JOHANNES KEPLER
UNIVERSITY LINZ

Author
Michael Preisach BSc.
1155264

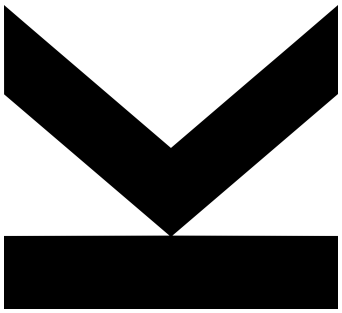
Submission
Institute for Networks and Security

Supervisor
Univ.-Prof. Priv.-Doz. Dr.
René Mayrhofer

Assistant Thesis Supervisor
Dr. **Michael Roland**

November 2021

System Integrity and Attestation for Biometric Sensors



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

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, November 2021

Abstract

The *Digital Shadow* project, developed at the Institute for Networks and Security, requires verifiable trust in many areas in order to recognize and authorize users based on their biometric data. This trust should give the user the opportunity to check the correctness of the system quickly and easily before he or she provides the system with biometric data. This master's thesis deals with the existing tools that can create such trust. The implemented system combines these tools in order to identify users in the Digital Shadow network with their biometric data. Incorrect use of this sensitive data should be excluded and the smallest possible set of metadata should be generated. Based on the implemented system, we discuss the properties of a trustworthy environment for software and explain the necessary framework requirements.

Kurzfassung

Das am Institut für Netzwerke und Sicherheit entwickelte Projekt *Digital Shadow* benötigt in vielen Bereichen ein prüfbares Vertrauen um Nutzer 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 mit den existierenden Werkzeugen, die ein solches Vertrauen schaffen können. Diese Werkzeuge werden kombiniert, um Nutzer im Digital Shadow Netzwerk mittels biometrischen Daten identifizieren zu können. Es soll dabei eine fälschliche Verwendung dieser sensiblen Daten ausgeschlossen sein und ein möglichst kleines Set von Metadaten erzeugt werden. Anhand des implementierten Systems werden die Eigenschaften einer vertrauenswürdigen Umgebung für Software diskutiert und notwendige Rahmenbedingungen erläutert.

Contents

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

1.1 Motivation

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 services is endless and will still grow in the future.

The downside of all these digital services is that using these services generates 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.

1.2 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 by 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.

1.3 Project Digidow

The Institute of Networks and Security is heavily using the cryptographic form of trust in the project *Digital Shadow* (Digidow). Digidow introduces an electronic authentication

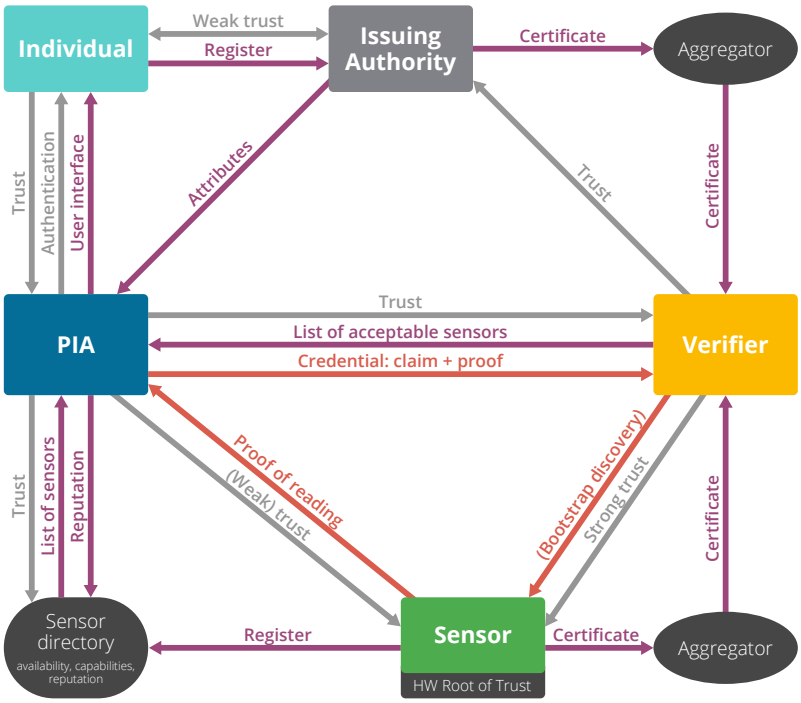


Figure 1.1: Overview of the Digidow network with its interactions [21]

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.

Mayrhofer et al. [21] provide an overview of the Digidow network as shown in Figure 1.1. The nodes using the Digidow network and their interactions are not fully specified at the time of this writing. Therefore the processes may be adapted when necessary. DigiDow introduces five main parties which are involved in a common authentication process.

- *Individual*: The human user who wants to be identified via the Digidow network.
- *Personal Identity Agent* (PIA): The PIA is the digital shadow of an individual. This individual is also the owner of the PIA and should be in control of sensitive data and software on it.
- *Issuing Authority*: This party acts as an authority for the individual's attributes. These attributes show an aspect of the individual's identity. After identifying the

individual via the Digidow network, these attributes may be used to allow or deny a certain action.

- *Verifier*: This is the party that verifies the whole authentication process and may finally trigger the desired action. It is usually strongly connected with the sensor which starts the identification process.
- *Sensor*: For authentication, an individual has to be uniquely identified. Therefore, the sensor records biometric data from the individual and passes it to the PIA via the Digidow network. Sensors are not limited to sensing biometric data. However, we focus in this thesis on developing a prototype of a biometric sensor (BS).

When an individual wants to be identified by Digidow, they will eventually step in front of a sensor. This defines the beginning of a Digidow transaction. The procedure is as follows:

1. The sensor will start recording a unique digital representation, triggered by the directly connected verifier or by hardware detection.
2. The sensor eventually finds one or a small group of eligible PIAs, where a secure communication channel is established.
3. After receiving the digital representation of the sensor, the PIA identifies the individual if possible.
4. When identification was possible, the PIA eventually sends a proof of identification and a claim of the requested attributes to the verifier.
5. The verifier will check the cryptographic proofs of the claim and the sensor data. If successful, the verifiifier will grant the desired action for the asking individual.

The above illustration is an early draft of the whole setup and is under constant development. Latest developments of the whole system will be published on the Digidow project page[1].

[1] https://digidow.eu

1.4 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.
- *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 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 verifier exceed the contribution of 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.
- We assume that any built system is secure against any hardware level threats. Attacks targeting the system without changing any running software on the system may remain undetected. For example, USB wire tapping or debug interfaces within the system may reveal sensitive information.

1.5 Organization

In the next chapter, we will indroduce and discuss existing contributions in the targeted scientific area. This includes especially the theoretical foundations of the network protocol which is part of our contribution. Together with that, we will introduce our theoretical solution for the previously stated problems in Chapter 3. We introduce then in Chapter 4 a working implementation with all necessary parts for provisioning the environment and the used hosts accordingly. Finally we will present the results and limitations in Chapter 6 and give an outlook on future work.

2 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).

2.1 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 [16].

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 TPM 2.0 are able to manage a TPM within the BIOS, even as mounted device.
- *Firmware TPM (fTPM).* This variant was introduced with the TPM 2.0 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 [11]. 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. [24], 1024 bit keys required in the worst case scenario 3 CPU months and 2048 bit keys needed 100 CPU years when using one core of an Intel Xeon E5-2650 v3 CPU. 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" [23] 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 also show 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.

2.1.1 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 [9] 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 [26].

2.1.2 The Hardware

With the previously mentioned software layers the TCG achieved independence of the underlying hardware. Hence, this design made the different flavors of TPMs possible.

With the TPM 2.0 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 for the TPM 2.0 standard, only few vulnerabilities were found in hardware implementations, as we showed in Section 2.1. This is caused

by a firm hardware definition and by using widely used and well proven cryptographic algorithms. Only the group signature scheme is the relatively new implementation of Camenisch et al. [4], [3]. They discuss a variety of concepts with their provable flaws [5] and show thereafter an own implementation. The prove of their concept is still valid at the time of this writing. Thus, we assume a hardware TPM 2.0 to be a reliable source for cryptographic algorithms.

2.1.3 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. 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 represents its own tree of keys, beginning with a root key. Since TPM 2.0 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.

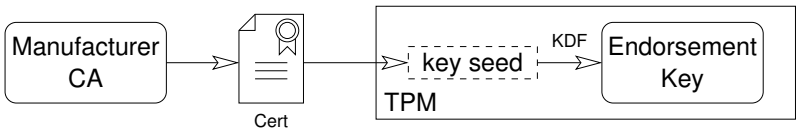


Figure 2.1: The manufacturer certifies every TPM it produces

2.1.4 Endorsement Key

The *Endorsement Key* (EK) is the root key for the corresponding hierarchy. Figure 2.1 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. The EK can then be generated with a *key derivation function* (KDF) which allows to add additional entropy to that of the TPM. According to chapter 15 of Arthur et al. [2], this additional entropy is not used and the parameter field is filled with zeroes. As a result, although it is possible to generate an arbitrary number of EKs, the TPM generates always the same by default. Only this default EK comes with a corresponding certificate which is signed by the TPM manufacturer by using its own root *Certificate Authority* (CA). Since the platform supports generating and using multiple root keys at a time, it is also possible to encrypt temporarily unused keys and store them 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.

2.2 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.

2.2.1 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:

$$\text{new\_PCR} = \mathit{hash}(\text{old\_PCR} \,||\, \text{data}). \tag{2.1}$$

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 TPM 1.2. For TPM 2.0, the process was only extended to support 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* [15] and shown in table 2.1. 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.

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

Table 2.1: 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

platform. Later, when UEFI became popular, the PCR descriptions got adopted for the new platform.

2.2.2 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 [15]. As the mainboard manufacturers 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.

2.2.3 Platform Handover to OS

The BIOS or UEFI performs the next measurements according to table 2.1 until PCRs 0–7 are written accordingly as shown in section 2.2.3 of the TCG PC Client Platform Firmware Profile [15]. 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 how the measurements are done in PCR 8 and higher is not defined by TCG. They only reserver PCR 8–15 for the OS. GRUB, for example, measures all executed GRUB commands, the kernel command line and the module command line by default into PCR 8, whereas any file read by GRUB will be measured into PCR 9 [12].

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.

2.2.4 Secure Boot

Secure boot is another technology to prevent malware from being executed before the OS kernel is loaded. Microsoft describe on their Documentation for Windows and UEFI what requirements are needed and how the secure boot process looks like [22]. It is part of the UEFI specification and uses, similar to trusted boot, checksums of firmware, option roms

and the boot loader. These checksums are checked against a signature database, which is held within the platform's NVRAM. The signatures are created with the platform key (PK) which is by default owned and managed by Microsoft. Although it is possible to install a new own PK and sign relevant software with it, you can only boot software signed from Microsoft by default when secure boot is enabled.

Shim is the gatekeeper for OSes not maintained by Microsoft. The binary is signed with the official PK and uses itself a self signed CA to sign further executables. A detailed description how shim works on Ubuntu is shown on their corresponding Wiki page [13]. Only this workflow enables secure boot when using Linux OSes.

Secure boot uses conventional non-volatile memory instead of the TPM to store private parts of its signing keys. Therefore, Secure and trusted boot can exist side by side on one system. However, Garrett shows that there exists a way to bypass secure boot without notifying the user [14]. He describes that the secure boot process only hands over environment variables to the kernel which tells that the previous boot steps were certified. An attacker could easily pass the same variables to the kernel, which boots then without noticing the broken secure boot process which breaks the effective trust in this system.

2.2.5 Intel Trusted Execution Technology

As TPM 1.2 chips became available, Intel published an own standard for trusted virtual environments which they called *Trusted Execution Technology* (TXT) [10]. It extends the standard from TCG by defining a measured launch, similar to trusted boot, for virtual machines. When booting a VM, it measures the boot process and provides the results to the hypervisor. As soon as the measuring results are available, the hypervisor is able to check those against known values and detect modified (= untrusted) machines. The hypervisor can then act accordingly to defined policies or attest the integrity of the guest machine.

2.2.6 Trusted Execution Environment

Although the name is similar to Intel's TXT, the goal of the *Trusted Execution Environment* (TEE) is quite different. Instead of *trusting* the whole system, TEE is a separate portion of the main processor with separated memory and storage capabilities.

ToDo!(In deinem Fall würde ich empfehlen, Intel TXT (sehr kurz, ein Absatz reicht) und TEE allgemeiner (konkrete ARM Trustzone) etwas ausführlicher (nicht mehr als eine halbe Seite, dafür vielleicht mit ein paar Referenzen wofür ARM TEEs schon heute verwendet werden, was als teilweise Ersatz von TPMs gesehen werden könnte) zu erwähnen. Aber noch weiter würde ich die Runde nicht ziehen.)

2.3 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 officially 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 [27].

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 2.2.1. 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.

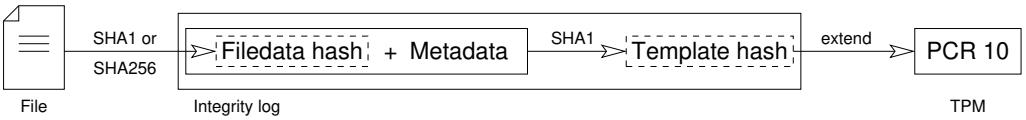


Figure 2.2: Overview of generating an entry in the integrity log

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

- `ima_template` sets the format of the produced 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.

2.3.1 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 shown in Figure 2.2. 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* which is independent of the previous algorithm a SHA1 checksum. 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. Unfortunately, recomputing the hash chain was not possible in the demonstration setup. We discuss that problem in Chapter 6.

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.

The first entry in every measurement file is called `boot_aggregate`. It is the trust link between trusted boot and IMA representing a cumulative hash of the PCR values 0 – 7. Consequently, the PCR result of trusted boot is also embedded in the measurement log and the corresponding hash chain of PCR 10.

2.3.2 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.

2.3.3 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 when accessed for read.

- `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 [18]. It is, for example, useful to exclude constantly changing log files from being measured to reduce useless entries in the measurement log.

2.3.4 Other System Integrity Approaches

Schear et al. [28] 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 use case. Instead, the system should be self contained as good as possible and it should be possible to get information about the system via anonymous attestation.

2.4 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. [3] [5], 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 in the TPM 1.2 standard.

This section describes the concept by Camenisch et al. [5] 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.

2.4.1 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 [7] introduced the algorithm based on the Schnorr Signature Scheme. It only assumes a collision resistant hash function $\mathcal{H} : \{0,1\}^* \rightarrow \{0,1\}^k$ 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. It 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 $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ that satisfies three properties [6] [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 [6] 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 $a \neq 1_{\mathrm{G}_1}$, $e(a, Y) = e(b, g_2)$ and $e(a, X) \cdot e(b, X)^m = e(c, g_2)$.

2.4.2 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. [5], 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.

Please note that these entities are different to the Digidow roles although they have the same name. When leaving the DAA context after this section, the DAA entities will clearly be qualified (e.g. *DAA* verifier). 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. [5] 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 $\mathsf{JOINPROCEED}(Q, \pi_1)$, $\mathcal{I}$ 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 $\mathsf{APPEND}(a, b, c, d, \pi_2)$ to $\mathcal{H}_j$
7. Upon receiving $\mathsf{APPEND}(a, b, c, d, \pi_2)$, $\mathcal{H}_j$
   - 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 $\mathtt{bsn}$.

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. [5] 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.

2.4.3 Standardization of DAA

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]. It implements a close variant of the previously described concept.

3 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.

3.1 Definition of the Biometric Sensor

The BS itself is defined as edge device within the Digidow network which 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 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.

3.2 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 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.

When a BS is working in production, it will usually be in an exposed environment. There may be an instance of a verifier next to the BS. The connection into the Digidow network may, however, based on untrusted networks. Furthermore the physical environment may

not be trustworthy. Given this environment, there are a number of threats that need to be considered when building a BS:

- *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*: Similar to metadata extraction, the adversary might directly access the attributes via wiretapping the USB cable. The adversary might be able to identify an individual with those attributes.
- *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.

Although all of these attack vectors should be mitigated when used in production, we will address only a subset for the prototype. First, we assume that only authorized personnel has access to the hardware itself. Any other person should only interact with the hardware sensor. Therefore any threat attacking communication between internal system components will not be addressed. Furthermore, we will assume an already established bidirectional channel between BS and PIA. Any algorithms on how the BS finds the corresponding PIA exceed the focus of this work.

On the other hand, any hardware modification including firmware upgrades should be detectable by the prototype system. In addition to this detection the BS should generate trust by having a provable system state before a Digidow transaction is executed. This includes mitigations against attacks on replaying attributes, blocking attribute transmission or aggregating them while running.

3.3 Prototype Concept

Given the threat model and the use cases described in Section 3.1, we will introduce a prototype which will address mainly the cryptographic aspects of trust. Furthermore we will discuss system integrity approaches and their limitations.

3.3.1 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 2.1, the TPM functions can be delivered in three different flavors: As dedicated or mounted device, as part of the platform firmware, or as software simulator. The firmware variant 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 2.2. 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 *empty* 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.

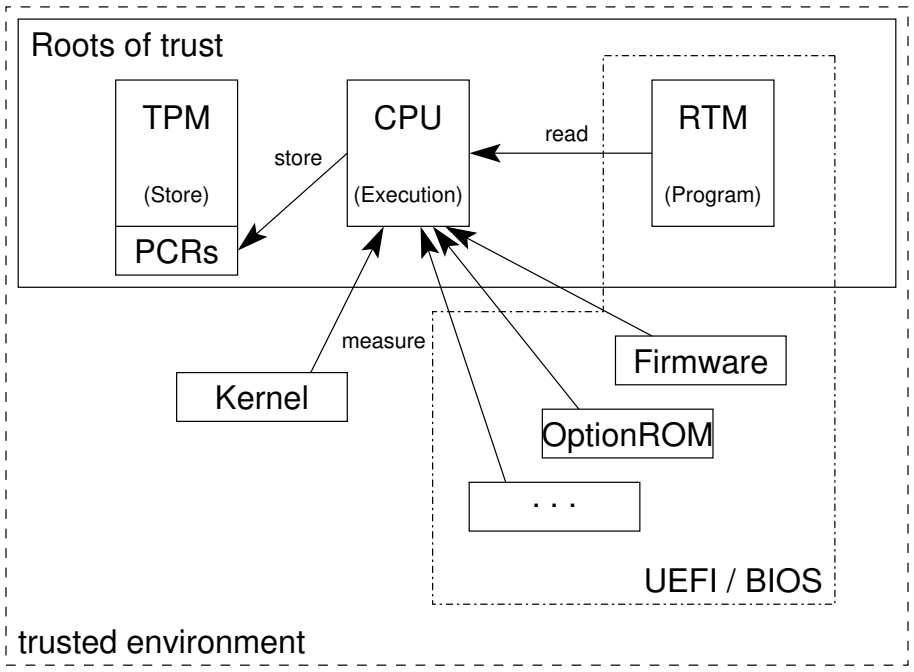


Figure 3.1: Extending trust from the Roots of Trust up to the kernel

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 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 3.1 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. 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.

3.3.2 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 2.3, 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 attached via USB.

3.3.3 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 with 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.

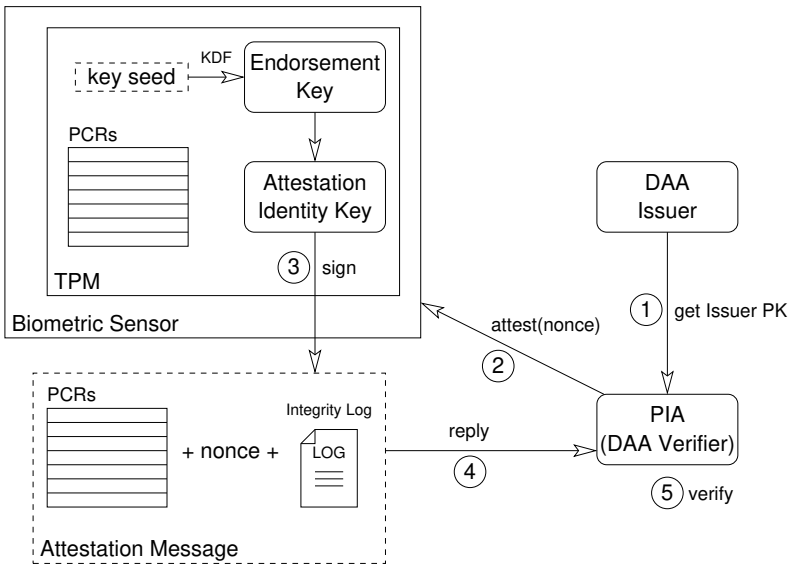


Figure 3.2: The DAA attestation process requires 5 steps. The PIA may trust the BS afterwards.

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 DAA issuer. During signup, DAA issuer and BS (DAA member) negotiate the membership credentials over the network. By being a member of the DAA group, the DAA issuer fully trusts that the BS is honest and acting accordingly the specification. The DAA issuer will not check any group members, since they can now act independently.

When the BS is then authenticating an individual, the process illustrated in Figure 3.2 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.

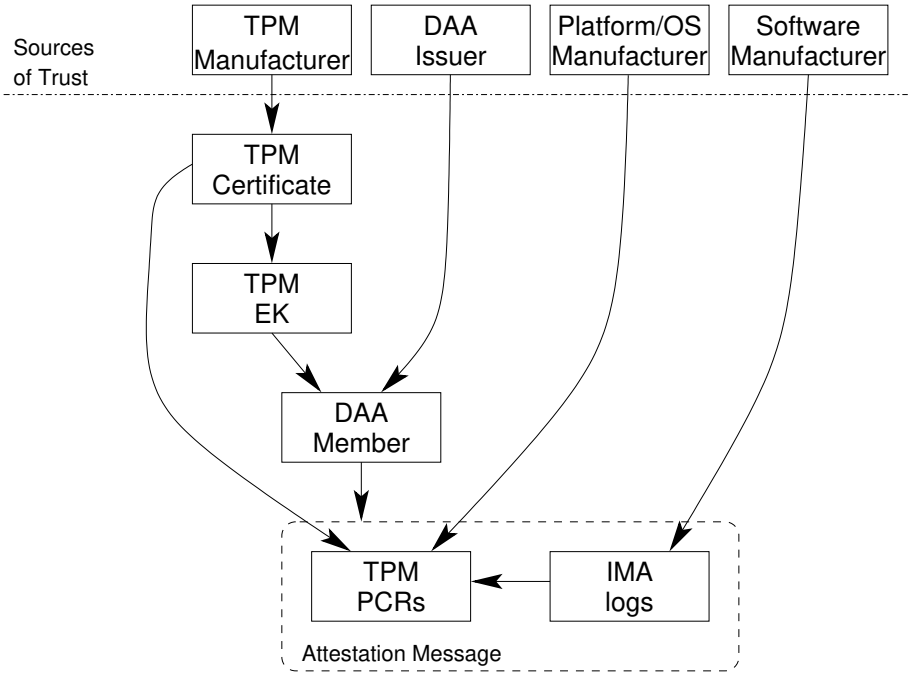


Figure 3.3: Overview of the Chain of Trust of the BS

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 3.3 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. For example, any TPM manufacturer has to implement the well defined standard from TCG. There exists, however, no such exact definition for hardware and firmware parts of the platform. Consequently, these parts should undergo a functional analysis before they receive a trust certificate. Trust means that, when the platform is defined trustworthy, the corresponding PCR values should be published to be verifiable when Digidow transactions occur.

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 3.3. 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 DAA verifier can now check the valid membership by checking the signature of the message against the DAA 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 integrity log against known values. PCR 10 represents the end of the hash chain fed by the integrity log entries.

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

4 Implementation

The concept described in Chapter 3 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 4.1 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.

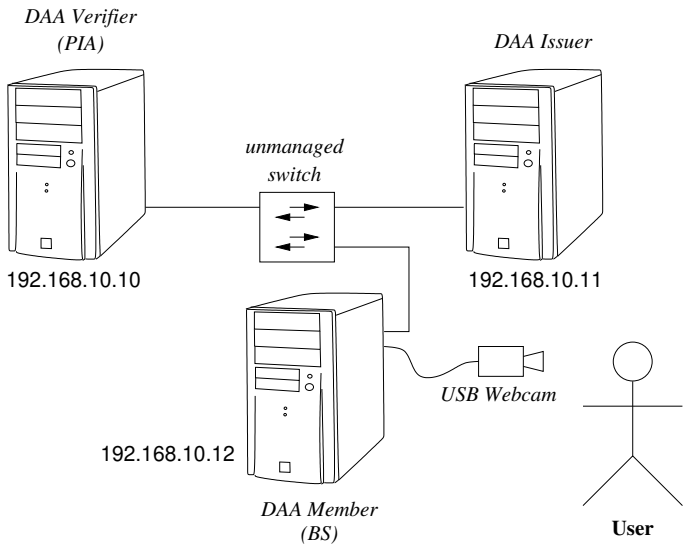


Figure 4.1: Prototype setup to show DAA features and the dataflow from BS to PIA

4.1 Hardware Setup

For demonstrating remote attestation via DAA over a simple network infrastructure, we use three systems with similar configuration. Table 4.1 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.

The used mainboards come with a dedicated TPM 2.0 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 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 TPM 2.0 even when coming with a prorietary header.

Table 4.1: Systems used for demonstration prototype

	System 1	*System 2*	*System 3*
Processor	AMD Athlon 240GE	Intel Pentium G4560T	Intel Pentium G5400T
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 module	Gigabyte TPM2.0_S	Gigabyte TPM2.0	Gigabyte TPM2.0_S
TPM chip	Infineon SLB9665TT2.0	Infineon SLB9665TT2.0	Infineon SLB9665TT2.0

4.2 Select the 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 [19] 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 compatibility was tested with an older OS version (18.04). When installing Ubuntu on the prototype, we used *Full Disk Encryption* (FDE) which leads to the disk allocation described in Table 4.2. Ubuntu installs Grub by default, which we will use in the following 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 4.2: Disk layout of the BS prototype

4.3 Trusted Boot

Parts of this section contain results of the master project which documented how to build an Linux environment with TPM backed full disk encryption. This includes the evaluation which Linux distributions support trusted boot, the scripts to handle the encryption key in the TPM and the documentation how to build the unified kernel.

By default, every mainboard with support for TPM 2.0 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 needs 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 2.1. According to the documentation [12], 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.

Embedded systems like a rolled out version of the BS do not need several boot options, making Grub unnecessary. Therefore we can replace Grub's 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 autonomously. Pornkitprasam [25] [26] and the Tevora company [30] 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 4.3 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 customized.

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 4.3: Memory layout of the Unified Kernel EFI file

4.3.1 Install and use Trusted Boot

The following shell scripts are available online[1]. All are tested on a Ubuntu 20.04 server minimal installation on all three devices. These packages need to be installed beforehand to make use of the scripts:

- `binutils` for objcopy, generating the unified kernel, and
- `tpm2-tools` to interact with the onboard TPM,

Installing trused boot is done in three steps, assuming being root on the target system:

1. *Execute* `install.sh`: It installs the shell scripts into `/usr/sbin` and adds a new random passphrase to LUKS. It furthermore adds TPM support to the initramfs and creates the unified kernel described above.
2. *Reboot*: During reboot, the new PCR values are generated.
3. *Execute* `update-luks-tpm.sh`: The new PCR values are used to seal the LUKS passphrase into the TPM.

After finishing these steps, the system should be able to decrypt the FDE without user interaction and end up at the login prompt. The automatic boot will now work as long as there are no updates affecting the unified kernel. If a kernel update happens during upgrading the system, a new EFI blob must be generated and the TPM seal is to be renewed. Similar to the installation procedure, this can be done in the following procedure:

1. *Execute* `update-kernel.sh`: This includes the latest updates of kernel, its parameters and the initramfs into the EFI blob.

[1] https://git.ins.jku.at/proj/digidow/trustedboot

Listing 4.1: `kernel-command-line.txt`: Command line for the Kernel

```
1 /vmlinuz 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 4.2: `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:␣"
```

2. *Reboot*: Again, the new PCR values need to be calculated.
3. *Execute `update-luks-tpm.sh`*: Similar to the installation, the new seal is generated and replaces the old one.

The result is a recent kernel with up-to-date configuration.

4.3.2 Detailed description of the scripts

Listing 4.1 shows the used command line which will be saved on `/boot/kernel-command-line.txt`. Relevant for trusted boot is the parameter where the root filesystem is located. This example works for the default Ubuntu LVM on LUKS configuration. The parameters activate also IMA which will be discussed later in this chapter.

If FDE is installed, the boot process needs 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 the passhrase with the PCR registers. The unseal operation itself is then done with Listing 4.2, which also needs to exist in the initramfs. We copy the script of Listing 4.3 to `/etc/initramfs-tools/hooks` to enable TPM access during boot after the next initramfs update. Next, the script of Listing 4.5 creates a new key for FDE 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. Using a clear text password file

Listing 4.3: `tpm2-hook.sh`: Script copying required TSS files into the initramfs

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

Listing 4.4: `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 $LATEST | grep -o 'version␣[^␣]*' | cut -d '␣' -f 2`
objcopy \
  --add-section .osrel="/usr/lib/os-release" --change-section-vma .osrel=0x20000 \
  --add-section .cmdline="/boot/kernel-command-line.txt" --change-section-vma .cmdline
      =0x30000 \
  --add-section .linux="/boot/vmlinuz-$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"
```

in the `/root` directory is not per se considered insecure as the disk must be decrypted beforehand, meaning that the passphrase also exists unencrypted in memory. Furthermore only root has access to the target directory. The backup is needed for updating the system and when issues with the TPM appear. When an updated kernel is booted, the seal cannot be opened anymore since the old PCR values cannot be restored anymore. It is, of course possible to generate a new passphrase every time when the PCR values change intendedly. In this case no passphrase need to be stored on the file system.

Finally, Listing 4.4 creates the unified kernel according to Table 4.3 using the command `objcopy` and copies it 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 4.6.

Listing 4.5: `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 4.6: `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"
```

Listing 4.7: `update-luks-tpm.sh`: Script for updating the Sealing of the TPM Object with new PCR values

```
1 #!/usr/bin/bash
2 echo "Updating TPM Policy with current available PCRs"
3
4 set +e
5 tpm2_evictcontrol -c 0x81000000
6
7 set -e
8 tpm2_flushcontext -t
9 tpm2_createprimary -C e -g sha256 -G ecc256 -c /root/keys/e-primary.context
10 tpm2_createpolicy --policy-pcr -l sha256:0,1,2,3,4,5,6,7 -L /root/keys/pcr-policy.
       digest
11 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
12 tpm2_flushcontext -t
13 tpm2_load -C /root/keys/e-primary.context -u /root/keys/obj.pub -r /root/keys/obj.
       priv -c /root/keys/load.context
14 tpm2_evictcontrol -C o -c /root/keys/load.context 0x81000000
15 # tpm2_unseal -c 0x81000000 -p pcr:sha1:0,1,2,3,4,5,6,7 -o /root/test.bin #proof that
        the persistence worked
16 rm -f /root/keys/load.context /root/keys/obj.priv /root/keys/obj.pub /root/keys/pcr-
       policy.digest
17 tpm2_flushcontext -t
```

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 4.7 shows how the passphrase is sealed into the TPM with all relevant PCR registers. Line 15 of the script shows the unseal operation which is used to get the passphrase out of the TPM. It should result in an exact copy of the original passphrase in `/root/keys`. This finishes the installation of trusted boot resulting in an unattended disk decryption process during system boot. 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.

4.4 Integrity Measurement Architecture

The result of trusted boot is a measured and therefore trusted kernel with its command line parameters and modules. IMA extends this trust to the file system as described in Section 2.3. All features of IMA are already implemented in the kernel sources and do not need additional packages. The Gentoo wiki page about IMA [17] describes which kernel compiler flags are required to enable IMA in a Gentoo kernel.

According to their blog [29], Redhat supports IMA since Enterprise Linux 8 and so would the Redhat clones like CentOS, Rocky or Alma Linux. Ubuntu enabled the required kernel compile flags by default on version 20.04 in the ubuntu kernel repository[2]. However, we found no official information on when IMA was introduced or whether IMA support will continue in future releases.

Every kernel supporting IMA creates a virtual file at `/sys/kernel/security/ima/ascii_runtime_measurements`. When IMA is disabled, which is the default, this file has only one entry representing the boot aggregate. By enabling IMA via kernel command line parameters, this virtual file gets filled according to the policies defined. The first four parameters used in the kernel command line as shown in Listing 4.1 define the behavior of IMA and how the measurement log should look like.

- `ima_appraise=fix` appends the filehash as an extended file attribute for every accessed file.
- `ima_policy=appraise_tcb` together with `ima_policy=tcb` analyze files owned by root and opened for execution.
- `ima_hash=sha256` sets the hashing algorithm.
- `rootflags=i_version` must be enabled when mounting the filesystem since IMA is checking the `i_version` number before re-hashing the resource. When it is disabled, the kernel has to hash the resource every time.

Unfortunately, the resulting integrity log is between 2000 and 3000 lines long when the system is freshly booted. The log can blow up to several 10000 lines when uptime exceeds

[2] https://kernel.ubuntu.com/git/kernel-ppa/mirror/ubuntu-5.4-focal.git/tree/security/integrity/ima/Kconfig, last visited on 9.7.2021

several days. Therefore, IMA has a policy language where rules for hashing files can be customized. It might, for example, be useful to exclude log files from being hashed.

A third party can comprehend the state of the attesting system, when parsing the integrity log. Together with the corresponding value of PCR 10, where the hashes of all entries are chained, the TPM can `certify` correctness of the log.

Besides the log entries, every hashed file gets an extended file attribute called `security.ima` which holds the current hash value in base64 encoding. The apt package `attr` contains the binary `getfattr` which can show these additional attributes with the following command:

```
getfattr -m - -d trustedboot/install.sh
```

Having attributes for all relevant files allows IMA to enforce appraisal. This means that the hash of every file is checked before accessing the file. Before enforcing IMA, the root filesystem should be parsed to create attributes for every file. Touching the file is not enough, the file must be opened for read. The following command opens every file in an ext4 filesystem for read, causing the kernel to create/update the file attributes, but does not read any data:

```
time find / -fstype ext4 -type f -uid 0 -exec dd if='{}' of=/dev/null count=0 status
    =none \;
```

It takes about fifteen minutes to parse the filesystem on a Ubuntu minimal server system given the hardware described above.

The final step is to enforce IMA appraisal via kernel command line parameter. By changing the parameter `ima_appraise=enforce` in `/boot/kernel-command-line.txt`, IMA enforces the hash and will deny access if the hash is not correct. The new parameters are activated by using the workflow for updating the kernel described in Subsection 4.3.1.

The result is a system which allows only resources with correct hash attribute to be read or executed. However, an adversary may still create the file hash itself and overwrite the file with the hash. Hence these hash values must be verified in some way before they can be trusted. Two options are therefore available:

- *Attest*: Let the complete integrity log be part of remote attestation and verify the hash values against a trusted known value database. This makes the verification part very complex but the state of the attesting system is well documented.

- *Sign*: Sign hash values for static files like executables or libraries. The host can directly check whether the file was tamepered with and act accordingly. Linux kernels support this feature with the *extended verification module* (EVM).

We use the variant where the integrity log is part of the message. Details about the message design will be discussed in Subsection 4.7.3. This variant makes the BS prototype implementation easier, however, it increases complexity to completely verify the message. It might be useful in the future to use EVM to prevent executing unsigned or modified code.

4.5 Runtime Analysis

IMA is a comprehensive tool for checking the integrity of a file or executable or library before it gets executed. When access is granted, IMA's job is done. Hence, IMA is a tool for static system analysis. An executable can however change its behaviour during runtime, either intended with remote procedure calls or unintended with a remote code execution vulnerability. These attack vectors can be addressed with runtime analysis.

One approach for runtime analysis is to create a log of syscalls for a certain process. Syscalls are the only interface from the program to the OS when managing resources needed by the program. Hence logging the trace of syscalls shows which resources it needs and which files are accessed or modified. Any unusual behavior could be detected immediately given a good knowledge of the used resources when executing a good version of the program.

The trace logs are created with `auditd` which requires the corresponding apt package to be installed.

4.6 Prove TPM 2.0 Certificate Chain

Every TPM has a corresponding certificate which is part of a certificate chain maintained by the TPM manufacturer. In our case, Infineon certifies its TPM with a number of intermediate CAs which itself are certified with Infineon's root CA. The TPM certificate is available for RSA and ECC cryptofamilies respectively. Since the verification workflow

is the same on all machines and for both cryptofamilies, we demonstrate on system 1 how the process works. Note that this works for Infineon TPMs. Other vendors like STM, AMD or Intel may provide certificates via download on their website.

1. Read the certificate from the TPM NVRAM. The RSA certificate is located at address `0x1c00002`, that for ECC on address `0x1c0000a`:

```
root@amd1:~# tpm2_nvread -C o 0x1c0000a -o amd1_ecc.crt
```

2. Download the certificates from the intermediate and root CA from Infineon's website:

```
root@amd1:~# wget https://www.infineon.com/dgdl/Infineon-TPM_ECC_Root_CA-C-
    v01_00-EN.cer?fileId=5546d46253f65057015404843f751cdc -O
    infineon_ecc_root_ca.crt #Infineon root CA
root@amd1:~# wget https://www.infineon.com/dgdl/Infineon-OPTIGA-ECC-
    Manufacturing-CA_036-Issued_by_RootCA.crt-C-v01_00-EN.crt?fileId=5546
    d46262475fbe0162486417b73cbe -O infineon_ecc_intermediate_ca_036.crt #
    Infineon intermediate CA
```

3. Convert all certificates into PEM format. OpenSSL can only verify a chain in PEM format.

```
root@amd1:~# openssl x509 -inform DER -outform PEM -in infineon_ecc_root_ca.
    crt -out infineon_ecc_root_ca.pem
root@amd1:~# openssl x509 -inform DER -outform PEM -in
    infineon_ecc_intermediate_ca_036.crt -out
    infineon_ecc_intermediate_ca_036.pem
root@amd1:~# openssl x509 -inform DER -outform PEM -in amd1_ecc.crt -out
    amd1_ecc.pem
```

4. Check the certificate chain with OpenSSL. The root certificate and the intermediate certificate need to be concatenated into one file to allow `openssl` to check the certificate chain as well.

```
root@amd1:~# openssl verify -verbose -CAfile <(cat
    infineon_ecc_intermediate_ca_036.pem infineon_ecc_root_ca.pem) amd1_ecc.
    pem
amd1_ecc.pem: OK
```

When OpenSSL returns `OK`, the certificate chain is intact and the TPM is indeed one from Infineon. To be exact: The website, probably hosted by Infineon, provides a certificate chain which matches and the links to the corresponding parent certificate are correct. Unfortunately, Infineon does neither provide any website certification nor any checksums

of the provided certificates. So, if the above described check fails, no source of trust can ensure that the root certificate is correct.

We found that both, ECC and RSA chains of all TPMs are intact. For documentation reasons, we provide the OpenSSL SHA256 checksums of all used certificates in Table 1 of the Appendix. These checksums were generated with:

```
openssl x509 -noout -fingerprint -sha256 -inform pem -in amd1_ecc.pem
```

4.7 Using the DAA Protocol

Direct anonymous attestation uses the TPM as cryptoprocessor and key store. The feature of identifiable instances of sensors is not required when interacting with the Digidow network. Only the *trusted* state of the sensor system and the membership in the corresponding group is relevant. Hence, the group membership is an essential part to provide trust to the users, requiring a deep knowledge on what hardware and software is installed and which vulnerabilities it might have.

The DAA group membership states that the system is provisioned from a trusted party, namely the DAA issuer. The level of trust is ultimative since the used version of DAA is only partly dynamic by lacking support of membership removal. During a Digidow transaction, the sensor attests its state by signing a message containing the integrity log and the PCR registers. Any party interacting with the sensor is then able to check trustworthiness via integrity and valid membership of the sensor.

We describe in the following which programs need to be installed and what configuration is required to demonstrate a working implementation of DAA.

4.7.1 Provision Hosts of Test Setup

The demonstration setup, shown in Figure 4.1, consists of three independent hosts which are connected together via TCP/IP. Every host represents one party in the DAA scheme, each requiring additional software to support the DAA protocol. Xaptums ECDAA library needs to be installed on all three hosts but only the sensor requires TPM support. Similar

to that, the ECDAA network wrapper is required to support the network communication part.

The member needs, besides DAA protocol support, software to capture and process an image of the USB webcam. We developed a small Rust program called `bs-capture` for capturing a face image from a webcam. For biometric processing, we transform the image into an embedding with the face recognition prototype of Digidow[3].

4.7.2 Installing Xaptum ECDAA Library

Xaptum's ECDAA library provides the cryptographic functions and the protocol primitives for DAA. A file based demonstration of the protocol is provided within the project. We have to build the ECDAA library from source since the provided deb packages do not officially support Ubuntu 20.04. Therefore we need the C build environment as follows:

```
root@amd1:~# apt install gcc cmake build-essential doxygen doxygen-latex parallel
```

The sensor host requires TPM support which is enabled with the additional package `libtss2-dev`:

```
root@amd1:~/ecdaa/build# apt install libtss2-dev
```

Download the repository from GitHub[4] and create the build folder on the filesystem:

```
root@amd1:~# git clone https://github.com/xaptum/ecdaa.git
root@amd1:~# mkdir -p ecdaa/build && cd ecdaa/build
```

The next step is to build and install the required dependencies from source. Cmake uses the environment variable `CMAKE_PREFIX_PATH` as installation target. Xaptum provided a shell script for the complete routine:

```
root@amd1:~/ecdaa/build# export CMAKE_PREFIX_PATH=/usr
root@amd1:~/ecdaa/build# ../.travis/install-amcl.sh ./amcl /usr FP256BN
```

Finally install the build of the project with `cmake`. Set the variable `DECDAA_TPM_SUPPORT` respectively:

[3] https://git.ins.jku.at/proj/digidow/prototype-facerecognition
[4] https://github.com/xaptum/ecdaa

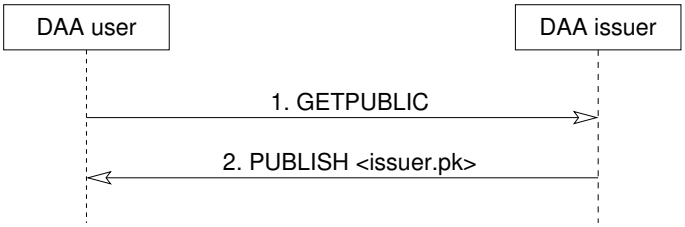


Figure 4.2: Protocol to get the DAA issuer's public key

```
root@amd1:~/ecdaa/build# cmake .. -DCMAKE_BUILD_TYPE=Release -DECDAA_CURVES=FP256BN
    -DCMAKE_INSTALL_PREFIX=/usr -DECDAA_TPM_SUPPORT=ON
root@amd1:~/ecdaa/build# cmake --build . --target install
```

Now, all prerequisities are installed to build and install the ECDAA network wrapper which is a contribution of this thesis.

4.7.3 DAA Network Protocol

The network protocol provided by `ecdaa-network-wrapper` adds a network communication layer to the cryptographic implementation of Xaptum's ECDAA project. It is designed to match the workflow of a Digidow transaction, affecting the decision which party is defined as listener and which as sender.

- *Start DAA issuer listener*: During startup of the issuer server, the program loads the public/private key pair if present. Otherwise, a new key pair will be created. The DAA issuer listener is always active to manage group subscriptions and queries for the group public key.
- *Broadcast DAA group public key*: The group public key created by the DAA issuer is necessary for group enrollment and for verification of any messages signed by a DAA group member. Consequently, DAA verifier and (potential) DAA member must get this key first. Figure 4.2 shows the two steps that are visible on the network. Since this communication contains only public data, no additional privacy measurements are required.
- *Enroll member to issuer's DAA group*: This step extends the DAA group and transfers the trust of the DAA group to the new member. This protocol requires the group

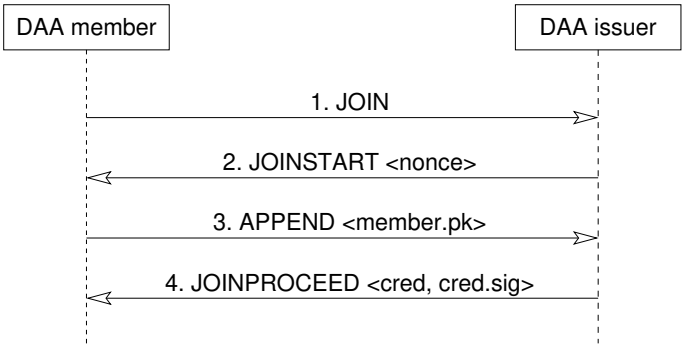


Figure 4.3: Protocol to add a new member to the issuer's DAA group

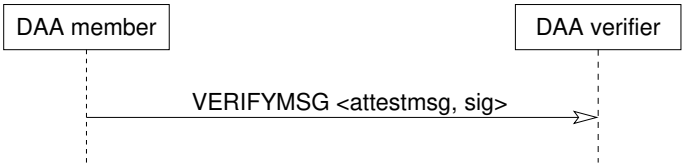


Figure 4.4: Protocol to send an attestation message to the DAA verifier

public key to be present at the DAA member. If this is not the case, the member asks for it automatically. The procedure is a four-way handshake, as shown in Figure 4.3. We describe in Subsection 2.4.2 that the key exchange is cryptographically secure, meaning that no adversary can extract any private keys when getting access to these messages. When the member is eventually part of the DAA group, it is *tusted* by the Issuer, requiring a thorough inspection of the member beforehand. Since we only show a working demo of the cryptographic concept, we skipped any efforts to check the member system in this implementation.

- *Send signed messages*: The diagram in Figure 4.4 shows that the DAA verifier listens and the member initiates the connection. This implementation reflects the Digidow transaction workflow in mind where the sensor (=member) sends a signed message to the PIA (=DAA verifier). Hence, sending attestation messages with biometric information of the user will happen once per transaction in the Digidow network.

The message contains, as shown in Listing 4.8 the face embedding and the attestation information about the sending host. Instead of putting the plain integrity log into this message, we decided to use only checksum and the number of entries to keep the message size rather small. We continue this discussion in more detail in the next chapter. Fur-

Listing 4.8: DAA message file format

```
1  embedding: <embedding in cev format>
2  sha1: <PCR SHA1 list>
3  sha256: <PCR SHA256 list>
4  imachecksum: <SHA512 sum of integrity log>
5  ima_lines: <number of entries in the integrity log>
```

thermore, the the ECDAA library limits the message size for signing to 1024 Bytes. Thus, we created a sha512 sum of the message and signed only this hash allowing us to send messages of arbitrary size and constant effort for signing.

4.7.4 Installing the DAA Network Protocol

The Network Wrapper can be downloaded from the Git repository[5]. Copy the folder `ecdaa-network-wrapper` to the build directory and change to this directory:

```
root@amd1:~# git clone https://git.ins.jku.at/proj/digidow/ecdaa-network-wrapper.git
root@amd1:~# cd ecdaa-network-wrapper
```

Initialize Cmake with the following command:

```
root@amd1:~/ecdaa-network-wrapper# cmake .
```

Then build the preferred targets, depending which host is used. For example, to build the member binary with TPM support, use:

```
root@amd1:~/ecdaa-network-wrapper# cmake --build . --target ecdaa_member_tpm -- -j 2
```

The following targets are available:

- `ecdaa_issuer`: Creates the binary for the DAA issuer.
- `ecdaa_member`: Builds the DAA member executable without TPM support. This should only be used for testing purposes.
- `ecdaa_member_tpm`: The DAA member binary with TPM support.
- `ecdaa_verifier`: Creates the DAA verifier binary.

[5] https://git.ins.jku.at/proj/digidow/ecdaa-network-wrapper

- `ecdaa_all`: Builds every binary listed above at once.

When all above steps are finished successfully, the host is capable of taking its role in the DAA protocol.

4.8 DAA Demo Application

For demonstration purposes, we use an USB webcam to take a photo of the person being in front of the sensor. This photo is then processed to generate a face embedding, which is small enough to be sent with the DAA attestation message.

The first part is done with a small Rust program called `bs-capture`, which is available online[6]. It uses the libraries from the video4linux project to capture a still image and save it to disk. Ubuntu 20.04 requires the following packages to be installed:

```
root@amd1:~# apt install rustc cargo libv4l-0 libv4l-dev
```

Then the program can be downloaded and executed:

```
1 root@amd1:~# git clone https://git.ins.jku.at/proj/digidow/bs-capture.git
2 root@amd1:~# cd bs-capture/
3 root@amd1:~/bs-capture# cargo run
```

Cargo creates a binary in `target/debug/bs-capture` which can then be used for this prototype. The program assumes that a webcam is available at `/dev/video0`. It takes a still image which is saved as `frame.jpg` in the working directory, in this example `~/bs-capture`.

This image needs to be processed to generate the face embedding data. Therefore we use the project *Prototype Facerecognition* which uses a trained tensorflow network to generate embeddings. The branch `retinaface-tflite` of the Git repository[7] contains an install script which takes care of installing all dependencies for setup:

```
1 root@amd1:~# git clone https://git.ins.jku.at/proj/digidow/prototype-facerecognition
      .git
2 root@amd1:~# cd prototype-facerecognition/
3 root@amd1:~/prototype-facerecognition# git checkout origin/retinaface-tflite
4 root@amd1:~/prototype-facerecognition# ./install.sh
```

[6] https://git.ins.jku.at/proj/digidow/bs-capture
[7] https://git.ins.jku.at/proj/digidow/prototype-facerecognition

This branch contains a tensorflow version which was compiled to work on our test systems. The default version uses an instruction set extension which is not available on the Pentium Gold processors of System 2 and 3.

When all parts are installed on the sensor, and issuer and verifier are ready to take connections from the sensor, the setup is finished. All tests of the following chapter are performed using above installation description.

5 Testing

With the setup described in the prevoius chapter, we created a system which is able to capture and process biometric data. The system encapsulates this data into an attestation message and sends it to the PIA which is the DAA verifier in this configuration. We show in the following how well each part of the setup work and which performance the tested devices show. Furthermore we analyze the footprint in memory as well as on the disk. The tests are only applied to System 1 and 3 since System 2 has a comparable hardware configuration to System 3 but uses a CPU of an older generation. Furthermore, only two TPMs support the cryptographic parts required for ECDAA. The TPM in System 2 uses an older firmware version which does not support the used implementation of ECDAA. We discuss this issue in further detail in Section 6.2. Consequently, System 2 was used as DAA verifier since this host does not require a TPM.

5.1 Trusted Boot

The first part of the setup is trusted boot which is well integrated in recent releases of kernel and GRUB bootloader. Furthermore, unlocking the optional disk encryption with the TPM works seamless with the kernel, even when using the manually generated unified kernel without Grub. Only when performing an update for Grub, it will modify the entries in the EFI boot manager. Consequently, we recommend a check of the boot entries after a system upgrade.

Having a backup boot option with Grub is useful for maintenance tasks during development. Especially for modifying the IMA configuration via kernel command line, it may be necessary to boot with a backup kernel. Hence, a backup boot process is strongly recommended for test setups. This requires a backup password for the disk encryption which allows to bypass the TPM during booting. Otherwise there are no updates possible

with the current setup since the affected PCRs are used by the EFI bootloader and cannot be precomputed. However, a backup bootloader is not required when operating the BS in unattended environments.

GRUB already supports trusted boot and activation requires a line in the corresponding config file. Unlike Grub, the unified kernel does not perform any measurements. Only when asking the TPM for the disk encryption key, the initramfs must have the TPM stack available. These files use about 62 KB of space in initramfs which is negligible compared with the complete image using about 80 MB. Furthermore, only in this case exists a measurable difference in the boot performace since asking the TPM takes around 4 s. This is the time when the boot process posts `unlocking via TPM` and stays there until the disk is unlocked. The used amount of memory was not tested here, but it should be comparable to the results when generating a TPM key for the member. According to these results, we assume that less than 10 KB are used to hold the TPM stack in memory.

5.2 IMA

IMA appears to have an easy setup but a complex set of consequences. First, we show the impact on disk usage when IMA is enabled. In this case, the file hashes will be stored as extended file attributes. According to the xattr man page [20], the ext4 file system creates therefore a single block per file where all extended attributes must find place. The block size of the used filesystem is 4 KB. The number of files is determined with the following command:

```
find / -fstype ext4 -type f -uid 0 | wc -l
```

For example, system 1 holds 156,947 files after setup as Biometric Sensor and several system updates. This results in estimated additional disk usage of 613 MB when adding the `xattr` block for each file. Further attributes, however, would not require more disk space on ext4 since every extended attribute must exist in this single block.

IMA holds all runtime information as virtual files in memory, including the integrity log. In context of memory usage, it is clear that the size of the virtual files without the redundant information (like PCR number and log format) is a rough estimation for the memory usage within the kernel since we did not analyze the corresponding data

Table 5.1: Average rebooting times with IMA (n=10)

Reboot time [s]	*System 1*			*System 3*		
	IMA off	IMA fix	IMA enf	IMA off	IMA fix	IMA enf
Shutdown	1.1	21.0	20.6	3.3	24.5	24.5
Boot to disk decryption	13.5	14.1	14.3	14.0	15.4	15.4
Decrypt disk	3.9	4.2	4.2	4.0	4.2	4.2
Boot with disk	10.2	27.4	27.0	10.6	31.5	30.6
Complete Reboot	28.7	66.7	66.1	31.8	75.6	74.6

structures within the kernel. Both memory and disk usage do not change between IMA's fixing and enforcing mode since enforcing only adds file access control.

Saving the file to disk, uses about 2 MB of size per 10,000 lines in the log. This indicates a linear dependency to the number of accessed files for the log and the kernel's memory usage. The log can easily get over 100,000 entries when the system is running long enough. System 1, for example, had a log file with 214,561 lines after about 15 days of uptime, resulting in about 40 MB of size. The log file showed that a major impact on the size was the performed system update. Generating the iniramfs blob and compiling programs from source together generates several 10,000 lines in the log.

When looking into the performance of IMA, there is a huge drop when it is enabled, especially when files are read the first time. We show in Table 5.1 the reboot performance of the used test systems given a setup for a biometric sensor described in Chapter 4 with TPM backed disk encryption enabled. This is an example for a reproducible, file intensive process where all resources have to pass IMA. The procedure was captured via camera and split into 4 stages. *Shutdown* represents the time from entering `reboot` into the shell until the monitor stops displaying content. Since all systems use the disk encryption feature supported with the TPM, we measured how log it takes to ask the TPM for the passphrase (*Boot to disk encryption*) and how long the query itself is executed (*Decrypt disk*). With boot disk and kernel available, the system starts all services until the login prompt on the CLI appears (*Boot with disk*). This ends the reboot procedure.

Although the number of runs is very small, the test clearly shows that enabling IMA doubles the time for rebooting. Furthermore, asking the TPM for the decryption key and decrypting the disk takes around 4 s. This may be caused by the `tpm2_unseal` operation which computes multiple relatively complex cryptographic instructions before the task is done.

5.3 Processing and Sending Biometric Data

Capturing and processing biometric data from the user is quite seamless and the cryptographic part of ECDAA is reliable enough for this prototype. During the following tests, all software and hardware parts worked as expected. Neither TPM nor software errors were encountered.

Analyzing the occupied resources is only meaningful for the DAA member. The implemented protoype of the DAA issuer does only negotiate the membership key. Revocation lists and group management are not implemented yet, although the ECDAA library provides datastrucutres and functions for that. Similarly, the DAA verifier only checks the signature of the received message. In a production setup, both entities must hold the revocation list and perform further checks to trust the DAA member and its messages:

We split the tasks of a Digidow sensor in several parts to document the conrtibution of each:

- *DAA TPM key generation*: Clear the TPM, generate a new EK and DAA key pair and persist the DAA key in the TPM's NVRAM.
- *DAA TPM join w/o keygen*: Use the DAA key which is already in place and negotiate a group membership with the DAA issuer.
- *DAA TPM keygen & join*: This combines the two steps above to give comparable time measurements to the join procedure without TPM.
- *DAA keygen & join*: Generate the DAA keypair and save it to disk. Join the DAA group by negotiating a secret with the DAA issuer.
- *Digidow sensor capture*: Create an image using `bs-capture` and save it to disk.

- *Digidow sensor embed*: Extract a face embedding using the tensorflow application `img2emb`.
- *Digidow sensor collect*: Collect the integrity log and save it to disk. Create a sha512sum of the file and put it together with all PCRs and the face embedding data into one message. Calculate another sha512sum from the message itself and save it to disk.
- *Digidow sensor send*: Sign the message's hash with the TPM DAA key and send it together with the message to the DAA verifier. The verifier saves message and hash for further procedures on its disk.

5.3.1 Disk Usage

In this early stage of the prototype, any statistics about how much disk space is required to run this setup is not useful. All programs besides the face embedding application are rather small, the binaries itself have a size of less than 100 kB. The installing process is still manual requiring a local build environment for C and Rust. Furthermore the programs require a list of dependencies which need to be installed with the package manager. Hence neither the size of the executables, nor the total disk occupation is informative for productive estimations. Simlarly, the face embedding application should be seen as example of a biometric sensor, making a detailed discussion about time and space efficiency less meaningful.

5.3.2 Memory Usage

First, we look into the memory footprint of each part by using `/usr/bin/time`. It measures the the maximum resident size in memory during lifetime, which includes stack, heap and data section. Table 5.2 shows the maximum usage of each task during 10,000 runs in the different IMA configurations `off`, `fix`, and `enforcing`. The memory allocation is constant for all parts in this table. Besides calculating the face embedding of the captured image, the whole transaction can be executed using few Megabytes of heap memory. This would fit on most embedded devices running a Linux kernel. However, the face embedding algorithm uses over 800 MB and requires the majority of the computation time as shown below.

Table 5.2: Maximum memory usage measured with `/usr/bin/time`

Task [kB]	*System 1*	*System 3*
DAA keygen & join	2,000	2,024
DAA TPM keygen & join	2,344	2,324
Digidow sensor capture	3,748	3,688
Digidow sensor embed	809,616	847,712
Digidow sensor collect	5,128	5,172
Digidow sensor send	2,604	2,628

5.3.3 Memory Safety

During these memory tests, valgrind showed a large number of possible memory leaks in the Python binary itself. The following example is executed:

```
root@amd1:~/jetson-nano-facerecognition# valgrind python3 img2emb.py data/test-
    images/test2.jpg
```

Valgrind ends with the report as follows:

```
==1648== HEAP SUMMARY:
==1648==  in use at exit: 32,608,730 bytes in 227,287 blocks
==1648== total heap usage: 810,162 allocs, 582,875 frees, 1,385,416,573 bytes
    allocated
==1648==
==1648== LEAK SUMMARY:
==1648== definitely lost: 3,144 bytes in 28 blocks
==1648== indirectly lost: 0 bytes in 1 blocks
==1648==  possibly lost: 523,629 bytes in 12,842 blocks
==1648== still reachable: 32,081,957 bytes in 214,416 blocks
==1648==                of which reachable via heuristic:
==1648==                  stdstring     : 537,414 bytes in 11,917 blocks
==1648==                  newarray      : 8,920 bytes in 5 blocks
==1648==    suppressed: 0 bytes in 0 blocks
==1648== Rerun with --leak-check=full to see details of leaked memory
==1648==
==1648== Use --track-origins=yes to see where uninitialised values come from
==1648== For lists of detected and suppressed errors, rerun with: -s
==1648== ERROR SUMMARY: 58173 errors from 914 contexts (suppressed: 0 from 0)
```

This report shows that the Python binary (here Python 3.8 from Ubuntu 20.04) is not memory safe, which is a significant drawback for the system and software integrity.

Any binary which is directly involved in the DAA protocol frees every allocated block. Furthermore, any binary in the TPM 2.0 software stack is memory safe according to valgrind. The used shell commands may not free every allocated block, however valgrind still finds no errors in these programs.

5.3.4 Performance

Table 5.3 shows the time consumption for each task with its minimum, average and maximum results over 10,000 runs. The *first* run is stated separately, because it is done immediately after a system reboot where the resources cached by the kernel are not loaded yet. The major delay in the first run is caused by the face embedding program, especially when IMA is enabled. As stated in Subsection 2.3.1, each resource has to be hashed and extended into PCR 10 before access is granted, making the first access significantly longer. Especially the tensorflow application requires significantly more time for the first run.

With IMA set to enforcing, the kernel furthermore manages access to the file asked to read. This decision does not require additional resources compared to the fixing mode. The file must be hashed in any case. As long as the file *integrity* is intact, PCR 10 and the integrity log file have to be written as well. Consequently, the difference between fixing and enforcing mode is to compare the computed filehash with the value in the extended attributes and the decision depending on that result.

Since IMA measures every loaded resource, the corresponding log file will constantly increase during testing. Unfortunately the integrity log is required to collect the data for the Digidow attestation message. Furthermore, it is clear that every Digidow transaction contributes to the log since the handover beween the single tasks is done based on files. Consequently, we expected a runtime dependent on the number of runs, making average and maximum runtime in Table 5.3 unavailable when IMA is enabled.

The graphs of Figure 5.1 show the runtime of each of the runs on both tested systems and with IMA in fixing or enforcing mode respectively. Each run is split into the four parts of a Digidow transaction. The graphs clearly show that our expectation of a linear relation between runtime and number of runs were not satisfied. In addition to the tests above, we measured the time of reading the IMA log file when there it held 266488 entries which is roughly 4 times the number of entries when above tests were finished. Reading this file

Table 5.3: Performance results of joining a DAA group and sending a Digidow transaction message (n=10000)

Task [s]		*System 1*			*System 3*		
		IMA off	IMA fix	IMA enf	IMA off	IMA fix	IMA enf
DAA keygen & join	min	0.03	0.03	0.03	0.03	0.03	0.03
	avg	0.05	0.05	0.05	0.03	0.03	0.03
	max	0.07	0.06	0.06	0.06	0.06	0.28
	first	0.04	0.07	0.07	0.04	0.13	0.04
DAA TPM keygen & join	min	0.33	0.33	0.33	0.35	0.35	0.35
	avg	0.34	0.34	0.34	0.37	0.37	0.37
	max	0.34	0.36	0.35	0.37	0.41	0.40
	first	0.37	0.41	0.41	0.40	0.42	0.35
Digidow sensor capture	min	0.92	0.91	0.92	0.91	0.91	0.91
	avg	1.07	1.05	1.05	1.06	1.06	1.06
	max	1.14	1.14	1.14	1.12	12.48	1.12
	first	1.36	1.42	1.47	1.34	1.46	1.45
Digidow sensor embed	min	3.48	3.51	3.51	4.07	4.09	4.10
	avg	3.53	3.53	3.55	4.12	4.14	4.14
	max	4.11	4.11	4.09	4.74	4.46	4.53
	first	5.41	19.93	19.88	5.99	40.21	40.23
Digidow sensor collect	min	0.07	0.14	0.14	0.09	0.19	0.19
	avg	0.08	n/a	n/a	0.10	n/a	n/a
	max	0.09	n/a	n/a	0.11	n/a	n/a
	first	0.09	0.18	0.22	0.11	0.24	0.25
Digidow sensor send	min	0.25	0.25	0.25	0.26	0.27	0.27
	avg	0.25	0.26	0.26	0.28	0.27	0.28
	max	0.26	0.27	0.27	0.28	0.29	0.29
	first	0.26	0.32	0.32	0.28	0.40	0.40
Digidow sensor transaction	min	4.75	4.84	4.85	5.38	5.50	5.49
	avg	4.92	n/a	n/a	5.56	n/a	n/a
	max	5.52	n/a	n/a	6.14	n/a	n/a
	first	7.12	21.92	21.89	7.72	42.31	42.33

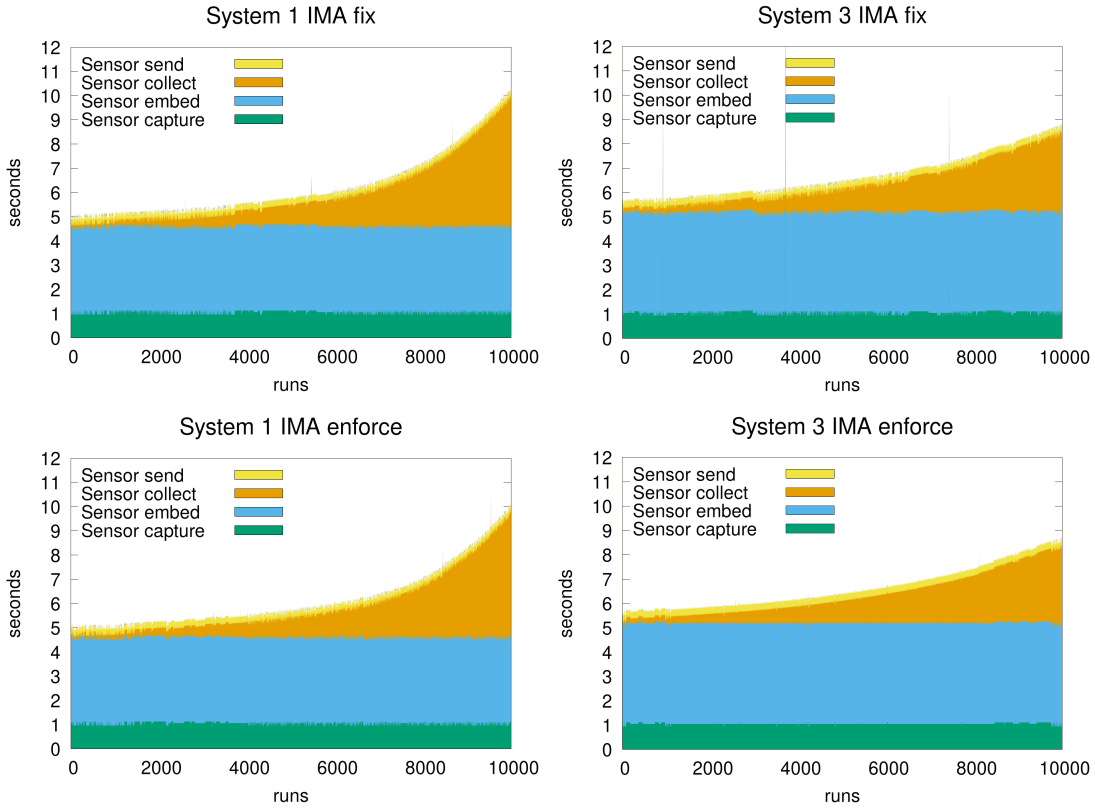


Figure 5.1: Time consumption of a Digidow transaction on the tested systems

Table 5.4: Number of additional entries in the integrity log while IMA is set to enforce

	System 1	*System 3*
Root login after boot	1912	2159
Digidow sensor capture	5	5
Digidow sensor embed	2561	2162
Digidow sensor collect	6	6
Digidow sensor send	12	12
Every other Digidow transaction	6	6

took the kernel of System 3 72.88 s. Compared to the 3.18s in the last run of above tests the slowdown is 22.91. This result supports the complexity estimation of $O(n^2)$ when collecting the integrity log by expecting a slowdown of above 16 for a 4 times larger n. Furthermore, it is interesting, that System 1 with the newer AMD processor seems to be faster in the beginning. When the number of runs reach 10,000, the system needs significantly more time than System 3 with the Intel processor, indicating an even worse asymptotic complexity. Since the software setup on both systems is comparable (Kernel version, Linux distro, installed programs, setup with respect to Chapter 4), it is not clear what the reason for this difference is. It may be found either in the microarchitectural implementation or in differently optimized code for the two processor architectures.

When IMA is in fixing or enforcing mode, the corresponding log will be filled with information about every accessed file. The numbers in Table 5.4 are taken from the integrity log after 10,000 Digidow transaction tests. IMA was set to enforcing and the DAA member key was already in the TPM. The root login happens when the `inputrc` file is loaded. In the IMA file, this file appears roughly in line 2000. Unfortunately, neither the number nor the sequence of entries in the integrity log is predictable in this setup. The number depends on services and daemons started during the boot sequence. Not every service is consistently loaded every time and—depending on its individual state—the number of files loaded by the service may differ over several boot attempts. Predicting the sequence of entries in the log is currently not possible since the kernel is taking advantage of the multicore CPU by starting services and daemons in parallel when possible.

In the example of Table 5.4, System 3 loaded parts of the Python environment before root could login. These resources were partly used by the tensorflow application. Consequently,

these two entries are very volatile and hence hard to predict. However, the contribution of capture, collect and send as well as every further Digidow transaction are consistent. The six entries per Digidow transaction are the changing working files for each run and one file in `/tmp`. Thus, after 10000 runs to start a Digidow transaction the integrity log ended up with about 65000 entries, taking the kernel already several seconds to simply output the file.

In the current setup, the single reliable information out of the integrity log is when one of the programs were executed on the system, the corresponding entries must be somewhere in the log. Furthermore you have to uniquely identify one program or script by the single file defining it.

5.4 Further Test Experiences

When IMA is set to enforcing, some unexpected problems appeared during updating Ubuntu. While executing `apt upgrade`, the package manager downloads the deb packages into its cache folder at `/var/cache/apt/`. These files, however, do not have the `security.ima` attribute when the download is finished. The kernel prevents any access due to this missing attribute and breaks the upgrade process. It is not clear why the files are not hashed although apt is run by root and every file created by root must be hashed according to the active IMA policy. Creating a text file via text editor shows exactly this behaviour.

The missing attributes are also a problem when attemping to turn IMA off again. This requires an updated command line and thereafter an updated unified kernel. Generating the new kernel works fine but moving the blob into the EFI partition fails due to the missing support of extended file attributes. The copy process seems to create the file, but it fails when trying to write the first blocks. However, we found two other ways to change the IMA setting of the kernel. One way is to boot a backup kernel with IMA set to `fix` or `off` and moving the generated file to the EFI partition. The second option is to create a backup of the complete EFI partition when IMA is not in enforcing mode. Creating a backup is done with the following command:

```
dd if=/dev/nvme0n1p1 of=/root/efi-backup.img
```

Listing 5.1: Attempt to recalculate the value of PCR 10

```
#!/usr/bin/bash
set -e
tpm2_pcrreset 16
cut -d '␣' -f 2 /sys/kernel/security/ascii_runtime_measurements > ima.hashes
while read i ;
do tpm2_pcrextend 16:sha1=$i ;
done < ima.hashes
tpm2_pcrread sha1:10,16
```

When the kernel runs with IMA in enforcing mode, the keernel still allows to restore the backup with:

```
dd if=/root/efi-backup.img of=/dev/nvme0n1p1
```

Another relevant part for attestation is to use the integrity log file do recalculate the produced hashes. This means on one hand to recalculate the hash in the integrity log entry. On the other hand the chain of hashes should result into the value held in PCR 10. A small script shown in Listing 5.1 tries to do this calculation. It uses the debug register PCR 16 which is resetable without reboot and has the same initial value as the first 10 PCRs. When IMA is off, the log holds only one entry of the boot aggregate. Then the SHA1 value can be computed with that script. However, it was not possible to reproduce the value in PCR 10 value when enabling IMA. Our tests took into account that PCR and log file could be modified when loading programs to read these resources the first time. Loading the log at least two times eventually ends up in a stable log and PCR value (it does not change anymore even when reading the log another time). The value of PCR 10 was still not reproducible. Furthermore the documentation of calculating these values does not mention how the sha256 hash in PCR 10 is calculated. `tpm2_pcrextend` requires a sha256 hash as input for the corresponding PCR bank, but the integrity log only provides sha1 hashes. Hence, any verification procedures regarding the sha256 bank of PCR 10 are currently not implemented.

6 State of Work and Outlook

We are able to demonstrate in this contribution a working prototype for a Digidow sensor. Although only a minimal feature set is implemented, it shows a direction where developments can step forward. Furthermore, this setup is one of the first applications using a group signature scheme. In context of the Digidow project, a working DAA demo can be useful for other use cases as well. Consequently, this contribution is essential for further research and development in the Digidow project.

We summarize in this chapter the test results and discuss the limitations of the current prototype. Consequently, we show a number of topics worth to investigate in the future.

6.1 State of Work

The test results show that the concept can be implemented on a basic level. Trusted boot works fine with recent Linux kernels and bootloaders if required. It is furthermore enabled by default on many Linux distributions and works out of the box when disk encryption is not needed.

Similarly, the DAA group signature scheme is working with and without using a TPM. It provides all necessary features for a partly dynamic group. Adding a device requires a controlled environment, where the sensor's hardware and software is analyzed before it gets the ultimate trust of the group. This environment is usually only available when building and provisioning the device. A DAA member key can only be invalidated by adding it to the revocation list. Since the DAA member key should never leave the TPM, it seems to be hard to remove a corrupted sensor from the group.

Linux distributions provide more and more support for IMA in recent years. Ubuntu's first LTS release with IMA was 20.04 which we used for this work. The different predefined rulesets work as expected but result in a large set of entries in the integrity log.

The application on top of this setup simulates the start of a Digidow transaction by capturing and processing an image. This processed image data is then sent to the PIA acting here as a DAA verifier. We showed in the test results that this demo application works reliable but yet very inefficient. The demonstration just shows a part of the workload of a full Digidow transaction.

6.2 Limitations

The main contribution to the computation delay is processing the image data which is computed with a tensorflow application. The tested systems have no GPU, requiring them to compute the neural network on the CPU. Similarly, the image capturing part, taking about 1 s, is very inefficient. Capturing an image from a camera and saving it to disk should be doable in less than 0.1 s. Furthermore, the dedicated TPM is a quite trustworthy but slow cryptoprocessor. Signing and sending adds another 0.25 s to the transaction. When capturing biometric data in an efficient way, the interaction with the TPM will be a major contribution to the whole transaction time.

Besides the slow demo implementation, we found several other limitations while building the environment. In the context of using the TPM, we faced several problems that are not solved yet:

- One TPM 2.0 module is not usable since it has an outdated firmware and we were not able to update the TPM. The manufacturer directed us to the TPM vendor for firmware updates and the vendor seems to ignore any request in that context. Although it is possible to update the TPM, we were not able to get a recent firmware blob for this TPM.
- The TPM manufacturer, Infineon in our case, hosts the certificate chain on a website where only the domain name leads to the manufacturing company. The website does not provide further cryptographic trust. When the certificate chain is broken,

it may not be clear whether the user posseses a corrupt TPM or the website just provides bogus certificates.

- The chain of trust between TPM manufacturer and DAA member key is not complete in the current implementation since there is no cryptographic link between the certified endorsement key and the DAA member key. The member key is only located in the endorsement hierarchy.
- The documentation of the TPM software stack is not beginner friendly. On one hand, the TCG documentation is free to use and provides a narrow definition of a TPM. On the other hand, it is optimized to be machine readable and it is usually necessary to read parts of Arthur et al. *A Practical Guide to TPM 2.0* [2] to understand how the documentation is managed. Although complex documentation is usual for standards, an interested developer may need more time as expected for understanding the interfaces.
- Using the TPM 2.0 tools for low effort interaction with the TPM is relatively easy. Unfortunately, the parameters are incompatible over its major releases, breaking any scripts depending on that. These differences are not documented and not announced publicly, making it hard to update any shell scripts depending on that.

Similar to using TPMs, the integrity of the sensor's hardware and software are facing some essential problems:

- The integrity log is currently too large to send it within the attestation message to the DAA verifier. In the current setup, the integrity documentation of this log is key to generate trust between sensor and PIA. Furthermore, it is still not defined how the PIA can efficiently compute an answer to the question whether to trust the sensor or not, given the attestation message and the integrity log.
- IMA is only able to provide proofs about static files and resources. Using a program which dynamically loads code from remote resources is an efficient way to circumvent any integrity restrictions. Since the test setup is using the network interface, this is a relevant attack vector against system integrity and trust.
- The features of IMA are poorly documented and some of the tools only support TPM 1.2 which is already outdated. Although the support of Linux distributions increases, it is not clearly stated which release supports IMA and which does not.

6.3 Future Work

Given the limitations in the current setup, we provide some thoughts which might be worth implementing on top of this contribution. Depending on the usage model, it might be worth extending the DAA scheme into a full dynamic group signature scheme. Camenisch et al. [3] discuss how the existing scheme could be extended to support the features of dynamic groups by implementing signature based revocation. Although their concept seems to work, there is currently no widespread public library available in the public.

We discussed in the previous section a missing link in the chain of trust of the TPM. This gap could be closed by certifying that the DAA membership key is in the endorsement hierarchy. Since the EK has no signing property, the certification procedure uses the attestation key (AK) and the attestation identity key (AIK) to prove this link. The theoretical concept is described by Arthur et al. at pages 109 ff [2]. Eric Chieng [8] wrote on his blogpost a practical approach to impelement this certification. This solution seems to close the missing link in the chain of trust.

Verifying the integrity of the sensor is still not solved. There are several approaches which reduce the size of the log. Several approaches should be investigated to mitigate this problem:

- Customize the IMA rules to omit unnecessary measurements where possible.
- Avoid writing and reading files during a Digidow transaction. Therefore the log should remain constant in size or at least independent of the number of transactions. An approach for that would be to transfer the data between the different steps via pipes.
- Sign the hashes of static files like binaries or configuration files with the kernel's extended verification module which works on top of IMA. When files with a valid signature are accessed, it might not be necessary to add them into the integrity log.
- Make the root partition read only where applicable. Every file read from this partition can be excluded from the integrity log. A Ramdisk could then be used to hold all working files of the sensor. This setup might satisfy similar goals compared to the previous approach, but the update procedure might be easier.

- Include the firmware of peripherals. In the current setup, the integrity log takes only libraries and drivers into account. However the peripherals like cameras or fingerprint sensors usually have a firmware onboard which should be measured similarly to BIOS or option ROMs. Unfortunately we could not test a sensor which allows to extract the firmware that is acually running on the hardware. If such hardware is available, adding the hashes into the system's attestation message should be possible.

As stated in the previous section, the integrity log only measures static content. Auditd investigates a program during execution and is able to log every system call during execution, which seems to fill the gap of dynamic execution. It might be necessary to configure the daemon and to integrate the auditd execution log of the sensor's transaction part to the attestation message.

The results in Subsection 5.3.3 show that memory safety is not given in every part of the demo. This is, however, a key contribution to the system's integrity and hence the trust in it. We recommend therefore to omit any use of scripting environments and to use memory-safe programming languages like Java or Rust. According to this, the programs should be precompiled and installed only as binaries together with the required external resources. The productive setup should not have any build tools installed.

Although we use an Ubuntu 20.04 server minimal setup, the majority of installed programs and services are not necessary for the Digidow sensor. Consequently, reducing the system to a bare minimum reduces the complexity of auditing and certifying the setup as well as the resulting integrity log and attestation information.

The demonstration setup is implemented in a way that the image capturing task should be easily replaceable with another sensor setup. Therefore, the sensor should support a variety of biometric features. It might be required to use hardware or GPU acceleration to process these features quick enough. Ideally the transaction delay is independent of which sensor type is used.

Many of the measures above make the computation and hence the transaction time more efficient. It is therefore necessary to define timing constraints to see whether further measures need to be implemented.

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. 26).

[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 pp. 10, 11, 71, 72).

[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 pp. 10, 19, 72).

[4] Jan Camenisch, Manu Drijvers, and Anja Lehmann. "Anonymous Attestation Using the Strong Diffie Hellman Assumption Revisited". In: *Trust and Trustworthy Computing*. Ed. by Michael Franz and Panos Papadimitratos. Cham: Springer International Publishing, 2016, pp. 1–20. DOI: 10.1007/978-3-319-45572-3_1 (cit. on p. 10).

[5] 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. 10, 19, 20, 21, 22, 23, 26).

[6] 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 pp. 21, 22).

[7] Jan Camenisch and Markus Stadler. "Efficient Group Signature Schemes for Large Groups". In: *CRYPTO '97* 1296 (Jan. 1997) (cit. on p. 20).

[8] Eric Chieng. *The Trusted Platform Module Key Hierarchy*. 2021. URL: https://ericchiang.github.io/post/tpm-keys/?utm_campaign=Go%20Full-Stack&utm_medium=email&utm_source=Revue%20newsletter#credential-activation (visited on 09/29/2021) (cit. on p. 72).

[9] TPM2 Software Community. *TPM2 Tools*. 2020. URL: https://github.com/tpm2-software/tpm2-tools (visited on 05/15/2020) (cit. on p. 9).

[10] Intel Corp. *Intel Trusted Execution Technology: White paper*. 2012. URL: http://www.intel.de/content/www/de/de/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html (visited on 12/19/2021) (cit. on p. 15).

[11] 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. 8).

[12] 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. 14, 40).

[13] Dann Frazier. *Secure Boot*. 2020. URL: https://wiki.ubuntu.com/UEFI/SecureBoot (visited on 07/24/2021) (cit. on p. 15).

[14] Matthew Garrett. *Why UEFI secure boot is difficult for Linux*. 2012. URL: https://mjg59.dreamwidth.org/9844.html (visited on 12/16/2021) (cit. on p. 15).

[15] 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. 12, 13, 14).

[16] Trusted Computing Group. *The TPM Library Specification*. 2019. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (visited on 05/16/2020) (cit. on p. 7).

[17] Gentoo Foundation Inc. *Integrity Measurement Architecture*. 2019. URL: https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture (visited on 07/07/2021) (cit. on p. 46).

[18] Gentoo Foundation Inc. *Integrity Measurement Architecture/Recipes*. 2019. URL: https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture/Recipes (visited on 07/07/2021) (cit. on p. 19).

[19] Xaptum Inc. *Source repository for the ECDAA C Library*. 2021. URL: https://github.com/xaptum/ecdaa (visited on 07/07/2021) (cit. on p. 39).

[20] Michael Kerrisk. *Xattr Man Page*. 2020. URL: https://man7.org/linux/man-pages/man7/xattr.7.html (visited on 09/19/2021) (cit. on p. 58).

[21] René Mayrhofer, Michael Roland, and Tobias Höller. *Poster: Towards an Architecture for Private Digital Authentication in the Physical World*. Network and Distributed System Security Symposium (NDSS Symposium 2020), Posters. Feb. 2020. URL: https://www.mroland.at/uploads/2020/02/Mayrhofer_2020_NDSS2020posters_Digidow.pdf (cit. on p. 3).

[22] Microsoft. *Secure Boot Overview*. 2014. URL: https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-8.1-and-8/hh824987(v=win.10) (visited on 07/24/2021) (cit. on p. 14).

[23] 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. 8).

[24] 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. 8).

[25] 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. 40).

[26] 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. 9, 40).

[27] 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. 16).

[28] 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. 19).

[29] Huzaifa Sidhpurwala. *How to use the Linux kernel's Integrity Measurement Architecture*. 2020. URL: https://www.redhat.com/en/blog/how-use-linux-kernels-integrity-measurement-architecture (visited on 07/09/2021) (cit. on p. 46).

[30] 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. 40).

Fingerprints of TPM certificates

This table shows the certificate fingerprints of the TPMs and their corresponding certificate chain.

Table 1: ECC certificate chain of used TPMs

Certificate	*SHA256 checksum*
ECC root CA	`CF:EB:02:FE:CD:55:AD:7A:73:C6:E1:D1:19:85:D4:C4:7D:EE:24:8A:B6:3D:CB:66:09:1A:24:89:66:04:43:C3`
ECC intermediate CA 036	`2C:BF:6D:6C:39:94:47:0A:EC:D2:B1:F8:7F:5D:21:11:08:D1:E0:A4:A8:1E:D9:CB:A2:6B:D2:98:96:5D:E0:1C`
ECC intermediate CA 011	`B5:59:A8:4E:63:32:A7:B4:E3:2B:4D:37:39:E3:72:E3:C5:17:BA:5A:4C:CF:FE:E1:DA:AF:80:BC:64:16:28:EE`
ECC System 1 TPM	`D8:C3:21:3E:BB:C2:CB:96:EA:14:2F:D5:57:61:79:04:BB:DE:8A:F9:21:2A:11:7D:4B:2E:FC:64:BD:35:C1:6E`
ECC System 2 TPM	`FE:1B:EF:42:8B:68:35:D3:FC:5A:13:A0:AE:12:19:BA:A1:60:D6:59:38:1D:79:8E:76:50:48:BE:5C:BD:83:A5`
ECC System 3 TPM	`92:29:68:6D:50:EE:34:08:30:DF:E7:30:D8:F3:C0:C7:13:3C:DF:F0:6D:9E:2B:2E:0F:54:76:AE:B8:D6:1A:DA`
RSA root CA	`89:9E:35:47:4C:98:07:EB:4C:7F:2F:7A:12:DA:00:28:FB:25:0C:D0:21:54:D0:00:9F:CA:7D:9C:66:57:4F:3B`
RSA intermediate CA 036	`21:6C:47:D2:77:FC:40:CE:90:F0:86:83:21:CB:5E:F5:91:FC:1D:D8:D0:E4:FD:A1:A2:C8:3C:17:BE:01:B0:7E`
RSA intermediate CA 011	`A8:33:79:F9:2A:34:1B:EB:61:B6:7F:03:50:66:44:94:0F:EB:4B:85:EA:50:4A:9D:22:13:BC:A5:2C:88:9F:89`
RSA System 1 TPM	`F1:C7:6A:00:CF:2B:63:4D:38:C0:2E:73:3C:84:BF:30:5C:C2:D3:61:DF:34:D8:95:BB:F1:0F:FB:6B:0C:79:E2`
RSA System 2 TPM	`CB:1F:7D:20:FE:B2:11:C4:2B:20:6B:4F:66:A6:14:1A:37:94:5F:85:93:6D:2E:92:85:57:BF:3A:BF:9E:DA:DD`
RSA System 3 TPM	`BF:0B:4E:77:80:18:86:9A:EF:09:06:96:E2:4D:72:A3:47:B6:E3:8F:AA:F9:9C:2E:C0:13:AB:70:E3:E4:5D:93`