JOHANNES KEPLER
UNIVERSITY LINZ

Author
Michael Preisach
1155264

Submission
Institute for Networks and Security

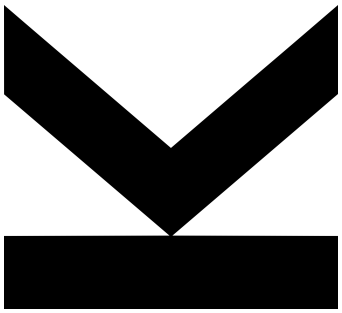
First Supervisor
Univ.-Prof. DI Dr. **René Mayrhofer**

Second Supervisor
DI **Tobias Höller**

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

July 2021

Project Digidow: Biometric Sensor



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

JOHANNES KEPLER
UNIVERSITY LINZ

Statutory Declaration

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

This printed thesis is identical with the electronic version submitted.

Linz, July 2021

JOHANNES KEPLER
UNIVERSITY LINZ

Abstract

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

Kurzfassung

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

JOHANNES KEPLER
UNIVERSITY LINZ

Contents

1 Introduction **1**
- 1.1 Trust . . . 2
- 1.2 Project DigiDow . . . 2
- 1.3 Our Contribution: Deriving Trust from the Biometric Sensor . . . 4
- 1.4 Description of structure . . . 6

2 Related Work **7**

3 Background **9**
- 3.1 Trusted Platform Module (TPM) . . . 9
  - 3.1.1 Using the TPM . . . 11
  - 3.1.2 The Hardware . . . 11
  - 3.1.3 TPM Key Hierarchies . . . 12
  - 3.1.4 Endorsement Key . . . 12
- 3.2 Trusted Boot . . . 13
  - 3.2.1 Platform Configuration Register . . . 13
  - 3.2.2 Static Root of Trust for Measurement . . . 15
  - 3.2.3 Platform handover to OS . . . 15
- 3.3 Integrity Measurement Architecture . . . 16
  - 3.3.1 Integrity Log . . . 17
  - 3.3.2 IMA Appraisal . . . 18
  - 3.3.3 IMA Policies . . . 18
  - 3.3.4 IMA extensions . . . 19
- 3.4 Direct Anonymous Attestation . . . 19
  - 3.4.1 Mathematical Foundations . . . 19
  - 3.4.2 DAA Protocol on LRSW Assumption . . . 22

4 Concept **26**
- 4.1 Definition of the Biometric Sensor . . . 26
- 4.2 Attack Vectors and Threat Model . . . 27
  - 4.2.1 Threat Model . . . 28
- 4.3 Prototype Concept . . . 29
  - 4.3.1 Integrity and Trust up to the Kernel . . . 29
  - 4.3.2 Integrity and Trust on OS Level . . . 32

4.3.3 Prove Trust with DAA 32
4.4 Trust and Security 35
4.5 Systems of Trust 35
4.5.1 Secure Boot, TXT, ... 35
4.5.2 TPM1.2 36
4.6 Trusted Boot 36
4.7 Integrity Measurements 36
4.8 Verify Trust with DAA 36
4.8.1 DAA History 36

5 Implementation **38**
5.1 Hardware Setup 39
5.2 Operating System 39
5.3 Trusted Boot 40
5.4 Integrity Measurement Architecture 42
5.4.1 Handling external hardware 42
5.5 Interaction with TPM2 42
5.6 Direct Anonymous Attestation 42

6 Conclusion and Outlook **43**
6.1 Testing 43
6.2 Limitations 43
6.3 Future Work 43
6.4 Outlook 43

Appendix **47**

TCP/IP Wrapper for the Xaptum ECDAA Protocol **50**
1 Common source files for all DAA parties 50
2 Source files for the DAA Issuer 57
3 Source files for the DAA Member 63
4 Source files for the DAA Member with TPM support 69
5 Source files for the DAA Verifier 87

JOHANNES KEPLER
UNIVERSITY LINZ

List of Figures

1.1 Overview of the DigiDow authentication process 3

3.1 TPM Certification 12
3.2 Integrity log entry 17

4.1 Extending trust from the Roots of Trust up to the Kernel 31
4.2 DAA Attestation procedure 33
4.3 Overview of the Chain of Trust of the BS 34

5.1 Prototype schematic 38

List of Tables

3.1 Usage of PCRs during an UEFI trusted boot process 14

5.1 Systems used for demonstration prototype 39
5.2 Disk layout of the BS prototype 40
5.3 Memory layout of the Unified Kernel EFI file 41

JOHANNES KEPLER
UNIVERSITY LINZ

1 Introduction

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

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

The downside of all these digital services is that using these services generate a lot of data. Besides of the intended exchange of information, many of the services try to extract metadata as well. Metadata answers some of the following questions. Which IP is connected? 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 and provide an API where an *Operating System* (OS) can connect to. Some companies deliver complete closed source

devices with internet connection. In this case a user has no chance to detect what the device is doing in this very moment.

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 sensible data is handled properly for only the designated usage. One may argue that a password can easily be changed when revealed to the public. Unfortunately, this does not apply to a fingerprint since a human usually has only ten of them during lifetime.

1.1 Trust

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

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

1.2 Project DigiDow

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

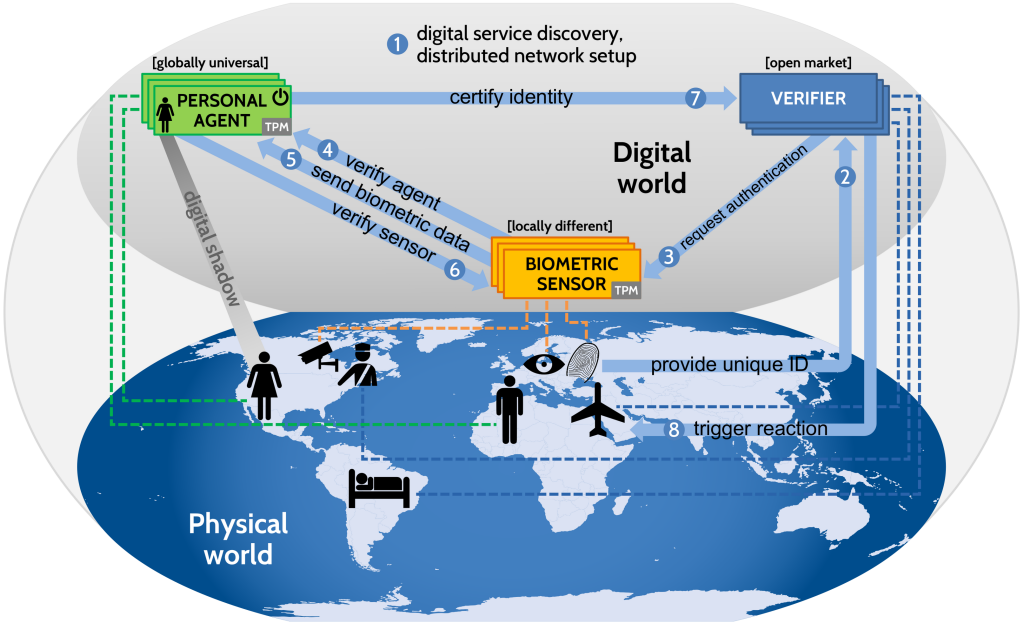


Figure 1.1: Overview of the DigiDow authentication process

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

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

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

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

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

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

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

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

1.3 Our Contribution: Deriving Trust from the Biometric Sensor

The DigiDow network is designed to preserve privacy and 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 the *Trusted Platform Module* (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 data set to continue with the authentication process. This data set

[1] https://digidow.eu

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

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

- Any network discovery (Step 1 in Figure 1.1) is omitted. BS and PIA are assumed to be reachable directly via TCP/IP
- We look into a protocol which proofs trustworthiness from BS to PIA. Any further proofs necessary for DigiDow's Verifier are also not focused in this thesis.

- The sensible data sets will be transmitted in cleartext between BS and PIA. It is considered easy to provide an additional layer of encryption for transportation. However this should be considered in the DigiDow network protocol design. This thesis focuses only on the trust part between BS and PIA.
- Any built system is considered secure on a hardware level. Any threats which are attacking the system without changing any running software on the system may be not detected. This includes USB wire tapping or debug interfaces within the system revealing sensible information.

1.4 Description of structure

In Chapter chapter 2 we will outline a variety of projects which do not contribute to this thesis. There is, however, scientific work that is used as scientific background to this thesis as described in chapter 3. This includes especially the theoretical foundations of the network protocol. Together with that, we will introduce our theoretical solution for the previously stated problems in chapter 4. Chapter 5 introduces then a working implementation with all necessary parts for a working prototype. Finally we will present the results and limitations in chapter 6 and give an overview of future work.

2 Related Work

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

Schear el. al. 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[15]. The concept of a well known virtual environment does, however, not apply to our contribution. Furthermore, the the system should be self contained as good as possible and it should be possible to get information about the system via anonymous attestation.

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

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

[1] /urlhttps://fidoalliance.org/specs/fido-v2.0-id-20180227/fido-ecdaa-algorithm-v2.0-id-20180227.html

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

3 Background

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

3.1 Trusted Platform Module (TPM)

The *Trusted Platform Module* (TPM) is a small coprocessor that introduces a variety of cryptographic features to the platform. This module is part of a standard developed by the Trusted Computing Group (TCG), which current revision is 2.0[17].

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 switch to another TPM.
- *Mounted device.* The dedicated chip is directly mounted on the target mainboard. Therefore any hardware modification is impossible. However most PC platforms provide BIOS features to control the TPM.
- *Firmware TPM (fTPM).* This variant was introduced with the TPM2.0 Revision. Firmware means in this context an extension of the CPU instruction set which provides the features of a TPM. Both Intel and AMD provide this extension for their

platforms for several years now. When activating this feature on BIOS level, all features of Trusted Computing are available to the user.

- *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 operating system. Features like Trusted Boot or in hardware persisted keys are not available.

Even the dedicated 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 Microcode 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[1]. 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 [12], 1048 bit keys required in the worst case scenario 3 CPU months and 2048 bit keys needed 100 CPU years. Infineon was able to fix that vulnerability per firmware update for all affected TPMs.

- *CVE-2019-16863*: This vulnerability is also known as *TPM fail* ([11]) and shows how to get an Elliptic Curve private key via timing and lattice attacks. The authors found TPMs from STMicorelectronics vulnerable, as well as Intel's fTPM implementation. Infineon TPM show also some non-expected behaviour, but this could not be used for data exfiltration. STMicro provided an update like Insineon did for the TPMs. Intel's fTPM lives in the Management Engine, which requires a BIOS update from the mainboard manufacturer to solve the issue.

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=tpm, last accessed on 15.05.2021

3.1.1 Using the TPM

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

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

The reference implementation of these APIs is published at Github[8] and is still under development. At the point of writing stable interfaces are available for C and C++, but other languages like Rust, Java, C# and others 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[14].

3.1.2 The Hardware

The TCG achieved with the previous mentioned software layers independence of the underlying hardware. Hence, TCG provided different flavors of of the TPM

TCG defined with the TPM2.0 standard 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 provided just the minimal set of algorithms and also the minimal amount of memory.

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

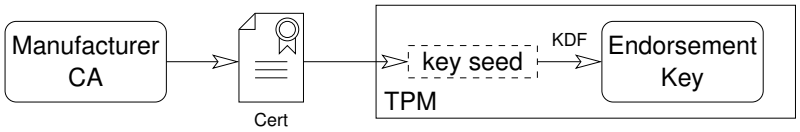


Figure 3.1: The manufacturer certifies every TPM it produces

3.1.3 TPM Key Hierarchies

A TPM comes with four different key hierarchies. These hierarchies fulfill different tasks and are therefore used in different use cases on the whole platform. Will Arthur et. al[1] provide a more detailed description on how the hierarchies workt together.

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

Each of the persistent hiearchies represent an 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 on key generation described below is similar to all three persistent hierarchies.

3.1.4 Endorsement Key

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

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

- Attestation Identity Key
- Key management

3.2 Trusted Boot

A boot process of modern platforms consists of several steps until the OS 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. In this common boot procedure exists no source of trust and hence no check for integrity or intended execution.

3.2.1 Platform Configuration Register

The *Trusted Computing Group* (TCG) introduced in 2004 their first standard for a new Trusted Computing Module (TPM). 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). *Measuring* means in this context a simple cryptographic extension function which works described in formula 3.1

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

The function of $||$ represents a concatenation of two binary strings and the hash function is either SHA1 or SHA256 hash. 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 in addition 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 change. 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 of TPM, version 1.2. For the recent TPM2.0 standard, the process was only extended with the support for the newer SHA256 standard.

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 be still 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*[10] and shown in table 3.1. Especially those PCRs involved in the boot process must only be reset according to a platform reset. During booting and running the system these registers can only be *extended* with new measurements.

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

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

When TCG introduced Trusted Boot in 2004, UEFI was not yet available for the ordinary PC platform. Consequently, TCG standardized the roles of every PCR only for the BIOS

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

3.2.2 Static Root of Trust for Measurement

The standard furthermore 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 by the binary representation of the firmware itself. This procedure is described in the TCG standard and the platform user has to *trust* the manufacturer, that it is performed as expected. 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[10]. As the manufacturer of the motherboards 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 furthermore must measure all platform initialization code like embedded drivers, host platform firmware, etc. as they are provided as part of the PC motherboard. If these measurements cannot be performed, the chain of trust is broken and consequently the platform cannot be trusted. One may see a zeroed PCR[0] or a value representing a hashed string of zeros as a strong indicator of a broken chain of trust.

3.2.3 Platform handover to OS

The BIOS or UEFI performs the next measurements according to table 3.1 until PCRs 1–7 are written accordingly. Before any further measurements are done, the control of the platform is handed over to the first part of the OS, which is usually the bootloader either in the Master Boot Record 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 takes control.

The Bootloader has then to continue the chain of trust by measuring the Kernel and the corresponding command line parameters into the next PCRs. The support and the way of how the measurements are done is not standardized. GRUB, for example, measures all executed grub commands, the kernel command line and the module command line into PCR 8, whereas any file read by GRUB will be measured into PCR 9[9].

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

3.3 Integrity Measurement Architecture

The *Integrity Measurement Architecture* (IMA) is a Linux kernel extension to extend the chain of trust to the running application. IMA is officialy supported by RedHat and Ubuntu and there exists documentation to enable IMA on Gentoo as well. Other OS providers may not use a kernel with the required compile flags and / or lack of supporting software outside the kernel. The IMA project page describes the required Kernel features for full support in their documentation[2].

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

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

[2] https://sourceforge.net/p/linux-ima/wiki/Home/#configuring-the-kernel, last visited on March 30, 2021

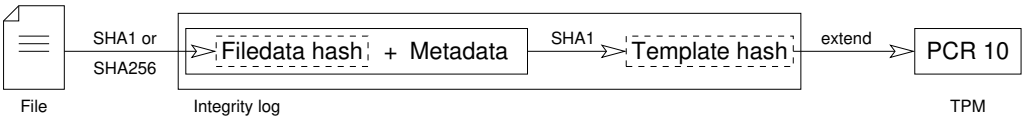


Figure 3.2: Overview of generating an entry in the integrity log

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.

3.3.1 Integrity Log

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

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

IMA knows three different file formats, where two of them can be used in recent applications. The only difference between these formats lie 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 writes also signatures of files into the log and includes them for calculating the template hash.

The older template `ima` uses only SHA1 and is fully replaceyble with the `ima-ng` template. Therefore, it should not be used for newer applications. **ToDo!**boot aggregate beschreiben

3.3.2 IMA Appraisal

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

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

3.3.3 IMA Policies

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

- `tcb`: All files owned by root will be measured.
- `appraise_tcb`: All executables which are run, all files mapped in memory for execution, all loaded Kernel modules and all files opened for read by root will be measured by IMA.

- `secure_boot`: All loaded modules, firmwares, executed Kernels and IMA policies are checked. Therefore these resources need to have a provable signature to pass the check. The corresponding public key must be provided by the system manufacturere 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 at the Gentoo Wiki[3]. It is, for example, useful to exclude constantly changing log files from being measured to reduce useless in the measurement log.

3.3.4 IMA extensions

3.4 Direct Anonymous Attestation

Direct Anonymous Attestation (DAA) is a cryptographic scheme which makes use of the functions provided by the TPM. DAA implements the concept of group signatures, where multiple secret keys can create a corresponding signature. These signatures can be verified with a single public key, when these 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 the discussion in [4] and [3] almost all schemes were proven insecure, since many of them had bugs in the protocol or allowed trivial public / secret key pairs. This includes also the impementation of DAA im the TPM1.2 standard.

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

3.4.1 Mathematical Foundations

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

[3] https://wiki.gentoo.org/wiki/Integrity_Measurement_Architecture/Recipes

Discrete Logarithm Problem

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

Signature Proof of Knowledge (SPK)

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

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

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

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

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

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

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

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

The verification holds since

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

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

Bilinear Maps

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

- *Bilinearity.* For all $P \in \mathbb{G}_1, Q \in \mathbb{G}_2$, for all $a, b \in \mathbb{Z} : e(P^a, Q^b) = e(P, Q)^{ab}$.
- *Non-degeneracy.* For all generators $g_1 \in \mathbb{G}_1, g2 \in \mathbb{G}_2 : e(g_1, g_2)$ generates $\mathbb{G}_T$.
- *Efficiency.* There exists an efficient algorithm that outputs the bilinear group $(q, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g_1, g_2)$ and an efficient algorithm for computing e.

Camenisch-Lysyanskaya Signature Scheme

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

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

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

3.4.2 DAA Protocol on LRSW Assumption

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

- *Issuer* $\mathcal{I}$. The issuer maintains a group and has evidence of hosts that are members in this group. This role equals the group manager of Bellare's generic definition.
- *Host* $\mathcal{H}$. The Host creates a platform with the corresponding TPM $\mathcal{M}$. Membership of groups are maintained by the TPM. Compared to Bellare et al., the role of a member is split into two cooperating parties, the key owner (TPM, passive) and the message author (Host, active).
- *Verifier* $\mathcal{V}$. A verifier can check, whether a Host with its TPM is in a group or not. Besides the group membership, no additional information is provided.

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

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

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

  - registers the public key (X, Y, π) at $\mathcal{F}_{ca}$ and stores the secret key,
  - 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. $\mathcal{I}$ upon receiving JOIN from $\mathcal{H}_j$, chooses a fresh nonce $n \leftarrow \{0,1\}^\tau$ and sends it back to $\mathcal{H}_j$.
  3. $\mathcal{H}_j$ upon receiving n from $\mathcal{I}$, 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. $\mathcal{I}$ upon input JOINPROCEED(Q, π_1) creates the CL-credential:
     - Verify that π_1 is correct.
     - Add $\mathcal{M}_i$ to $\mathcal{L}$.
     - Choose $r \xleftarrow{\$} \mathbb{Z}_q$ and compute $a \leftarrow g_1^r$, $b \leftarrow a^y$, $c \leftarrow a^x \cdot Q^{rxy}$, $d \leftarrow Q^{ry}$.
     - Create the prove $\pi_2 \xleftarrow{\$} SPK\{(t) : b = g_1^t \wedge d = Q^t\}$.
     - Send APPEND(a, b, c, d, π_2) to $\mathcal{H}_j$
  7. $\mathcal{H}_j$ upon receiving APPEND(a, b, c, d, π_2)
     - verifies, that $a \neq 1$, $e(a, Y) = e(b, g_2)$ and $e(c, g_2) = e(a \cdot d, X)$.
     - forwards (b, d, π_2) to $\mathcal{M}_i$.

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

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

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

  1. $\mathcal{H}_j$ upon input $\mathsf{SIGN}(m, \mathtt{bsn})$ 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. $\mathcal{M}_i$ upon receiving $(m, \mathtt{bsn}, r)$

     - 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.

1. $\mathcal{V}$ upon input $\mathsf{VERIFY}(m, \mathtt{bsn}, \sigma)$
   - parses $\sigma \leftarrow (a, b, c, d, \pi, \mathtt{nym})$,
   - verifies π with respect to $(m, \mathtt{bsn})$ and $\mathtt{nym}$ 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.

  1. $\mathcal{V}$ on input $\mathsf{LINK}(\sigma, m, \sigma', m', bsn)$ verifies the signatures and compares the pseudonyms contained in σ, σ':
     - Check, that $\mathtt{bsn} \neq \perp$ and that both signatures σ, σ' are valid.
     - Parse the signatures $\sigma \leftarrow (a, b, c, d, \pi, \mathtt{nym})$, $\sigma' \leftarrow (a', b', c', d', \pi', \mathtt{nym}')$.
     - If $\mathtt{nym} = \mathtt{nym}'$, set $f \leftarrow 1$, otherwise $f \leftarrow 0$.
     - Output $\mathsf{LINK}(f)$.

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

- *Non-frameability*: No one can create signatures that the Platform never signed, but that link to messages signed from that platform.
- *Correctness of link*: Two signatures will link when the honest platform signs it with the same basename.
- *Symmetry of Link*: It does not matter in which order the linked signatures will be proven. The link algorithm will always output the same result.

4 Concept

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

4.1 Definition of the Biometric Sensor

The BS itself is defined as edge device within the DigiDow network. According to the schema shown in Figure 1.1, the BS will be placed in a public area (e.g. a checkpoint in an airport or as access control system at a building) to interact directly with the DigiDow users. There, the BS is the gateway 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 step.

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

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

4.2 Attack Vectors and Threat Model

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

In addition to autonomy, the BS should also ensure proper handling of received and generated data. The recorded dataset from a sensor is *sensitive data* due to its ability to identify an individual. Due to its narrow definition, it is affordable to protect sensitive data. Besides that, *metadata* is information generated during the whole transaction phase. Timestamps and host information are metadata as well as connection lists, hash sums 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 notwork providers are able to combine lots of metadata to traces of individuals. Eventually an action of those traced individuals might unveil their identity. Consequently, a central goal of DigiDow is to minimize the amount to minimize the risk of traces.

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

4.2.1 Threat Model

To fulfill the Sensor's use case, we need to consider the following attack vectors.

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

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

4.3 Prototype Concept

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

4.3.1 Integrity and Trust up to the Kernel

We decided to use the PC platform as hardware base for the prototype. There are lots of different form factors available you can extend the system with a broad variety of sensors. Furthermore the TPM support is implemented to support integrity analysis on the system. 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 of TPM, the amount of available software and the ease of installation is better on the PC platform.

As described in section 3.1, the TPM functions can be delivered in three different flavors: As dedicated or mounted device and as part of the processor's firmware. The fTPM is part of a large proprietary environment from AMD or Intel which which introduces, besides implementation flaws, additional attack surfaces for the TPM. Hence we will use dedicated TPM chips on the platform, which are pluggable, to gain most control over the functionality.

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

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

To further reduce the attack surface, the prototype will not use a bootloader like GRUB. Instead, the Kernel should be 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's firmware. The tiny program just measures all parts of the firmware and feeds the TPM with the results. However, the program is maintained by the mainboard manufacturer and the source is not available to the public. We have to trust that this piece of software is working correctly,

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

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

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

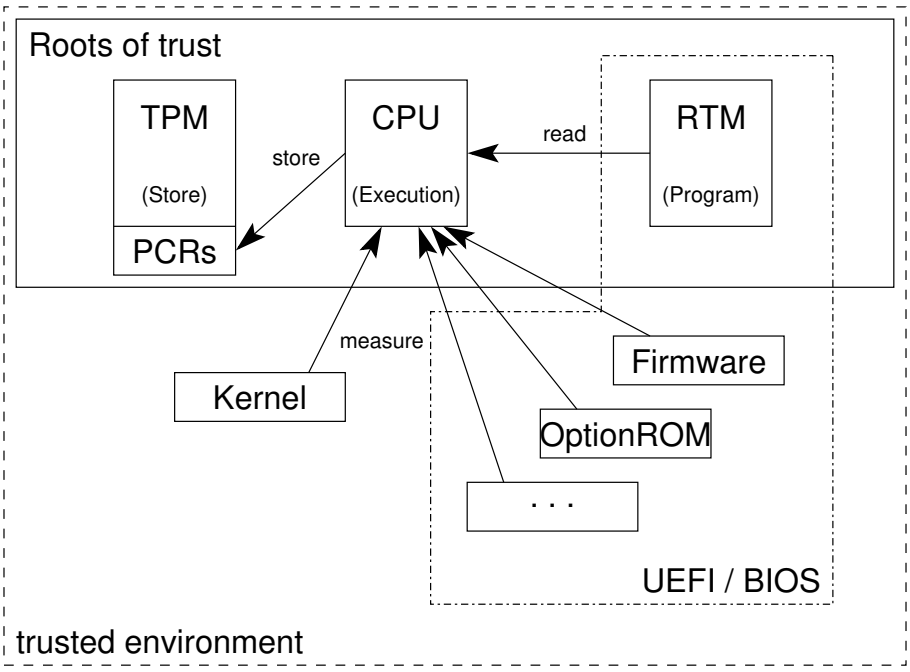


Figure 4.1: Extending trust from the Roots of Trust up to the Kernel

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

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

When the roots of trust are honest, the trusted environment can be constructed during booting the platform with the PCR measurements. We get then a system, where all active parts in the booting process are trusted up to the Linux kernel with its extensions and execution parameters.

4.3.2 Integrity and Trust on OS Level

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

4.3.3 Prove Trust with DAA

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

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

1. *Network Discovery*: The PIA is already identified over the DigiDow network and there exists a bidirecional channel between BS and PIA

2. *Secure Communication Channel*: The bidirectional channel is assumed to be hardened against wire tapping, metadata extraction and tampering. The prototype will take no further action to encrypt any payload besides the cryptographic features that come along with DAA itself.

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

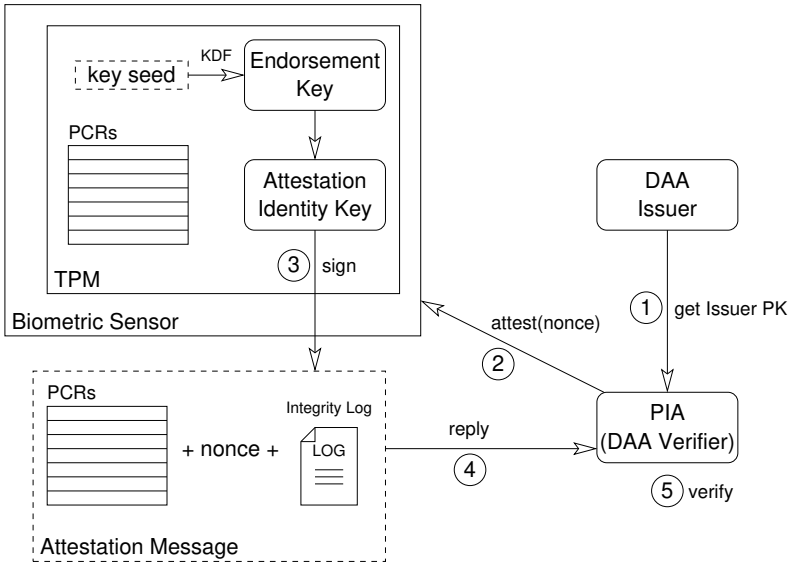


Figure 4.2: The DAA attestation process requires 5 steps. The PIA may trust the Biometric Sensor afterwards.

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

1. The PIA gets once and independently of any transaction the public key of the BS group.
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 SK.
4. The Attestation Message will be sent back to the PIA.
5. The PIA checks the signature of the message, checks the entries of the Integrity log against known values, and proves the PCR values accordingly.

Figure 4.3 shows how the sources of trust will be represented in the final attestation message.

- DONE Definition of sensitive data / privacy / metadata

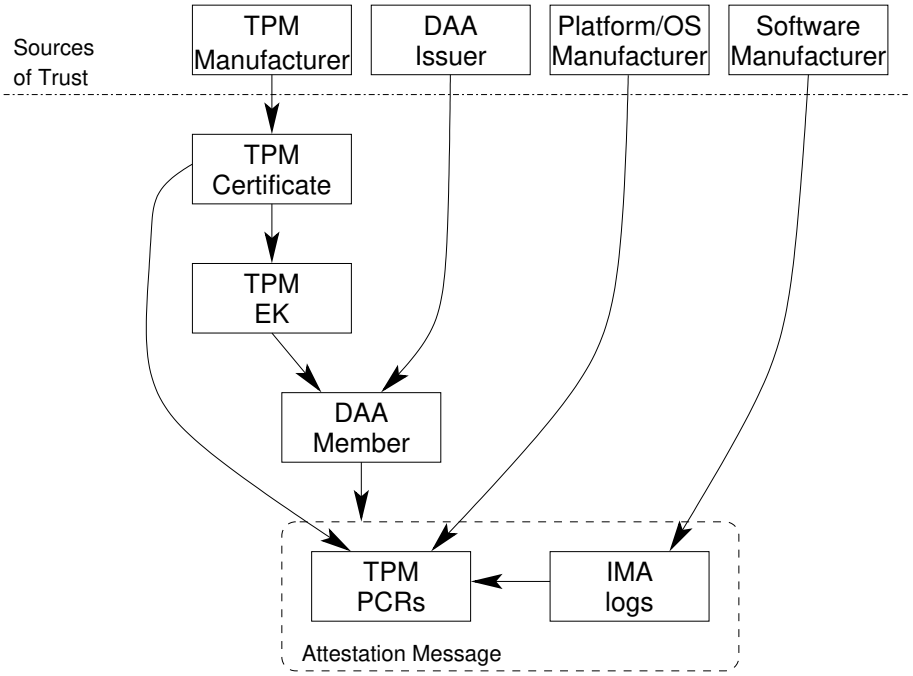


Figure 4.3: Overview of the Chain of Trust of the BS

- This version of BS is not owned by the user, there is no personal data in the System
- Rogue Personal Identity Agent (PIA)
- Metadata Extraction
- Attribute extraction
- Sensor Data Modification/manipulation
- Wiretap between Sensor and System (USB or network)
- Physical Manipulation of the BS-System
- Network - Retransmission of sensor data of a rogue BS
- Network - Blocking Data transmission of a rogue BS
- Rogue BS Sensor Data aggregation
- Rogue BS Sensor data modification before transmission

4.4 Trust and Security

Trust is an essential term in this thesis. In the world of IT security, the term *trusted computing* defines a secured environment where special or confidential computing jobs are dispatched. This environment or product usually meets the following requirements

- *Minimalization.* The number of features and hence the complexity must be as low as possible.
- *Sound definitions.* Every function should be well defined. There should be no margin for interpretation left. Security Engineers should be involved in the development.
- *Complete testing.* Testing for trusted computing includes a threat analysis and exhaustive testing if possible.

Since software and hardware testing is never complete, it is hard to find a good balance between feature set and testing completeness.

However trust in IT is not equal to security. It defines a subset of IT security where this small well defined environment is embedded in a larger system which is usually untrusted. Claiming a system *secure* spans the constraints of trust over the complete system, which is not affordable for commodity computers these days. However it is possible to use the trusted environment to get some guarantees on the untrusted parts of a system as well In Chapter 3 we will show how trust will be extended in a commodity PC.

Differentiation between trust and security — and the problem that not everyone is using that right.

4.5 Systems of Trust

All trust systems are built on the standards of Trusted Computing Group.

4.5.1 Secure Boot, TXT, . . .

Trusted Boot is not the same as Secure Boot. Explain the difference

4.5.2 TPM1.2

Initial Version of the crypto-coprocessor, successfully spread into many systems, but hardly any integration in Trust/security Software

4.6 Trusted Boot

4.7 Integrity Measurements

As described in the previous section, when the boot process is eventually finished, the OS is then responsible for extending the chain of trust. Given a valid trusted boot procedure, the binary representation of the kernel is already measured. Therefore the Kernel itself has the responsibility to keep track of everything happening on the platform from the OS point of view.

Soon after the first TPM standard was published, the *Integrity Measurement Architecture* (IMA) for the Linux Kernel was introduced. Since Kernel 3.7 it is possible to use all IMA features, when the compiler options of the Kernel are set correspondingly.

IMA

Extend the Chain of Trust beyond the boot process. The Kernel can measure many different types of Resources. What is a useful set of measurements

4.8 Verify Trust with DAA

4.8.1 DAA History

Direct Anonymous Attestation (DAA) is a cryptographic protocol, which aims to provide evidence that a device is a honest member of a group without providing any identification information. Brickell, Camenisch and Chen[2] introduce DAA and implement the protocol for the TPM 1.2 standard. However it supports only RSA and has limitations in verifying attestation signatures. Hence, DAA is not used with the TPM 1.2 standard.

Since the DAA protocol is quite complex, it is difficult to provide a sound security model for DAA and formally prove the security properties of it. Chen, Morissey and Smart[7] add linkability to the protocol. Their approach for a formal proof is not correct, since a trivial key can be used for pass verification[4]

Camenisch, Drijvers and Lehmann[4] developed a DAA scheme for the new TPM 2.0 standard. It supports linkability and the proves for security and correctness still hold. Furthermore, RSA and ECC cryptography is supported which makes it practicable for a wider variety of use cases. However, Camenisch et al. proposed a fix in the TPM 2.0 API to guarantee all requirements necessary for DAA. Xaptum implemented this DAA-variant including the fixes in the TPM API. The implementation will be discussed in Chapter 4.

Analyzing the security and integrity of this scheme would exceed the scope of this thesis. Hence this thesis describes the DAA protocol and assumes the correctness and integrity.

5 Implementation

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

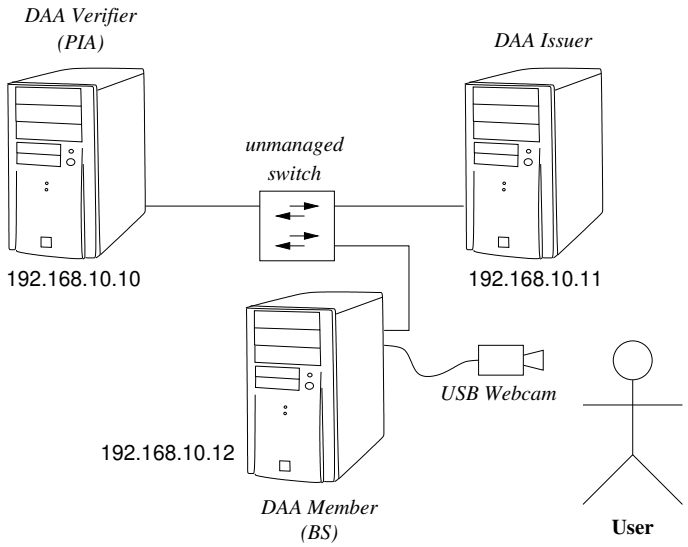


Figure 5.1: Prototype setup to show DAA features and the Dataflow from BS to PIA

Figure 5.1 shows the setup on a connection level. To show the features of DAA, it is necessary to have three independent systems which are connected via a TCP/IP network. Every host is connected via Ethernet to the other systems. To keep the setup minimal, the IP addresses are static and Internet is only required during installation.

Table 5.1: Systems used for demonstration prototype

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

5.1 Hardware Setup

For demonstrating remote attestation via DAA over a simple network infrastructure, we use 3 systems with similar configuration. Table 5.1 show 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 TPM2.0 header which may differ from board to board. A 19-pin header is available on the older platform of *System 2*. As long as TPM and mainboard have the same 19-pin connector they will be compatible to each other. The newer Gigabyte mainboards come with a proprietary 11-pin connector which is only compatible with Gigabyte's TPM2.0_S module. All other modules are however electrical compatible since only unused pins of the full size connector are removed. With a wiring adapter any TPM board would work on any mainboard supporting TPM2.0 even when coming with a prorietary header.

5.2 Operating System

The Operating System need 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

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

Table 5.2: Disk layout of the BS prototype

Xaptum ECDAA[1] 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 according Kernel compile parameters must be set.

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

5.3 Trusted Boot

By default, every Mainboard with support for TPM2.0 supports also Trusted Boot. When a TPM becomes available, the 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, Trusted Boot is directly supported and needs just to be enabled in the configuration. In this case, Grub will be measured from the BIOS to the PCRs 4 and 5, as shown in 3.1. Grub itself uses PCR 8 for executed commands, the Kernel command line and all commands forwarded to Kernel modules. PCR 9 is used to measure all files read by Grub[2].

There is however a more efficient way of booting for embedded systems since there is often only one bootable Kernel in place and the device has to boot autonomously. Pawit Pornkitprasam [13][14] and Karl O from Tevora [16] introduced the concept of a *Unified Kernel* for Ubuntu and Arch respectively.

[1] https://github.com/xaptum/ecdaa
[2] https://www.gnu.org/software/grub/manual/grub/html_node/Measured-Boot.html (visited on 19.11.2020)

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

Table 5.3: Memory layout of the Unified Kernel EFI file

We create a large EFI file which contains the initramfs, Kernel command line and the Kernel itself. This EFI file replaces that from Grub in the EFI boot partition. Listing 2 shows the used command line which will be saved on `/boot/kernel-command-line.txt` The parameters activate also IMA which is discussed later in this chapter. The shell script shown in Listing 5 uses the command `objcopy` to create a single EFI file which contains the Kernel with corresponding release information and parameters and the initial ramdisk. The memory layout is shown in 5.3 With this Unified Kernel in place, no additional PCRs are used and everything is measured by the BIOS. It furthermore omits the bootloader which is not necessary since the BS is ideally an embedded system with a single boot option in the end.

So, when the BIOS hands over the system to the bootloader, all PCR values are already set. The Trusted Boot chain can now be used to authenticate the Kernel against the system. Therefore a second key is added to the LUKS header, which is a random number of 32 byte length. This key is saved in the TPM and sealed with the values of PCR 0–7. If the BIOS measurements calculate the same values as those of the sealing, the TPM is able to reveal the key for the FDE and the boot process can continue. The *trusted* environment is now extended to the Kernel and the modules loaded at boot.

- Trusted Boot with GRUB 2.04: TPM support available; PCR mapping
- Secure Boot with Unified Kernel; another PCR mapping
- Benefits and Drawbacks of both variants
- describe automated unlocking

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

5.4 Integrity Measurement Architecture

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

5.4.1 Handling external hardware

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

5.5 Interaction with TPM2

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

5.6 Direct Anonymous Attestation

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

6 Conclusion and Outlook

6.1 Testing

These are the test results

6.2 Limitations

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

6.3 Future Work

6.4 Outlook

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

Bibliography

[1] Will Arthur, David Challener, and Kenneth Goldman. *A Practical Guide to TPM 2.0*. Jan. 2015. DOI: 10.1007/978-1-4302-6584-9 (cit. on p. 12).

[2] Brickell, Camenisch, and Chen. "Direct Anonymous Attestation". In: *SIGSAC: 11th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2004 (cit. on p. 36).

[3] Jan Camenisch, Liqun Chen, Manu Drijvers, Anja Lehmann, David Novick, and Rainer Urian. "One TPM to Bind Them All: Fixing TPM 2.0 for Provably Secure Anonymous Attestation". In: May 2017, pp. 901–920. DOI: 10.1109/SP.2017.22 (cit. on p. 19).

[4] Jan Camenisch, Manu Drijvers, and Anja Lehmann. "Universally Composable Direct Anonymous Attestation". In: *Public-Key Cryptography – PKC 2016*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Berlin, Heidelberg: Springer Berlin Heidelberg, Mar. 2016, pp. 234–264. ISBN: 978-3-662-49386-1. DOI: 10.1007/978-3-662-49387-8_10 (cit. on pp. 19, 21, 22, 25, 37).

[5] Jan Camenisch and Anna Lysyanskaya. "Signature Schemes and Anonymous Credentials from Bilinear Maps". In: vol. 3152/2004. Aug. 2004, pp. 56–72. DOI: 10.1007/978-3-540-28628-8_4 (cit. on p. 21).

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

[7] Liqun Chen, Dan Page, and Nigel Smart. "On the Design and Implementation of an Efficient DAA Scheme". In: Nov. 2010, pp. 223–237. DOI: 10.1007/978-3-642-12510-2_16 (cit. on p. 37).

[8] TPM2 Software Community. *TPM2 Tools*. 2020. URL: https://github.com/tpm2-software/tpm2-tools (visited on 05/15/2020) (cit. on p. 11).

[9] Free Software Foundation. *GRUB 2.04 User Manual: Measuring Boot Components*. 2019. URL: https://www.gnu.org/software/grub/manual/grub/html_node/Measured-Boot.html (visited on 03/29/2021) (cit. on p. 16).

[10] 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. 14, 15).

[11] 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. 10).

[12] 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. 10).

[13] 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).

[14] 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. 11, 40).

[15] 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. 7).

[16] 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).

[17] *The TPM Library Specification*. 2019. URL: https://trustedcomputinggroup.org/resource/tpm-library-specification/ (visited on 05/16/2020) (cit. on p. 9).

JOHANNES KEPLER
UNIVERSITY LINZ

Sealing LUKS encryption key with PCRs in a TPM

Listing 1: `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 2: `kernel-command-line.txt`: Command line for the Kernel

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

Listing 3: `passphrase-from-tpm.sh`: Initramfs-script to ask the TPM for the LUKS key

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

Listing 4: `update-luks-tpm.sh`: Script for updating the Sealing of the TPM Object with new PCR values

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

```

set +e
tpm2_evictcontrol -c 0x81000000

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

Listing 5: `update-kernel.sh`: Script for updating the unified Kernel

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

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

Listing 6: `install.sh`: Script to install Trusted Boot on Ubuntu

```
#!/bin/bash
```

```
2 set -e
3
4 cp -vf ./passphrase-from-tpm.sh /usr/sbin/ || exit 1
5 cp -vf ./update-luks-tpm.sh /usr/sbin || exit 1
6 cp -vf ./update-kernel.sh /usr/sbin || exit 1
7 cp -vf ./create-luks-tpm.sh /usr/sbin || exit 1
8
9 cp -vf ./tpm2-hook.sh /etc/initramfs-tools/hooks/ || exit 2
10 awk -i inplace '/luks/{print␣$0␣",discard,initramfs,keyscript=/usr/sbin/passphrase-
       from-tpm.sh"}' /etc/crypttab
11
12 cp -vf ./kernel-command-line.txt /boot/ || exit 3
13 /usr/sbin/create-luks-tpm.sh
14 /usr/sbin/update-kernel.sh
15 efibootmgr --create --disk /dev/nvme0n1 --part 1 --label "ubuntu␣unified" --loader "\
       EFI\Linux\Linux.efi" --verbose
16 echo "Installed␣successfully!␣Please␣reboot␣and␣execute␣update-luks-tpm.sh␣
       afterwards"
```

JOHANNES KEPLER
UNIVERSITY LINZ

TCP/IP Wrapper for the Xaptum ECDAA Protocol

1 Common source files for all DAA parties

Listing 1: `common.h`

```
1
2  #ifndef ECDAA_COMMON_H
3  #define ECDAA_COMMON_H
4
5  #include <sys/random.h>
6  #include <stdio.h>
7  #include <string.h>
8  #include <stdint.h>
9
10 #define ISSUERIP "192.168.10.11"
11 #define ISSUERPORT 6590
12 #define MEMBERIP "192.168.10.12"
13 #define MEMBERPORT 6591
14 #define VERIFIERIP "192.168.10.10"
15 #define VERIFIERPORT 6592
16
17 #define MAX_CLIENTS 10
18 #define MAX_BUFSIZE 20480
19 #define MAX_MSGSIZE ((MAX_BUFSIZE - 1536) / 2)
20 #define MAX_BSNSIZE 128
21 #define NONCE_SIZE 384
22
23 typedef int (*conn_handler)(char *buffer);
24
25 void ecdaa_rand(void *buffer, size_t buflen);
26
27 size_t ecdaa_decode(const char *in_enc, uint8_t *out_dec, size_t outlen);
28
29 size_t ecdaa_encode(const uint8_t *in_dec, char *out_enc, size_t inlen);
30
31 #endif //ECDAA_COMMON_H
```

Listing 2: `common.c`

```
1  #include "common.h"
2
3  static const char base64_table[65] = "
       ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";
4
5  static const uint8_t base64_index[256] = {
6    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
7    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
8    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 62, 63, 62, 62, 63,
9   52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 0, 0, 0, 0, 0, 0,
10   0, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14,
11   15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 0, 0, 0, 0, 63,
12   0, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40,
13   41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51 };
14
15
16 void ecdaa_rand(void *buffer, size_t buflen) {
17    getrandom(buffer, buflen, 0);
18 }
19
20 char bin2hex(uint8_t byte) {
21    uint8_t word = byte & 0x0f;
22    char hex = 0;
23    if (word >= 0 && word <= 9) hex = word + '0';
24    else if (word >= 10 && word <= 15) hex = word - 10 + 'A';
25    return hex;
26 }
27
28 uint8_t hex2bin(char hex) {
29    uint8_t byte = 0;
30    if (hex >= '0' && hex <= '9') byte = hex - '0';
31    else if (hex >= 'a' && hex <= 'f') byte = hex - 'a' + 10;
32    else if (hex >= 'A' && hex <= 'F') byte = hex - 'A' + 10;
33    return byte;
34 }
35
36 size_t hex_decode(const char *in_hex, uint8_t *out_bin, size_t outlen) {
37    size_t i = 0;
38    size_t j = 0;
39    for (; i < outlen; i++, j+=2) {
40       uint8_t val = hex2bin(in_hex[j]);
41       val += hex2bin(in_hex[j+1]) * 16;
42       out_bin[i] = (char) val;
43    }
44    return i;
45 }
46
47 size_t hex_encode(const uint8_t *in_bin, char *out_hex, size_t inlen) {
48    size_t i = 0;
```

```
49    size_t j = 0;
50    for (; i < inlen; i++, j+=2) {
51       out_hex[j] = bin2hex(in_bin[i]);
52       out_hex[j+1] = bin2hex(in_bin[i] >> 4);
53    }
54    return i;
55 }
56
57 size_t base64_encode(const uint8_t *in_dec, char *out_enc, size_t inlen) {
58    size_t outlen = 4 * ((inlen + 2) / 3);
59    size_t i = 0;
60    size_t j = 0;
61
62    while(i < inlen) {
63       uint32_t octet_a = i < inlen ? in_dec[i++] : 0;
64       uint32_t octet_b = i < inlen ? in_dec[i++] : 0;
65       uint32_t octet_c = i < inlen ? in_dec[i++] : 0;
66
67       uint32_t triple = (octet_a << 16) + (octet_b << 8) + octet_c;
68
69       out_enc[j++] = base64_table[(triple >> 18) & 0x3F];
70       out_enc[j++] = base64_table[(triple >> 12) & 0x3F];
71       out_enc[j++] = base64_table[(triple >> 6) & 0x3F];
72       out_enc[j++] = base64_table[(triple) & 0x3F];
73    }
74    switch (inlen % 3) {
75    case 1:
76       out_enc[j-2] = '=';
77    case 2:
78       out_enc[j-1] = '=';
79    default:
80       break;
81    }
82    return j;
83 }
84
85 size_t base64_decode(const char *in_enc, uint8_t *out_dec, size_t outlen) {
86    size_t inlen = 4 * ((outlen + 2) / 3);
87    size_t i = 0;
88    size_t j = 0;
89
90    while (i < inlen) {
91       uint32_t sextet_a = in_enc[i] == '=' ? 0 & i++ : base64_index[in_enc[i++]];
92       uint32_t sextet_b = in_enc[i] == '=' ? 0 & i++ : base64_index[in_enc[i++]];
93       uint32_t sextet_c = in_enc[i] == '=' ? 0 & i++ : base64_index[in_enc[i++]];
94       uint32_t sextet_d = in_enc[i] == '=' ? 0 & i++ : base64_index[in_enc[i++]];
95
96       uint32_t triple = (sextet_a << 18) + (sextet_b << 12) + (sextet_c << 6) +
             sextet_d;
97
```

```
98        if (j < outlen) out_dec[j++] = (triple >> 16) & 0xFF;
99        if (j < outlen) out_dec[j++] = (triple >> 8) & 0xFF;
100       if (j < outlen) out_dec[j++] = triple & 0xFF;
101    }
102    return i;
103 }
104
105 size_t ecdaa_encode(const uint8_t *in_dec, char *out_enc, size_t inlen) {
106    return base64_encode(in_dec, out_enc, inlen);
107 }
108
109 size_t ecdaa_decode(const char *in_enc, uint8_t *out_dec, size_t outlen) {
110    return base64_decode(in_enc, out_dec, outlen);
111 }
```

Listing 3: `client.h`

```
1  //
2  // Created by root on 10/30/19.
3  //
4
5  #ifndef ECDAA_ISSUER_CLIENT_H
6  #define ECDAA_ISSUER_CLIENT_H
7  #include <stdlib.h>
8  #include <string.h>
9  #include <stdio.h>
10 #include <sys/socket.h>
11 #include <unistd.h>
12 #include <arpa/inet.h>
13 #include "common.h"
14
15 int client_connect(conn_handler handler, char *ip_str, int16_t port);
16
17 #endif //ECDAA_ISSUER_CLIENT_H
```

Listing 4: `client.c`

```
1  #include "client.h"
2
3  int client_open(char *servip, int16_t port) {
4     struct sockaddr_in servaddr;
5     size_t servaddr_len = 0;
6     int connfd = 0;
7
8     connfd = socket(AF_INET, SOCK_STREAM, 0);
9     if (-1 == connfd) {
10       printf("client_listen:␣failed␣to␣create␣endpoint.\n");
11       return -1;
12    }
13    bzero(&servaddr, sizeof(servaddr));
```

```
14
15    servaddr.sin_family = AF_INET;
16    servaddr.sin_addr.s_addr = inet_addr(servip);
17    servaddr.sin_port = htons(port);
18    servaddr_len = sizeof(servaddr);
19    if (0 != connect(connfd, (const struct sockaddr *) &servaddr, servaddr_len)) {
20       printf("client_accept:␣connection␣to␣server␣failed\n");
21       close(connfd);
22       return -1;
23    }
24    return connfd;
25 }
26
27 int client_connect(conn_handler handler, char *servip, int16_t port) {
28    int connfd = 0;
29    char buffer[MAX_BUFSIZE];
30    int ret = 0;
31    int len = 0;
32
33    connfd = client_open(servip, port);
34    if(0 >= connfd) {
35       return -1;
36    }
37    bzero(buffer, MAX_BUFSIZE);
38    for (ret = 0; 0 == ret;) {
39       ret = handler(buffer);
40       if(0 != ret)
41          break;
42
43       if (0 >= write(connfd, buffer, sizeof(buffer))) {
44          printf("client_connect:␣cannot␣write␣to␣socket\n");
45          ret = -1;
46       }
47
48       bzero(buffer, MAX_BUFSIZE);
49       len = read(connfd, buffer, sizeof(buffer));
50       if (0 > len) {
51          printf("client_connect:␣cannot␣read␣from␣socket\n");
52          ret = -1;
53       } else if (0 == len) {
54          printf("client_connect:␣server␣closed␣connection\n");
55          ret = 1;
56       }
57    }
58
59    if (0 != close(connfd)) {
60       printf("client_connect:␣failed␣to␣close␣server␣connection␣properly\n");
61    }
62
63    return ret;
```

```
64 }
```

Listing 5: `server.h`

```
1 //
2 // Created by root on 10/30/19.
3 //
4
5 #ifndef ECDAA_ISSUER_SERVER_H
6 #define ECDAA_ISSUER_SERVER_H
7 #include <stdio.h>
8 #include <string.h>
9 #include <stdlib.h>
10 #include <sys/socket.h>
11 #include <netinet/in.h>
12 #include <unistd.h>
13 #include "common.h"
14
15 int server_start(conn_handler handler, int16_t port);
16
17 #endif //ECDAA_ISSUER_SERVER_H
```

Listing 6: `server.c`

```
1 #include "server.h"
2
3 int server_open(int16_t port) {
4    struct sockaddr_in servaddr;
5    int connfd = 0;
6
7    connfd = socket(AF_INET, SOCK_STREAM, 0);
8    if (-1 == connfd) {
9       printf("server_open:␣failed␣to␣create␣endpoint.\n");
10      return -1;
11   }
12   servaddr.sin_family = AF_INET;
13   servaddr.sin_addr.s_addr = htonl(INADDR_ANY);
14   servaddr.sin_port = htons(port);
15
16   if (0 != bind(connfd, (const struct sockaddr *) &servaddr, sizeof(servaddr))) {
17      printf("server_open:␣failed␣to␣bind␣socket\n");
18      close(connfd);
19      return -1;
20   }
21   return connfd;
22 }
23
24 int server_start(conn_handler handler, int16_t port) {
25   struct sockaddr_in client;
26   unsigned int client_len = 0;
```

```
27    int connfd = 0;
28    int clientfd = 0;
29    int len = 0;
30    int ret = 0; //<0 .. failure, 0 .. continue, 1 .. exit, 2 .. shutdown
31    char buffer[MAX_BUFSIZE];
32
33    if(NULL == handler) {
34       printf("server_start:␣received␣empty␣handler,␣stopping\n");
35       return -1;
36    }
37    connfd = server_open(port);
38    if(-1 == connfd) {
39       printf("server_start:␣could␣not␣open␣port,␣stopping\n");
40       return -1;
41    }
42    if (0 != listen(connfd, MAX_CLIENTS)) {
43       printf("server_start:␣listen␣failed,␣stopping\n");
44       return -1;
45    }
46    printf("server_start:␣listening\n");
47    for(ret = 1; 1 == ret;) {
48       client_len = sizeof(client);
49       clientfd = accept(connfd, (struct sockaddr *) &client, &client_len);
50       if (0 >= clientfd) {
51          printf("server_start:␣connection␣to␣client␣failed\n");
52          ret = -1;
53       } else {
54          for(ret = 0; 0 == ret;) {
55             bzero(buffer, MAX_BUFSIZE);
56             len = read(clientfd, buffer, sizeof(buffer));
57             if (0 > len) {
58                printf("server_start:␣cannot␣read␣from␣socket\n");
59                ret = -1;
60             } else if(0 == len) {
61                printf("server_start:␣client␣closed␣connection\n");
62                ret = 1;
63             } else {
64                ret = handler(buffer);
65             }
66
67             if (0 <= ret && 0 >= write(clientfd, buffer, sizeof(buffer))) {
68                printf("server_start:␣cannot␣write␣to␣socket\n");
69                ret = -1;
70             }
71
72          }
73          if (0 != close(clientfd)) {
74             printf("server_start:␣failed␣to␣close␣client␣connection␣properly\n");
75             ret = -1;
76          }
```

```
77        }
78     }
79     printf("server_start:␣closing␣connection\n");
80     if (0 != close(connfd)) {
81        printf("server_start:␣failed␣to␣close␣server␣port␣properly\n");
82        ret = -1;
83     }
84     return ret;
85 }
```

2 Source files for the DAA Issuer

Listing 7: `issuer.h`

```
1 //
2 // Created by root on 11/5/19.
3 //
4
5 #ifndef ECDAA_ISSUER_H
6 #define ECDAA_ISSUER_H
7 #include <ecdaa.h>
8 #include "server.h"
9 #include "client.h"
10 #include "common.h"
11
12 int process_issuer(char *buffer);
13 const char* issuer_public_key_file = "ipk.bin";
14 const char* issuer_secret_key_file = "isk.bin";
15
16 #endif //ECDAA_ISSUER_H
```

Listing 8: `issuer.c`

```
1 #include "issuer.h"
2
3 typedef enum issuer_state {
4    ON,
5    JOINSTART,
6    JOINPROCEED,
7    READY
8 } issuerstate_e;
9
10 typedef struct issuer {
11    struct ecdaa_issuer_public_key_FP256BN ipk;
12    struct ecdaa_issuer_secret_key_FP256BN isk;
13    uint8_t nonce[NONCE_SIZE];
14    struct ecdaa_member_public_key_FP256BN mpk;
15    issuerstate_e state;
16    struct ecdaa_credential_FP256BN cred;
```

```
17    struct ecdaa_credential_FP256BN_signature cred_sig;
18 } issuer_t;
19
20 issuer_t issuer;
21
22 int issuer_setup();
23 int issuer_reset(char *buffer);
24 int issuer_joinstart(char *buffer);
25 int issuer_joinproceed(char *buffer);
26 int issuer_publish(char *buffer);
27
28 int main() {
29    issuer.state = ON;
30    if(ON == issuer.state) {
31       if (0 == issuer_setup()) {
32          issuer.state = READY;
33       } else {
34          printf("issuer␣setup␣failed\n");
35          return 1;
36       }
37    }
38
39    if (2 != server_start(&process_issuer, ISSUERPORT)) {
40       printf("server␣failed\n");
41    }
42    return 0;
43 }
44
45 int process_issuer(char *buffer) {
46    int ret = 0;
47
48    printf(">␣ISSUER:␣%s\n", buffer);
49
50    if (0 == strncasecmp("OK", buffer, 2)) {
51       switch (issuer.state) {
52          case JOINPROCEED:
53             issuer.state = READY;
54             break;
55          default:
56             bzero(buffer, MAX_BUFSIZE);
57             strncpy(buffer, "ERR\n", 4);
58             break;
59       }
60    } else if (0 == strncasecmp("ERR", buffer, 3)) {
61       switch (issuer.state) {
62          case JOINPROCEED:
63             printf("command␣failed␣at␣client\n");
64             issuer.state = READY;
65             break;
66          default:
```

```
67             bzero(buffer, MAX_BUFSIZE);
68             strncpy(buffer, "ERR\n", 4);
69             break;
70          }
71       } else if (0 == strncasecmp("RESET", buffer, 4)) {
72          switch (issuer.state) {
73             case READY:
74                printf("generate new issuer identity\n");
75                if (0 == issuer_setup(buffer)) {
76                   issuer.state = READY;
77                } else {
78                   printf("issuer setup failed\n");
79                   return 2;
80                }
81                if(0 == issuer_joinstart(buffer)) {
82                   issuer.state = JOINSTART;
83                }
84                break;
85             default:
86                bzero(buffer, MAX_BUFSIZE);
87                strncpy(buffer, "ERR\n", 4);
88                break;
89          }
90       } else if (0 == strncasecmp("JOIN", buffer, 4)) {
91          switch (issuer.state) {
92             case READY:
93                if(0 == issuer_joinstart(buffer)) {
94                   issuer.state = JOINSTART;
95                }
96                break;
97             default:
98                bzero(buffer, MAX_BUFSIZE);
99                strncpy(buffer, "ERR\n", 4);
100               break;
101         }
102      } else if (0 == strncasecmp("APPEND", buffer, 6)) {
103         switch (issuer.state) {
104            case JOINSTART:
105               if(0 == issuer_joinproceed(buffer)) {
106                  issuer.state = READY;
107               }
108               break;
109            default:
110               bzero(buffer, MAX_BUFSIZE);
111               strncpy(buffer, "ERR\n", 4);
112               break;
113         }
114      } else if (0 == strncasecmp("PUBLISH", buffer, 7)) {
115         switch (issuer.state) {
116            case READY:
```

```
117             issuer_publish(buffer);
118             issuer.state = READY;
119             break;
120          default:
121             bzero(buffer, MAX_BUFSIZE);
122             strncpy(buffer, "ERR\n", 4);
123             break;
124       }
125    } else if (0 == strncasecmp("EXIT", buffer, 4)) {
126       printf("exit()\n");
127       bzero(buffer, MAX_BUFSIZE);
128       strncpy(buffer, "OK\n", 3);
129       ret = 1;
130    } else if (0 == strncasecmp("SHUTDOWN", buffer, 8)) {
131       bzero(buffer, MAX_BUFSIZE);
132       strncpy(buffer, "OK\n", 3);
133       ret = 2;
134    } else {
135       printf("error()\n");
136       bzero(buffer, MAX_BUFSIZE);
137       strncpy(buffer, "ERR\n", 4);
138       ret = 0;
139    }
140
141    printf("<␣ISSUER:␣%s", buffer);
142    return ret;
143 }
144
145 // "JOIN" > "JOINSTART <issuer.nonce>"
146 int issuer_joinstart(char *buffer) {
147    ecdaa_rand(issuer.nonce, NONCE_SIZE);
148    char* current;
149    bzero(buffer, MAX_BUFSIZE);
150    strncpy(buffer, "JOINSTART␣", 10);
151    current = &buffer[10];
152    ecdaa_encode(issuer.nonce, current, NONCE_SIZE);
153    buffer[2 * NONCE_SIZE + 10] = '\n';
154    return 0;
155 }
156
157 // "APPEND <member.mpk>" > "JOINPROCEED <member.cred><member.cred_sig>"
158 int issuer_joinproceed(char *buffer) {
159    char *current = &buffer[7];
160    uint8_t binbuf[MAX_BUFSIZE];
161    bzero(binbuf, MAX_BUFSIZE);
162    int ret = 0;
163
164    ecdaa_decode(current, binbuf, ECDAA_MEMBER_PUBLIC_KEY_FP256BN_LENGTH);
165    bzero(buffer, MAX_BUFSIZE);
```

```
166     ret = ecdaa_member_public_key_FP256BN_deserialize(&issuer.mpk, binbuf, issuer.
            nonce, NONCE_SIZE);
167     if(-1 == ret) {
168        strncpy(buffer, "ERR\n", 4);
169        printf("issuer_joinproceed:␣member␣public␣key␣is␣malformed!\n");
170        return -1;
171     } else if (-2 == ret) {
172        strncpy(buffer, "ERR\n", 4);
173        printf("issuer_joinproceed:␣signature␣of␣member␣public␣key␣is␣invalid\n");
174        return -1;
175     }
176
177     if (0 != ecdaa_credential_FP256BN_generate(&issuer.cred, &issuer.cred_sig, &issuer.
            isk, &issuer.mpk, ecdaa_rand)) {
178        strncpy(buffer, "ERR\n", 4);
179        printf("issuer_joinproceed:␣error␣generating␣credential\n");
180        return -1;
181     }
182     bzero(buffer, MAX_BUFSIZE);
183     strncpy(buffer, "JOINPROCEED␣", 12);
184
185     current = &buffer[12];
186     bzero(binbuf, MAX_BUFSIZE);
187     ecdaa_credential_FP256BN_serialize(binbuf, &issuer.cred);
188     ret = ecdaa_encode(binbuf, current, ECDAA_CREDENTIAL_FP256BN_LENGTH);
189
190     current = &current[ret];
191     bzero(binbuf, MAX_BUFSIZE);
192     ecdaa_credential_FP256BN_signature_serialize(binbuf, &issuer.cred_sig);
193     ret = ecdaa_encode(binbuf, current, ECDAA_CREDENTIAL_FP256BN_SIGNATURE_LENGTH);
194
195     current[ret] = '\n';
196     return 0;
197 }
198
199 // "RESET > RESETDONE"
200 int issuer_reset(char *buffer) {
201     printf("issuer_reset:␣generating␣new␣keys␣and␣save␣them␣to␣disk\n");
202     if (0 != ecdaa_issuer_key_pair_FP256BN_generate(&issuer.ipk, &issuer.isk,
            ecdaa_rand)) {
203        printf("issuer_reset:␣Error␣generating␣issuer␣key-pair\n");
204        strncpy(buffer, "ERR\n", 4);
205        return -1;
206     }
207     if(0 != ecdaa_issuer_public_key_FP256BN_serialize_file(issuer_public_key_file, &
            issuer.ipk) ||
208           0 != ecdaa_issuer_secret_key_FP256BN_serialize_file(issuer_secret_key_file,
                  &issuer.isk)) {
209        printf("issuer_reset:␣Error␣saving␣key-pair␣to␣disk\n");
210        strncpy(buffer, "ERR\n", 4);
```

```
211       return -1;
212    }
213    bzero(buffer, MAX_BUFSIZE);
214    strncpy(buffer, "RESETDONE\n", 10);
215    return 0;
216 }
217
218 //Load or generate issuer keypair initially
219 int issuer_setup() {
220    printf("setup()\n");
221    if (0 == ecdaa_issuer_public_key_FP256BN_deserialize_file(&issuer.ipk,
           issuer_public_key_file)) {
222       if (0 == ecdaa_issuer_secret_key_FP256BN_deserialize_file(&issuer.isk,
              issuer_secret_key_file)) {
223          printf("issuer_setup:␣loaded␣keys␣from␣disk\n");
224          return 0;
225       }
226    }
227    printf("issuer_setup:␣generating␣new␣keys␣and␣save␣them␣to␣disk\n");
228    if (0 != ecdaa_issuer_key_pair_FP256BN_generate(&issuer.ipk, &issuer.isk,
           ecdaa_rand)) {
229       printf("issuer_setup:␣Error␣generating␣issuer␣key-pair\n");
230       return -1;
231    }
232    if(0 != ecdaa_issuer_public_key_FP256BN_serialize_file(issuer_public_key_file, &
           issuer.ipk) ||
233          0 != ecdaa_issuer_secret_key_FP256BN_serialize_file(issuer_secret_key_file,
                 &issuer.isk)) {
234       printf("issuer_setup:␣Error␣saving␣key-pair␣to␣disk\n");
235       return -1;
236    }
237    return 0;
238 }
239
240 // "PUBLISH" > "PUBLISH <issuer.ipk>"
241 int issuer_publish(char *buffer) {
242    char *current;
243    uint8_t binbuf[MAX_BUFSIZE];
244    bzero(buffer, MAX_BUFSIZE);
245    int ret = 0;
246
247    strncpy(buffer, "PUBLISH␣", 8);
248
249    current = &buffer[8];
250    bzero(binbuf, MAX_BUFSIZE);
251    ecdaa_issuer_public_key_FP256BN_serialize(binbuf, &issuer.ipk);
252    ret = ecdaa_encode(binbuf, current, ECDAA_ISSUER_PUBLIC_KEY_FP256BN_LENGTH);
253
254    current[ret] = '\n';
255
```

```
256     return 0;
257 }
```

3 Source files for the DAA Member

Listing 9: `member.h`

```
1
2 #ifndef ECDAA_MEMBER_H
3 #define ECDAA_MEMBER_H
4 #include <ecdaa.h>
5 #include <ecdaa.h>
6 #include "server.h"
7 #include "client.h"
8 #include "common.h"
9
10 /* int process_member(char *buffer); */
11
12 const char* member_public_key_file = "mpk.bin";
13 const char* member_secret_key_file = "msk.bin";
14 const char* member_credential_file = "mcred.bin";
15 const char* member_nonce_file = "mnonce.bin";
16
17 #endif //ECDAA_ISSUER_H
```

Listing 10: `member.c`

```
1 #include "member.h"
2
3 typedef enum memberstate {
4    ON,
5    ISSUERPUB,
6    RCVPUBLIC,
7    JOIN,
8    APPEND,
9    JOINPROCEED,
10   JOINED,
11 } memberstate_e;
12
13 typedef struct member {
14    struct ecdaa_member_public_key_FP256BN mpk;
15    struct ecdaa_member_secret_key_FP256BN msk;
16    memberstate_e state;
17    uint8_t nonce[NONCE_SIZE];
18    struct ecdaa_credential_FP256BN cred;
19    struct ecdaa_issuer_public_key_FP256BN ipk;
20    uint8_t bsn[MAX_BSNSIZE];
21    size_t bsn_len;
22 } member_t;
```

```
23
24 member_t member;
25 uint8_t msg[MAX_MSGSIZE];
26 size_t msg_len;
27 int member_join(char *buffer);
28
29 int member_verifymsg(char *buffer);
30
31 int member_publish(char *buffer);
32
33 int member_joinappend(char *buffer);
34
35 int member_joinfinish(char *buffer);
36
37 int main(int argc, char **argv) {
38    char buffer[MAX_BUFSIZE];
39    char *remote_ip = argv[2];
40    int ret = 0;
41    //strncpy(member.bsn, "mybasename", 10);
42    //member.bsn_len = strlen(member.bsn);
43    switch(argc) {
44       case 3:
45          if( 0 == strncasecmp("--join", argv[1], 6) || 0 == strncasecmp("-j", argv[1],
                2)) {
46             member.state = ON;
47             ret = client_connect(&member_join, remote_ip, ISSUERPORT);
48             if (0 >= ret || JOINED != member.state) {
49                printf("Join process failed!\n");
50                return 1;
51             } else {
52                printf("Join process was successful\n");
53             }
54          } else {
55       printf("2 arguments but not join\n");
56     }
57          break;
58       case 4:
59          if( 0 == strncasecmp("--send", argv[1], 6) || 0 == strncasecmp("-s", argv[1],
                2)) {
60             msg_len = ecdaa_read_from_file(msg, MAX_MSGSIZE, argv[3]);
61    if (msg_len < 0) {
62                printf("Could not open message file %s\n", argv[3]);
63                return 1;
64    }
65             if (0 > ecdaa_read_from_file(member.nonce, NONCE_SIZE, member_nonce_file)
                    ||
66                 0 != ecdaa_member_secret_key_FP256BN_deserialize_file(&member.msk,
                       member_secret_key_file) ||
67                 0 != ecdaa_member_public_key_FP256BN_deserialize_file(&member.mpk,
                       member_public_key_file, member.nonce, NONCE_SIZE) ||
```

```
68                   0 != ecdaa_credential_FP256BN_deserialize_file(&member.cred,
                         member_credential_file)) {
69                printf("Could not import key files. importing from %s, %s, %s or %s
                      was not successful\n",
70                   member_nonce_file, member_secret_key_file, member_public_key_file,
                         member_credential_file);
71                return 1;
72             }
73             member.state = JOINED;
74             ret = client_connect(&member_verifymsg, remote_ip, VERIFIERPORT);
75             if (0 >= ret || JOINED != member.state) {
76                printf("connection to verifier failed\n");
77             }
78          } else {
79        printf("3 arguments but not send\n");
80      }
81          break;
82       default:
83          printf("Usage: \n Join an issuer's group: %s --join <issuer's IPv4>\n",
               argv[0]);
84          printf("Send a signed message to the verifier: %s --send <verifier's IPv4
               > <msgfile>\n", argv[0]);
85          printf("Before sending a DAA-signed message, the member must join a DAA
               group\n");
86          break;
87    }
88    return 0;
89 }
90
91 int member_join(char *buffer) {
92    int ret = 0;
93
94    switch (member.state) {
95       case ON:
96          bzero(buffer, MAX_BUFSIZE);
97          strncpy(buffer, "PUBLISH\n", 8);
98          member.state = ISSUERPUB;
99          break;
100      case ISSUERPUB:
101         if (0 == strncasecmp("PUBLISH", buffer, 7)) {
102            printf("ISSUER > MEMBER: %s", buffer);
103            uint8_t binbuf[MAX_BUFSIZE];
104            char *current = &buffer[8];
105            ecdaa_decode(current, binbuf, ECDAA_ISSUER_PUBLIC_KEY_FP256BN_LENGTH);
106            ret = ecdaa_issuer_public_key_FP256BN_deserialize(&member.ipk, binbuf);
107            if (-1 == ret) {
108               printf("member_getpublic: issuer public key is malformed!\n");
109               ret = -1;
110            } else if (-2 == ret) {
```

```
111                printf("member_getpublic:␣signature␣of␣issuer␣public␣key␣is␣invalid\
                       n");
112                ret = -1;
113             } else {
114                bzero(buffer, MAX_BUFSIZE);
115                strncpy(buffer, "JOIN\n", 5);
116                member.state = APPEND;
117       ret = 0;
118             }
119          } else {
120             printf("member_getpublic:␣did␣not␣get␣public␣key␣from␣issuer\n");
121             member.state = ON;
122             ret = -1;
123          }
124          break;
125       case APPEND:
126          if (0 == strncasecmp("JOINSTART", buffer, 9)) {
127             printf("ISSUER␣>␣MEMBER:␣%s", buffer);
128             member_joinappend(buffer);
129             member.state = JOINPROCEED;
130          } else {
131             printf("member_join:␣did␣not␣get␣nonce␣from␣issuer\n");
132             member.state = RCVPUBLIC;
133             ret = -1;
134          }
135          break;
136       case JOINPROCEED:
137          if (0 == strncasecmp("JOINPROCEED", buffer, 11)) {
138             printf("ISSUER␣>␣MEMBER:␣%s", buffer);
139             member_joinfinish(buffer);
140             member.state = JOINED;
141             ret = 1;
142          } else {
143             printf("member_getpublic:␣did␣not␣get␣credentials␣from␣issuer\n");
144             member.state = RCVPUBLIC;
145             ret = -1;
146          }
147          break;
148       default:
149          ret = -1;
150    }
151    if (0 == ret) {
152       printf("ISSUER␣<␣MEMBER:␣%s", buffer);
153    }
154    return ret;
155 }
156
157 //"VERIFYMSG" > "VERIFYMSG <attestval>"
158 int member_verifymsg(char *buffer) {
159    char *current = buffer;
```

```
160   uint8_t binbuf[MAX_BUFSIZE];
161   uint8_t has_nym = member.bsn_len > 0 ? 1 : 0;
162   struct ecdaa_signature_FP256BN sig;
163   size_t sig_len = has_nym ? ecdaa_signature_FP256BN_with_nym_length() :
          ecdaa_signature_FP256BN_length();
164   int ret = 0;
165
166   if (0 == strncasecmp("OK", buffer, 2)) {
167      return 1;
168   } else if (0 == strncasecmp("ERR", buffer, 3)) {
169      printf("member_verifymsg:␣Verifier␣refused␣signature\n");
170 return 1;
171   }
172
173   bzero(buffer, MAX_BUFSIZE);
174   strncpy(current, "VERIFYMSG␣", 10);
175   current = &current[10];
176
177   ret = ecdaa_encode(msg, current, msg_len);
178   current = &current[2 * MAX_MSGSIZE];
179   if(has_nym) {
180      if (0 != ecdaa_signature_FP256BN_sign(&sig, msg, msg_len, member.bsn, member.
              bsn_len, &member.msk, &member.cred, ecdaa_rand)) {
181         printf("member_verifymsg:␣Signing␣message␣failed\n");
182      }
183      current[0] = '1';
184      current = &current[1];
185      strncpy(current, (char *)member.bsn, MAX_BSNSIZE);
186      current = &current[MAX_BSNSIZE];
187   } else {
188      if (0 != ecdaa_signature_FP256BN_sign(&sig, msg, msg_len, NULL, 0, &member.msk,
             &member.cred, ecdaa_rand)) {
189         printf("member_verifymsg:␣Signing␣message␣failed\n");
190      }
191      current[0] = '0';
192 current = &current[1];
193   }
194
195   bzero(binbuf, MAX_BUFSIZE);
196   ecdaa_signature_FP256BN_serialize(binbuf, &sig, has_nym);
197   ret = ecdaa_encode(binbuf, current, sig_len);
198   printf("member_verifymsg:␣has_nym:␣%u,␣sig_len:␣%lu\n",has_nym, sig_len);
199   printf("member_verifymsg:␣msg:␣%s,␣len:␣%lu\n",msg, msg_len);
200   printf("member_verifymsg:␣bsn:␣%s,␣len:␣%lu\n",(char *)member.bsn, strlen((char
          *)member.bsn));
201   printf("member_verifymsg:␣sig:␣%s,␣len:␣%lu\n", current, sig_len);
202
203   current[ret] = '\n';
204   return 0;
205 }
```

```
206
207 //"PUBLISH" > "PUBLISH <member.mpk>"
208 int member_publish(char *buffer) {
209    char *current;
210    int ret = 0;
211    uint8_t binbuf[MAX_BUFSIZE];
212    bzero(buffer, MAX_BUFSIZE);
213
214    strncpy(buffer, "PUBLISH ", 8);
215
216    current = &buffer[8];
217    bzero(binbuf, MAX_BUFSIZE);
218    ecdaa_member_public_key_FP256BN_serialize(binbuf, &member.mpk);
219    ret = ecdaa_encode(binbuf, current, ECDAA_MEMBER_PUBLIC_KEY_FP256BN_LENGTH);
220
221    current[ret] = '\n';
222
223    return 0;
224 }
225
226 //"JOINSTART <issuer.nonce>" > "APPEND <member.mpk>"
227 int member_joinappend(char *buffer) {
228    char *current = &buffer[10];
229    uint8_t binbuf[MAX_BUFSIZE];
230    int ret = ecdaa_decode(current, member.nonce, NONCE_SIZE);
231    ecdaa_write_buffer_to_file(member_nonce_file, member.nonce, NONCE_SIZE);
232 //   if (0 != ecdaa_member_key_pair_TPM_FP256BN_generate(&member.mpk, member.nonce,
       NONCE_SIZE)) {
233    if (0 != ecdaa_member_key_pair_FP256BN_generate(&member.mpk, &member.msk, member.
        nonce, NONCE_SIZE, ecdaa_rand)) {
234       fprintf(stderr, "Error generating member key-pair\n");
235       return -1;
236    }
237    bzero(buffer, MAX_BUFSIZE);
238    strncpy(buffer, "APPEND ", 7);
239
240    current = &buffer[7];
241    bzero(binbuf, MAX_BUFSIZE);
242    ecdaa_member_public_key_FP256BN_serialize(binbuf, &member.mpk);
243    ret = ecdaa_encode(binbuf, current, ECDAA_MEMBER_PUBLIC_KEY_FP256BN_LENGTH);
244    current[ret] = '\n';
245    return 0;
246 }
247
248 //"JOINPROCEED <member.cred><member.cred_sig>" > ""
249 int member_joinfinish(char *buffer) {
250    char *current = &buffer[12];
251    uint8_t *bincur;
252    uint8_t binbuf[MAX_BUFSIZE];
253    int ret = 0;
```

```
254     bzero(binbuf, MAX_BUFSIZE);
255     ret = ecdaa_decode(current, binbuf, ECDAA_CREDENTIAL_FP256BN_LENGTH);
256
257     current = &current[ret];
258     bincur = &binbuf[ECDAA_CREDENTIAL_FP256BN_LENGTH];
259     ecdaa_decode(current, bincur, ECDAA_CREDENTIAL_FP256BN_SIGNATURE_LENGTH);
260     ret = ecdaa_credential_FP256BN_deserialize_with_signature(&member.cred, &member.
            mpk, &member.ipk.gpk, binbuf, bincur);
261     if(-1 == ret) {
262        printf("member_joinfinish:␣credential␣is␣malformed!\n");
263        ret = -1;
264     } else if(-2 == ret) {
265        printf("member_joinfinish:␣siganture␣of␣credential␣is␣invalid\n");
266        ret = -1;
267     }
268     printf("member_joinfinish:␣writing␣key-pair␣and␣credential␣to␣disk\n");
269     if(0 != ecdaa_member_public_key_FP256BN_serialize_file(member_public_key_file, &
            member.mpk) ||
270          0 != ecdaa_member_secret_key_FP256BN_serialize_file(member_secret_key_file,
                 &member.msk) ||
271          0 != ecdaa_credential_FP256BN_serialize_file(member_credential_file, &member.
                 cred)) {
272        printf("issuer_setup:␣Error␣saving␣key-pair␣or␣credential␣to␣disk\n");
273        return -1;
274     }
275
276     return ret;
277 }
```

4 Source files for the DAA Member with TPM support

Listing 11: member-tpm.h

```
1 //
2 // Created by root on 11/5/19.
3 //
4
5 #ifndef ECDAA_MEMBER_TPM_H
6 #define ECDAA_MEMBER_TPM_H
7 #include <tss2/tss2_sys.h>
8 #include <tss2/tss2_tcti.h>
9 #include <tss2/tss2_tcti_device.h>
10 #include <ecdaa.h>
11 #include <ecdaa-tpm.h>
12 #include "server.h"
13 #include "client.h"
14 #include "common.h"
15
```

```
16 #define ECP_FP256BN_LENGTH 130
17
18 int process_member(char *buffer);
19
20 const char* member_public_key_file = "mpk.bin";
21 const char* member_credential_file = "mcred.bin";
22 const char* member_nonce_file = "mnonce.bin";
23
24 #endif //ECDAA_MEMBER_TPM_H
```

Listing 12: member-tpm.c

```
1  #include "member-tpm.h"
2
3  typedef enum memberstate {
4     ON,
5     ISSUERPUB,
6     RCVPUBLIC,
7     JOIN,
8     APPEND,
9     JOINPROCEED,
10    JOINED,
11 } memberstate_e;
12
13 typedef struct member {
14    struct ecdaa_member_public_key_FP256BN mpk;
15    memberstate_e state;
16    uint8_t nonce[NONCE_SIZE];
17    struct ecdaa_credential_FP256BN cred;
18    struct ecdaa_issuer_public_key_FP256BN ipk;
19    uint8_t bsn[MAX_BSNSIZE];
20    size_t bsn_len;
21    struct ecdaa_tpm_context ctx;
22    TPM2_HANDLE pk_handle;
23    unsigned char pk_in[ECP_FP256BN_LENGTH];
24    unsigned char tcti_buffer[256];
25 } member_t;
26
27 member_t member;
28 uint8_t msg[MAX_MSGSIZE];
29 size_t msg_len;
30 const char *pub_key_filename = "pub_key.txt";
31 const char *handle_filename = "handle.txt";
32
33 int init_tpm();
34 int free_tpm();
35 static int read_public_key_from_files(uint8_t *public_key, TPM2_HANDLE *key_handle,
       const char *pub_key_filename, const char *handle_filename);
36 int member_join(char *buffer);
37 int member_verifymsg(char *buffer);
```

```
38 int member_publish(char *buffer);
39 int member_joinappend(char *buffer);
40 int member_joinfinish(char *buffer);
41
42 int main(int argc, char *argv[]) {
43    char buffer[MAX_BUFSIZE];
44    char *remote_ip = argv[2];
45    int ret = 0;
46    TPM2_HANDLE sk_handle = 0;
47
48    //strncpy(member.bsn, "mybasename", 10);
49    //member.bsn_len = strlen(member.bsn);
50    switch(argc) {
51       case 3:
52          if( 0 == strncasecmp("--join", argv[1], 6) || 0 == strncasecmp("-j", argv[1],
                2)) {
53    if (0 != read_public_key_from_files(member.pk_in, &sk_handle, pub_key_filename,
       handle_filename)) {
54                printf("Error:␣error␣reading␣in␣public␣key␣files␣'%s'␣and␣'%s'\n",
                       pub_key_filename, handle_filename);
55       return 1;
56    }
57             member.state = ON;
58             ret = client_connect(&member_join, remote_ip, ISSUERPORT);
59             if (0 >= ret || JOINED != member.state) {
60                printf("Join␣process␣failed!\n");
61                return 1;
62             } else {
63                printf("Join␣process␣was␣successful\n");
64             }
65          } else {
66             printf("2␣arguments␣but␣not␣join\n");
67          }
68          break;
69       case 4:
70          if( 0 == strncasecmp("--send", argv[1], 6) || 0 == strncasecmp("-s", argv[1],
                2)) {
71             msg_len = ecdaa_read_from_file(msg, MAX_MSGSIZE, argv[3]);
72             if (msg_len < 0) {
73                printf("Could␣not␣open␣message␣file␣%s\n", argv[3]);
74                return 1;
75             }
76    if (0 != read_public_key_from_files(member.pk_in, &sk_handle, pub_key_filename,
       handle_filename)) {
77                printf("Error:␣error␣reading␣in␣public␣key␣files␣'%s'␣and␣'%s'\n",
                       pub_key_filename, handle_filename);
78       return 1;
79    }
80             if (0 > ecdaa_read_from_file(member.nonce, NONCE_SIZE, member_nonce_file)
                    ||
```

```
81                   0 != ecdaa_member_public_key_FP256BN_deserialize_file(&member.mpk,
                         member_public_key_file, member.nonce, NONCE_SIZE) ||
82                   0 != ecdaa_credential_FP256BN_deserialize_file(&member.cred,
                         member_credential_file)) {
83                printf("Could not import key files. importing from %s, %s or %s was
                       not successful\n",
84                   member_nonce_file, member_public_key_file, member_credential_file);
85                return 1;
86             }
87             member.state = JOINED;
88             ret = client_connect(&member_verifymsg, remote_ip, VERIFIERPORT);
89             if (0 >= ret || JOINED != member.state) {
90                printf("connection to verifier failed\n");
91             }
92          } else {
93             printf("3 arguments but not send\n");
94          }
95          break;
96       default:
97          printf("Usage: \n Join an issuer's group: %s --join <issuer's IPv4>\n",
               argv[0]);
98          printf("Send a signed message to the verifier: %s --send <verifier's IPv4
               > <msgfile>\n", argv[0]);
99          printf("Before sending a DAA-signed message, the member must join a DAA
               group\n");
100         break;
101   }
102   return 0;
103 }
104
105 //int init_tpm() {
106 // TSS2_TCTI_CONTEXT *tctiContext = NULL;
107 // member.pk_handle = 0x81010000;
108 // const char* passwd = NULL;
109 // uint16_t passwdlen = 0;
110 // TSS2_RC retval = 0;
111 // size_t bufsize = tss2_tcti_getsize_device();
112 // uint8_t tctiBuffer[bufsize];
113 // bzero(tctiBuffer, bufsize);
114 // const char* devicePath = "/dev/tpm0";
115 //
116 // tctiContext = tctiBuffer;
117 // retval = tss2_tcti_init_device(devicePath, strlen(devicePath), tctiContext);
118 // switch (retval & 0xFF) {
119 //    case TSS2_RC_SUCCESS:
120 //       printf("tcti context established\n");
121 //       break;
122 //    default:
123 //       printf("tcti context failed\n");
124 //       break;
```

```
125 // }
126 // //initialize ecdaa tpm context
127 // if(0 != ecdaa_tpm_context_init(&member.ctx, handle, passwd, passwdlen, tctiContext
        )) {
128 //    printf("\necdaa context failed\n");
129 //    return -1;
130 // }
131 // printf("\necdaa context initialized\n");
132 // return 0;
133 //}
134
135 int init_tpm()
136 {
137    const char *device_conf = "/dev/tpm0";
138
139    int init_ret;
140
141    memset(member.tcti_buffer, 0, sizeof(member.tcti_buffer));
142
143    TSS2_TCTI_CONTEXT *tcti_ctx = (TSS2_TCTI_CONTEXT*)member.tcti_buffer;
144
145    size_t size;
146    init_ret = Tss2_Tcti_Device_Init(NULL, &size, device_conf);
147    if (TSS2_RC_SUCCESS != init_ret) {
148       printf("Failed␣to␣get␣allocation␣size␣for␣tcti␣context\n");
149       return -1;
150    }
151    if (size > sizeof(member.tcti_buffer)) {
152       printf("Error:␣device␣TCTI␣context␣size␣larger␣than␣pre-allocated␣buffer\n");
153       return -1;
154    }
155    init_ret = Tss2_Tcti_Device_Init(tcti_ctx, &size, device_conf);
156    if (TSS2_RC_SUCCESS != init_ret) {
157       printf("Error:␣Unable␣to␣initialize␣device␣TCTI␣context\n");
158       return -1;
159    }
160
161    //initialize ecdaa tpm context
162    if(0 != ecdaa_tpm_context_init(&member.ctx, member.pk_handle, NULL, 0, tcti_ctx))
           {
163       printf("\necdaa␣context␣failed\n");
164       return -1;
165    }
166    printf("\necdaa␣context␣initialized\n");
167    return 0;
168 }
169
170 int free_tpm() {
171    ecdaa_tpm_context_free(&member.ctx);
172    return 0;
```

```
173 }
174
175 int member_join(char *buffer) {
176    int ret = 0;
177
178    switch (member.state) {
179       case ON:
180          bzero(buffer, MAX_BUFSIZE);
181          strncpy(buffer, "PUBLISH\n", 8);
182          member.state = ISSUERPUB;
183          break;
184       case ISSUERPUB:
185          if (0 == strncasecmp("PUBLISH", buffer, 7)) {
186             printf("ISSUER␣>␣MEMBER:␣%s", buffer);
187             uint8_t binbuf[MAX_BUFSIZE];
188             char *current = &buffer[8];
189             ecdaa_decode(current, binbuf, ECDAA_ISSUER_PUBLIC_KEY_FP256BN_LENGTH);
190             ret = ecdaa_issuer_public_key_FP256BN_deserialize(&member.ipk, binbuf);
191             if (-1 == ret) {
192                printf("member_getpublic:␣issuer␣public␣key␣is␣malformed!\n");
193                ret = -1;
194             } else if (-2 == ret) {
195                printf("member_getpublic:␣signature␣of␣issuer␣public␣key␣is␣invalid\
                       n");
196                ret = -1;
197             } else {
198                bzero(buffer, MAX_BUFSIZE);
199                strncpy(buffer, "JOIN\n", 5);
200                member.state = APPEND;
201                ret = 0;
202             }
203          } else {
204             printf("member_getpublic:␣did␣not␣get␣public␣key␣from␣issuer\n");
205             member.state = ON;
206             ret = -1;
207          }
208          break;
209       case APPEND:
210          if (0 == strncasecmp("JOINSTART", buffer, 9)) {
211             printf("ISSUER␣>␣MEMBER:␣%s", buffer);
212             member_joinappend(buffer);
213             member.state = JOINPROCEED;
214          } else {
215             printf("member_join:␣did␣not␣get␣nonce␣from␣issuer\n");
216             member.state = RCVPUBLIC;
217             ret = -1;
218          }
219          break;
220       case JOINPROCEED:
221          if (0 == strncasecmp("JOINPROCEED", buffer, 11)) {
```

```
222             printf("ISSUER_>_MEMBER:_%s", buffer);
223             member_joinfinish(buffer);
224             member.state = JOINED;
225             ret = 1;
226          } else {
227             printf("member_getpublic:_did_not_get_credentials_from_issuer\n");
228             member.state = RCVPUBLIC;
229             ret = -1;
230          }
231          break;
232       default:
233          ret = -1;
234    }
235    if (0 == ret) {
236       printf("ISSUER_<_MEMBER:_%s", buffer);
237    }
238    return ret;
239 }
240
241 //"VERIFYMSG" > "VERIFYMSG <attestval>"
242 int member_verifymsg(char *buffer) {
243    char *current = buffer;
244    uint8_t binbuf[MAX_BUFSIZE];
245    uint8_t has_nym = member.bsn_len > 0 ? 1 : 0;
246    struct ecdaa_signature_FP256BN sig;
247    size_t sig_len = has_nym ? ecdaa_signature_FP256BN_with_nym_length() :
           ecdaa_signature_FP256BN_length();
248    int ret = 0;
249
250    if (0 == strncasecmp("OK", buffer, 2)) {
251       return 1;
252    } else if (0 == strncasecmp("ERR", buffer, 3)) {
253       printf("member_verifymsg:_Verifier_refused_signature\n");
254       return 1;
255    }
256
257    bzero(buffer, MAX_BUFSIZE);
258    strncpy(current, "VERIFYMSG_", 10);
259    current = &current[10];
260
261    ret = ecdaa_encode(msg, current, msg_len);
262    current = &current[2 * MAX_MSGSIZE];
263    if(has_nym) {
264       if (0 != ecdaa_signature_TPM_FP256BN_sign(&sig, msg, msg_len, member.bsn,
              member.bsn_len, &member.cred, ecdaa_rand, &member.ctx)) {
265          printf("member_verifymsg:_Signing_message_failed\n");
266       }
267       current[0] = '1';
268       current = &current[1];
269       strncpy(current, (char *)member.bsn, MAX_BSNSIZE);
```

```
270       current = &current[MAX_BSNSIZE];
271    } else {
272       if (0 != ecdaa_signature_TPM_FP256BN_sign(&sig, msg, msg_len, NULL, 0, &member.
              cred, ecdaa_rand, &member.ctx)) {
273          printf("member_verifymsg:␣Signing␣message␣failed\n");
274       }
275       current[0] = '0';
276       current = &current[1];
277    }
278
279    bzero(binbuf, MAX_BUFSIZE);
280    ecdaa_signature_FP256BN_serialize(binbuf, &sig, has_nym);
281    ret = ecdaa_encode(binbuf, current, sig_len);
282    printf("member_verifymsg:␣has_nym:␣%u,␣sig_len:␣%lu\n",has_nym, sig_len);
283    printf("member_verifymsg:␣msg:␣%s,␣len:␣%lu\n",msg, msg_len);
284    printf("member_verifymsg:␣bsn:␣%s,␣len:␣%lu\n",(char *)member.bsn, strlen((char
           *)member.bsn));
285    printf("member_verifymsg:␣sig:␣%s,␣len:␣%lu\n", current, sig_len);
286
287    current[ret] = '\n';
288    return 0;
289 }
290
291 //"PUBLISH" > "PUBLISH <member.mpk>"
292 int member_publish(char *buffer) {
293    char *current;
294    int ret = 0;
295    uint8_t binbuf[MAX_BUFSIZE];
296    bzero(buffer, MAX_BUFSIZE);
297
298    strncpy(buffer, "PUBLISH␣", 8);
299
300    current = &buffer[8];
301    bzero(binbuf, MAX_BUFSIZE);
302    ecdaa_member_public_key_FP256BN_serialize(binbuf, &member.mpk);
303    ret = ecdaa_encode(binbuf, current, ECDAA_MEMBER_PUBLIC_KEY_FP256BN_LENGTH);
304
305    current[ret] = '\n';
306
307    return 0;
308 }
309
310 //"JOINSTART <issuer.nonce>" > "APPEND <member.mpk>"
311 int member_joinappend(char *buffer) {
312    char *current = &buffer[10];
313    uint8_t binbuf[MAX_BUFSIZE];
314    int ret = ecdaa_decode(current, member.nonce, NONCE_SIZE);
315    ecdaa_write_buffer_to_file(member_nonce_file, member.nonce, NONCE_SIZE);
316    if (0 != ecdaa_member_key_pair_TPM_FP256BN_generate(&member.mpk, member.pk_in,
           member.nonce, NONCE_SIZE, &member.ctx)) {
```

```
317       fprintf(stderr, "Error␣generating␣member␣key-pair\n");
318       return -1;
319    }
320    bzero(buffer, MAX_BUFSIZE);
321    strncpy(buffer, "APPEND␣", 7);
322
323    current = &buffer[7];
324    bzero(binbuf, MAX_BUFSIZE);
325    ecdaa_member_public_key_FP256BN_serialize(binbuf, &member.mpk);
326    ret = ecdaa_encode(binbuf, current, ECDAA_MEMBER_PUBLIC_KEY_FP256BN_LENGTH);
327    current[ret] = '\n';
328    return 0;
329 }
330
331 //"JOINPROCEED <member.cred><member.cred_sig>" > ""
332 int member_joinfinish(char *buffer) {
333    char *current = &buffer[12];
334    uint8_t *bincur;
335    uint8_t binbuf[MAX_BUFSIZE];
336    int ret = 0;
337    bzero(binbuf, MAX_BUFSIZE);
338    ret = ecdaa_decode(current, binbuf, ECDAA_CREDENTIAL_FP256BN_LENGTH);
339
340    current = &current[ret];
341    bincur = &binbuf[ECDAA_CREDENTIAL_FP256BN_LENGTH];
342    ecdaa_decode(current, bincur, ECDAA_CREDENTIAL_FP256BN_SIGNATURE_LENGTH);
343    ret = ecdaa_credential_FP256BN_deserialize_with_signature(&member.cred, &member.
           mpk, &member.ipk.gpk, binbuf, bincur);
344    if(-1 == ret) {
345       printf("member_joinfinish:␣credential␣is␣malformed!\n");
346       ret = -1;
347    } else if(-2 == ret) {
348       printf("member_joinfinish:␣siganture␣of␣credential␣is␣invalid\n");
349       ret = -1;
350    }
351    printf("member_joinfinish:␣writing␣key-pair␣and␣credential␣to␣disk\n");
352    if(0 != ecdaa_member_public_key_FP256BN_serialize_file(member_public_key_file, &
           member.mpk) ||
353          0 != ecdaa_credential_FP256BN_serialize_file(member_credential_file, &member.
                 cred)) {
354       printf("issuer_setup:␣Error␣saving␣key-pair␣or␣credential␣to␣disk\n");
355       return -1;
356    }
357
358    return ret;
359 }
360
361 int read_public_key_from_files(uint8_t *public_key, TPM2_HANDLE *key_handle, const
       char *pub_key_filename, const char *handle_filename)
362 {
```

```
363     int ret = 0;
364
365     FILE *pub_key_file_ptr = fopen(pub_key_filename, "r");
366     if (NULL == pub_key_file_ptr)
367        return -1;
368     do {
369        for (unsigned i=0; i < ECP_FP256BN_LENGTH; i++) {
370           unsigned byt;
371           if (fscanf(pub_key_file_ptr, "%02X", &byt) != 1) {
372              ret = -1;
373              break;
374           }
375           public_key[i] = (uint8_t)byt;
376        }
377     } while(0);
378     (void)fclose(pub_key_file_ptr);
379     if (0 != ret)
380        return -1;
381
382     FILE *handle_file_ptr = fopen(handle_filename, "r");
383     if (NULL == handle_file_ptr)
384        return -1;
385     do {
386        for (int i=(sizeof(TPM2_HANDLE)-1); i >= 0; i--) {
387           unsigned byt;
388           if (fscanf(handle_file_ptr, "%02X", &byt) != 1) {
389              ret = -1;
390              break;
391           }
392           *key_handle += byt<<(i*8);
393        }
394        if (0 != ret)
395           break;
396     } while(0);
397     (void)fclose(handle_file_ptr);
398
399     return ret;
400 }
```

Listing 13: `daa-test-tpm.`

```
1
2 #ifndef ECDAA_TESTTPM_H
3 #define ECDAA_TESTTPM_H
4 #include <ecdaa.h>
5 #include "server.h"
6 #include "client.h"
7 #include "common.h"
8 #include <string.h>
9 #include <stdlib.h>
```

```
10
11 #include <ecdaa-tpm.h>
12 #include <tss2/tss2_sys.h>
13 #include <tss2/tss2_tcti.h>
14 #include <tss2/tss2_tcti_device.h>
15 #endif
```

Listing 14: `create_tpm_key-util.c`

```
1 /*******************************************************************************
2  *
3  * Copyright 2020 Xaptum, Inc.
4  *
5  *   Licensed under the Apache License, Version 2.0 (the "License");
6  *   you may not use this file except in compliance with the License.
7  *   You may obtain a copy of the License at
8  *
9  *      http://www.apache.org/licenses/LICENSE-2.0
10 *
11 *   Unless required by applicable law or agreed to in writing, software
12 *   distributed under the License is distributed on an "AS IS" BASIS,
13 *   WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
14 *   See the License for the specific language governing permissions and
15 *   limitations under the License
16 *
17 *******************************************************************************/
18
19 #include "daa-test-tpm.h"
20
21 static TPMA_SESSION empty_session_attributes = {0}; // attributes for password either
       can't be set or don't make sense
22 const char *pub_key_filename = "pub_key.txt";
23 const char *handle_filename = "handle.txt";
24
25 void parse_cmd_args(int argc, char *argv[]) {
26    if (3 != argc) {
27       printf("usage:␣%s␣<public␣key␣output␣file>␣<handle␣output␣file>\n", argv[0]);
28       exit(1);
29    }
30
31    pub_key_filename = argv[1];
32    handle_filename = argv[2];
33    printf("Saving␣public␣key␣to␣%s␣and␣handle␣to␣%s\n", pub_key_filename,
          handle_filename);
34 }
35
36 struct test_context {
37    TSS2_SYS_CONTEXT *sapi_ctx;
38    TPM2_HANDLE primary_key_handle;
39    TPM2_HANDLE signing_key_handle;
```

```
40    TPM2_HANDLE persistent_key_handle;
41    TPM2B_PUBLIC out_public;
42    TPM2B_PRIVATE out_private;
43    unsigned char tcti_buffer[256];
44    unsigned char sapi_buffer[4200];
45
46 };
47
48 static void initialize(struct test_context *ctx);
49 static void cleanup(struct test_context *ctx);
50
51 static void create_key(const char* pub_key_filename, const char* handle_filename);
52 static int clear(struct test_context *ctx);
53 static int create_primary(struct test_context *ctx);
54 static int create(struct test_context *ctx);
55 static int load(struct test_context *ctx);
56 static int save_public_key_info(const struct test_context* ctx, const char*
       pub_key_filename, const char* handle_filename);
57 static int evict_control(struct test_context *ctx);
58
59 int main(int argc, char *argv[])
60 {
61    // Included in the utils header, but we don't need them.
62 //   (void)tpm_initialize;
63 //   (void)tpm_cleanup;
64
65 // parse_cmd_args(argc, argv);
66
67    printf("Saving public key to %s and handle to %s\n", pub_key_filename,
           handle_filename);
68    create_key(pub_key_filename, handle_filename);
69 }
70
71 void initialize(struct test_context *ctx)
72 {
73    const char *device_conf = "/dev/tpm0";
74
75    int init_ret;
76
77    memset(ctx->tcti_buffer, 0, sizeof(ctx->tcti_buffer));
78    memset(ctx->sapi_buffer, 0, sizeof(ctx->sapi_buffer));
79
80    TSS2_TCTI_CONTEXT *tcti_ctx = (TSS2_TCTI_CONTEXT*)ctx->tcti_buffer;
81
82    size_t size;
83    init_ret = Tss2_Tcti_Device_Init(NULL, &size, device_conf);
84    if (TSS2_RC_SUCCESS != init_ret) {
85       printf("Failed to get allocation size for tcti context\n");
86       exit(1);
87    }
```

```
88    if (size > sizeof(ctx->tcti_buffer)) {
89       printf("Error:_device_TCTI_context_size_larger_than_pre-allocated_buffer\n");
90       exit(1);
91    }
92    init_ret = Tss2_Tcti_Device_Init(tcti_ctx, &size, device_conf);
93    if (TSS2_RC_SUCCESS != init_ret) {
94       printf("Error:_Unable_to_initialize_device_TCTI_context\n");
95       exit(1);
96    }
97
98    ctx->sapi_ctx = (TSS2_SYS_CONTEXT*)ctx->sapi_buffer;
99    size_t sapi_ctx_size = Tss2_Sys_GetContextSize(0);
100
101   TSS2_ABI_VERSION abi_version = TSS2_ABI_VERSION_CURRENT;
102   init_ret = Tss2_Sys_Initialize(ctx->sapi_ctx,
103                           sapi_ctx_size,
104                           tcti_ctx,
105                           &abi_version);
106
107   ctx->out_public.size = 0;
108   ctx->out_private.size = 0;
109 }
110
111 void cleanup(struct test_context *ctx)
112 {
113    TSS2_TCTI_CONTEXT *tcti_context = NULL;
114    TSS2_RC rc;
115
116    if (ctx->sapi_ctx != NULL) {
117       rc = Tss2_Sys_GetTctiContext(ctx->sapi_ctx, &tcti_context);
118
119       Tss2_Tcti_Finalize(tcti_context);
120
121       Tss2_Sys_Finalize(ctx->sapi_ctx);
122    }
123 }
124
125 void create_key(const char* pub_key_filename, const char* handle_filename)
126 {
127    struct test_context ctx;
128    initialize(&ctx);
129
130    int ret = 0;
131
132 // ret = clear(&ctx);
133 // if(ret != TSS2_RC_SUCCESS) {
134 //    printf("TPM Clear failed: %x\n",ret);
135 // }
136
137    ret = create_primary(&ctx);
```

```
138    if(ret != TSS2_RC_SUCCESS) {
139       printf("TPM_Create_Primary_failed:_%x\n",ret);
140    }
141
142    ret = create(&ctx);
143    if(ret != TSS2_RC_SUCCESS) {
144       printf("TPM_Create_failed:_%x\n",ret);
145    }
146
147    ret = load(&ctx);
148    if(ret != TSS2_RC_SUCCESS) {
149       printf("TPM_Load_failed:_%x\n",ret);
150    }
151
152    ret = evict_control(&ctx);
153    if(ret != TSS2_RC_SUCCESS) {
154       printf("TPM_Evict_Control_failed:_%x\n",ret);
155    }
156
157    ret = save_public_key_info(&ctx, pub_key_filename, handle_filename);
158    if(ret != TSS2_RC_SUCCESS) {
159       printf("Save_Public_Key_Info_failed:_%x\n",ret);
160    }
161
162    cleanup(&ctx);
163 }
164
165 int save_public_key_info(const struct test_context *ctx, const char* pub_key_filename,
       const char* handle_filename)
166 {
167    int write_ret = 0;
168
169    FILE *pub_key_file_ptr = fopen(pub_key_filename, "w");
170    if (NULL == pub_key_file_ptr)
171       return -1;
172    do {
173       if (fprintf(pub_key_file_ptr, "%02X", 4) != 2)
174          break;
175
176       for (unsigned i=0; i < ctx->out_public.publicArea.unique.ecc.x.size; i++) {
177          if (fprintf(pub_key_file_ptr, "%02X", ctx->out_public.publicArea.unique.ecc.
                x.buffer[i]) != 2) {
178             write_ret = -1;
179             break;
180          }
181       }
182       if (0 != write_ret)
183          break;
184
185       for (unsigned i=0; i < ctx->out_public.publicArea.unique.ecc.y.size; i++) {
```

```
186          if (fprintf(pub_key_file_ptr, "%02X", ctx->out_public.publicArea.unique.ecc.
                 y.buffer[i]) != 2) {
187             write_ret = -1;
188             break;
189          }
190       }
191       if (0 != write_ret)
192          break;
193    } while(0);
194    (void)fclose(pub_key_file_ptr);
195
196    (void)handle_filename;
197    FILE *handle_file_ptr = fopen(handle_filename, "w");
198    if (NULL == handle_file_ptr)
199       return -1;
200    write_ret = 0;
201    do {
202       for (int i=(sizeof(ctx->persistent_key_handle)-1); i >= 0; i--) {
203          if (fprintf(handle_file_ptr, "%02X", (ctx->persistent_key_handle >> i*8) & 0
                 xFF) != 2) {
204             write_ret = -1;
205             break;
206          }
207       }
208       if (0 != write_ret)
209          break;
210    } while(0);
211    (void)fclose(handle_file_ptr);
212
213    return write_ret;
214 }
215
216 int clear(struct test_context *ctx)
217 {
218    TPMI_RH_CLEAR auth_handle = TPM2_RH_LOCKOUT;
219
220    TSS2L_SYS_AUTH_COMMAND sessionsData = {};
221    sessionsData.auths[0].sessionHandle = TPM2_RS_PW;
222    sessionsData.auths[0].sessionAttributes = empty_session_attributes;
223    sessionsData.count = 1;
224
225    TSS2L_SYS_AUTH_RESPONSE sessionsDataOut = {};
226    sessionsDataOut.count = 1;
227
228    TSS2_RC ret = Tss2_Sys_Clear(ctx->sapi_ctx,
229                          auth_handle,
230                          &sessionsData,
231                          &sessionsDataOut);
232
233    printf("Clear␣ret=%#X\n", ret);
```

```
234
235     return ret;
236 }
237
238 int create_primary(struct test_context *ctx)
239 {
240     TPMI_RH_HIERARCHY hierarchy = TPM2_RH_ENDORSEMENT;
241
242     TSS2L_SYS_AUTH_COMMAND sessionsData = {};
243     sessionsData.auths[0].sessionHandle = TPM2_RS_PW;
244     sessionsData.auths[0].sessionAttributes = empty_session_attributes;
245     sessionsData.count = 1;
246
247     TSS2L_SYS_AUTH_RESPONSE sessionsDataOut = {};
248     sessionsDataOut.count = 1;
249
250     TPM2B_SENSITIVE_CREATE inSensitive = {};
251
252     TPM2B_PUBLIC in_public = {};
253     in_public.publicArea.type = TPM2_ALG_ECC;
254     in_public.publicArea.nameAlg = TPM2_ALG_SHA256;
255     in_public.publicArea.objectAttributes = TPMA_OBJECT_FIXEDTPM |
256        TPMA_OBJECT_FIXEDPARENT |
257        TPMA_OBJECT_SENSITIVEDATAORIGIN |
258        TPMA_OBJECT_USERWITHAUTH |
259        TPMA_OBJECT_DECRYPT |
260        TPMA_OBJECT_RESTRICTED;
261     in_public.publicArea.parameters.eccDetail.symmetric.algorithm = TPM2_ALG_AES;
262     in_public.publicArea.parameters.eccDetail.symmetric.keyBits.aes = 128;
263     in_public.publicArea.parameters.eccDetail.symmetric.mode.sym = TPM2_ALG_CFB;
264     in_public.publicArea.parameters.eccDetail.scheme.scheme = TPM2_ALG_NULL;
265     in_public.publicArea.parameters.eccDetail.curveID = TPM2_ECC_NIST_P256;
266     in_public.publicArea.parameters.eccDetail.kdf.scheme = TPM2_ALG_NULL;
267
268     TPM2B_DATA outsideInfo = {};
269
270     TPML_PCR_SELECTION creationPCR = {};
271
272     TPM2B_CREATION_DATA creationData = {};
273     TPM2B_DIGEST creationHash = {};
274     TPMT_TK_CREATION creationTicket = {};
275
276     TPM2B_NAME name = {};
277
278     TPM2B_PUBLIC public_key = {};
279
280     TSS2_RC ret = Tss2_Sys_CreatePrimary(ctx->sapi_ctx,
281                                 hierarchy,
282                                 &sessionsData,
283                                 &inSensitive,
```

```
284                                 &in_public,
285                                 &outsideInfo,
286                                 &creationPCR,
287                                 &ctx->primary_key_handle,
288                                 &public_key,
289                                 &creationData,
290                                 &creationHash,
291                                 &creationTicket,
292                                 &name,
293                                 &sessionsDataOut);
294
295     printf("CreatePrimary␣ret=%#X\n", ret);
296
297     return ret;
298 }
299
300 int create(struct test_context *ctx)
301 {
302     TSS2L_SYS_AUTH_COMMAND sessionsData = {};
303     sessionsData.auths[0].sessionHandle = TPM2_RS_PW;
304     sessionsData.auths[0].sessionAttributes = empty_session_attributes;
305     sessionsData.count = 1;
306
307     TSS2L_SYS_AUTH_RESPONSE sessionsDataOut = {};
308     sessionsDataOut.count = 1;
309
310     TPM2B_SENSITIVE_CREATE inSensitive = {};
311
312     TPM2B_PUBLIC in_public = {};
313     in_public.publicArea.type = TPM2_ALG_ECC;
314     in_public.publicArea.nameAlg = TPM2_ALG_SHA256;
315     in_public.publicArea.objectAttributes = TPMA_OBJECT_FIXEDTPM |
316        TPMA_OBJECT_FIXEDPARENT |
317        TPMA_OBJECT_SENSITIVEDATAORIGIN |
318        TPMA_OBJECT_USERWITHAUTH |
319        TPMA_OBJECT_SIGN_ENCRYPT;
320     in_public.publicArea.parameters.eccDetail.symmetric.algorithm = TPM2_ALG_NULL;
321     in_public.publicArea.parameters.eccDetail.scheme.scheme = TPM2_ALG_ECDAA;
322     in_public.publicArea.parameters.eccDetail.scheme.details.ecdaa.hashAlg =
          TPM2_ALG_SHA256;
323     in_public.publicArea.parameters.eccDetail.scheme.details.ecdaa.count = 1;
324     in_public.publicArea.parameters.eccDetail.curveID = TPM2_ECC_BN_P256;
325     in_public.publicArea.parameters.eccDetail.kdf.scheme = TPM2_ALG_NULL;
326
327     TPM2B_DATA outsideInfo = {};
328
329     TPML_PCR_SELECTION creationPCR = {};
330
331     TPM2B_CREATION_DATA creationData = {};
332     TPM2B_DIGEST creationHash = {};
```

```
333     TPMT_TK_CREATION creationTicket = {};
334
335     TSS2_RC ret = Tss2_Sys_Create(ctx->sapi_ctx,
336                             ctx->primary_key_handle,
337                             &sessionsData,
338                             &inSensitive,
339                             &in_public,
340                             &outsideInfo,
341                             &creationPCR,
342                             &ctx->out_private,
343                             &ctx->out_public,
344                             &creationData,
345                             &creationHash,
346                             &creationTicket,
347                             &sessionsDataOut);
348
349     printf("Create␣ret=%#X\n", ret);
350
351     return ret;
352 }
353
354 int load(struct test_context *ctx)
355 {
356     TSS2L_SYS_AUTH_COMMAND sessionsData = {};
357     sessionsData.auths[0].sessionHandle = TPM2_RS_PW;
358     sessionsData.auths[0].sessionAttributes = empty_session_attributes;
359     sessionsData.count = 1;
360
361     TSS2L_SYS_AUTH_RESPONSE sessionsDataOut = {};
362     sessionsDataOut.count = 1;
363
364     TPM2B_NAME name = {};
365
366     int ret = Tss2_Sys_Load(ctx->sapi_ctx,
367                         ctx->primary_key_handle,
368                         &sessionsData,
369                         &ctx->out_private,
370                         &ctx->out_public,
371                         &ctx->signing_key_handle,
372                         &name,
373                         &sessionsDataOut);
374
375     printf("Load␣ret=%#X\n", ret);
376
377     return ret;
378 }
379
380 int evict_control(struct test_context *ctx)
381 {
382     TSS2L_SYS_AUTH_COMMAND sessionsData = {};
```

```
383     sessionsData.auths[0].sessionHandle = TPM2_RS_PW;
384     sessionsData.auths[0].sessionAttributes = empty_session_attributes;
385     sessionsData.count = 1;
386
387     TSS2L_SYS_AUTH_RESPONSE sessionsDataOut = {};
388     sessionsDataOut.count = 1;
389
390     ctx->persistent_key_handle = 0x81010000;
391
392     TSS2_RC ret = Tss2_Sys_EvictControl(ctx->sapi_ctx,
393                                 TPM2_RH_OWNER,
394                                 ctx->signing_key_handle,
395                                 &sessionsData,
396                                 ctx->persistent_key_handle,
397                                 &sessionsDataOut);
398
399     printf("EvictControl ret=%#X\n", ret);
400
401     return ret;
402 }
```

5 Source files for the DAA Verifier

Listing 15: `verifier.h`

```
1
2 #ifndef ECDAA_VERIFIER_H
3 #define ECDAA_VERIFIER_H
4 #include <ecdaa.h>
5 // #include <ecdaa-tpm.h>
6 #include "server.h"
7 #include "client.h"
8 #include "common.h"
9
10 int process_verifier(char *buffer);
11 const char* verifier_message_file = "vmsg.txt";
12
13 #endif //ECDAA_VERIFIER_H
```

Listing 16: `verifier.c`

```
1 #include "verifier.h"
2
3 typedef enum verifierstate {
4     ON,
5     ASKISSUER,
6     GOTISSUER,
7     ASKATTEST,
8 } verifierstate_e;
```

```
9
10 typedef struct verifier {
11    struct ecdaa_issuer_public_key_FP256BN ipk;
12 //   struct ecdaa_member_public_key_FP256BN mpk;
13    struct ecdaa_revocations_FP256BN revocations;
14    verifierstate_e state;
15 } verifier_t;
16
17 verifier_t verifier;
18
19 int verifier_getissuer(char *buffer);
20
21 //int verifier_getmember(char *buffer);
22 int verifier_attestmember(char *buffer);
23
24 int verifier_checklink(char *buffer);
25
26 int verifier_checkattest(char *buffer);
27
28 int main() {
29    verifier.revocations.sk_list = NULL;
30    verifier.revocations.bsn_list = NULL;
31
32    if (2 != server_start(&process_verifier, VERIFIERPORT)) {
33       printf("server␣failed\n");
34    }
35    return 0;
36 }
37
38 int process_verifier(char *buffer) {
39    int ret = 0;
40    char remote_ip[16];
41
42    printf(">␣VERIFIER:␣%s\n", buffer);
43
44    if (0 == strncasecmp("VERIFYMSG", buffer, 9)) {
45       switch (verifier.state) {
46          case GOTISSUER:
47             ret = verifier_checkattest(&buffer[10]);
48             bzero(buffer, MAX_BUFSIZE);
49             if (-1 == ret) {
50                printf("process_verifier:␣member␣public␣key␣is␣malformed!\n");
51                strncpy(buffer, "ERR\n", 4);
52             } else if (-2 == ret) {
53                printf("process_verifier:␣signature␣of␣member␣public␣key␣is␣invalid\
                      n");
54                strncpy(buffer, "ERR\n", 4);
55             } else {
56                strncpy(buffer, "OK\n", 3);
57             }
```

```
58              break;
59           default:
60              bzero(buffer, MAX_BUFSIZE);
61              strncpy(buffer, "ERR\n", 4);
62        }
63  ret = 0;
64    } else if (0 == strncasecmp("ATTEST", buffer, 6)) {
65        strncpy(remote_ip, &buffer[7], 15);
66        ret = client_connect(&verifier_attestmember, remote_ip, MEMBERPORT);
67        if (0 >= ret) {
68           printf("process_verifier:␣member␣verification␣failed\n");
69           bzero(buffer, MAX_BUFSIZE);
70           strncpy(buffer, "ERR\n", 4);
71        } else {
72           bzero(buffer, MAX_BUFSIZE);
73           strncpy(buffer, "OK\n", 3);
74        }
75        ret = 0;
76    } else if (0 == strncasecmp("LINK", buffer, 4)) {
77        bzero(buffer, MAX_BUFSIZE);
78        verifier_checklink(buffer);
79    } else if (0 == strncasecmp("GETPUBLIC", buffer, 9)) {
80        verifier.state = ON;
81        int iplen = strlen(&buffer[10]);
82  if (iplen >= 7 && iplen <= 15) {
83           strncpy(remote_ip, &buffer[10], 15);
84           ret = client_connect(&verifier_getissuer, remote_ip, ISSUERPORT);
85           if (0 >= ret || GOTISSUER != verifier.state) {
86              printf("process_verifier:␣issuer␣connection␣failed\n");
87              bzero(buffer, MAX_BUFSIZE);
88              strncpy(buffer, "ERR\n", 4);
89           } else {
90              bzero(buffer, MAX_BUFSIZE);
91              strncpy(buffer, "OK\n", 3);
92           }
93  } else {
94           printf("process_verifier:␣no␣valid␣ip\n");
95           bzero(buffer, MAX_BUFSIZE);
96           strncpy(buffer, "ERR\n", 4);
97  }
98        ret = 0;
99    } else if (0 == strncasecmp("EXIT", buffer, 4)) {
100       printf("exit()\n");
101       bzero(buffer, MAX_BUFSIZE);
102       strncpy(buffer, "OK\n", 3);
103       ret = 1;
104   } else if (0 == strncasecmp("SHUTDOWN", buffer, 8)) {
105       bzero(buffer, MAX_BUFSIZE);
106       strncpy(buffer, "OK\n", 3);
107       ret = 2;
```

```
108    } else {
109       printf("error()\n");
110       bzero(buffer, MAX_BUFSIZE);
111       strncpy(buffer, "ERR\n", 4);
112       ret = 0;
113    }
114
115    printf("<_VERIFIER:_%s\n", buffer);
116    return ret;
117 }
118
119 //"GETPUBLIC <IPv4>" > "OK"
120 int verifier_getissuer(char *buffer) {
121    int ret = 0;
122
123    switch (verifier.state) {
124       case ON:
125          bzero(buffer, MAX_BUFSIZE);
126          strncpy(buffer, "PUBLISH\n", 8);
127          verifier.state = ASKISSUER;
128          break;
129       case ASKISSUER:
130          if (0 == strncasecmp("PUBLISH", buffer, 7)) {
131             printf("ISSUER_>_VERIFIER:_%s", buffer);
132             uint8_t binbuf[MAX_BUFSIZE];
133             char *current = &buffer[8];
134             ecdaa_decode(current, binbuf, ECDAA_ISSUER_PUBLIC_KEY_FP256BN_LENGTH);
135             ret = ecdaa_issuer_public_key_FP256BN_deserialize(&verifier.ipk, binbuf);
136             if (-1 == ret) {
137                printf("verifier_getpublic:_issuer_public_key_is_malformed!\n");
138                ret = -1;
139             } else if (-2 == ret) {
140                printf("verifier_getpublic:_signature_of_issuer_public_key_is_
                       invalid\n");
141                ret = -1;
142             } else {
143                verifier.state = GOTISSUER;
144                ret = 1;
145             }
146          } else {
147             printf("verifier_getpublic:_did_not_get_public_key_from_issuer\n");
148             verifier.state = ON;
149             ret = -1;
150          }
151          break;
152       default:
153          ret = -1;
154    }
155    if (0 == ret) {
156       printf("ISSUER_<_VERIFIER:_%s", buffer);
```

```
157    }
158    return ret;
159 }
160
161
162 int verifier_attestmember(char *buffer) {
163    int ret = 0;
164
165    switch (verifier.state) {
166       case GOTISSUER:
167          bzero(buffer, MAX_BUFSIZE);
168          strncpy(buffer, "ATTEST\n", 7);
169          verifier.state = ASKATTEST;
170          break;
171       case ASKATTEST:
172          if (0 == strncasecmp("ATTEST", buffer, 6)) {
173             printf("MEMBER > VERIFIER: %s", buffer);
174             ret = verifier_checkattest(&buffer[7]);
175             if (-1 == ret) {
176                printf("verifier_attestmember: group public key is malformed!\n");
177                ret = -1;
178             } else if (-2 == ret) {
179                printf("verifier_attestmember: signature of group public key is
                       invalid\n");
180                ret = -1;
181             } else {
182                verifier.state = GOTISSUER;
183                ret = 1;
184             }
185          } else {
186             printf("verifier_attestmember: did not get correct message from member\
                       n");
187             ret = -1;
188          }
189          break;
190       default:
191          ret = -1;
192    }
193    if (0 == ret) {
194       printf("MEMBER < VERIFIER: %s", buffer);
195    }
196    return ret;
197 }
198
199 //"ATTEST <msg>0<signature w/o bsn>" or
200 //"ATTEST <msg>1<signature with bsn>"
201 int verifier_checkattest(char *buffer) {
202    char *current = buffer;
203    char msg[MAX_MSGSIZE];
204    size_t msg_len = 0;
```

```
205    int has_nym = 0;
206    char bsn[MAX_BSNSIZE];
207    size_t bsn_len = 0;
208    uint8_t binbuf[MAX_BUFSIZE];
209    size_t sig_len = 0;
210    struct ecdaa_signature_FP256BN sig;
211    int ret = 0;
212
213    bzero(msg, MAX_MSGSIZE);
214    ret = ecdaa_decode(current, msg, MAX_MSGSIZE);
215    msg_len = strlen(msg);
216    current = &current[2 * MAX_MSGSIZE];
217    has_nym = current[0] - '0';
218    current = &current[1];
219
220    if (has_nym) {
221       bzero(bsn, MAX_BSNSIZE);
222       strncpy(bsn, current, MAX_BSNSIZE);
223       bsn_len = strlen(bsn);
224       current = &current[MAX_BSNSIZE];
225   sig_len = ecdaa_signature_FP256BN_with_nym_length();
226    } else {
227
228   sig_len = ecdaa_signature_FP256BN_length();
229
230    }
231    bzero(binbuf, MAX_BUFSIZE);
232    ecdaa_decode(current, binbuf, sig_len);
233
234    ret = ecdaa_signature_FP256BN_deserialize(&sig, binbuf, has_nym);
235    if (0 != ret) {
236       printf("verifier_checkattest:_error_reading_signature\n");
237       return -1;
238    }
239
240    printf("verifier_checkattest:_msg:_%s,_len:_%lu\n",msg, msg_len);
241    printf("verifier_checkattest:_bsn:_%s,_len:_%lu\n",bsn, bsn_len);
242    printf("verifier_checkattest:_sig:_%s,_len:_%lu\n", current, sig_len);
243    ret = ecdaa_signature_FP256BN_verify(&sig, &verifier.ipk.gpk, &verifier.
           revocations, (uint8_t *) msg, msg_len,
244          (uint8_t *) bsn, bsn_len);
245    if (0 != ret) {
246       printf("verifier_checkattest:_signature_not_valid,_ret_=_%i\n", ret);
247       return -1;
248    }
249
250    printf("writing_message_to_%s\n", verifier_message_file);
251    ecdaa_write_buffer_to_file(verifier_message_file, msg, msg_len);
252    return 0;
253 }
```

```
254
255 //"LINK" > "NOT IMPLEMENTED"
256 int verifier_checklink(char *buffer) {
257    strncat(buffer, "NOT_IMPLEMENTED\n", 17);
258    return 0;
259 }
```