measurement

The standard defines which part of the platform or firmware has to perform the measurement. Since the TPM itself is a purely passive element, executing instructions provided by the CPU, the BIOS/UEFI firmware has to initiate the measurement beginning with the binary representation of the firmware itself. This procedure is described in the TCG standard and the platform user has to *trust* the manufacturer for expected behavior. It is called the *Static Root of Trust for Measurement* (SRTM) and is defined in section 2.2 of the TCG PC Client Platform Firmware Profile[11]. As the mainboard manufacturer do not publish their firmware code, one may have to reverse engineer the firmware to prove correct implementation of the SRTM.

The SRTM is a small immutable piece of the firmware which is executed by default after the platform was reset. It is the first piece of software that is executed on the platform and measures itself into PCR 0. It must measure all platform initialization code like embedded drivers, host platform firmware, etc. as they are provided as part of the mainboard. If these measurements cannot be performed, the chain of trust is broken and consequently the platform cannot be trusted. When PCR 0 is zeroed or filled with the hashed representation of a string of zeroes, the SRTM did not act as expected. This indicates a broken chain of trust and should only appear when using the TPM simulator.

3.2.3 Platform Handover to OS

The BIOS or UEFI performs the next measurements according to table 3.1 until PCRs 1–7 are written accordingly. Before any further measurements are done, the control of the platform is handed over to the kernel of either a bootloader or the OS when booting without any bootloaders. In any case, these binaries are stored in the *Master Boot Record* (MBR) or provided as EFI blob in the EFI boot partition. It is noteworthy that the bootloader itself and its configuration payload is measured in PCR 4 and 5 before the handover is done. This guarantees that the chain of trust keeps intact when the bootloader/OS takes control.

The bootloader has to continue the chain of trust by measuring the kernel and the corresponding command line parameters into the next PCRs. The support and the way of how the measurements are done is not standardized. GRUB, for example, measures all

executed GRUB commands, the kernel command line and the module command line into PCR 8, whereas any file read by GRUB will be measured into PCR 9[9].

The whole process from initialization over measuring all software parts until the OS is started, is called *Trusted Boot*. The user can check the resulting values in the written PCR registers against known values. These values can either be precomputed or just the result of a previous boot. If all values match the expectations, the chain of trust exists between the SRTM and the kernel.

3.3 Integrity Measurement Architecture

The *Integrity Measurement Architecture* (IMA) is a Linux kernel extension to extend the chain of trust to the running application. IMA is officialy supported by RedHat and Ubuntu and there exists documentation to enable IMA on Gentoo as well. Other OS providers may not use a kernel with the required compile flags and/or do not provide userland software required to manage IMA. The IMA project page describes the required kernel features for full support in their documentation[17].

The process of keeping track of system integrity becomes far more complex on the OS level compared to the boot process. First, there are far more file system resources involved in running a system. Even a minimal setup of a common Linux Distribution like Ubuntu or RedHat will load several hundred files until the kernel has completed its boot process. Second, all these files will be loaded in parallel to make effective use of the available CPU resources. It is clear that parallelism introduces non-determinism to the order of executing processes and, of course, the corresponding system log files. Hence when using PCRs, this non-determinism results in different values, as stated in subsection 3.2.1. The system, however, might still be in a trustworthy state.

Finally, the user might know some additional data to the current value in the PCR register. Since the value itself does not tell anything to the user, a measurement log must be written for every operation on this PCR index.

IMA comes with three property variables which set the behaviour of the architecture:

- `ima_template` sets the format of the produced log.

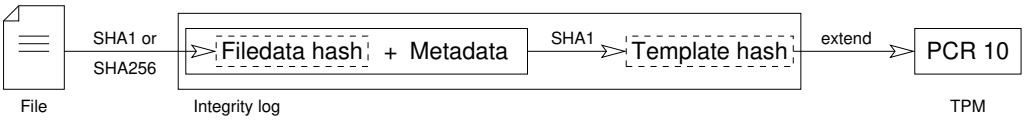


Figure 3.2: Overview of generating an entry in the integrity log

- `ima_appraise` changes the behaviour when a file is under investigation.
- `ima_policy` finally defines which resources should be analzyed.

These settings will be discussed in more detail in the following.

3.3.1 Integrity Log

IMA uses the *integrity log* to keep track of any changes of local filesystem resources. This is a virtual file that holds every measurement that leads to a change on the IMA PCR. When IMA is active on the system, the integrity log can be found in `/sys/kernel/security/ima/ascii_runtime_measurements`.

Before a file is accessed by the kernel, IMA creates an integrity log entry as it is shown in Figure 3.2. Depending on the settings for IMA, a SHA1 or SHA256 hash is created for the file content. The resulting *filedata hash* will be concatenated with the corresponding metadata. This concatenation will again be hashed into the so called *template hash*. Finally, the template hash is the single value of the whole computation that will be extended into the PCR. The integrity log holds at the end the filedata hash, the metadata and the template hash as well as the PCR index and the logfile format.

IMA knows three different file formats, where two of them can be used in recent applications. The only difference between these formats lies in the used and logged metadata:

- `ima-ng` uses, besides the filedata hash, also the filedata hash length, the pathname length and the pathname to create the template hash.
- `ima-sig` uses the same sources as ima-ng. When available, it also writes signatures of files into the log and includes them for calculating the template hash.

The older template `ima` uses only SHA1 and is fully replaceable with the `ima-ng` template. Therefore, it should not be used for newer applications.

ToDo!boot aggregate beschreiben

3.3.2 IMA Appraisal

IMA comes with four different runtime modes. These modes set the behaviour especially when there exists no additional information about the file in question.

- `off`: IMA is completely shut down. The integrity log just holds the entry of the boot aggregate.
- `log`: Integrity measurements are done for all relevant resources and the integrity log is filled accordingly.
- `fix`: In addition to writing the log file, the filedata hashes are also written as extended file attribute into the file system. This is required for the last mode to work.
- `enforce`: Only files with a valid hash value are allowed to be read. Accessing a static resource without a hash or an invalid hash will be blocked by the kernel.

3.3.3 IMA Policies

The IMA policies define which resources are targeted with IMA. There exist three template policies which can be used concurrently:

- `tcb`: All files owned by root will be measured.
- `appraise_tcb`: All executables which are run, all files mapped in memory for execution, all loaded kernel modules and all files opened for read by root will be measured by IMA.
- `secure_boot`: All loaded modules, firmwares, executed kernels and IMA policies are checked. Therefore, these resources need to have a provable signature to pass the check. The corresponding public key must be provided by the system manufacturer within the provided firmware or as Machine Owner Key in shim.

In addition to these templates, the system owner can define custom policies. Some example policies can be found in the Gentoo Wiki[10]. It is, for example, useful to exclude constantly changing log files from being measured to reduce useless entries in the measurement log.

3.3.4 IMA Extensions

Extended Verification Module (EVM)

3.4 Direct Anonymous Attestation

Direct Anonymous Attestation (DAA) is a cryptographic scheme which makes use of the functions provided by the TPM. DAA implements the concept of group signatures, where multiple secret keys can create a corresponding signature. These signatures can be verified with a single public key when private keys are member of the same group.

The scientific community is researching on TPM-backed DAA since the first standard of TPM went public in 2004. Since then many different approaches of DAA were discussed. According to Camenisch et al. in [4] and [3], almost all schemes were proven insecure, since many of them had bugs in the protocol or allowed trivial public/secret key pairs. This also includes the impementation of DAA im the TPM1.2 standard.

This section describes the concept by Camenisch et al. [4] including the cryptographic elements used for DAA. Unlike the description in the original paper, we describe the practical approach, which will be used in the following concept.

3.4.1 Mathematical Foundations

The following definitions form the mathematical building blocks for DAA. It is noteworthy that these definitions work with RSA encryption as well as with *Elliptic Curve Cryptography* (ECC).

Discrete Logarithm Problem

Given a cyclic group $G = \langle g \rangle$ of order n, the discrete logarithm of $y \in G$ to the base g is the smallest positive integer α satisfying $g^\alpha = y$ if this x exists. For sufficiently large n and properly chosen G and g, it is infeasible to compute the reverse $\alpha = \log_g y$. This problem is known as *Discrete Logarithm Problem* and is the basis for the following cryptographic algorithms.

Signature Proof of Knowledge (SPK)

A SPK is a signature of a message which proves that the creator of this signature is in possession of a certain secret. The secret itself is never revealed to any other party. Thus, this algorithm is a *Zero Knowledge Proof of Knowledge* (ZPK).

Camenisch and Stadler [6] introduced the algorithm based on the Schnorr Signature Scheme. It only assumes a collision resistant hash function $\mathcal{H} : \{0,1\}^* \to \{0,1\}^k$ for signature creation. For instance,

$$SPK\{(\alpha) : y = g^\alpha\}(m)$$

denotes a proof of knowledge of the secret α, which is embedded in the signature of message m. The one-way protocol consists of three procedures:

1. *Setup.* Let m be a message to be signed, α be a secret and $y := g^\alpha$ be the corresponding public representation.
2. *Sign.* Choose a random number r and create the signature tuple (c, s) as

   $$c := \mathcal{H}(m \,||\, y \,||\, g \,||\, g^r) \quad \text{and} \quad s := r - c\alpha \pmod{n}.$$

3. *Verify.* The verifier knows the values of y and g, as they are usually public. The message m comes with the signature values c and s. She computes the value

   $$c' := \mathcal{H}(m \,||\, y \,||\, g \,||\, g^s y^c) \quad \text{and verifies, that} \quad c' = c\,.$$

The verification holds since

$$g^s y^c = g^r g^{-c\alpha} g^{c\alpha} = g^r .$$

This scheme is extensible to prove knowledge of an arbitrary number of secrets as well as more complex relations between secret and public values.

Bilinear Maps

Bilinear Maps define a special property for mathematical groups which form the basis for verifying the signatures in DAA. Consider three mathematical groups $\mathbb{G}_1$, $\mathbb{G}_2$, with their corresponding base points g_1, g_2, and $\mathbb{G}_T$. Let $e : \mathbb{G}_1 \times \mathbb{G}_2 \rightarrow \mathbb{G}_T$ that satisfies three properties [4, 5]:

- *Bilinearity.* For all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$, for all $a, b \in \mathbb{Z} : e(P^a, Q^b) = e(P, Q)^{ab}$.
- *Non-degeneracy.* For all generators $g_1 \in \mathbb{G}_1, g_2 \in \mathbb{G}_2 : e(g_1, g_2)$ generates $\mathbb{G}_T$.
- *Efficiency.* There exists an efficient algorithm that outputs the bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ and an efficient algorithm for computing e.

Camenisch-Lysyanskaya Signature Scheme

The Camenisch-Lysyanskaya (CL) Signature Scheme [5] is based on the LRSW assumption and allows efficient proofs for signature posession and is the basis for the DAA scheme discussed below. It is based on a bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ that is available to all steps in the protocol.

- *Setup.* Choose $x \leftarrow \mathbb{Z}_q$ and $y \leftarrow \mathbb{Z}_q$ at random. Set the secret key $sk \leftarrow (x, y)$ and the public key $pk \leftarrow (g_2^x, g_2^y) = (X, Y)$.
- *Sign.* Given a message m and the secret key sk, choose a at random and output the signature $\sigma \leftarrow (a, a^y, a^{x+xym}) = (a, b, c)$.
- *Verify.* Given message m, signature σ and public key pk, verify that $a \neq 1_{\mathbb{G}_1}, e(a, Y) = e(b, g_2)$ and $e(a, X) \cdot e(b, X)^m = e(c, g_2)$.

Camenisch et al. stated in section 4.2 of their paper [4] that one has to verify the equation against $e(g_1, b)$ and $e(g_1, c)$ which is not correct.

3.4.2 DAA Protocol on LRSW Assumption

DAA is a group signature protocol, which aims with a supporting TPM to reveal no additional information about the signing host besides content and validity of the signed message m. According to Camenisch et al. [4], the DAA protocol consists of three parties:

- *Issuer* $\mathcal{I}$. The issuer maintains a group and has evidence of hosts that are members in this group.
- *Host* $\mathcal{H}$. The host creates a platform with the corresponding TPM $\mathcal{M}$. Membership of groups are maintained by the TPM. Only the key owner (TPM, passive) and the message author (Host, active) form a full group member.
- *Verifier* $\mathcal{V}$. A verifier can check, whether a host with its TPM is in a group or not. Besides the group membership, no additional information is provided.

A certificate authority $\mathcal{F}_{ca}$ is providing a certificate for the issuer itself. The basename `bsn` is some clear text string, whereas `nym` represent the encrypted basename bsn^{gsk}. $\mathcal{L}$ is the list of registered group members which is maintained by $\mathcal{I}$. The paper of Camenisch et al. [4] introduces further variables that are necessary for their proof of correctness. These extensions were omitted in the following to understand the protocol more easily.

- *Setup.* During setup, $\mathcal{I}$ is generating the issuer secret key isk and the corresponding issuer public key ipk. The public key is published and assumed to be known to everyone.

  1. On input SETUP, $\mathcal{I}$
     - generates $x, y \leftarrow \mathbb{Z}_q$ and sets $isk = (x, y)$ and $ipk \leftarrow (g_2^x, g_2^y) = (X, Y)$. Initialize $\mathcal{L} \leftarrow \emptyset$,
     - generates a proof $\pi \xleftarrow{\$} SPK\{(x, y) : X = g_2^x \wedge Y = g_2^y\}$ that the key pair is well formed,
     - registers the public key (X, Y, π) at $\mathcal{F}_{ca}$ and stores the secret key, and

  - outputs SETUPDONE.

- *Join.* When a platform, consisting of host $\mathcal{H}_j$ and TPM $\mathcal{M}_i$, wants to become a member of the issuer's group, it joins the group by authenticating to the issuer $\mathcal{I}$.

  1. On input JOIN, host $\mathcal{H}_j$ sends the message JOIN to $\mathcal{I}$.
  2. Upon receiving JOIN from $\mathcal{H}_j$, $\mathcal{I}$ chooses a fresh nonce $n \leftarrow \{0,1\}^\tau$ and sends it back to $\mathcal{H}_j$.
  3. Upon receiving n from $\mathcal{I}$, $\mathcal{H}_j$ forwards n to $\mathcal{M}_i$.
  4. $\mathcal{M}_i$ generates the secret key:
     - Check that no completed key record exists. Otherwise, it is already a member of that group.
     - Choose $gsk \xleftarrow{\$} \mathbb{Z}_q$ and store the key as $(gsk, \perp)$.
     - Set $Q \leftarrow g_1^{gsk}$ and compute $\pi_1 \xleftarrow{\$} SPK\{(gsk) : Q = g_1^{gsk}\}(n)$.
     - Return (Q, π_1) to $\mathcal{H}_j$.
  5. $\mathcal{H}_j$ forwards JOINPROCEED(Q, π_1) to $\mathcal{I}$.
  6. Upon input JOINPROCEED(Q, π_1), $\mathcal{I}$ creates the CL credential:
     - Verify that π_1 is correct.
     - Add $\mathcal{M}_i$ to $\mathcal{L}$.
     - Choose $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $a \leftarrow g_1^r$, $b \leftarrow a^y$, $c \leftarrow a^x \cdot Q^{rxy}$, $d \leftarrow Q^{ry}$.
     - Create the prove $\pi_2 \xleftarrow{\$} SPK\{(t) : b = g_1^t \wedge d = Q^t\}$.
     - Send APPEND(a, b, c, d, π_2) to $\mathcal{H}_j$
  7. Upon receiving APPEND(a, b, c, d, π_2), $\mathcal{H}_j$
     - verifies that $a \neq 1$, $e(a, Y) = e(b, g_2)$ and $e(c, g_2) = e(a \cdot d, X)$, and
     - forwards (b, d, π_2) to $\mathcal{M}_i$.

8. $\mathcal{M}_i$ receives (b, d, π_2) and verifies π_2. The join is completed after the record is extended to $(gsk, (b, d))$. $\mathcal{M}_i$ returns JOINED to $\mathcal{H}_j$.

9. $\mathcal{H}_j$ stores (a, b, c, d) and outputs JOINED.

- *Sign.* After joining the group, a host $\mathcal{H}_j$ and TPM $\mathcal{M}_i$ can sign a message m with respect to basename bsn.

  1. Upon input $\mathsf{SIGN}(m, \mathtt{bsn})$, $\mathcal{H}_j$ re-randomizes the CL credential:

     - Retrieve the join record (a, b, c, d) and choose $r \xleftarrow{\$} \mathbb{Z}_q$.
       Set $(a', b', c', d') \leftarrow (a^r, b^r, c^r, d^r)$.
     - Send $(m, \mathtt{bsn}, r)$ to $\mathcal{M}_i$ and store (a', b', c', d').

  2. Upon receiving $(m, \mathtt{bsn}, r)$, $\mathcal{M}_i$

     - checks, that a complete join record $(gsk, (b, d))$ exists, and
     - stores $(m, \mathtt{bsn}, r)$.

  3. $\mathcal{M}_i$ completes the signature after it gets permission to do so.

     - Retrieve group record $(gsk, (b, d))$ and message record $(m, \mathtt{bsn}, r)$.
     - Compute $b' \leftarrow b^r, d' \leftarrow d^r$.
     - If $\mathtt{bsn} = \perp$ set $\mathtt{nym} \leftarrow \perp$ and compute
       $\pi \xleftarrow{\$} SPK\{(gsk) : d' = b'^{gsk}\}(m, \mathtt{bsn})$.
     - If $\mathtt{bsn} \neq \perp$ set $\mathtt{nym} \leftarrow H_1(\mathtt{bsn})^{gsk}$ and compute
       $\pi \xleftarrow{\$} SPK\{(gsk) : \mathtt{nym} = H_1(\mathtt{bsn})^{gsk} \wedge d' = b'^{gsk}\}(m, \mathtt{bsn})$.
     - Send $(\pi, \mathtt{nym})$ to $\mathcal{H}_j$.

  4. $\mathcal{H}_j$ assembles the signature $\sigma \leftarrow (a', b', c', d', \pi, \mathtt{nym})$ and outputs $\mathsf{SIGNATURE}(\sigma)$.

- *Verify.* Given a signed message, everyone can check, whether the signature with respect to bsn is valid and the signer is member of this group. Furthermore, a revocation list RL holds the private keys of corrupted TPMs, whose signatures are no longer accepted.

  Upon input $\mathsf{VERIFY}(m, \mathtt{bsn}, \sigma)$, $\mathcal{V}$

  - parses $\sigma \leftarrow (a, b, c, d, \pi, \mathtt{nym})$,
  - verifies π with respect to $(m, \mathtt{bsn})$ and nym if $\mathtt{bsn} \neq \perp$,
  - checks that $a \neq 1$, $b \neq 1$ $e(a, Y) = e(b, g_2)$ and $e(c, g_2) = e(a \cdot d, X)$,
  - checks that for every $gsk_i \in \mathtt{RL} : b^{gsk_i} \neq d$,
  - sets $f \leftarrow 1$ if all test pass, otherwise $f \leftarrow 0$, and
  - outputs $\mathsf{VERIFIED}(f)$.

- *Link.* After proving validity of the signature, the verifier can test, whether two different messages with the same basename $\mathtt{bsn} \neq \perp$ are generated from the same TPM.

  On input $\mathsf{LINK}(\sigma, m, \sigma', m', bsn)$, $\mathcal{V}$ verifies the signatures and compares the pseudonyms contained in σ, σ':

  - Check that $\mathtt{bsn} \neq \perp$ and that both signatures σ, σ' are valid.
  - Parse the signatures $\sigma \leftarrow (a, b, c, d, \pi, \mathtt{nym})$, $\sigma' \leftarrow (a', b', c', d', \pi', \mathtt{nym}')$.
  - If $\mathtt{nym} = \mathtt{nym}'$, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
  - Output $\mathsf{LINK}(f)$.

Camenisch et al. [4] extend the general group concept scheme with their concept. The feature of linking messages together requires further security features within the DAA scheme, which the authors also prove in their paper along with the other properties of the scheme:

- *Non-frameability*: No one can create signatures that the platform never signed, but that link to messages signed from that platform.

- *Correctness of link*: Two signatures will link when the honest platform signs it with the same basename.
- *Symmetry of Link*: It does not matter in which order the linked signatures will be proven. The link algorithm will always output the same result.

4 Concept

In this chapter we define the constraints for the *Biometric Sensor* (BS) as well as a generic attempt for a prototype. The constraints include a discussion about the attack vectors to the BS. We explain which requirements can and will be addressed and how sensitive data is processed in the BS.

4.1 Definition of the Biometric Sensor

The BS itself is defined as edge device within the Digidow network. According to the schema shown in **??**, the BS will be placed in a public area (e.g. a checkpoint in an airport or as access control system at a building) to interact directly with the Digidow users. There, the BS acts as interface to the to the Digidow network. By providing a biometric property, the user should be able to authenticate itself and the network may then trigger the desired action, like granting access or logging presence. Depending on the biometric property, the sensor may not be active all the time, but activated when an authentication process is started.

The following enumeration shows the steps of the BS for identifying the interacting person.

1. *Listen*: Either the sensor hardware itself (e.g. a detection in a fingerprint sensor) or another electrical signal will start the authentication process.
2. *Collect*: Measure sensor data (picture, fingerprint) and calculate a biometric representation (attribute).
3. *Discover*: Start a network discovery in the Digidow network and find the PIA corresponding to the present person. It may be necessary to interact with more than one PIA within this and the next steps.

4. *Transmit*: Create a trusted and secure channel to the PIA and transmit the attribute.
5. *Reset*: Set the state of the system as it was before this transaction.

Since the BS handles biometric data—which must be held confidential outside the defined use cases—a number of potential threats must be considered when designing the BS.

4.2 Attack Vectors and Threat Model

As mentioned before, the BS will work in an exposed environment. Neither the user providing biometric data nor the network environment should be trusted for proper function. There should only be a connection to the Digidow network for transmitting the recorded data. This assumption of autonomy provides independence to the probably diverse target environments and use cases.

In addition to autonomy, the BS should also ensure proper handling of received and generated data. The recorded dataset from a sensor is *sensitive data* due to its ability to identify an individual. Due to its narrow definition, it is affordable to protect sensitive data. Besides that, *metadata* is information generated during the whole transaction phase. Timestamps and host information are metadata as well as connection lists, hashes and log entries and much more (What? Where? When?) There exists no exact definition or list of metadata which makes it hard to prevent any exposure of it. Metadata does not directly identify an individual. However huge network providers are able to combine lots of metadata to traces of individuals. Eventually an action of those traced individuals might unveil their identity. Consequently, a central goal of Digidow is to minimize the amount to minimize the risk of traces.

Privacy defines the ability of individuals to keep information about themselves private from others. In the context of the BS, this is related to the recorded biometric data. Furthermore, to prevent tracking, any interaction with a sensor should not be matched to personal information. Only the intended and trusted way of identification within the Digidow network should be possible.

4.2.1 Threat Model

To fulfill the sensor's use case, we need to consider the following attack vectors:

- *Rogue Hardware Components*: Modified components of the BS could, depending on their contribution to the system, collect data or create a gateway to the internal processes of the system. Although the produced hardware piece itself is fine, the firmware on it is acting in a malicious way. This threat addresses the manufacturing and installation of the system.
- *Hardware Modification*: Similar to rogue hardware components, the system could be modified in the target environment by attaching additional hardware. With this attack, adversaries may get direct access to memory or to data transferred from or to attached devices.
- *Metadata Extraction*: The actual sensor like camera or fingerprint sensor is usually attached via USB or similar cable connection. It is possible to log the protocol of those attached devices via Man-in-the-Middle attack on the USB cable.
- *Attribute Extraction*: The actual sensor, like camera or fingerprint sensor, is usually attached via USB or a similar cable connection. It is possible to log the protocol of those attached devices via wiretapping the USB cable. With that attack, an adversary is able to directly access the attributes to identify individuals.
- *Modification or aggregation of sensitive data within BS*: The program which prepares the sernsor data for transmission could modify the data before sealing it. The program can also just save the sensitive data for other purposes.
- *Metadata extraction on network*: During transmission of data from the sensor into the Digidow network, there will be some metadata generated. An adversary could use these datasets to generate tracking logs and eventually match these logs to individuals.
- *Replay of sensor data of a rogue BS*: When retransmitting sensor data, the authentication of an individual could again be proven. Any grants provided to the successfully identified individual could then be given to another person.

- *Rogue Biometric Sensor blocks transmission*: By blocking any transmission of sensor data, any transaction within the Digidow network could be blocked and therefore the whole authentication process is stopped.
- *Rogue Personal Identity Agent*: A rogue PIA might receive the sensor data instead of the honest one. Due to this error, a wrong identity and therefore false claims would be made out of that.

4.3 Prototype Concept

Given the threat model and the use cases described in section 4.1, we will introduce a prototype which will address many of the defined requirements. Any threats adressing the physical integrity of the BS will, however, be omitted. These threats can be addressed with physical intrusion and vandalism protection like they are available for ATMs. We will instead focus on the integrity of the system when the BS is operating.

4.3.1 Integrity and Trust up to the Kernel

We decided to use the PC platform as hardware base for the prototype. There are lots of different form factors available and you can extend the system with a broad variety of sensors. Furthermore, the platform provides full TPM support to enable cryptographic and integrity features. Finally, the platform can run almost all Linux variants and supports relevant pieces of software for this project. A flavour of Linux supporting all features described in this chapter, will be used as OS platform. The ARM platform seem to be capable of all these features as well. However, the support for TPMs, the amount of available software and the ease of installation is better on the PC platform.

As described in section 3.1, the TPM functions can be delivered in three different flavors: As dedicated or mounted device and as part of the platform firmware. The fTPM is part of larger proprietary environments from AMD and Intel which introduces, besides implementation flaws, additional attack surfaces for the TPM. Hence, we will use plugged TPM chips on the platform. Then we are able to deactivate the TPM for demonstration purposes by simply unplugging it.

Any recent PC platform supports TPMs and consequently trusted boot as mentioned in section 3.2. The system will describe its hardware state in the PCRs $0-7$ when the EFI/BIOS hands over to the bootloader. We use these PCR values to detect any unauthorized modifications on hardware or firmware level. It is important to also include *epmty* PCRs to detect added hardware on the PCI bus with an Option ROM, for example.

With these PCR values we can seal a passphrase in the TPM. The disk, secured with Full Disk Encryption (FDE), can only be accessed when the hardware underneath is not tampered with.

To further reduce the attack surface, the prototype will not use a bootloader like GRUB. Instead, the kernel is run directly from the UEFI/BIOS. Therefore, the kernel is packed directly into an EFI file, together with its command line parameters and the initial file system for booting. This *Unified Kernel* is directly measured by the UEFI/BIOS and is also capable of decrypting the disk, given the correct PCR values.

This setup starts with two sources of trust that are formally defined:

- *TPM*: The TPM acts as certified Root of Trust for holding the PCRs and for the cryptographic function modifying those.
- *RTM*: The Root of Trust for Measurement is part of the mainboard firmware. The tiny program just measures all parts of the firmware and feeds the TPM with the results. However, the program is maintained by the mainboard manufacturer and the source is not available to the public. We have to trust that this piece of software is working correctly.

We implicitly assume that the CPU, executing all these instructions and interacting with the TPM, is working correctly.

All parts contributing to the boot phase will be measured into one of the PCRs before any instruction is executed. Decrypting the disk can then be interpreted as authorization procedure against the encrypted disk. Consequently, only a *known* kernel with a *known* hardware and firmware setup underneath can access the disk and finish the boot process in the OS.

The disk encryption is, however, only an optional feature which can be omitted in a production environment when there is no sensitive data on the disk that must not be

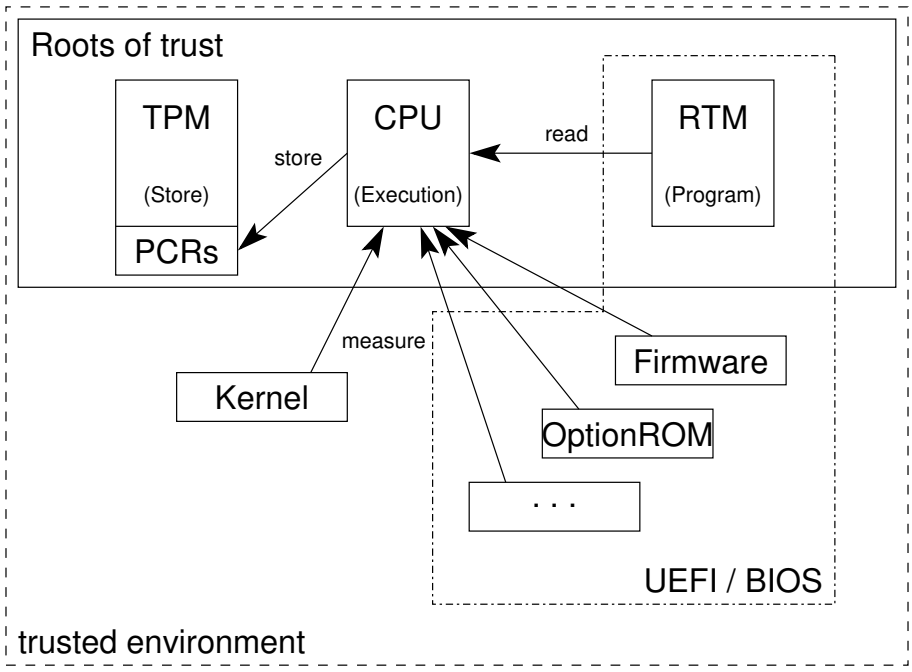


Figure 4.1: Extending trust from the Roots of Trust up to the kernel

revealed to the public. The system needs to check its integrity on the OS level and summarize that by publishing an attestation message, before any transaction data is used.

Figure 4.1 illustrates how above processes extend the trust on the system. The TPM is the cryptographic root of trust, storing all measurement results and the target values for validation. Since the RTM is the only piece of code which lives in the platform firmware and is executed *before* it is measured, it is an important part in the trust architecture of the system. An honest RTM will measure the binary representation of itself, which makes the code at least provable afterwards. Finally, the CPU is assumed to execute all the code according to its specification. Proving correctness of the instruction set cannot be done during the boot process.

When the roots of trust are honest, the trusted environment can be constructed during booting the platform with the PCR measurements. We get a trusted boot chain from firmware up to the kernel with its extensions and execution parameters as a result.

4.3.2 Integrity and Trust on OS Level

With the trusted kernel and IMA, we can include the file system into the trusted environment. According to section 3.3, every file will be hashed once IMA is activated and configured accordingly. By enforcing IMA, the kernel allows access to only those files having a valid hash. Consequently, every file which is required for proper execution needs to be hashed beforehand, i.e. before IMA is enforced. The IMA policy in place should be `appraise_tcb`, to analyze kernel modules, executable memory mapped files, executables and all files opened by root for read. This policy should also include drivers and kernel modules for external hardware like a camera for attached via USB.

4.3.3 Proving Trust with DAA

The features described above take care of building a trusted environment on the system level. DAA will take care of showing the *trust* to a third party which has no particular knowledge about the BS. In the Digidow context, the PIA should get, together to the biometrical measurements, a proof that the BS is a trusted system acting honestly.

To reduce the complexity of this problem, we consider two assumptions:

1. *Network Discovery*: The PIA is already identified over the Digidow network and there exists a bidirecional channel between BS and PIA
2. *Secure Communication Channel*: The bidirectional channel is assumed to be hardened against wire tapping, metadata extraction and tampering. The prototype will take no further action to encrypt any payload besides the cryptographic features that come along with DAA itself.

For the scope of this thesis, the DAA protocol should be applied on a simple LAN, where all parties are connected locally. The BS will eventually become a member of the group of sensors managed by the issuer. During signup, issuer and BS (member) negotiate the membership credentials over the network. By being a member of the DAA group, the issuer fully trusts that the BS is honest and acting according the specification. The issuer will not check any group members, since they can now act independently of the issuer.

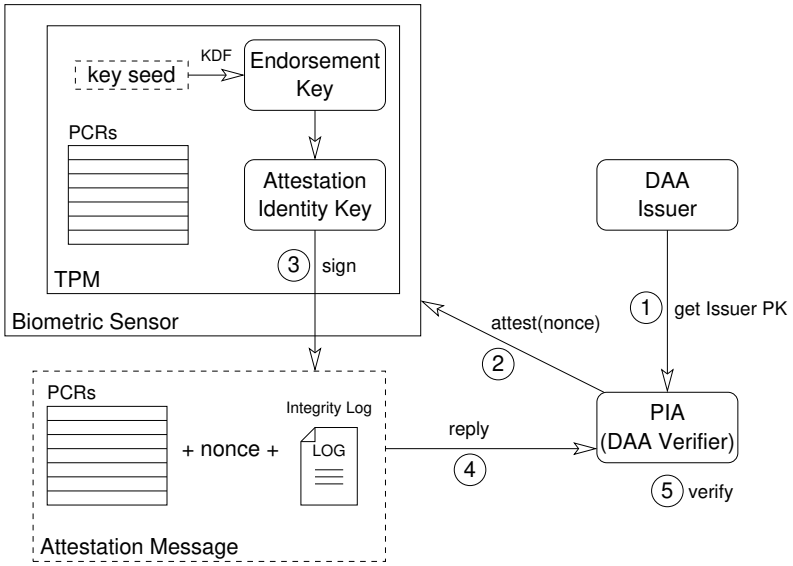


Figure 4.2: The DAA attestation process requires 5 steps. The PIA may trust the BS afterwards.

When the BS is then authenticating an individual, the process illustrated in Figure 4.2 will be executed.

1. The PIA gets the public key of the BS group once and independently of any transaction.
2. During the transaction, the PIA will eventually ask the BS for attestation together with a nonce.
3. The BS will collect the PCR values, the integrity log and the nonce into an Attestation message signed with the member's secret key (SK).
4. The attestation message will be sent back to the PIA.
5. The PIA checks the signature of the message, checks the entries of the integrity log against known values, and proves the PCR values accordingly.

Figure 4.3 shows how the sources of trust will be represented in the final attestation message. The four sources of trust are defined as groups which deliver parts of the prototype, but cannot be verified on a cryptographic level. Hence, suppliers must be manually added to these groups by using a well defined check for trustworthiness. Any TPM manufacturer has to implement the well defined standard from TCG. There exists,

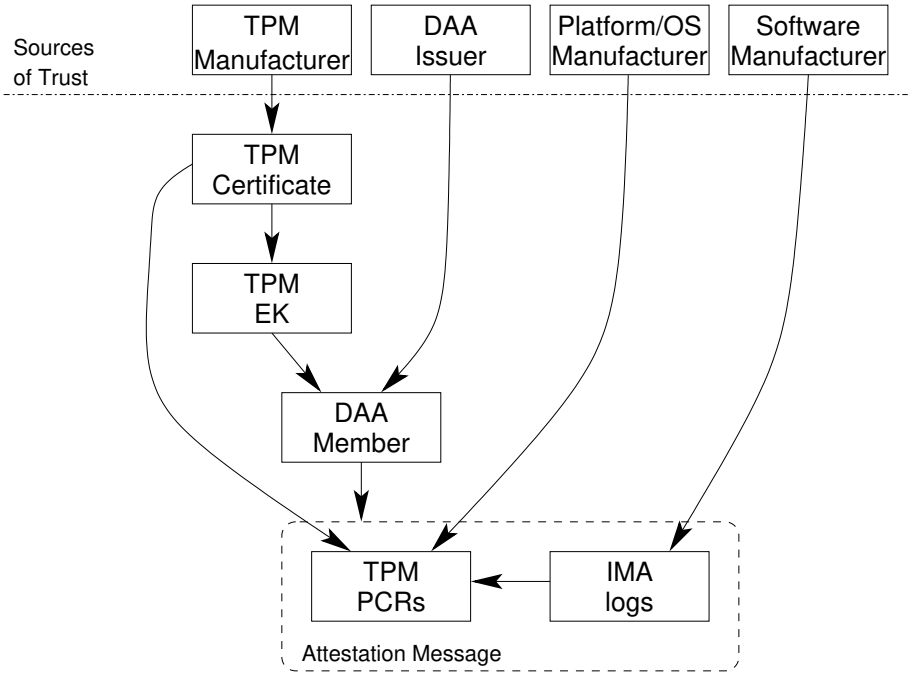


Figure 4.3: Overview of the Chain of Trust of the BS

however, no such exact definition for hardware and firmware parts of the platform. Consequently, these parts should undergo a functional analysis before they are trusted. Trust means that, when the platform is defined trustworthy, the corresponding PCR values should be published.

The same procedure should be done for the kernel and the used OS environment and of course, the used software. There, only the kernel with its parameters have a corresponding PCR value. Furthermore, a hash value should be published for any relevant file on the file system.

We can then build a cryptographic representation of the chain of trust in Figure 4.3. The TPM has a signed certificate from its manufacturer, where it derives the endorsement key (EK) from. When all of the above checks against platform, OS and TPM are good, the DAA issuer will assign the platform to the group of trusted BS. The BS has now a member SK for signing its attestation message.

The verifier can now check the valid membership by checking the signature of the message against the issuer's public key (PK). Furthermore, it can check the state of the platform by

comparing the PCR values against known values. Finally, it can check the integrity of the running software by checking the hashes in the IMA log against known values. PCR 10 represents the end of the hash chain fed by the IMA log entries.

If all values are good, the BS can be trusted and the Digidow transaction can be continued at the PIA.

5 Implementation

The concept decscribed in chapter 4 will be implemented as a prototype to demonstrate a working implementation and to analyze the speed of those parts of a transaction. Although the goal is to put all these features on a highly integrated system, we decided to start with widely available hardware based on Intel's x86 architecture.

Figure 5.1 shows the setup on a connection level. To show the features of DAA, it is necessary to have three independent systems which are connected via a TCP/IP network. Every host is connected via ethernet to the other systems. To keep the setup minimal, the IP addresses are static and internet is only required during installation. Hence, Service Discovery is done statically, every host knows the IP addresses and functions of each other directly.

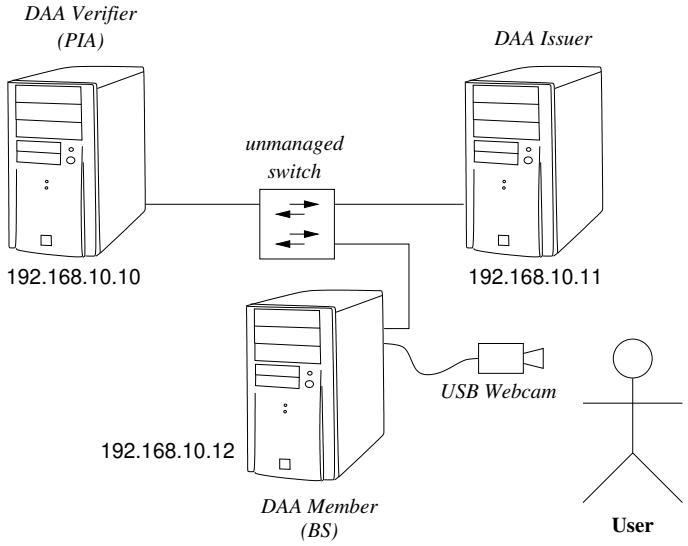


Figure 5.1: Prototype setup to show DAA features and the Dataflow from BS to PIA

5.1 Hardware Setup

For demonstrating remote attestation via DAA over a simple network infrastructure, we use three systems with similar configuration. Table 5.1 shows the specification of these systems. We decided to order one system with an AMD processor in it to find differences in handling the TPM between Intel and AMD systems. All features used in this thesis were available on both platform types, so there were no differences found.

Table 5.1: Systems used for demonstration prototype

	System 1	*System 2*	*System 3*
Processor	AMD Athlon 240GE	Intel Pentium G4560T	Intel Pentium G4560T
Mainboard	Gigabyte B450I Aorus Pro Wifi	Gigybyte GA H110N	Gigabyte GA H310N
Memory	8GB DDR4	8GB DDR4	8GB DDR4
Storage	NVMe SSD 128GB	NVMe SSD 128GB	NVMe SSD 128GB
TPM	Gigabyte TPM2.0_L	Gigabyte TPM2.0_L	Gigabyte TPM2.0_L

The used mainboards come with a dedicated TPM2.0 header which may differ from board to board. A 19-pin header is available on the older platform of *System 2*. As long as TPM

and mainboard have the same 19-pin connector they will be compatible to each other. The newer Gigabyte mainboards come with a proprietary 11-pin connector which is only compatible with Gigabyte's TPM2.0_S module. All other modules are however electrical compatible since only unused pins of the full size connector are removed. With a wiring adapter any TPM board would work on any mainboard supporting TPM2.0 even when coming with a prorietary header.

5.2 Operating System

The OS needs to fulfill three requirements for this prototype. First, the TPM must be supported by the kernel. Second, the OS has to support a recent version of the TPM software stack (TSS 3.0.x or newer at the point of writing) for using the Xaptum ECDAA[12] project with enabled hardware TPM. Similarly, the `tpm2-tools` must be available in a version newer than `4.0.0`. Finally, the support for the Integrity Measurement Architecture (IMA) must be activated in the kernel and supported by the OS. This feature is available in the mainline Linux kernel. However, the corresponding kernel compile parameters must be set.

Ubuntu 20.04 LTS does fulfill above mentioned requirements by default. Ubuntu is also supported by the Xaptum ECDAA project, although it was tested with an older version (18.04). When installing Ubuntu on the prototype, we used *Full Disk Encryption* (FDE) which leads to the disk allocation described in Table 5.2. Ubuntu installs Grub by default, and it is used as a fallback bootloader.

Partition	*Size*	*Mountpoint*	*Comment*
`nvme0n1p1`	512M	`/boot/efi`	EFI boot partition
`nvme0n1p2`	1G	`/boot`	Bootloader partition (Grub)
`nvme0n1p3`	118G		lvm on dm_crypt
`ubuntu-vg-ubuntu-lv`	118G	`/`	root partition on lvm

Table 5.2: Disk layout of the BS prototype

5.3 Trusted Boot

By default, every mainboard with support for TPM2.0 must support trusted boot. When a TPM becomes available, the UEFI/BIOS itself takes all required measures until the boot process is handed over to the OS bootloader (e.g. GRUB). Since Ubuntu uses GRUB 2.04 as bootloader which has TPM support by default, trusted boot just to be enabled in the GRUB configuration. In this case, GRUB will be measured from the BIOS to the PCRs 4 and 5, as shown in Table 3.1. Grub itself uses PCR 8 for executed commands, the kernel command line and all commands forwarded to kernel modules. PCR 9 is used to measure all files read by GRUB[9].

Embedded systems like a productive version of the BS do not need several boot options. Therefore we replace the bootloader EFI file with a blob containing all required information to load the kernel directly. This kernel decrypts the disk and boots the remaining system. Pawit Pornkitprasam [15], [16] and Karl O from Tevora [19] introduced the concept of a *Unified Kernel* for Ubuntu and Arch respectively.

This large EFI file contains the initramfs, kernel command line and the kernel itself. Table 5.3 shows the content of the EFI blob with the corresponding offset addresses as well as the sources in the file system.

All binary resources are available as blobs which can be imported directly. Only the command line parameters need to be defined manually. Listing 5.1 shows the used command line which will be saved on `/boot/kernel-command-line.txt`. The parameters activate also IMA which will be discussed later in this chapter.

If FDE is installed, the boot process need to be aware of how to decrypt the disk. Therefore, the initramfs needs the luks binaries as well as the TPM software stack to unseal

Address	*Source path*	*Comment*
`0x0000000`	`/usr/lib/systemd/boot/efi/linuxx64.efi.stub`	Linux EFI Stub
`0x0020000`	`/usr/lib/os-release`	Linux OS release information
`0x0030000`	`/boot/kernel-command-line.txt`	Kernel command line parameters
`0x0040000`	`/boot/vmlinuz`	latest kernel image
`0x3000000`	`/boot/initrd`	latest initial ramdisk

Table 5.3: Memory layout of the Unified Kernel EFI file

Listing 5.1: `kernel-command-line.txt`: Command line for the Kernel

```
1 /vmlinuz-5.4.0-42-generic ima_appraise=fix ima_policy=appraise_tcb ima_policy=tcb
      ima_hash=sha256 root=/dev/mapper/ubuntu--vg-ubuntu--lv ro rootflags=i_version
```

Listing 5.2: `passphrase-from-tpm.sh`: Initramfs-script to ask the TPM for the LUKS key

```
1 #!/bin/sh
2 echo "Unlocking␣via␣TPM" >&2
3 export TPM2TOOLS_TCTI="device:/dev/tpm0"
4 /usr/bin/tpm2_unseal -c 0x81000000 -p pcr:sha256:0,1,2,3,4,5,6,7
5 if [ $? -eq 0 ]; then
6   exit
7 fi
8 /lib/cryptsetup/askpass "Unlocking␣the␣disk␣fallback␣$CRYPTTAB_SOURCE␣(
      $CRYPTTAB_NAME)\nEnter␣passphrase:␣"
```

the passhrase with the PCR registers. The unseal operation itself is then done with Listing 5.2, which also needs to exist in the initramfs. We copy the script of Listing 5.3 to `/etc/initramfs-tools/hooks` to enable TPM access during boot after the next initramfs update. Next, a new key for FDE is created by using the random number generator of the TPM. It is saved in clear text in `/root/keys` to be able to update the sealing operation when new PCR values are used. The passphrase needs to be available, when an update resulted in new PCR values. In this case, the passphrase in the TPM would not be accessible anymore. Listing 5.5 shows the script for creating the LUKS passphrase. When the new phrase is added to LUKS, the user is asked for an existing FDE password. We keep the first password as backup when decryption via TPM fails. Finally, Listing 5.4 shows how the unified kernel is created with the command `objcopy` and copied on the EFI disk partition. The offset addresses need to be chosen according to the size of the included blobs. All steps described above are summarized in Listing 5.6.

Listing 5.3: `tpm2-hook.sh`: Script copying required TSS files into the initramfs

```
1 #!/bin/sh -e
2 if [ "$1" = "prereqs" ]; then exit 0; fi
3 . /usr/share/initramfs-tools/hook-functions
4 copy_exec /usr/bin/tpm2_unseal
5 copy_exec /usr/lib/x86_64-linux-gnu/libtss2-tcti-device.so.0
6 copy_exec /usr/sbin/passphrase-from-tpm.sh
```

Listing 5.4: `update-kernel.sh`: Script for updating the unified Kernel

```
#!/usr/bin/bash
set -e
PARTITION_ROOT=/dev/mapper/ubuntu--vg-ubuntu--lv
DISK=/dev/nvme0n1

mkdir -p /boot/efi/EFI/Linux
update-initramfs -u -k all
LATEST=`ls -t /boot/vmlinuz* | head -1`
VERSION=`file -bL $LATEST | grep -o 'version␣[^␣]*' | cut -d '␣' -f 2`
### echo "/vmlinuz-$VERSION root=/dev/mapper/vg-root rw loglevel=3 cryptdevice=
    PARTUUID=$(blkid -o value $PARTITION_ROOT | tail -n 1):lvm:allow-discards rd.luks.
    options=discard" > /boot/kernel-command-line.txt #Arch command line
# echo "/vmlinuz-$VERSION root=$PARTITION_ROOT ro ima_appraise=fix ima_policy=tcb
    ima_policy=appraise_tcb rootflags=i_version" > /boot/kernel-command-line.txt #
    Ubuntu command line
objcopy \
  --add-section .osrel="/usr/lib/os-release" --change-section-vma .osrel=0x20000 \
  --add-section .cmdline="/boot/kernel-command-line.txt" --change-section-vma .cmdline
      =0x30000 \
  --add-section .linux="/boot/vmlinuz-$VERSION" --change-section-vma .linux=0x40000 \
  --add-section .initrd="/boot/initrd.img-$VERSION" --change-section-vma .initrd=0
      x3000000 \
  "/usr/lib/systemd/boot/efi/linuxx64.efi.stub" "/boot/efi/EFI/Linux/Linux.efi"
```

Listing 5.5: `create-luks-tpm.sh`: Script to create a new LUKS key

```
#!/bin/bash
set -e

CRYPTFS=/dev/nvme0n1p3

echo "creating␣secret␣key"
mkdir -p /root/keys
tpm2_getrandom 32 -o /root/keys/fde-secret.bin
chmod 600 /root/keys/fde-secret.bin
cryptsetup luksAddKey $CRYPTFS /root/keys/fde-secret.bin

# /usr/sbin/update-luks-tpm.sh #not reqired before reboot
```

Listing 5.6: `install.sh`: Script to install Trusted Boot on Ubuntu

```
#!/bin/bash
set -e

cp -vf ./passphrase-from-tpm.sh /usr/sbin/ || exit 1
cp -vf ./update-luks-tpm.sh /usr/sbin || exit 1
cp -vf ./update-kernel.sh /usr/sbin || exit 1
cp -vf ./create-luks-tpm.sh /usr/sbin || exit 1

cp -vf ./tpm2-hook.sh /etc/initramfs-tools/hooks/ || exit 2
awk -i inplace '/luks/{print␣$0␣",discard,initramfs,keyscript=/usr/sbin/passphrase-
    from-tpm.sh"}' /etc/crypttab

cp -vf ./kernel-command-line.txt /boot/ || exit 3
/usr/sbin/create-luks-tpm.sh
/usr/sbin/update-kernel.sh
efibootmgr --create --disk /dev/nvme0n1 --part 1 --label "ubuntu␣unified" --loader "\
    EFI\Linux\Linux.efi" --verbose
echo "Installed␣successfully!␣Please␣reboot␣and␣execute␣update-luks-tpm.sh␣
    afterwards"
```

When the unified kernel is installed, the system needs to be rebooted to generate the PCR values for the new boot chain. The FDE decryption with the TPM will of course fail, since there is no sealed passphrase available yet. This step is done now since the new unified kernel is measured the first time. Listing 1 shows how the passphrase is sealed into the TPM with all relevant PCR registers. The result is a trusted boot chain which ensures, that the system only has access to the encrypted disk when the kernel with its parameter is known—and therefore trusted.

- update initramfs with tpmtools
- hook script takes PCRs to unseal key, optional use secondary key for manual decryption (admin access)
- compile unified kernel
- optional update kernel and tpm values after system upgrade – one reboot required for generating the PCR values; impossible to precompute.
- Trusted Boot with GRUB 2.04: TPM support available; PCR mapping
- Secure Boot with unified kernel; another PCR mapping

Listing 5.7: `update-luks-tpm.sh`: Script for updating the Sealing of the TPM Object with new PCR values

```
#!/usr/bin/bash
echo "Updating TPM Policy with current available PCRs"

set +e
tpm2_evictcontrol -c 0x81000000

set -e
tpm2_flushcontext -t
tpm2_createprimary -C e -g sha256 -G ecc256 -c /root/keys/e-primary.context
tpm2_createpolicy --policy-pcr -l sha256:0,1,2,3,4,5,6,7 -L /root/keys/pcr-policy.
    digest
tpm2_create -g sha256 -u /root/keys/obj.pub -r /root/keys/obj.priv -C /root/keys/e-
    primary.context -L /root/keys/pcr-policy.digest -a "noda|adminwithpolicy|
    fixedparent|fixedtpm" -i /root/keys/fde-secret.bin
tpm2_flushcontext -t
tpm2_load -C /root/keys/e-primary.context -u /root/keys/obj.pub -r /root/keys/obj.
    priv -c /root/keys/load.context
tpm2_evictcontrol -C o -c /root/keys/load.context 0x81000000
# tpm2_unseal -c 0x81000000 -p pcr:sha1:0,1,2,3,4,5,6,7 -o /root/test.bin #proof that
     the persistence worked
rm -f /root/keys/load.context /root/keys/obj.priv /root/keys/obj.pub /root/keys/pcr-
    policy.digest
tpm2_flushcontext -t
```

- Benefits and Drawbacks of both variants
- describe automated unlocking

Limitations due to bad implementation on BIOS-Level, no Certificate Verification Infrastructure available for TPMs? Needs to be proven for correctness.

5.4 Integrity Measurement Architecture

Available on Ubuntu, RedHat and optionally Gentoo. The kernel has the correct compile options set.

5.4.1 Handling external hardware

4 How can camera and fingerprint sensor be trusted? What is the limitation of this solution?

5.5 Interaction with TPM2

tpm2-tools 4.x are usable to interact with the TPM from the command line. Available on all major releases after summer 2019. Fallback is using the TPM2 ESAPI or SAPI, which is available on almost all Linux distributions.

5.6 Direct Anonymous Attestation

DAA Project from Xaptum: Working DAA handshake and possible TPM integration. Requires an Attestation Key which is secured with a password policy.

6 Conclusion and Outlook

6.1 Testing

These are the test results

6.2 Limitations

Still hard to set up a system like that. Documentation is available, but hardly any implementations for DAA and IMA.

6.3 Future Work

6.4 Outlook

Hardening of the system beyond IMA useful. Minimization also useful, because the logging gets shorter.

Bibliography

[1] FIDO Alliance. *FIDO ECDAA Algorithm Implementation Draft*. 2018. URL: https://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.html (visited on 07/07/2021) (cit. on p. 8).

[2] Will Arthur, David Challener, and Kenneth Goldman. *A Practical Guide to TPM 2.0*. Jan. 2015. DOI: 10.1007/978-1-4302-6584-9 (cit. on p. 13).

[3] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. "One TPM to Bind Them All: Fixing TPM 2.0 for Provably Secure Anonymous Attestation". In: May 2017, pp. 901–920. DOI: 10.1109/SP.2017.22 (cit. on p. 20).

[4] Jan Camenisch, Manu Drijvers, and Anja Lehmann. "Universally Composable Direct Anonymous Attestation". In: *Public-Key Cryptography – PKC 2016*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2016, pp. 234–264. ISBN: 978-3-662-49386-1. DOI: 10.1007/978-3-662-49387-8_10 (cit. on pp. 20, 22, 23, 26).

[5] Jan Camenisch and Anna Lysyanskaya. "Signature Schemes and Anonymous Credentials from Bilinear Maps". In: vol. 3152/2004. Aug. 2004, pp. 56–72. DOI: 10.1007/978-3-540-28628-8_4 (cit. on p. 22).

[6] Jan Camenisch and Markus Stadler. "Efficient Group Signature Schemes for Large Groups". In: *CRYPTO '97* 1296 (Jan. 1997) (cit. on p. 21).

[7] TPM2 Software Community. *TPM2 Tools*. 2020. URL: https://github.com/tpm2-software/tpm2-tools (visited on 05/15/2020) (cit. on p. 12).

[8] MITRE Corporation. *Search Results for "tpm" in the CVE Database*. 2021. URL: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=tpm (visited on 05/15/2021) (cit. on p. 11).

[9] Free Software Foundation. *GRUB 2.04 User Manual: Measuring Boot Components*. 2019. URL: https://www.gnu.org/software/grub/manual/grub/html_node/Measured-Boot.html (visited on 03/29/2021) (cit. on pp. 17, 41).

[10] Inc Gentoo Foundation. *Integrity Measurement Architecture/Recipes*. 2019. URL: https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture/Recipes (visited on 07/07/2021) (cit. on p. 20).

[11] Trusted Computing Group. *TCG PC Client Platform Firmware Profile Specification Revision 1.04*. 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_PCClientSpecPlat_TPM_2p0_1p04_pub.pdf (visited on 08/01/2020) (cit. on pp. 15, 16).

[12] Xaptum Inc. *Source repository for the ECDAA C Library*. 2021. URL: https://github.com/xaptum/ecdaa (visited on 07/07/2021) (cit. on p. 40).

[13] Daniel Moghimi, Berk Sunar, Thomas Eisenbarth, and Nadia Heninger. "TPM-FAIL: TPM meets Timing and Lattice Attacks". In: *29th USENIX Security Symposium (USENIX Security 20)*. Boston, MA: USENIX Association, Aug. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi (cit. on p. 11).

[14] Matus Nemec, Marek Sys, Petr Svenda, Dusan Klinec, and Vashek Matyas. "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 1631–1648. ISBN: 9781450349468. DOI: 10.1145/3133956.3133969. URL: https://doi.org/10.1145/3133956.3133969 (cit. on p. 11).

[15] Pawit Pornkitprasan. *Full Disk Encryption on Arch Linux backed by TPM 2.0*. July 2019. URL: https://medium.com/@pawitp/full-disk-encryption-on-arch-linux-backed-by-tpm-2-0-c0892cab9704 (visited on 02/27/2020) (cit. on p. 41).

[16] Pawit Pornkitprasan. *Its certainly annoying that TPM2-Tools like to change their command line parameters*. Oct. 2019. URL: https://medium.com/@pawitp/its-certainly-annoying-that-tpm2-tools-like-to-change-their-command-line-parameters-d5d0f4351206 (visited on 02/27/2020) (cit. on pp. 12, 41).

[17] David Safford, Dmitry Kasatkin, and Mimi Zohar. *Integrity Measurement Architecture (IMA) Wiki Page*. 2020. URL: https://sourceforge.net/p/linux-ima/wiki/Home/ (visited on 03/20/2021) (cit. on p. 17).

[18] Nabil Schear, Patrick T. Cable, Thomas M. Moyer, Bryan Richard, and Robert Rudd. "Bootstrapping and Maintaining Trust in the Cloud". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACSAC '16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 65–77. ISBN: 9781450347716. DOI: 10.1145/2991079.2991104. URL: https://doi.org/10.1145/2991079.2991104 (cit. on p. 8).

[19] Tevora. *Configuring Secure Boot + TPM 2*. June 2019. URL: https://threat.tevora.com/secure-boot-tpm-2/ (visited on 06/19/2020) (cit. on p. 41).

[20] *The TPM Library Specification*. 2019. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (visited on 05/16/2020) (cit. on p. 10).