JOHANNES KEPLER
UNIVERSITY LINZ

Author
Maximilian Arthofer, BSc
12006655

Submission
**Institute of
Networks and Security**

Thesis Supervisor
Univ.-Prof. DI Dr.
René Mayrhofer

Assistant Thesis
Supervisor
Dr. **Michael Roland**
DI Dr. **Tobias Höller**

December 2025

Secure Embedded Sensor Systems with Remote Attestation

Master's Thesis

to confer the academic degree of

Diplom-Ingenieur

in the Master's Program

Computer Science

**JOHANNES KEPLER
UNIVERSITY LINZ**
Altenberger Straße 69
4040 Linz, Austria
jku.at

Abstract

With the most populous countries on the planet relying on biometric identity systems (e.g. China, India) and a push towards digital identities within the EU (e.g. EUID), it is important to consider how and where the data linked to these digital identities is handled. Centralized sets of identifying data lend themselves to abuse, necessitating their decentralized storage. This is exactly what Project Digidow aims to provide: a distributed, scalable biometric authentication system. To offer its services, Digidow relies on a fleet of sensors which users have to present their biometric profiles to. Implicitly trusting that these sensors are trustworthy is potentially dangerous because sensitive data could be passed to malicious actors. This thesis aims to bridge this gap, by enabling the sensors to perform remote attestation. Using this process, they gain the capability of providing evidence to users and their agents, that they are currently in a known state. Based on this, sensor manufacturers can provide reference values for trustworthy sensor states, which users can then use to derive trust from the evidence they received. This thesis shows how attestation can be done using off-the-shelf hardware like a Raspberry Pi, while also highlighting the inherent limitations of such an implementation and how to potentially overcome them. The practical output provides all the necessary code changes and additions to existing Digidow components to run such an attestation in a secure and trustworthy manner.

Kurzfassung

Da die Verwaltung der bevölkerungsreichsten Länder der Welt zu Teilen bereits auf biometrischen Identitäten (z.B. China, Indien) basiert und es auch innerhalb der EU Anstrengungen zur Einführung von digitalen Identitäten (z.B. EUID) gibt, ist es wichtig sich mit der Frage auseinanderzusetzen wie und wo diese hochsensiblen Daten gespeichert und verarbeitet werden. Zentralisierte Datenbanken eröffnen Wege diese zu missbrauchen, was dezentralisierte Speicherungsmethoden notwendig macht. Hier setzt das Projekt Digidow an und bietet ein verteiltes und skalierbares biometrisches Authentifizierungssystem. Um dies zu ermöglichen, verwendet es ein Netz aus Sensoren, welchen die Benutzer ihr biometrisches Profil präsentieren müssen. Den Sensoren blind zu vertrauen birgt Risiken, da nicht ohne Weiteres ausgeschlossen werden kann, dass Dritte Daten an modifizierten Sensoren abgreifen. Die vorliegende Arbeit versucht diese Lücke zu schließen, indem sie den Sensoren die Möglichkeit eröffnet "Remote Attestation" zu betreiben. Dadurch erhalten diese die Fähigkeit, Beweise für ihren aktuellen Zustand zu liefern. Basierend darauf können Hersteller Referenzdaten verbreiten, was es Nutzern dann erlaubt aus den von den Sensoren zur Verfügung gestellten Daten Vertrauen abzuleiten. Diese Arbeit zeigt wie sich "Remote Attestation" mit leicht zugänglicher Hardware wie einem Raspberry Pi umsetzen lässt, welche Probleme dabei auftreten können und wie man diese potenziell löst. Der praktische Teil beeinhaltet alle notwendigen Code Änderungen in bestehenden Digidow Komponenten, um den Prozess der "Remote Attestation" zu ermöglichen.

Acknowledgements

This work has been carried out within the scope of Digidow, the Christian Doppler Laboratory for Private Digital Authentication in the Physical World. We gratefully acknowledge financial support by the Austrian Federal Ministry of Economy, Energy and Tourism, the National Foundation for Research, Technology and Development, the Christian Doppler Research Association, 3 Banken IT GmbH, ekey biometric systems GmbH, Kepler Universitätsklinikum GmbH, NXP Semiconductors Austria GmbH & Co KG, and Österreichische Staatsdruckerei GmbH.

Contents

Chapter 1

Introduction

1.1 Motivation

As more and more of the interactions between a countries government and its citizens are digitalized [5, 31], digital identities are quickly gaining significance. These hold a user's identifying information in a digital wallet to present them for authentication, just like one would be using a traditional ID. Of all the different forms these identifying datasets can take, biometric profiles are especially interesting, as they provide a number of major advantages over traditional password based authentication [38]:

1. **Convenience** - one does not have to bring an ID along.
2. **Security** - chosen biometric modalities are hard to steal or fake.
3. **Liability** - as the features in use are part of human beings, they cannot be transferred, making it harder for individuals to deny intention.
4. **Negative Identification** - contrary to traditional authentication methods, biometrics allow a system to tell whether a person is known or not (e.g. no second registration with different email).

Building large scale biometric authentication systems is therefore a desirable endeavor. Traditional biometric systems can be simplified to the process pictured in figure 1.1. In the enrollment phase, participating users present a specific biometric modality to a sensor, which captures it and creates what is often referred to as a "biometric embedding". These user embeddings are then

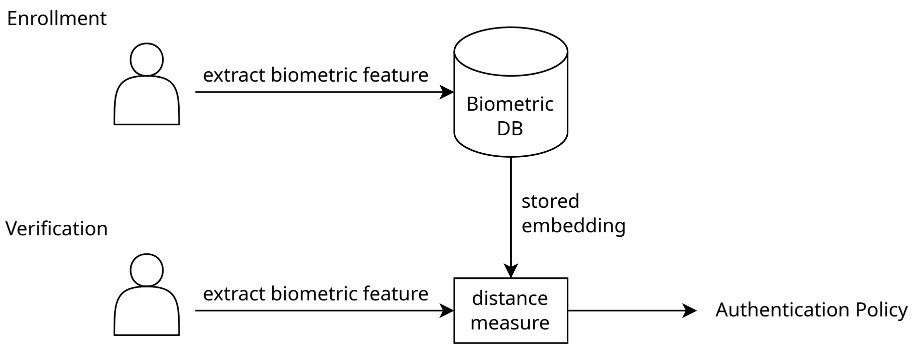


Figure 1.1: Simplified architecture of a biometric system based on [38]

stored in a database. During the verification phase, the same biometric modality is measured and the result is compared to the stored one. In this context, the comparison entails the calculation of a distance metric (e.g. Hamming distance), giving insight on how close these two features are to each other. This distance metric is then interpreted using a use-case specific authentication policy (i.e. distance between features is below a certain threshold). The problem with these systems is, that biometric data is highly sensitive and lends itself to abuse. Users are effectively at the mercy of the entity controlling the biometric database. If this entity is a state actor with vast resources, such data sets enable mass surveillance or at least make such a lot more feasible.

Digidow[29], which in this thesis is used as the short form for “Christian Doppler Laboratory for Private Digital Authentication in the Physical World”, aims to provide the means to solve this problem by decentralizing the storage of its users biometric embeddings. It does so by introducing so-called “Personal Identity Agents” (PIA). Each PIA is representing exactly one individual and holds its biometric data. The identification of individuals is done by a network of sensors, each capable of extracting certain biometric features. Since these sensors need to be able to compare the features they extract with some known reference (formerly the entries of a biometric database), the PIA will have to give its biometric profile to a sensor in a process called “registration”.

Figure 1.2 shows a data flow diagram of the current situation. Users provide/ store their embedding to/in the PIA on the left side. They also interact with sensors (on the right) which derive embeddings based on what they are sensing. PIAs then discover sensors and register with them, transferring their stored embedding. That way, the sensor is able to recognize specific users and notify them through a callback when their related individual is recognized. The problem right now is, that the PIA has no way of knowing whether the sensor can be trusted and will treat the user’s biometric embedding with the necessary care. It would be desirable for a PIA to discover the trustworthiness of a sensor, before interacting with it in the first place—a perfect application for remote

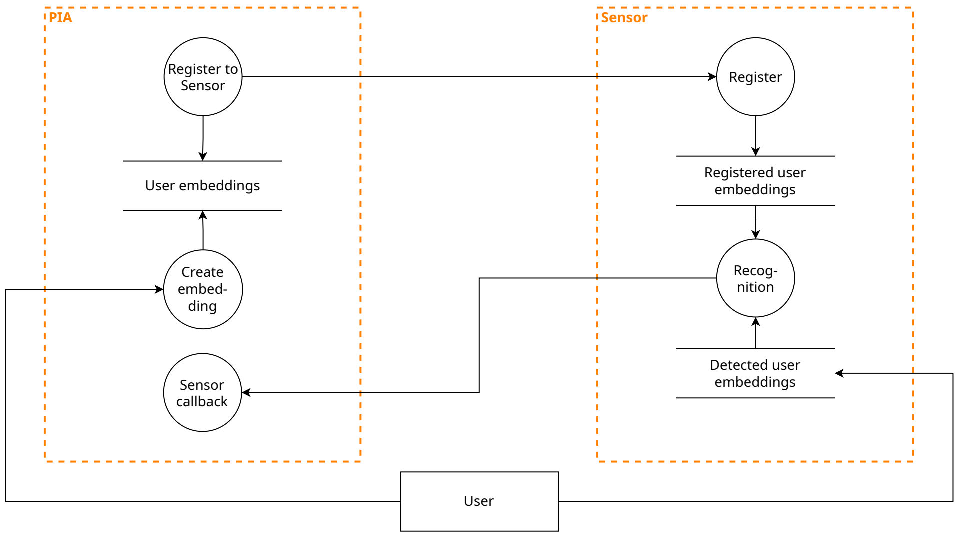


Figure 1.2: Data flow model for the sensor registration flow.

attestation.

1.2 Objectives

The Goals for this thesis are to:

- enable the sensors to produce measurements with which they can prove their current system state,
- enable the PIA to interpret these measurements and derive a decision on the respective sensor's trustworthiness,
- define a process to guide the interactions between PIA and the sensors,
- develop strategies on how to enable manufacturers to provide the necessary reference data, and
- identify potential shortcomings in the resulting solution and proposals of how to overcome them.

1.3 Outline

The thesis will start by introducing the necessary background topics like the Digidow architecture, the hardware in use etc. in chapter 2. Chapter 3 will present related work and place the topic in the existing research landscape. In chapter 4 an overview of the proposed solution is presented. Chapter 5 explains the most important implementation bits. Chapter 6 tests the created artifacts and derives key observations. Chapter 7 concludes the thesis by outlining its limitations and possible avenues for future work.

Chapter 2

Background

2.1 Digidow

As introduced in section 1.1, this thesis was developed as part of Project Digidow. The name is a portmanteau word of Digital and Shadow and aims to convey the core system idea of providing a nearly invisible way for users to authenticate themselves against a digital system using their biometrics. Figure 2.1 shows a simplified version of the architecture as presented in [29].

2.1.1 PIA

As described initially, the personal identity agent (PIA) is the entity holding a user's biometric data, preventing the need for a big centralized database. A PIA more or less "represents" a user within the system boundaries and acts on its behalf. To enable the user to exercise full control over one's biometric data, the PIA could either be hosted by the users themselves or through a trusted third party hosting provider. Besides the user's biometric profile, the PIA also holds a set of attributes retrieved from an issuing authority upon registering with it. These attributes can be used to authenticate the user towards a verifier, being

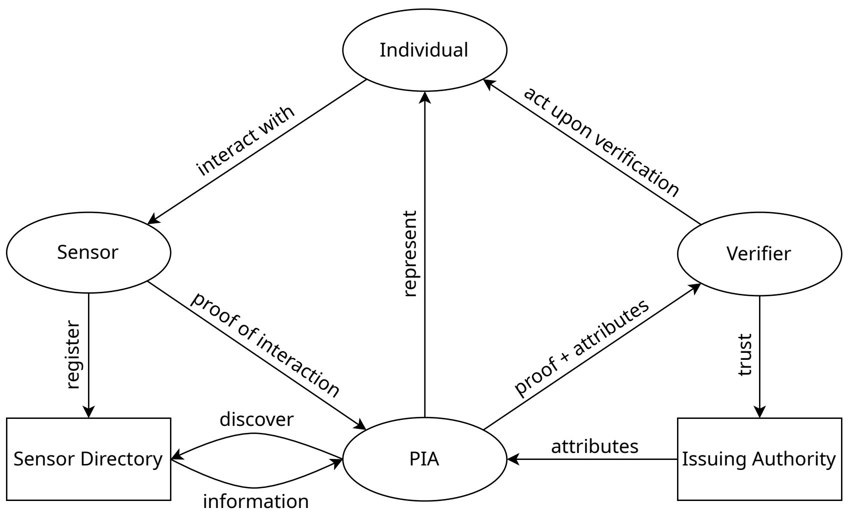


Figure 2.1: Digidow overview adjusted from [29].

proof that an issuing authority vows that the PIA indeed represents the person it claims to be.

2.1.2 Sensor

Sensors are the parts of the system that detect and recognize users. They rely on some biometric modality. Detection works without the knowledge of a user's biometric data. Recognizing specific users and contacting their respective PIAs on the other hand, requires a biometric profile as well as the address of the related PIA. Once a PIA decides to provide this information, the sensor will contact it whenever a person depicted by the biometric profile is detected. The single modality currently available as a research prototype utilizes face recognition and is running on a Raspberry Pi 5.

2.1.3 Sensor Directory

In order for the PIA to find sensors that its user is likely going to interact with (e.g. based on the user's location), it can query sensors with certain attributes from the sensor directory. When a sensor starts up, it registers with this directory in order to be discoverable by PIAs.

2.1.4 Issuing Authority

An issuing authority issues the attributes that the PIAs use to represent their users. Its main purpose is to create the relationship between an actual person and their personal identity agent. The check whether that is actually the case, is based on an out-of-band check utilizing traditional IDs (e.g. a driving license). Potential issuing authorities include government institutions or any form of businesses (e.g. access control to corporate buildings).

2.1.5 Verifier

The verifiers are the entities the users want to authenticate with. They trust certain issuing authorities to have properly checked an individual's identity and when presented with a set of valid attributes as well as the proof of interaction from a sensor, they provide some sort of feedback or service. A simple example of such a reaction would be opening the door to a restricted building.

2.2 Remote Attestation

When employing remote attestation, the overall goal is for one system to convince another one, that it is in a certain state. This could, for example, be a sensor, which wants to prove to a PIA, that it can be entrusted with face embeddings. When looking at such procedures the following terminology is used (defined in [3]):

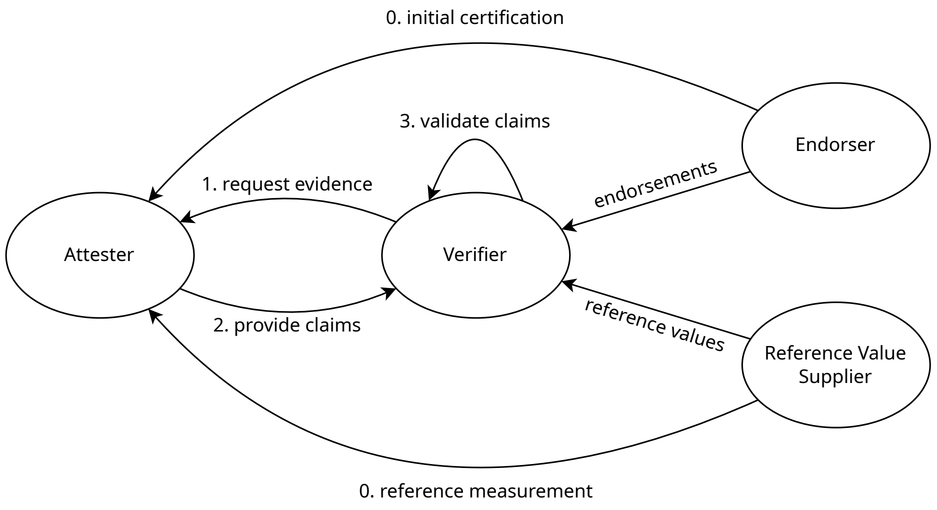


Figure 2.2: The most basic form of remote attestation, adapted from [3].

- **Attester:** This is the entity which seeks to prove its trustworthiness or its overall system state. To this end it gathers different claims and assembles these as evidence.
- **Claims/Evidence:** These are properties of or facts about the system the Attester garners during evidence assembly, e.g. "This computer is running x, y and z".
- **Verifier:** This is the entity which the Attester is trying to convince. It is the recipient of the evidence and based on it, it derives a trust decision. To derive this decision, the Verifier is reliant on reference data and/or knowledge about the sensor.
- **Endorser:** A party which provides evidence to the Verifier, that certain procedures within the Attester did indeed run, and produced a certain output. An example would be the Endorsement Key inside a TPM, which can be used to prove, that a certain element is present within a genuine TPM. To do this, the Endorser is reliant on public key cryptography, certificates and signing algorithms.
- **Reference Value Provider:** Another entity supporting the Verifier in its decision, by providing known good values for the evidence presented by the Attester. The trust in the reference value provider's ability to measure a good state alone, is what gives trust in the system after the attestation.

Figure 2.2 shows a very basic form of remote attestation, wherein the verifier requests evidence and validates the claims afterward, using endorsements and reference values. The two most important aspects to be highlighted in the following, are evidence and the trust that this evidence is correctly supplied. Together they allow a verifier to argue about the state of an attester.

2.2.1 Evidence

Johnson et al. [26] distinguish two different types of evidence:

- **Static:** values and measurements that don't (or rarely) change over time, like the checksum of a bootloader, the checksum of the binary of some face detection software, ...
- **Dynamic:** values and measurements that do change over time, like main memory contents, log files, ...

According to Johnson et al. [26], static evidence is traditionally not enough to stop every attacker, because an attack could simply emanate from a location not covered by static evidence collection (like main memory). On the other hand, dynamic evidence is hard to capture and hard to compare to, because it does not stay constant over extended periods of times. Striking a balance between these two types is important and basically a question of how easy evidence is to work with versus how thoroughly it represents the system state.

2.2.2 Trust

One of the most important problems in remote attestation is how the attester manages to convince the verifier, that the information that is being sent is genuine. This "trust" can be derived in a number of different ways. In related literature ([2, 15, 26]) these trust anchors are divided into different classes depending on how they achieve this:

- Software-based approaches rely (as the name already implies) solely on software to prove trust. The way this traditionally works is that the verifier closely measures the time it takes the attestation procedure measuring the attester to fulfill its task. They are based on the idea that if an attester tries to send bogus information, additional CPU cycles will be used, leading to a measurable difference in reaction time. These procedures are very cheap as they do not need any additional hardware, however they also open up a number of problems. For one, the verifier needs to be on the same network as the attester as network delays through routing protocols etc. make it very hard if not impossible to gain accurate timing. On the other hand, the verifier relies on the assumption, that the attester cannot execute the attestation procedure any faster as doing so would defeat the timing assumptions. This inflicts additional pressure on the implementation of the attestation procedure as it has to be, per definition, optimal.
- Hardware-based approaches use hardware trust anchors for attestation. They are based around the idea, that hardware is harder to alter and (in some cases) can even operate independently of the system it is installed in. Given a properly isolated hardware security module, an attacker is forced to attack the device on the physical layer which is potentially very expensive (e.g. fault injection). Additionally, any attack of this kind is preceded by the acquisition of the actual device which (depending on the access controls in place) bloats the attacks cost even more.
- Hybrid approaches aim to combine software and hardware based attestation to draw from either approaches advantages, i.e. the strong security

guarantees of hardware and the lower cost of software based solutions. Francillion et al. [15] did exactly that and defined a set of minimal requirements on an attestation procedure, including both hard- and software aspects, without relying on fully developed hardware modules like a TPM. These boil down to:

- Hardware capable of securing some secret k
- Secure memory erasure and system reset mechanisms
- The ability to disable interrupts to ensure the uninterruptability of the attestation code
- Hardware that ensures when and how the attestation code can be run

Some of these requirements will become important later in this thesis (e.g. the first one). The idea of reducing expensive hardware to a minimum and running the rest of the attestation in software is the main characteristic of hybrid attestation approaches.

2.2.3 Principles

In a cornerstone paper in the area of remote attestation, Coker et al. [4] state a set of principles for remote attestation:

- **Fresh Information:** Attestation evidence needs to incorporate the current state of the system, not just static evidence from system startup.
- **Comprehensive information:** The provided evidence should enable the verifier to paint a clear picture of the internal state of the attester.
- **Constrained disclosure:** The attester should be able to control which information is sent to which verifier and which information is omitted.
- **Semantic explicitness:** The verifier should be able to derive information and predictions on the attester by applying logical operations (e.g. A and B where measured, meaning C must hold). This has implications on how the evidence is delivered.
- **Trustworthy mechanism:** The verifier should be able to grasp the trustworthiness of the attestation procedure. Furthermore, both parties should be aware of how the attestation is performed.

Given the requirements on the attestation procedure, derived from these principles, the authors acknowledge that an implemented attestation procedure would likely not fulfill all of them. The presented procedure will for example not focus on the constrained disclosure part, as the sensors acting as the attester are supposed to be transparent in the first place.

2.3 The Trusted Platform Module (TPM)

In order to enable the sensor to report on its current software state during the remote attestation process, a “Trusted Platform Module” (short TPM) was built into the sensor platform. A TPM is a secure coprocessor which does not share its internal state with the system it is built into. This is probably its most important property, as this means one does not need to trust the majority of the system, it's enough to trust the roots of trust in and around the module. A TPM is often referred to as a "Hardware Root of Trust" and is thus a prime candidate if one seeks to report on the internal state of a machine. The following explanations are based on the TPM specifications [46, 47, 48] provided by the Trusted Computing Group (henceforth referred to as the TCG).

Trust always needs to be based off of something, it will not spawn out of thin air. We always want to keep this something as small and compact as possible as there is at least the perceived notion, that a smaller system (or part thereof) is easier to evaluate and thus easier to be trusted. The TCG describes this as the "Roots of Trust", the parts of a system for which trust cannot be trivially derived. They describe three different roots of trust some of which can be provided by a TPM and some of which have to be provided by the platform manufacturer.

- The "Core Root of Trust for Measurement (CRTM)" which is responsible for capturing the system state. There exist two different types of measurement roots:
  - The "Static Root of Trust for Measurement (S-RTM)" is referring to the first instructions run by a system. It's the code that runs before the TPM gets involved and is normally represented by the firmware of a system. There is no way to tell what exactly it does and thus one needs to trust that it is not doing anything malicious. Once this code runs, it shall measure whatever code runs next and store that measurement somewhere safe. That way we do not have to trust whatever code that is as long as we trust the code that did the initial measurement. This is what the prototype in this thesis will rely on. A depiction of the SRTM can be seen in figure 2.3.
  - The "Dynamic Root of Trust for Measurement (D-RTM)" [52] on the other hand is not run at boot time, but at a later stage. To still be able to trust it, the TPM is supported by the system's CPU which implements special commands creating a temporary measurement environment. The latter will ensure this environment's isolation and thus allow the system to run trustworthy measurements in this stage. This way attestation evidence can be collected, without having to restart the entire system.
- The "Root of Trust for Storage (RTS)" describes a storage location shielded from external access. The results one gets from the S-RTM are worthless if they cannot be kept from being modified. This functionality is covered by a TPM as outlined below.

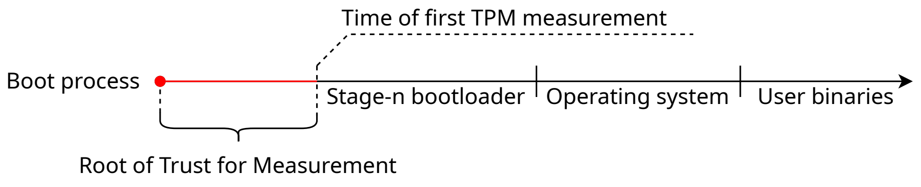


Figure 2.3: Depiction of the S-RTM.

- The “Root of Trust for Reporting (RTR)” is the key for remote attestation. It describes the ability to report on various different aspects of the system while providing strong evidence to the other party, that this reporting can be trusted. For this purpose, every TPM manufactured according to the TCG’s specification has its own endorsement key with an accompanying certificate signed by the manufacturer. The private portion of this endorsement key never leaves the TPM and can thus be trusted to be known by no one else. This means that if someone were to take the certificate, check that it was indeed signed by the manufacturer, trusts the manufacturer to be able to build a secure TPM, encrypts a secret using the public portion of the endorsement key, sends it to the TPM and receives the correct plaintext, one knows that the information must have been passed into a genuine TPM. This scheme can be used to establish signing keys which can then be used to sign further communication thus providing a “Root of Trust for Reporting”.

Enabling these “Roots of Trust” demands a set of more or less basic features, two of which are especially important for the work presented in this thesis:

- **Cryptographic subsystem**: provides elements for symmetric and asymmetric encryption, signature and hashing algorithms, key derivation functions (which we will specifically encounter in a later chapter) and random number generators. This enumeration is not exhaustive and the complete list of features can be taken from the TCG’s specification.
- **Protected storage**: Storage on a TPM comes in two different kinds:
  - **Volatile**: comprised of the Platform configuration registers, any keys used for currently running operations, active sessions etc.
  - **Non-Volatile**: contains for example the seed from which the endorsement key is generated. This storage area entails any information that is supposed to still be available after a power cycle.

2.3.1 Platform Configuration Registers

The platform configuration registers (PCRs) are storage elements that hold a single hash value. A TPM might have one or multiple banks of 24 PCRs, based on the hashing algorithms it supports. They are meant to hold checksum measurements issued by the system and cannot be set to arbitrary values. They change their content based on an *extend* function, as shown in figure 2.4.

If a new checksum is to be added to a specific PCR, its old value is concatenated with the new measurement and the corresponding hash function is calculated over that concatenated byte array. The result is then stored in this register, representing the new value. This ensures, that measurements cannot simply be forged by setting the register to a known good state. The process is depicted in figure 2.4. The knowledge of the initial measurement (checksum) is necessary to end up on the same value.

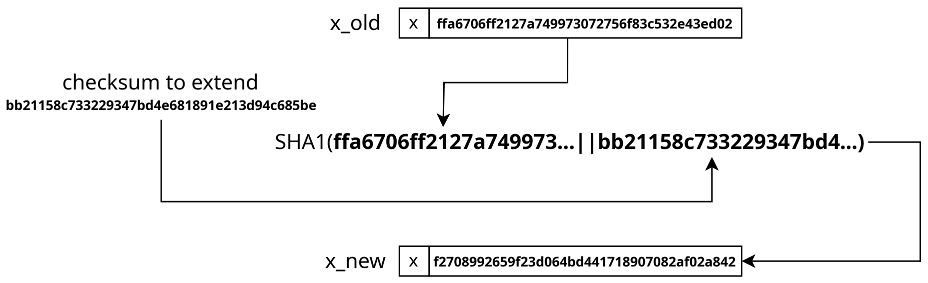

x_old
x
ffa6706ff2127a749973072756f83c532e43ed02
checksum to extend
bb21158c733229347bd4e681891e213d94c685be
SHA1(ffa6706ff2127a749973...||bb21158c733229347bd4...)
x_new
x
f2708992659f23d064bd441718907082af02a842

Figure 2.4: PCR extend operation for SHA1.

2.3.2 Chain of Trust - System Measurement

On a system using a TPM, all trust depends on the fact, that the combination of early system ROM firmware and TPM are doing their respective work correctly. Taking this fact as a given and trusting that they do, we can extend this trust to other parts of the system. When the root of trust measures whatever it is loading (another bootloader or operating system), what it basically does is calculating a checksum of the related data and extending it into the TPMs PCRs. This value can then be used to compare to what is expected for a valid instance of this data/code. If these two values are the same, the chain of trust was successfully extended. This new code can now also load something else (e.g. a binary) and again measure it in the process.
In this way, the trust that was put into the Roots, is extended to the whole system and without trusting anything outside the minimal startup code and the TPM, one ends up with a valid system. The theoretical chain, for an arbitrary system is pictured in figure 2.5. These measurements effectively build a representation of the system's current state inside the PCRs.

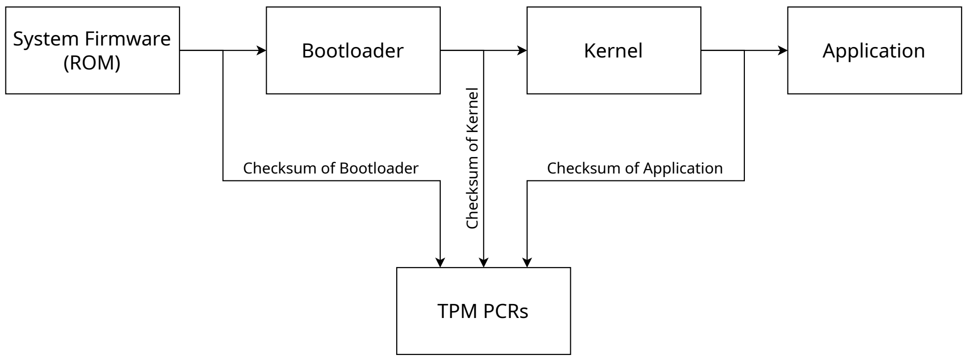


Figure 2.5: Chain of trust starting at the root of trust for measurement, spanning up to a running application.

2.3.3 Platform Certificates

As was already established, adding a TPM to a platform is not enough in itself. The TPM does not control the RTM and can thus not ensure that it ran, wasn't interrupted or modified. Since a modification of the RTM would break the chain of trust, there needs to be a guarantee that the measurement process ran at some point. The result is the need of an additional type of endorsement, provided by the platform's manufacturer, the so-called "Platform Certificate" [43]. It attests that a specific platform, being in the possession of a specific key, has a TPM as well as an immutable RTM, which cannot be interrupted or replaced. This certificate and the associated endorsement bridges the gap between the platform's RTM and the TPM giving proof during the attestation, that measurements from it can be trusted. A major problem is, that these platform certificates are not easy to come around. Only a handful of manufacturers provide a way to retrieve them, which typically involves a not insignificant amount of related work. Examples for companies providing such certificate are the enterprise divisions of both HP [24] and Dell [23]. An overview of how these should be used to ensure supply chain security is provided in [39].

2.3.4 Talking to the TPM

A TPM can be implemented in a number of different ways. In terms of hardware the main distinction is between TPMs which are implemented as their own chips (e.g. Infineon SLB9670) and TPMs implemented in a standard CPU, which are also called firmware TPMs (fTPMs). Dedicated TPMs have to be connected to the rest of the system using a bus system like SPI.
In order to "talk" to a connected TPM, the TCG defines a number of different APIs with varying levels of abstraction. The one used in this thesis is the "libtss2-esys" [41], developed by the "tpm2-software community", implementing the "TSS 2.0 Enhanced System API (ESAPI) specification" [45] by the TCG. Just as its underlying "libtss2-sys" ("TCG TSS 2.0 System Level API (SAPI) specification") it provides "1-to-1 mappings for TPM2 commands" as described in part 3 of the TPM2 specification [48]. The thesis implementation itself uses a crate[32], which provides bindings, in combination with some abstractions, to/from the "libtss2-esys" library. For debug and exploratory purposes, "tpm2-tools" [42] was used, which is a command line based tool providing access to many of the TPM commands, also developed by the "tpm2-software community". Throughout the next subsections, commands from this tool will be used to underpin what is being explained to make the concepts in question more tangible.

2.3.5 Handles and Contexts

During operation a TPM interacts with a plethora of different objects like keys, sessions, key seeds, etc. Every one of these elements currently present on the TPM is assigned its own, unique handle. A handle is a 32-bit integer identifying that element inside memory. An example for such an object is the endorsement key:

```
sensor@raspberrypi:~ $ tpm2_createek -Q -c - -G rsa -u ek.pub
sensor@raspberrypi:~ $ tpm2_getcap handles-persistent
- 0x81000000
```

In this example, the endorsement key is generated from the TPM resident endorsement seed. The "-c" option lets us define at which handle to store the resulting key. Passing "-" tells the program to look for a suitable handle, which then turns out to be "0x81000000". There are different types of handles, differentiated by their respective most significant octet. The one at hand ("0x81") refers to "persistent object handles" which is a group of handles which do not get erased by power cycling the system, which makes sense in this case, as the endorsement key does not change. The other handle types are available in chapter 13 of the tpm2 architecture specification [46].

As one can imagine, if there were a lot of processes using a single TPM, its memory would fill up very quickly not to mention the fact that other applications could potentially get access to sensitive TPM contents. Therefore, TPMs are able to store the context of a handle (and with that the object associated to it) in a context file outside the TPMs memory. To this end, a key is derived and the object data encrypted. The integrity of the result is guaranteed via an HMAC. The resulting structure is returned and written to a file on the systems mass storage. At a later point in time, this data can be fed back to the TPM to load the object again and assign a new handle. The related TPM commands used in this thesis are the following:

- **TPM2_ContextSave**: save the current context for a TPM object/handle
- **TPM2_ContextLoad**: load the stored context for a stored TPM object/handle
- **TPM2_FlushContext**: erase handle/context information from TPM

Not all objects can be stored using a context file. The subset of objects that can are either "transient objects" (handles with most significant octet of "0x80") or sessions used for authorization, which will be described next.

2.3.6 Sessions and Authorization

A TPM and its contents are potentially available for a number of different processes on the system it is installed in. Let's imagine one process creates an attestation key and loads its context into the TPM. Another process could then retrieve the handle and use it for something different, potentially against the will of its creator. This is why, for a subset of elements on the TPM, authorizations are required. There are four (arguably three) different types of authorizations (sorted from "weakest" to "strongest"):

- **nullauth**: no authorization required
- **password**: plaintext password provided as authValue
- **HMAC**:session based authorization based on providing a password as authValue "protected" by using an HMAC (password does not get transmitted directly)

- **policy**: session based authorization providing a session digest called auth-Policy

The first two types are pretty self-explanatory: either no authorization is required or a password referred to as "authValue". The latter two types are so called "session-based" authorization types. Like other objects, sessions can be loaded from and stored in context files. They get assigned handles like any other object and handed over when making use of certain commands requiring either HMAC or policy authorization. When this happens, the internal value of the session changes. This is also why, besides authorization, sessions can be used to prove that a certain sequence of commands was executed, as this sequence is reflected in the session's internal value.
When an HMAC session is used, an HMAC gets calculated from the associated password (in combination with additional data), which is then handed over for authorization. A policy session on the other hand, does not have a password associated. Authorization using policy session works by calling any number of so-called policy assertions. These policy assertions alter the session's internal value when the respective assertion holds. Presenting the resulting session digest thus proves to the TPM that the policy made up of a specific set of assertions holds for that session. It is this exact reason that policy sessions cannot be used as audit sessions, as the value their internal state holds is necessary for authorization.

There are a number of related TPM commands that find use in the thesis implementation:

- **TPM2_StartAuthSession**: Starts an authentication session of either type and returns the handle.
- **TPM2_PolicySecret**: Asserts that the caller knows the authValue (secret) related to a certain object identified by its handle.

2.3.7 Attestation

Serving as the "Root of Trust for Reporting" a TPM also provides a lot of functionality to facilitate remote attestation. The following is a list of attestation related commands provided by a TPM. The exact command usage will be made clearer in section 5.

- **TPM2_Quote**: creates a quote over selected PCRs and signs it using the key indicated by the passed handle
- **TPM2_PCR_Read**: reads all the selected PCRs and returns their values
- **TPM2_CreatePrimary**: used to "create an object under the primary seed" [48]. In this context this command is used to re-create the endorsement key from the endorsement seed.
- **TPM2_Create**: used to create the attestation key as a child element to the endorsement key
- **TPM2_MakeCredential**: used to create a credential which ties some object attributes to a secret, encrypted by a key stored on a TPM.

- **TPM2_ActivateCredential**: here the TPM verifies that the object referenced by the credential is actually present on the TPM and if (and only if) it is, the credential is decrypted using the endorsement key and the resulting secret is returned.

2.4 IMA

The Integrity Measurement Architecture (IMA) is security feature of the Linux kernel and was first introduced in version 2.6.30 [7]. Its main goal is to make sure that files are not being altered or at the very least such an alteration is noticed. To this end it provides 3 core services [21]:

- **IMA-Measurement**: Upon accessing a file, a hash (checksum) is calculated and stored in a log. If a TPM is present on the system, that measured hash is also extended into PCR10, enabling reporting through the TPM on measured IMA values.
- **IMA-Appraisal**: Stores measured file hashes in its attributes. If an already measured file is accessed again, its checksum is recalculated and compared to the stored hash. If they differ, the access is denied.
- **IMA-Audit**: Includes IMA measurement in additional log files, supposedly improving system audit possibilities.

In the context of this thesis, the measurement feature is of particular interest. As described in section 2.3, it is used to extend the system's measurement into the "Root of Trust for Reporting". When the TPM produces a signed quote over its PCRs that also includes PCR10, meaning the IMA access log is transitively signed by the TPM. An entity with the goal of verifying the IMA log can now "replay" all the measurements taken and together with individual PCR values should be able to reproduce the quote, thus proving the contents of the IMA log. Having access to a trustworthy IMA log is interesting because it gives a remote party insight into the running system and whether files produce expected checksums. If the log would, for example, contain a binary with a checksum value not known to the verifier, this could hint at the sensor being modified.

To decide which files to measure and apply its features to, IMA uses policies. This is especially important, because files that change often are hard to include into an IMA log audit because it would necessitate constant updates on the reference log values. We thus use policies to exclude files that change often as part of systems operation to keep reference material valid for longer. IMA policies are made up from "measure" and "dont_measure" statements followed by one of a few possible arguments creating a "scope" which files potentially match into. If they do, the respective rule applies. The following section shows a short excerpt from IMA's built in TCB policy: [21]:

```
dont_measure fsmagic=0x9fa0         # PROC_SUPER_MAGIC
...
measure func=FILE_CHECK mask=MAY_READ uid=0
measure func=MODULE_CHECK
...
```

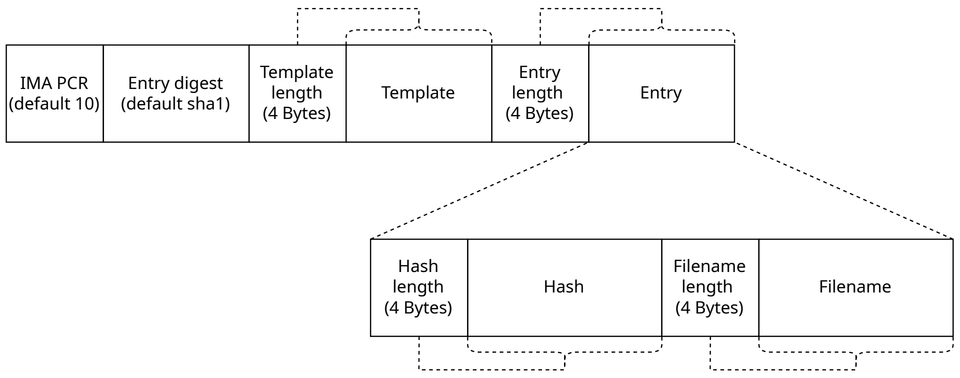


Figure 2.6: IMA log entry layout.

From top to bottom these policy entries state, that the proc-filesystem is not to be measured which is good as it changes a lot (representation of processes). On the other hand it also defines what to measure i.e. every file (func=FILE_CHECK) read (MAY_READ) by root (uid=0) and all the kernel modules (MODULE_CHECK). With these building blocks one has to build a policy that on the one hand, covers as much of the system as possible and on the other hand, does not measure too many (if at all) frequently changing files, which has a significant impact on how easy it is to check the log.
If IMA is configured correctly, measurement logs show up in */sys/kernel/security/integrity/*. The following is an example of what such log entries might look like:

```
10 6b...dd ima-ng sha256:7b6436b0c98f62380866d9432c2af0ee...61 boot_aggregate
10 cc...2d ima-ng sha256:94a66d567efa9ca67f80f9cb3e67ecd5...f4 /usr/bin/kmod
10 c3...27 ima-ng sha256:3dd934e1fb34248b9c7a657d1651964d...64 /etc/ld.so.cache
10 af...34 ima-ng sha256:060ff11d8b8a2bbceddcc4a8034b11d1...ad /init
10 d4...b8 ima-ng sha256:aa1c8f7b1f36084b34b952de06d06430...ba /usr/bin/sh
...
```

The log file is also available as binary data, which then has to be parsed in order to extract the information necessary for the attestation. Figure 2.6 shows the entry layout for template “ima-ng”:

Besides “ima-ng” there are other available templates like “ima-sig”, which contains the same information together with a signature. A complete list of available templates is available in the IMA documentation [21]. Independent of which template is used, the digest of an entry which gets extended into the assigned PCR is always the hash value over the complete “Entry” (marked in figure 2.6 via a bracket) using the hashing algorithm of the PCR bank to be extended. Linux kernels before 6.10.6 [21] only had one binary measurement file with a default digest using SHA1. On these machines the digest for SHA256 needed to be calculated again in order to check the value against PCR10 in the SHA256 bank. On newer kernels, IMA holds log files for all supported hash algorithms, meaning that there already is an IMA log file containing the SHA256 entry digests.

2.5 Transparency Logs

A key requirement for the resulting remote attestation to derive any trust is that the values which the attestation results are compared to are distributed in a way that ensures they are not being tampered with. To this end, this thesis uses a transparency log which contains entries for every valid reference data set. This way, it doesn't matter how the reference data is distributed since a proof of inclusion in the log is enough to vouch for its correctness. The log itself is built from a verifiable data structure as described in [28]. These structures use Merkle Trees to achieve the verifiability property. Such a tree stores the checksums of the data sets, for which the inclusion should be proved, in its leaf nodes. Every structural node above the leaf level is associated to at most two child nodes. These structural nodes also hold checksums which are generated by first concatenating its children's hash and then feeding the resulting value into the related hash function. The root node in a tree built according to these rules holds a hash representing all the currently included entries of the tree. An example for such a tree can be seen in figure 2.7. The root node is signed by the transparency log and is needed to prove the inclusion of specific data sets. Say the inclusion of data set N3 is to be proved. First, the proof is requested from the log, which if a certain data set is contained, returns a set of hashes. The fact that an entry is present (indicated by the set of hashes returned), is however not enough to show inclusion. To do this, the root hash has to be recalculated. The set of hashes returned by the inclusion proof request contains all those hash values needed to verify the root hash value given the value of N3. This set of hashes is called the "Audit Path" of N3, consisting of the hash values for N2, N6 and N8. First N2 and N3 are used to compute N7 and then N6 and N7 are used to derive the hash of N9 which can then be used together with N8 to derive the root value. If this matches the signed value, the inclusion was successfully proved. Transparency logs do not hold or distribute the data associated with them. The only information they convey is that some data is included and whether the structure was tempered by someone else. If only trusted values are inserted into a log, inclusion can thus prove the trustworthiness of a value.

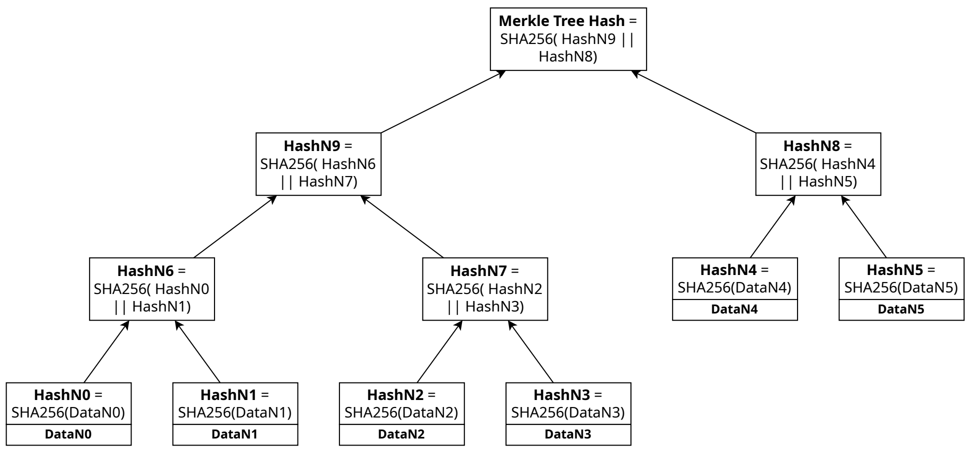

Merkle Tree Hash = SHA256(HashN9 || HashN8)
HashN9 = SHA256(HashN6 || HashN7)
HashN8 = SHA256(HashN4 || HashN5)
HashN6 = SHA256(HashN0 || HashN1)
HashN7 = SHA256(HashN2 || HashN3)
HashN4 = SHA256(DataN4)
DataN4
HashN5 = SHA256(DataN5)
DataN5
HashN0 = SHA256(DataN0)
DataN0
HashN1 = SHA256(DataN1)
DataN1
HashN2 = SHA256(DataN2)
DataN2
HashN3 = SHA256(DataN3)
DataN3

Figure 2.7: Example for a Merkle tree used to implement Transparency logs.

Chapter 3

Related Work

Having covered the basic background information necessary to understand this thesis, it's important to place it within the broader academic landscape of remote attestation.

First, it is important to differentiate what the word "sensor" actually refers to. Oftentimes the word is applied to small sensing platforms with very strict limits on power and cost. This has obvious implications on the type of attestation that is possible within these systems, as a dedicated hardware security module might be out of scope due to these limitations. Ankergård et al. [1] give an overview of different software based methods, aiming them at already deployed sensing platforms. They describe a decrease of interest in software-based attestation, due to the fact that they are deemed less safe than ones, where the trust is bound in hardware. While the construction of such protocols is rather interesting, they are wholly unsuitable for the kind of sensor required by Digidow. Having the components communicate over the Tor network, makes measuring attestation delays reliably almost impossible.

Given the limitations of software based approaches, some research has been directed towards so called Physically Unclonable Functions (PUFs). These are reliant on the inherent variability in computer chips, introduced by their respective manufacturing mechanism (e.g. uninitialized memory regions stabilizing at either logical 1 or 0). They produce device specific values and can be used for fingerprinting or authentication. An identity attestation procedure based on that technique was proposed by Román et al. [36]. The scheme is not reliant on hardware trust anchors but still able to provide unique IDs. Identity attestation does not necessarily compare to the attestation done in this thesis, but the idea of having a function reliant on a specific set of hardware could potentially enable additional security even if one has access to a hardware root of trust (see replay attack in section 4.2).

Systems which are less bound by cost or power factors, circumvent these issues by introducing hardware based roots of trust like TPM chips. Given the shortcomings of software based approaches TPMs are also being introduced in less powerful sensor platforms. An example is the attestation process described by Tan et al. [40] where each sensor node is equipped with its own Trusted Platform Module. This is getting closer to what this thesis aims to achieve, though the sensors in use still fall short of what the prototype sensor platform for Digidow (Raspberry Pi 4/5) can do in terms of performance.

Since the face detection sensors in Digidow have to run models for face detection and recognition, they already require a substantial amount of processing power, which means, that introducing a hardware root of trust is not a big concern power or cost wise. A prototype for attesting face detection sensors within the Digidow system was shown in 2022 by Michael Preisach [33]. The goal was to engineer a system which future research could be based on. The assumptions that were made during the creation of the prototype, do however not reflect the current status of the Digidow system. Some major points of contention with the proposed solution are:

- The attestation prototype was built on x86 hardware whose compatibility to TPMs is far superior to the aarch64 based Raspberry Pi platform in use by the sensor today. The Grub bootloader on x86 supports TPM measurements out of the box, whereas on the Pi one has to resort to third stage bootloaders to get any kind of TPM support.
- The attestation procedure was implemented using tpm2-tools in combination with a set of bash-scripts. While it would theoretically be possible to call these from within the Rust binaries (which represent today's Digidow component prototypes), this would be far from a satisfactory result.
- The thesis spends quite a lot of time introducing and implementing Direct Anonymous Attestation. The idea with that is, that the sensor does not have to share its endorsement key, making it harder to track it across attestation attempts. Since sensors are identifiable via their onion address anyway, DAA does not provide any meaningful protection. Besides, that the sensor is meant to be transparent anyway.
- It lacks functionality to check SHA256 digests for the IMA log, which is necessary given that SHA1 was retired by NIST in late 2022 [30].

Looking at work using a similar hardware setup, Usman et al. [50] present a remote attestation procedure called "RASUES" which beyond the attestation itself, is able to facilitate software updates. For their prototype, they use the same combination of Raspberry Pi 4 and Infineon sLB9670 TPM as is being used in this thesis. The most interesting part from this thesis perspective is the fact that they managed to "activate UEFI". Upon further research it is likely that the authors were referring to the EDK2 bootloader which seemingly also supports measured boot. This could be an alternative bootloader candidate for future endeavors. It is furthermore interesting because the process of updating a node that is attested to periodically, is an important one and not directly addressed in this thesis.

Lastly, it is worth shifting the view from academic research prototypes to systems actually utilizing remote attestation. One such example is the "Key- and ID-Attestation" in Android [18]. To run the attestation, Android is reliant on a hardware trust anchor, which most of the time comes in the form of ARM TrustZone (TEE - Trusted Execution Environment). This essentially divides the execution on the processor in two different domains: the secure and the non-secure world [17]. The key management is handled within this secure world, protecting them from potential attackers and forming a "hardware-backed key storage".
The goal of key attestation in Android is to prove that a certain key is secured within the key store. To this end, Android devices store an attestation key signed by Google. This is how they derive the endorsement, that the TEE and its related software parts were implemented correctly. Using this attestation key, the key storage can thus prove that keys are held within by signing them (comp. credentials in TPMs). Similarly, Android can also attest hardware identifiers like the device IMEI (International Mobile Equipment Identity

- unique identifier) or the serial number. Just like the attestation key, copies of the devices identifiers are safely stored within the TEE during manufacturing to prevent them from being edited after the fact. Once inside the TEE, they can be attested to by using the attestation key, same as in key-attestation. Beyond that, Android is also capable of attesting the correctness of its boot, system and vendor related code in the form of its *vbmeta-digest* (verified boot meta data digest) [16]. This is essentially a digest over the systems static partitions and as such precomputable. Once a systems software state is final, its *vbmeta-digest* can be stored in hardware-backed secure storage and later attested to like the hardware identifiers mentioned above. While similar to the remote attestation presented in this thesis (in that both are reliant on hardware trust anchors), Android attestation is dealing with significantly less data, limited to static descriptors as opposed to a "full system attestation" giving information about the whole system, especially the dynamic parts in the form of executed binaries.

Chapter 4

Overview

4.1 Proposed Solution

Given the context outlined in the previous chapters, additions to Digidow's architecture and feature set have to be made, to facilitate remote attestation. The following provides an overview over the proposed solution and the general idea of how the process is going to work. A more detailed walkthrough is then provided in section 5. Figure 4.1 shows an overview over the different solution parts.

- **Manufacturer/Developer/Maintainer:**
  These entities either manufacture/develop or maintain the software and hardware components that a running Digidow sensor is made of. In the optimal case, the whole software (operating system with all its applications) is provided in the form of a single, bootable sensor firmware image. This, in turn, should be the output of a build process based on an openly accessible source repository. The image built from this source and booted on a specific set of hardware then produces a characteristic set of PCR values. If these measurements are taken from a system state the manufacturer deems trustworthy, they can be made available in the form of reference values.
- **Transparency Log:**
  The reference data contains information on the sensors hardware and software, as well as the data necessary for a verifier to derive an attestation decision. It is included into a transparency log by its manufacturer, so that no matter where the reference values are sourced from, the verifier can always check that they are identical to what was measured in the trusted reference environment of the manufacturer. The integration of the transparency log is not part of this thesis, but was implemented as part of a master's project.
- **TPM:**
  A TPM is introduced into the sensor node, providing the PCRs into which the system state is measured as well as the *Root of Trust for Reporting* to provide the collected evidence.
- **Sensor Directory:**
  In order for to be discovered by PIAs, sensors register to the sensor directory. PIAs could also learn of the existence of a specific sensor from a different source (by getting told its address), which is why the sensor directory could also be used to just query information about a sensor. The current plan is that the sensor directory will also be responsible for distributing the reference data.

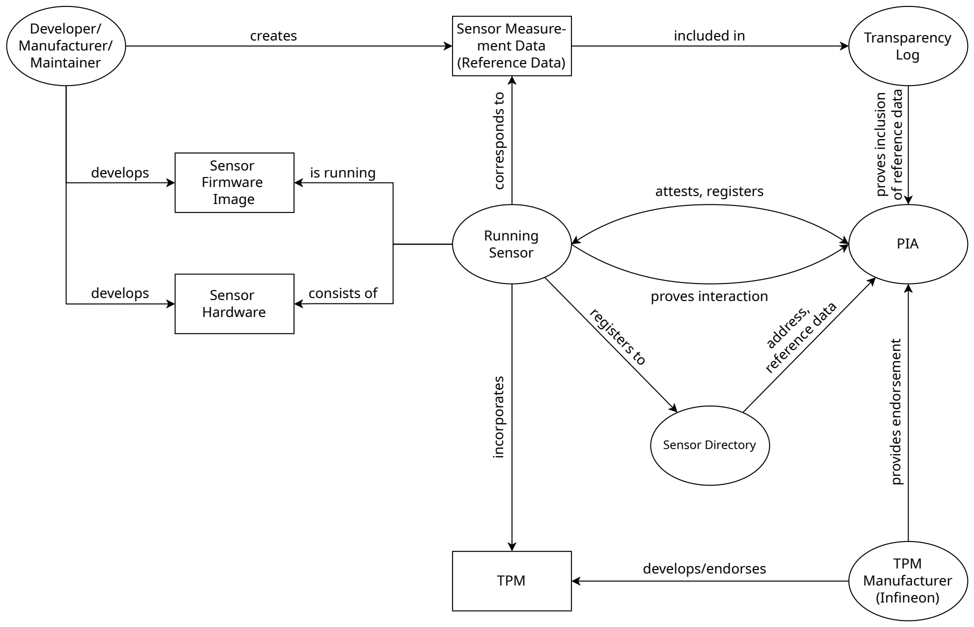


Figure 4.1: Architecture of the proposed sensor attestation.

- **PIA:**
  Contrary to before, the sensor will now provide claims about its running state, which is to be verified by the PIA. It will make sure that it can be trusted to handle the sensitive information (of embedding, time and location) it is passed. Only after the attestation is successfully completed (using the endorsement and reference material described earlier), will the PIA register to the sensor and the traditional operation of Digidow will commence.

4.2 Threat Model

Figure 4.2 contains all the elements that the attestation procedure introduces into the project, or affects in some way. Based on this data flow diagram, a simplified threat modelling process is done to identify potential threats the prototype is exerted to. The following assets were considered:

- **PIA:** user biometric embeddings, reference data, attestation evidence
- **Sensor:** Root of Trust for Measurement, bootloader, operating system, applications, user embeddings, IMA log, attestation evidence
- **TPM:** PCR values, attestation key, endorsement key/cert
- **Deployment system:** sensor image, reference data
- **Transparency Log:** reference data inclusion proof

Based on the above, the following threats/assumptions were made:

1. **PIA embedding disclosure**
   Vulnerabilities in the PIA implementation could lead to the disclosure of the

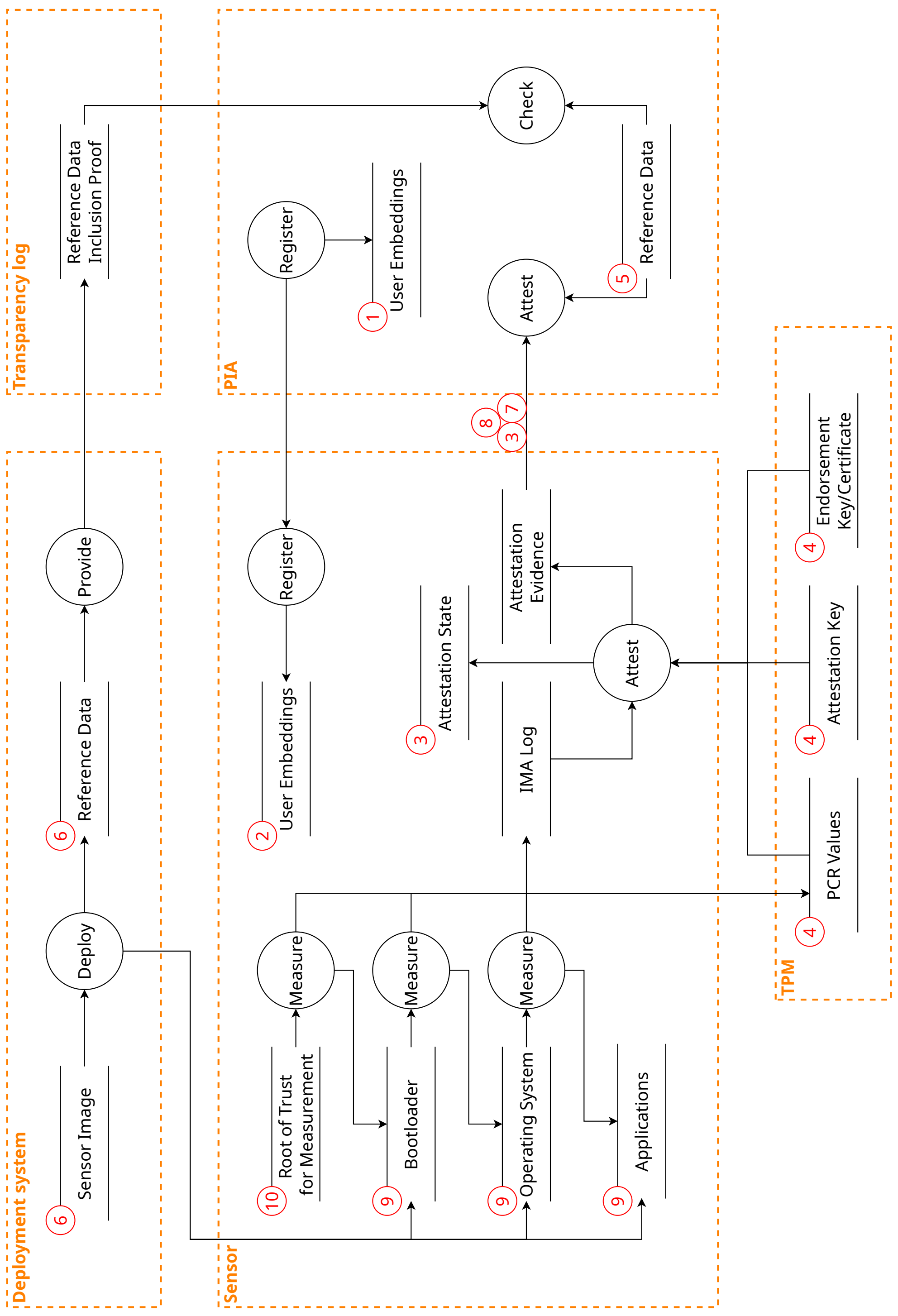


Figure 4.2: Data flow model for the newly introduced attestation procedure.

user embedding. Since the security of the PIA itself is not within the scope of this thesis, it is assumed that the PIA is capable of keeping the user's embedding reasonably safe.

2. **Sensor embedding disclosure**
Even while using remote attestation, the sensor implementation might have arbitrary "traditional" vulnerabilities in the services it exposes. To better focus on the remote attestation procedure, it is assumed that the sensor image is hardened as much as possible, and while it is in a valid state, it is capable of keeping the user's data secure.

3. **Denial-of-service attack on sensor**
An attacker could potentially deny access to the sensor by requesting numerous concurrent attestation procedures. This could, on the one hand, bind TPM resources, draining access from valid attestation runs, while on the other hand, fill up the sensor's state store. Possible controls will be presented in section 7.2.

4. **TPM information disclosure**
The extraction of the private portions of either the endorsement or attestation key from the TPMs storage would break the attestation procedure and allow an attacker to forge evidence. Attacks on the TPM itself are subject of its specification and out of scope for this thesis. Going forward, it is thus assumed that the TPM's storage secure.

5. **Supplying wrong reference values to the PIA**
The distribution of the reference data is expected to be carried out by the sensor directory. If an attacker was able to control this data provision process, the attestation could be broken because the PIA would compare the attestation quote to invalid reference data. To some extent, this threat is mitigated architecturally by fielding a transparency log, used to prove that a certain reference data set is trustworthy. The trust into the transparency log is upheld by its structural properties as well as a set of witnesses, which constantly monitor the state of the log.

6. **Supply chain attack on Deployment system**
A supply chain attack is conceivable, in which attackers modify the sensor images and/or corresponding reference data rolled out to the sensors. This threat entails all the problems of the previous one, with the added problem that the sensor can be directly modified. Countering supply chain attacks is incredibly complex and out of scope for this thesis. It is thus assumed, that the deployment systems are trustworthy. This argument becomes a lot easier if the software image is built from an openly accessible source repository, as the transparency afforded by this will add to the trustworthiness of the sensor image.

7. **Replay of attestation quote**
An attacker could record an arbitrary successful attestation and replay it at a later time to falsely claim legitimacy. This threat demands a freshness check to be included into the attestation procedure.

8. **Relay of attestation procedure by malicious actor to valid sensor**
An attacker might not even have to record a valid attestation at all; any freshness checks could simply be bypassed by relaying the attestation to a known good sensor. This sensor has no way of knowing who it is talking to

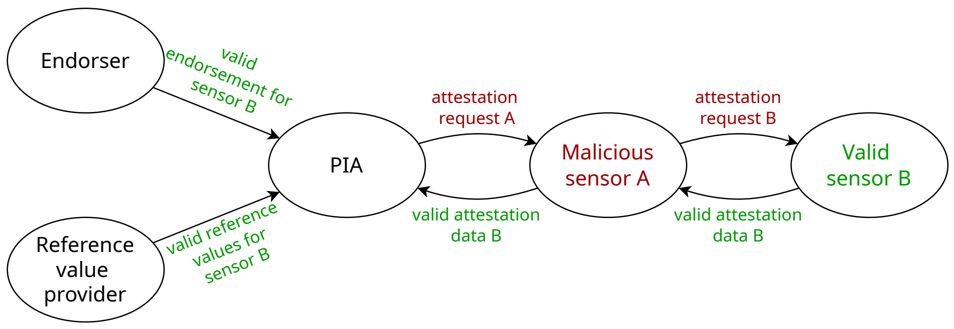


Figure 4.3: A malicious actor relaying the attestation process to a valid sensor obtaining "good" attestation data.

and will thus provide all of its attestation material including endorsement certificates, correct PCR values and IMA log. Since the attestation material is coming from a known good sensor, the attestation will go through and the PIA will send its biometric material to an untrusted party, the very threat this thesis seeks to avoid. Figure 4.3 depicts the message flow causing this threat. By attaching their own address to the signed attestation data, sensors can protect PIAs from such attacks, as they would provide irrefutable evidence of the quote's origin. This scenario gets even more complicated though, if an attacker is in control of both the malicious and the benign sensor. The structure of the Tor network would allow for the benign node to think it is in possession of a certain address while the malicious node would be the one receiving all the traffic towards it. In this case the aforementioned measure would break. This special case is currently out of scope and left for future work, while the more trivial case is addressed by adding the sensor address as described.

9. **Modification of sensor software (bootloader, operating system or application)**
In this scenario, an attacker managed to compromise the sensor and altered the system or its applications (e.g. exfiltrating user embeddings, loosening IMA policies etc). Since the TPM does not share its trust boundary with the PIA and assuming that the root of trust for measurement was not compromised, the system measurement would catch any such modifications and make it visible to the PIA, prior to its interaction with the compromised sensor. This is the exact reason why remote attestation is used in this scenario.

10. **Modifying the Root of Trust for Measurement (RTM)**
As will be discussed in a later section, the Root of Trust for Measurement is not always comprised of software that is exclusively loaded from system ROM. In cases where the system firmware is not capable of doing measurements involving a TPM, the RTM will also include software read from mass storage (e.g. third stage bootloaders like U-Boot on Raspberry Pi). In these cases the measurement process can be subverted, allowing the modification of later stage software like operating system and applications while still producing correct measurements. This threat will be further explored in section 7.2.1.

Chapter 5

Implementation

This section covers the most important implementation steps to arrive at the provided attestation prototype.

5.1 Setup

Before the development of the attestation mechanism can start, essential Digidow components need to be set up. For the previously outlined remote attestation this means: at least one PIA, at least one sensor, an issuing authority and a sensor directory. Figure 5.1 depicts this setup and how components communicate.

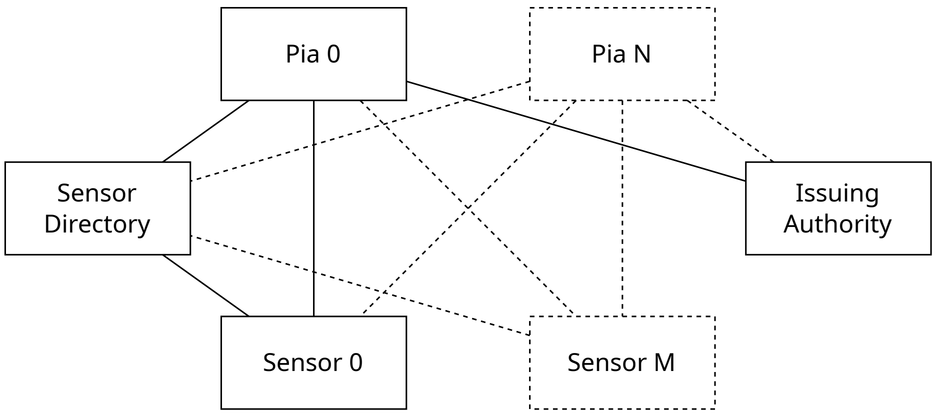


Figure 5.1: Components needed for this thesis's setup.

A verifier is not required for this setup, as the attestation process does not require that interaction. Once the PIA holds the user's biometric embedding, the issuing authority is likewise no longer necessary for testing. To implement the attestation only the PIA and the sensor needed to be modified as they are the two participating parties. The modification of further components to improve the attestation is discussed in section 7.2.2.

5.1.1 Issuing Authority

The issuing authority[1] is built and run using Nix [9]. It furthermore automatically creates all of its configuration files. It's important to take note of the generated onion address it prints to its console on startup. The web interface (default port 9000) it provides is later used to manually check PIA identities.

5.1.2 Sensor Directory

At the time of writing the thesis, the sensor directory[2] is subject of ongoing development efforts. The current version of the PIA does, nevertheless, rely on a preliminary version of the sensor directory in order to function. This version is also built and run using Nix. The only configuration needed is to write down the respective addresses into its state file (*state.data*):

```
"sensors": [
 {
  "type": "face",  #Biometric modality of the sensor
  "addr":  "..."   #Sensor onion address
  "pk": "..."      #Sensor public key
  "lat": x,        #Latitude
  "lon":y          #Longitude
 },
 ...
]
```

Sensor addresses are generated from the seeds assigned to each device and printed into its console upon startup.

5.1.3 Sensor

Contrary to when the development of this thesis started, the sensor[3] will now automatically generate a default configuration file, making any prior configurations unnecessary. It can also be built using Nix, and is best run using the *start.sh* script contained within it's source repository.

5.1.4 PIA

Before one can start the PIA[4] binary to begin the enrollment, the onion address for the sensor directory needs to be set. This is hard-coded into the PIA's source, the respective line can be found in *./core/src/net/sensor_directory.rs*:

```
9: const SENSOR_DIRECTORY_ADDR: &str = "...";
```

[1]https://git.ins.jku.at/proj/digidow/issuing-authority
[2]https://git.ins.jku.at/proj/digidow/sensor-directory
[3]https://git.ins.jku.at/proj/digidow/sensor
[4]https://git.ins.jku.at/proj/digidow/pia

Upon startup, the binary will look for a file named *Data.json*, where it expects to find the onion address of an issuing authority. Once the PIA has started, the web service it opened can be accessed to start the enrollment. The user is asked to upload a picture, which the PIA can then present to the Issuing Authority. There, an operator needs to validate the identity of the PIA and its user, based on traditional means of authentication (i.e. personal ID). If all checks succeed, the Issuing Authority returns a set of attributes the PIA can present to verifiers as a proof of the legitimacy of its identity, concluding the traditional PIA setup.

5.2 Sensor Setup

One of the main parts of this thesis was to set up the sensor in a way that enables system measurement. This section is divided into a number of subsections, representing the different configuration steps needed to achieve this.

The operating system used to power the sensor throughout the thesis is *Raspberry Pi OS* [13]. The SD card with the respective image was set up using the *Raspberry Pi Imager* also provided by the Raspberry Pi foundation. After setting up the OS and creating a user called *sensor*, the TPM software stack needs to be installed, which is achieved by the following command:

```
sudo apt-get install libtss2-dev
```

Lastly, to build and run the sensor code, the Nix package manager needs to be installed. The necessary instructions were taken from[11]:

```
sh <(curl --proto '=https' --tlsv1.2 -L https://nixos.org/nix/install) --no-daemon
```

5.2.1 Adding the TPM

First the hardware (Infineon SLB-9670 by LetsTrust) needs to be added to the sensor. The sensor platform used during the creation of this thesis is the Raspberry Pi 4, which offers a 40 pin GPIO interface for this purpose. Some of these pins are designated for SPI bus communication, which needs to be activated through an entry in the *config.txt* file on the Raspberry Pi's boot partition. Once active, the TPM can be connected to the respective pins, as outlined in figure 5.2.

After the TPM is connected and the SPI bus is activated, the device tree needs to be updated to include the new device. On a traditional desktop hardware discovery is done via ACPI (Advanced Configuration and Power Interface). On devices without a UEFI or BIOS (like the Raspberry Pi) however, this happens through the use of so-called device tree files. This tree contains a description for all the hardware the device is made out of (e.g. SoC or connected devices). There is a "base" device tree for every Raspberry Pi variant provided by its manufacturer. Changes to this device tree happen through overlays, which get patched into the base device tree by the bootloader. The Linux kernel source repository provided by the Raspberry Pi foundation already includes a device tree overlay for the SLB-9670 which can be loaded to provide basic TPM functionality. Running an official Raspberry Pi kernel the changes in *config.txt* thus boil down to:

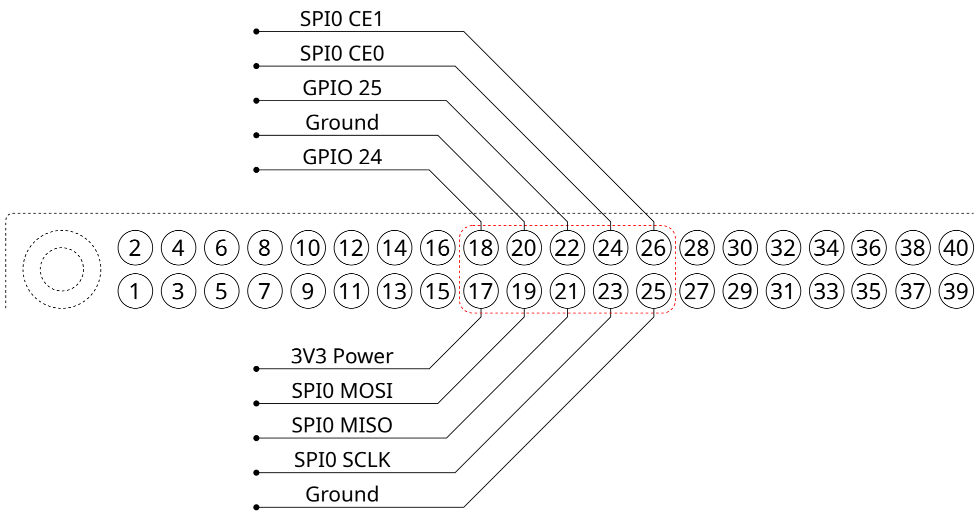


Figure 5.2: Raspberry Pi pins the TPM module is connected to. Based on [20].

```
dtparam=spi=on
dtoverlay=tpm-slb9670
```

5.2.2 Measured Boot

The first step to enabling measured boot on a Raspberry Pi, is to install a bootloader capable of interacting with TPMs. There are a number of different options with this feature like EDK2 or U-Boot. Since the thesis started out on the Raspberry Pi 5, which at the moment has no maintained support for EDK2, U-Boot was chosen. Later it turned out that the U-Boot version currently available for Pi 5 was also not suitable doing measurements, which prompted switching the hardware to a Raspberry Pi 4, whilst waiting for software support on Pi 5 to catch up. Later research found, that on Pi 4, EDK2 would also have been able to do measurements on boot [50], since there was already progress towards U-Boot however, no switch was made on the software side. Accordingly, this section will cover the installation and configuration of U-Boot for measured boot.

The first step is to download the source code and configure/build an image for Raspberry Pi. The following configuration is based on [19]. Instructions on how to cross compile software for the Raspberry Pi are provided here [14]. The commands below download the current version of the U-Boot source, set up a config file for BCM2711 (which is the SoC on Pi 4) and open up the configuration menu:

```
git clone https://github.com/u-boot/u-boot.git
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- rpi_4_defconfig
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

After the menu opens, the following settings need to be applied:

```
Boot options --->
 UEFI Support --->
  [ ] Support running UEFI applications

Device Drivers  --->
 [*] SPI Support  --->
  [*] Enable Driver Model for SPI drivers
  [*] Soft SPI driver

Library routines  --->
 Security support  --->
  [*] Trusted Platform Module (TPM) Support
 Hashing Support  --->
  -*- Enable SHA256 support

Device Drivers  --->
 TPM support  --->
  [ ] TPMv1.x support
  [*] TPMv2.x support (NEW)
  [*] Enable support for TPMv2.x SPI chips
  (65536) EventLog size (NEW)

Boot options  --->
 Boot images  --->
  [*] Measure boot images and configuration when booting without EFI
  [*] Measure the devicetree image (NEW)
```

The build itself is then simply started by calling:

```
make CROSS_COMPILE=aarch64-linux-gnu-
```

The result of this is (among other files) the u-boot.bin binary in the build folder.

Instead of using the Raspberry Pi's internal SPI controller, we switch to a "Soft SPI driver", meaning an SPI device realized in software. For that, the device tree overlay needs to be changed as well, telling this driver which pin plays which role in the SPI protocol. The adjusted device tree overlay was sourced from [13]. The following snippet shows how the Raspberry's GPIO pins are manually assigned to their SPI role:

```
gpio-sck = <&gpio 11 0>;
gpio-mosi = <&gpio 10 0>;
gpio-miso = <&gpio 9 0>;
cs-gpios = <&gpio 7 1>;
```

Another important change that needs to be made, is that U-Boot needs to get told where in memory the eventlog is to be stored. Every measurement taken by the bootloader implies the extension of this event log, containing information on what a measurement was about. In the configuration provided above, the event log is needed for the measurement to operate correctly. In [35] and [25] the authors suggest adding the *sml-base* as well as the *sml-size* directives to the device depicted in the overlay:

```
linux,sml-base = <0x00 0xfb600000>;
linux,sml-size = <0x10000>;
```

To arrive at these specific memory values, one has to consult the */proc/iomem* file on the running sensor [25]. This shows, the main memory address regions as well as which areas are reserved by the system:

```
40000000-fbffffff : System RAM
f7000000-fb5fffff : reserved
fb6a0000-fb75ffff : reserved
fb760000-fb760fff : reserved
fb761000-fb7e4fff : reserved
fb7e6000-fb7e8fff : reserved
fb7e9000-fb7fcfff : reserved
fb7fd000-fbffffff : reserved
```

Then a memory region towards the end, which is not reserved is chosen. The size is derived from the event log size set in the configuration earlier. The default value for this is 65536 which corresponds to 0x10000 in hex.

The rest of the overlay file stays unchanged to [19]. While editing, device tree overlay files have the human-readable *dts* format. To get the binary overlay usable by the Pi, the following command can be used (where *tpm-soft-spi.dts* is the input and *tpm-soft-spi.dtbo* the output):

```
dtc -O dtb -b 0 -@ tpm-soft-spi.dts -o tpm-soft-spi.dtbo
```

Once all the necessary files (u-boot image, device tree overlay, Linux image) are placed inside the Pi's boot partition, the device is ready to start up. To tell it how to do so, U-Boot needs a startup script containing the necessary commands to start the kernel image, which can be found here [27].

```
setenv bootargs 'console=serial0,115200 ... ima_hash=sha256'
fatload mmc 0:1 ${kernel_addr_r} kernel8.img
booti ${kernel_addr_r} - ${fdt_addr}
```

- **setenv**: Sets the name and value of an environment variable. In the script above this is used to set the memory location and its respective size to be later used when loading the kernel image. Furthermore, a variable *bootargs* is created, containing the arguments passed to the Linux kernel once it is booted. Using this method, specific boot argument values will be different for every installation, as the *PARTUUID* of the root partition is bound to be different for every system. One way to remedy this was also presented in [27], though not implemented.
- **fatload**: A command used to load an image to a specific memory location, in this specific case the Linux kernel image with the name *kernel8.img*. The memory location the kernel was loaded to is then stored in the variable *kernel_addr_r*.
- **booti**: One of the boot commands provided by U-Boot, in this case referring to "boot image". This first sets up the environment and then starts the kernel (internally using the bootm command).

Table 5.1: Measurement results for the attestation procedure

PCR index	Content
0	*EV_SEPARATOR* + Digest of U-Boot version string
1	*EV_SEPARATOR* + Digest of Device Tree + Linux *bootargs*
2-7	*EV_SEPARATOR*
8	Digest of kernel image
9	Digest of initrd buffer
10	IMA log entry digests

These commands are written to a simple text file (e.g. *boot.txt*) which then has to be converted into a U-Boot image called *boot.scr* which will be automatically found and executed by the bootloader. This is done by invoking the *mkimage* program provided through the package repositories of most common Linux distributions:

```
mkimage -A arm -T script -C none -n "Boot script" -d "boot.txt" boot.scr
```

The last necessary step is to tell the Raspberry's firmware to run the U-Boot binary instead of the Linux kernel, which can be done by adjusting the following line of *config.txt* inside the Pi's boot partition:

```
kernel=u-boot.bin
```

Restarting the system, U-Boot measures the boot process according to its "Legacy measured boot" rules [35]. The individual PCR values can be read using tpm2-tools:

```
sha256:
 0 : 0x88A54DB2A99C82EADE0FF8F9D146C3B81BE080C381070995B30D192D8D5E98B5
 1 : 0xC20F3DFAFBCFBF5305906F1DDB0E8CFAA847A767A523500748C3A82DC8B8F6E6
 2 : 0xE21B703EE69C77476BCCB43EC0336A9A1B2914B378944F7B00A10214CA8FEA93
 ...
 7 : 0xE21B703EE69C77476BCCB43EC0336A9A1B2914B378944F7B00A10214CA8FEA93
 8 : 0xBE6244593A83906D20A752A7C823C78ED8655C2EA62039FF85376737D0D2BFCC
 9 : 0xCFC7D8042593E188C59D2FD523F07A95D06DD3160F0955D8C34B0EB067F517B6
 10: 0xCCD1DB2D7C4FAD9C5536507FC83A64630F9E011A55DFE48D0458D0EB381895C8
```

The registers 2-7 contain the same value. The reason for that is, that the "Server Management Domain Firmware Profile Specification" of the TCG [44], demands a so called *EV_SEPARATOR* to be inserted into PCRs 0 to 7 after the "Pre-OS" measurements conclude. This separator has the value *0xFFFFFFFF*, the SHA256 digest of which is *ad95131bc0b799c0b1af477fb14fcf26a6a9f76079e48bf090acb7e8367bfd0e* , which extended into an empty PCR results in the observed hash. The table 5.1 shows what the specific PCRs contain. U-Boot uses PCRs 0, 1, 8 and 9 for meaningful measurements.

The event log, which the memory was reserved for via *linux,sml-base* and *linux,sml-size*, can be printed using *tpm2_eventlog* (part of *tpm2-tools*) and is located under */sys/kernel/security/tpm0/binary_bios_measurements*. It contains all the events that led to measurements into the PCRs, here for example, the measurement of the device tree:

```
- EventNum: 4
  PCRIndex: 1
  EventType: EV_TABLE_OF_DEVICES
  DigestCount: 2
  Digests:
  - AlgorithmId: sha1
    Digest: "4edb71d79074010da5f709bd0796c9a9a6a29cc2"
  - AlgorithmId: sha256
    Digest: "6e04880cbaec2a79939f3398f3a6cfcb106413af8532b7a0e20af994af2b30d3"
  EventSize: 4
  Event: "64747300"
```

According to */proc/iomem*, the event log memory area does not stay reserved once the kernel is running.

5.2.3 Enabling IMA

Since the default kernel used by Raspberry Pi OS does not have IMA support enabled, it needs to be recompiled with the specific flags set. Detailed instructions on how to do so can be taken from the Pi's documentation [14]. First, the latest version of the Linux kernel source, provided by the Raspberry Pi foundation, needs to be procured:

```
git clone --depth=1 https://github.com/raspberrypi/linux
```

Next, the necessary build dependencies need to be installed, which is dependent on the distribution installed on the build host. On a Debian system this looks like the following:

```
sudo apt-get install bc bison flex libssl-dev make libc6-dev libncurses5-dev \
crossbuild-essential-arm64
```

When building the kernel, the system needs to be told which version to build and what to call the output. This makes it easier to distinguish the built kernel containing the modifications:

```
KERNEL=kernel8
CONFIG_LOCALVERSION="-v7l-Remote-Attestation"
```

The U-Boot repository already contains a base config file for the Raspberry Pi 4, which can be built using the following command:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- bcm2711_defconfig
```

Just as with the U-Boot build before, the *menuconfig* can be utilized to set the necessary configuration flags:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- menuconfig
```

The changes are depicted below and are based on a sample implementation provided by the TPM's manufacturer [51]:

```
Security options  --->
 [*] Enable different security models
 [*] Integrity subsystem
 [*]   Integrity Measurement Architecture(IMA)
  Default template (ima-ng (default))  --->
  Default integrity hash algorithm (SHA256)  --->
  [*] Enable multiple writes to the IMA policy
  [*] Enable reading back the current IMA policy
 [*] SELinux Support

Device Drivers  --->
 [*] SPI support  --->
  <*> GPIO-based bitbanging SPI Master
 Character devices  --->
  -*- TPM Hardware Support  --->
  <*>   TPM Interface Specification 1.3 Interface / TPM 2.0 FIFO Interface - (SPI)
```

The documentation provided by the manufacturer of the TPM in use suggests using the onboard *BCM2835* SPI controller. When enabling both measured boot and IMA measurements this doesn't work. There is likely a problem when using the TPM using software based SPI in U-Boot and then using the dedicated hardware in the kernel. Using the software based driver for both eliminates this issue.

Another problem that was encountered when using the dedicated SPI hardware with IMA is, that for some older kernel versions the TPM would initialize after IMA started up. This would lead to the subsystem switching into a "passthrough" mode, simply not extending the measurements into the PCRs. For these kernel versions, the TPM's manufacturer provides kernel changes solving this issue, outlined in [51]. Newer kernels already include this fix, rendering the change obsolete.

After adjusting the configuration, the kernel can be built:

```
make ARCH=arm64 CROSS_COMPILE=aarch64-linux-gnu- Image modules dtbs
```

In a last step, the build output needs to be written to the Raspberry Pi's mass storage. To this end, the SD card is mounted and then the image itself, the base device tree as well as the overlays are copied over:

```
sudo cp mnt/boot/$KERNEL.img mnt/boot/$KERNEL-backup.img
sudo cp arch/arm64/boot/Image mnt/boot/$KERNEL.img
sudo cp arch/arm64/boot/dts/broadcom/*.dtb mnt/boot/
sudo cp arch/arm64/boot/dts/overlays/*.dtb* mnt/boot/overlays/
sudo cp arch/arm64/boot/dts/overlays/README mnt/boot/overlays/
```

The next step is to define the measurement policy to be used. This specifies which files are being measured by IMA, written to the IMA log and extended into PCR 10. There are a number of different built-in policies available and the policy used for this implementation is based on the built-in TCB (Trusted Computing Base) policy (defined in the Linux kernel, annotations copied from [21]):

```
dont_measure fsmagic=0x9fa0         # PROC_SUPER_MAGIC
dont_measure fsmagic=0x62656572     # SYSFS_MAGIC
dont_measure fsmagic=0x64626720     # DEBUGFS_MAGIC
dont_measure fsmagic=0x1021994      # TMPFS_MAGIC
dont_measure fsmagic=0x1cd1         # DEVPTS_SUPER_MAGIC
dont_measure fsmagic=0x42494e4d     # BINFMTFS_MAGIC
dont_measure fsmagic=0x73636673     # SECURITYFS_MAGIC
dont_measure fsmagic=0xf97cff8c     # SELINUX_MAGIC
dont_measure fsmagic=0x43415d53     # SMACK_MAGIC
dont_measure fsmagic=0x27e0eb       # CGROUP_SUPER_MAGIC
dont_measure fsmagic=0x63677270     # CGROUP2_SUPER_MAGIC
dont_measure fsmagic=0x6e736673     # NSFS_MAGIC
dont_measure fsmagic=0xde5e81e4     # EFIVARFS_MAGIC
measure func=MMAP_CHECK mask=MAY_EXEC
measure func=BPRM_CHECK mask=MAY_EXEC           # binary executed
measure func=FILE_CHECK mask=^MAY_READ euid=0
measure func=FILE_CHECK mask=^MAY_READ uid=0    # root opened r/o, r/w
measure func=MODULE_CHECK
measure func=FIRMWARE_CHECK
measure func=POLICY_CHECK
```

This policy starts by defining a number of different file systems which are not to be measured. The rules after that (starting with “measure”) define the kind of files which are going to be included:

- Every file which is memory mapped into a process (i.e. visible through /proc/pid/maps) which includes libraries or shared objects (.so)
- Every program for which the kernel creates a binary parameter structure, i.e. every program which is getting executed
- Anything read by root, with the read bit set
- Kernel modules as they get loaded
- Firmware being loaded
- Custom IMA policies as they are applied

While this policy already covers a lot of what is happening on the system, it has a major flaw. It tends to measure files which do not stay constant across reboots, like:

- */var/log/lastlog*: reporting user logins
- */var/lib/NetworkManager/timestamps*: network connection timestamps
- */etc/fake-hwclock.data*: Contains a timestamp from when the system was last shut down for continuity reasons

These files pose a problem, since the goal is to measure the IMA log at a specific point in time, and then use that log to confirm new measurements in the future. One has to expect that the system could have been restarted, otherwise reference values would have to be taken every reboot. Also, someone would have to make sure, that the system ends up in a secure state after every power cycle which could be difficult (who could tell what was done to the system since the last reboot?). In the optimal case, the parts of the system observed by IMA stay the same from the moment the manufacturer of a sensor is taking the reference measurements all the way up to the point where the system needs to be updated which would force remeasuring the reference values. The goal is therefore to find a tradeoff between what files are being measured, and having the measurement remain constant across a given timespan of the system running. Taking too many files out of the measurement potentially reduces its significance while keeping too many (also changing) files in affects the operation of the remote attestation.

There is no functionality within IMA to exclude specific files from the measurement procedure. IMA is however capable of applying rules on the basis of SELinux labels. These were used to gain a more fine-grained control over the exclusions. Per default, SELinux is disabled, which is why the respective configuration option was chosen in the kernel configuration's security section. Starting with the TCB policy, whenever a file with changed checksum occurred, its associated SELinux label (which can be printed via *ls -Z filename*) was added to the exclusion list, resulting in the following additions to the policy:

```
dont_measure obj_type=etc_t
dont_measure obj_type=lastlog_t
dont_measure obj_type=wtmp_t
dont_measure obj_type=ld_so_cache_t
dont_measure obj_type=NetworkManager_var_lib_t
```

In the case of */etc/fake-hwclock.data* this led to the exclusion of the label *etc_t* which encompasses parts of the *etc* directory. This in turn means that this approach will lead to many files not being measured even though they would be relevant. This approach also has the problem that the files which came up as changed during testing might not form an exhaustive list of files to exclude. Overall this solution is unsatisfactory and highlights the complexity of working with IMA logs. Possible solutions to these inconsistencies will be presented in a chapter 7.1.3.

5.3 Attestation Process

At this point, the Digidow components are set up to support the measurement of the sensor's state, which means that now the procedure itself should be introduced. Figure 5.3 gives an overview of the most important events taking place during the attestation process.

1. Upon starting up, the PIA retrieves a list of all available sensor nodes from the sensor directory and initiates the attestation procedure with each one, by calling the sensor's REST api (the *attest_init* endpoint to be specific).

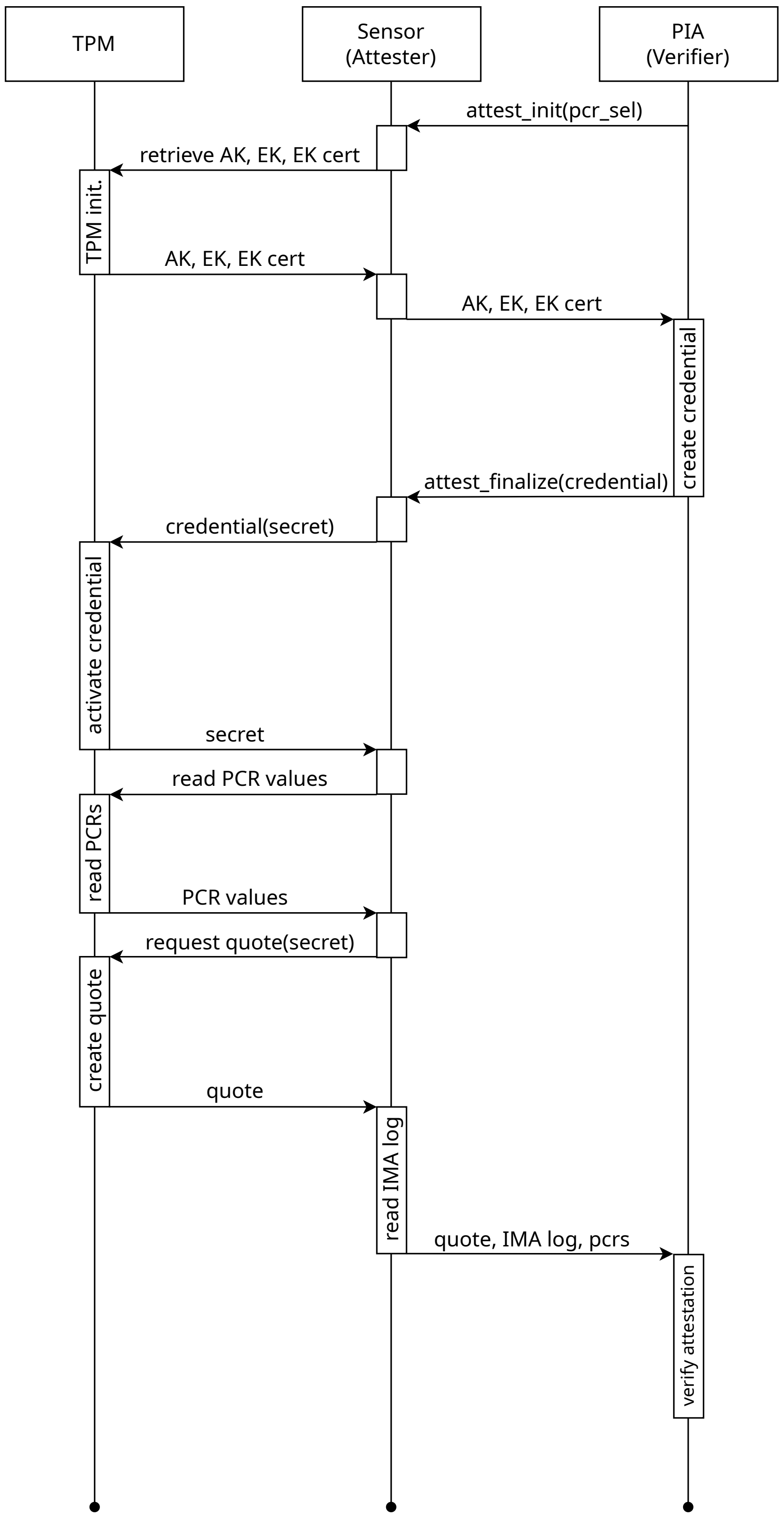


Figure 5.3: Remote attestation process implemented in the prototype.

2. When a sensor receives an attestation request, it creates a new attestation key by calling into its TPM. Since the EK is not necessarily present in the TPM, it also has to be generated from the stored seed. The attestation and the endorsement key, together with the endorsement certificate are returned to the PIA. The handles and context variables generated during this phase are stored in a shared data structure on the sensor.
3. The PIA then checks the endorsement key certificate and ensures that it trusts in it stemming from a genuine TPM (is signed by a trusted party, e.g. Infineon). It then generates a random secret and protects it in the form of a credential, a process explained in section 5.4.2. This credential is sent to the sensor in the form of an *attest_finalize* request.
4. After receiving the second request, the sensor first retrieves the stored attestation structure from its shared object. This is then used to open the credential and derive the attestation reply. Once the three TPM calls to open the credential, read the individual PCR values and create the PCR digest are done, the information is formatted accordingly and returned to the PIA.
5. At this point, the verifier can utilize the reference values associated with this specific sensor to verify the reported measurements and derive an attestation decision. The process outlining the various checks is described in 5.4.1.

5.4 PIA

This section describes the changes to the PIA to enable the attestation. First, the PIA contacts the sensor directory to retrieve a list of applicable sensors. At the time of writing, the sensor directory is not able to deliver sensors based on the location or detected intentions of the PIA so all sensors that are stored within it are returned. For each of these sensors, a new *SensorConnection* structure is created. The creation of such a structure triggers the registration of the PIA with the respective sensor, therefore copying the locally stored embedding to the sensor. Furthermore, an individual callback address is created, enabling the sensor to notify the PIA if the sent embedding is detected. The code for remote attestation has to go in between the start of the creation function and the registration itself, in order to be able to cut off the interaction in case the sensor is not trustworthy.

First, a new function *send_http_request(shared,request)* is introduced, allowing a user to send an arbitrary http request using the shared zwuevi instance.

```
async fn send_http_request(shared: &Shared,request: Request<String>)
 -> Result<String,...>{
 let mut personality_request_tries = 0;
 loop{
  let locked = shared.read().await;
  let reply = locked...send_http_request(request.clone()).await;
  drop(locked);
  match reply {
   Ok(reply) => {
    return Ok(reply);
   }
   Err(error) => {...}
  }
 };
}
```

Sending the request is tried *MAX_ATTESTATION_REQUEST_TRIES* amount of times, before the try is aborted. Using this function, the attestation starts with the retrieval of the reference data. Currently, this is just being stored in a file in the PIA's working directory. In the future the reference material will most likely be distributed using the sensor directory:

```
  let Ok(reference_data_file) =
   std::fs::read("./reference_psr...tqd.onion.json") else{...};
```

After that, the reference material needs to be checked against the state present in the transparency log. To this end one has to first request the latest log root, to retrieve the current size of the tree, and then request the inclusion proof for the individual reference set. First, the log root request:

```
 let Ok(latest_log_root_request) = http::Request::get(
  format!(
   "http://{}/log/latest-signed-log-root?log_id={}",
   PERSONALITY_ADDRESS,
   TREE_ID
  )
 )
```

And after that the request for the inclusion proof, where *body* is referring to the JSON structure representing the reference data:

```
let Ok(inclusion_proof_request) = http::Request::post(
 format!(
  "http://{}/log/inclusion-proof?log_id={}&tree_size={}",
  PERSONALITY_ADDRESS,
  TREE_ID,
  latest_log_root.tree_size
 )
).header("Content-Type", "application/json").body(body)
```

As described in section 2.5, getting a return value for an inclusion proof is not enough. The consistency of the tree based on the value returned by the transparency log needs to be shown, the process of which was also outlined previously. If the inclusion of the reference data could be proven, the attestation can continue. If not, and no trustworthy reference could be retrieved, the verification cannot be executed correctly, making the attestation meaningless. On such terms, the attestation (and by extension the registration) should be aborted. Once the reference data is secured, the PIA initiates the attestation process by calling the sensor's *attest_init* endpoint, including the list of PCR values to incorporate:

```
let attest_init_request = http::Request::post(
 format!("http://{}.onion/v1/attest_init", &sensor.url)
)
.body(
 json!({"pcrselect": PCR_SELECT,}).to_string()
)
let Ok(credential_data) =
 send_http_request(&shared, attest_init_request).await else{...};
```

At the moment, only PCRs 0-10 are selected, as these are the ones that the chosen measurement mechanisms (Measured Boot and IMA) use:

```
const PCR_SELECT: [u32; 11] = [0,1,2,3,4,5,6,7,8,9,10];
```

This array is later converted into a vector using *to_vec()*. The result is then passed to the *generate_credential()* function, which parses the returned data, checks the EK certificate as well as the key attributes associated with the endorsement key and returns the credential blob itself, the attestation key that was generated by the sensor for this attestation run, and the secret the sensor will have to return later:

```
let Ok((credential,attestation_key,secret)) =
 Attestation::genrate_credential(credential_data).await else{...};
```

This credential is then packed into the *attest_finalize request* and sent off to the sensor.

```
let attest_finalize_request = http::Request::post(
 format!("http://{}.onion/v1/attest_finalize", sensor.url.clone())
)
.body(credential)
```

Based on this, the sensor is then able to produce the data necessary for verification, the attestation evidence. This is returned and then handed to the PIA's *check_quote()* function, the result of which either aborts, or allows the following registration:

```
match Attestation::check_quote(
 quote_data, reference_data, attestation_key,
 credential_secret, sensor.url.clone(), PCR_SELECT.to_vec()
) {
 Ok(result) => {
  if result {
   info!("Quote verification for sensor {} went through.",sensor.url);
  }else{
   info!("Quote verification for sensor {} failed.",sensor.url);
   return;
  }
 }
 Err(err) => {
  info!("...",sensor.url, err);
  return;
 }
}
```

Figure 5.4 presents an overview of these HTTP requests:

5.4.1 Quote Verification

The checking operation starts by deserializing and unmarshalling the quote data, which is represented by a *QuoteReply* JSON structure:

```
pub struct QuoteReply{
 pub quote: Vec<u8>,
 pub signature: Vec<u8>,
 pub pcr_slots: IndexMap<u32, Vec<u8>>,
 pub ima_log: Vec<u8>,
}
```

The variables within the structure hold binary data encoded in BASE64 strings. After retrieving the binary data, the PIA starts with its checks. Figure 5.5 shows a waterfall diagram of these and the conditions that need to hold for each of these checks.

1. The secret is a 32 byte array, filled with random data. It is generated by the PIA upon generating a credential, which is then passed to the sensor. When the sensor passes it back to the PIA, it is compared to the one stored for two reasons:
   - **Endorsement:** to prove that the quote was created by a TPM which is endorsed by its respective manufacturer as the credential that encapsulated it can only be open if the endorsement key of the TPM the credential was aimed at is known.

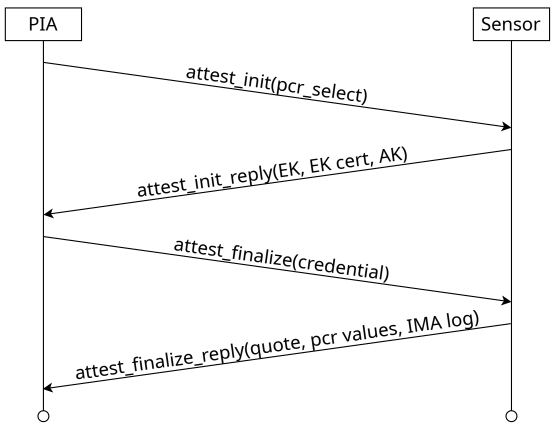

PIA
Sensor
attest_init(pcr_select)
attest_init_reply(EK, EK cert, AK)
attest_finalize(credential)
attest_finalize_reply(quote, pcr values, IMA log)

Figure 5.4: HTTP message flow of attestation procedure.

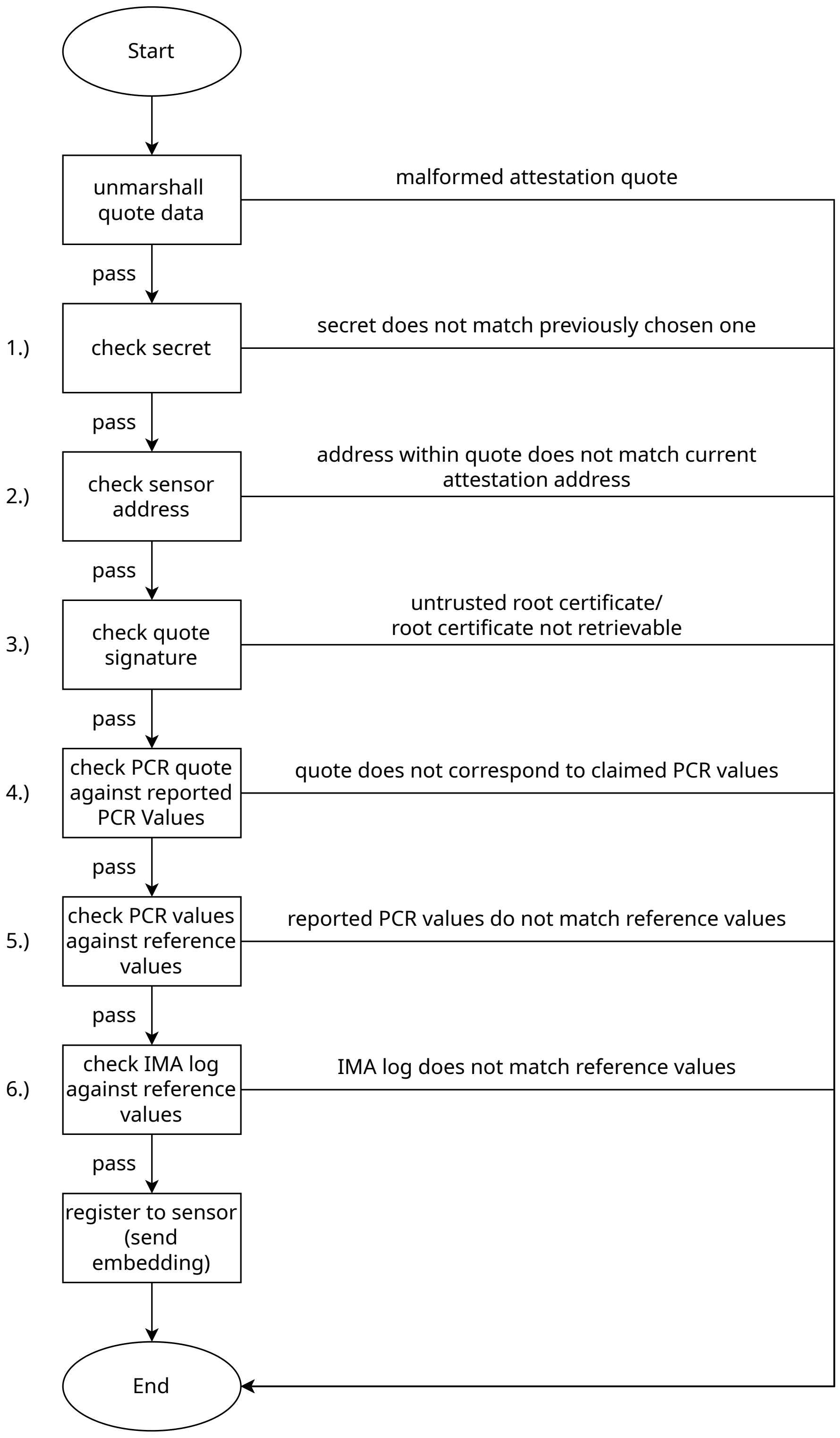

Start
unmarshall quote data
malformed attestation quote
pass
1.)
check secret
secret does not match previously chosen one
pass
2.)
check sensor address
address within quote does not match current attestation address
pass
3.)
check quote signature
untrusted root certificate/ root certificate not retrievable
pass
4.)
check PCR quote against reported PCR Values
quote does not correspond to claimed PCR values
pass
5.)
check PCR values against reference values
reported PCR values do not match reference values
pass
6.)
check IMA log against reference values
IMA log does not match reference values
pass
register to sensor (send embedding)
End

Figure 5.5: Checks performed to derive an attestation decision.

  - **Freshness:** to prove that the quote is not a replay from a previous attestation procedure, as the secret would have been a different one.

2. The attestation quote also contains the sensor address, which needs to be compared to the one that the attestation requests were sent to. That way, a malicious node will not be able to pose as a valid sensor using some other sensor's attestation data. Any attempt to do so would be thwarted by the fact that the PIA is then able to compare the address of the sensor it thinks it's attesting to, to the address of the sensor that is actually providing the attestation evidence. This addresses threat 8, presented in the threat model in chapter 4.
3. The quote is signed using the attestation key promoted to the PIA when requesting the credential. When the PIA checked the secret, it automatically proved the endorsement, meaning that the attestation key is resident on a genuine TPM and can thus be trusted. The PIA can then check the signature on the quote with the public portion of the attestation key which proves that this quote is coming from inside the TPM.
4. The sensor returns both the individual PCR values and a digest over all of them, only the latter of which is signed by the TPM. For this reason, the digest over the claimed individual values needs to be calculated and compared to the digest in order to extend the trust provided by the signature to the individual values.
5. Once the correctness of the PCR values has been established, they can be compared to their reference counterparts. This only works for PCRs 0-9 as PCR 10 contains the IMA log which is bound not to hold the same value as the one in the reference data.
6. In a last step, the IMA log needs to be checked. First, the correctness of the IMA log needs to be verified. This happens by taking the entries from the provided log and calculating the final digest. An important point while doing this is, to keep in mind that the IMA log is potentially newer than the PCR values it is provided with, meaning that the log could potentially contain more entries than are necessary to calculate the claimed hash in PCR 10. That means one has to go through the log and accumulate the digest. If the digest at any point matches the claimed PCR 10 value, the log up to this point is confirmed. Any log entries beyond that point are not attested and as such no longer interesting. If the calculated digest matches, one can start to check the individual entries and compare them to the IMA log provided as part of the reference data. The IMA policy was constructed in a way, that every entry in the newly acquired log should be the same in the reference log. If the new log contains an entry not within the reference data or a hash which does not match, the log has to be deemed malicious. All of this functionality is implemented in the *check_ima_log()* function, which heavily relies on the structures defined in section 2.4.

5.4.2 Credential Generation

Credentials are needed to prove that a genuine TPM is present on a system. This works through the endorsement key embedded into the TPM. The public portion of it is used to encrypt some data and tied to some object attributes, for example a name. If the TPM receives such a credential and an object with the specified name exists on the TPM, it uses the private portion of its endorsement key to retrieve the encrypted data. If this data is presented to someone else, it proves that a TPM is present as the only way it could have been encrypted would be the knowledge of the private endorsement key for which the manufacturer signed a certificate (endorsement). Thus, before the credential is created, the PIA needs to run two major checks:

- Check the endorsement certificate chain
- Check the endorsement key attributes, showing that it is fixed to a TPM, does not have a parent key and is not restricted (meaning it is not used for signing and does not have an associated signing scheme)

If these checks fail, creating a credential is not necessary, as the endorsement key cannot be trusted to begin with.

The process of credential creation only involves the name of the object, the public portion of the endorsement key as well as some data chosen by the verifier (the secret mentioned earlier). It is not dependent on any internal TPM state and as such does not need one to be present on the system. This also means that the function can be implemented in software only. The problem is that at the point of the creation of this thesis, there were no Rust library functions covering this use case. According to the specification [49], TPMs provide this functionality for convenience. Expecting every PIA to contain a TPM however is an unsatisfactory assumption as in its current design it would only be used for credential generation (which could change in the future). These circumstances prompted the reimplementation of the credential generation in Rust. This way the PIA can create the credential itself without relying on the environment it is running in, while also making the generation a lot faster.

The credential generation process is specified in [49] and includes two functions:

- The key derivation function described on page 45 of the specification.
- The credential generation function itself, outlined on page 167 of the specification.

Key Derivation Function: KDFa()

The key derivations function's signature as well as the interpretation of the respective parameters:

```
fn tpm2_kdfa(
 hash_alg:u16,
 key:Vec<u8>,
 label:Vec<u8>,
 context_u:Vec<u8>,
 context_v:Vec<u8>,
 bits:u32
) -> Result<Vec<u8>, AttestationError>
```

- **hash_alg**: the hash algorithm to be used for key derivation

- **key**: secret keying material used in the derivation
- **label**: character data containing information on the use of the key
- **context_u & context_v**: arbitrary data vectors
- **bits**: the desired length for the resulting key

The function starts by checking *hash_alg*, as it currently only supports SHA256. If any other ID is encountered it returns an error. The *bits* parameter is converted to bytes, the counter *i* is initialized with 1 and the derivation loop starts:

```
if hash_alg != SHA256_ALG_ID {return Err(...)}
let mut result: Vec<u8> = Vec::new();
let Ok(bytes) = usize::try_from(bits/8) else {return Err(...));};
let mut i: u32 = 1;
...
```

The loop runs as along as the result vector does not contain enough bytes. As soon as it does the result is truncated to the length defined by *bits* and returned. In the loop first an HMAC object is initialized using the secret *key*. After that, the counter *i*, the *label*, *context_u*, *context_v* as well as the desired length *bits* are accumulated into a byte buffer and fed to the HMAC object. If *label* is empty or does not end on a zero byte, one is inserted in between the label and the context area.

```
...
while result.len() < bytes {
 let hmac = Hmac::<Sha256>::new_from_slice(&key.as_slice());
 let Ok(mut hmac) = hmac else{...};
 let mut buffer: Vec<u8> = Vec::new();
 buffer.append(&mut i.to_be_bytes().to_vec());
 buffer.append(&mut label.clone());
 match label.last() {
  Some(last) => {
   if *last != 0u8 {
    buffer.append(&mut 0_u8.to_be_bytes().to_vec())
   }
  }
  None => buffer.append(&mut 0_u8.to_be_bytes().to_vec())
 }
 buffer.append(&mut context_u.clone());
 buffer.append(&mut context_v.clone());
 buffer.append(&mut bits.to_be_bytes().to_vec());
 hmac.update(buffer.as_slice());
 result.append(&mut hmac.finalize().into_bytes().to_vec());
 i+=1;
}
result.truncate(bytes);
Ok(result)
```

Credential Generation Function: MakeCredential()

The credential generation functions head as well as the interpretation of its respective parameters:

```
fn make_credential(
 ekpublic:Public,
 secret:Vec<u8>,
 akname:Vec<u8>
) -> Result<Vec<u8>,AttestationError>
```

- **ekpublic**: the public endorsement key of the TPM this credential is being created for
- **secret**: the secret data to be encrypted inside the credential
- **akname**: the name of the object which's presence is to be checked

Just like the key derivation function, this implementation of the credential generation is assuming a narrow selection of encryption, padding and hashing schemes and checks that it is adhered to. The code doing all these checks is omitted to avoid clutter. This function makes use of *Tpm2b* structures, which in the context of a TPM denotes a buffer which when serialized is preceded by its length. In a first step, such a buffer is initialized with the *secret*, creating the credential to encrypt:

```
let cv: Tpm2b = Tpm2b::new(secret);
```

Next a seed is generated "using methods of the asymmetric EK"[49]. In the case of an RSA key this means the seed should be randomly generated (using a cryptographically secure random number generator according to [37]) and encrypted with the endorsement key using "Optimal Asymmetric Encryption Padding" (OAEP) and the label "IDENTITY":

```
let mut seed= vec![0u8; SHA256_LENGTH];
let Ok(mut rng) = Hc128Rng::try_from_os_rng() else{...};
if rng.try_fill_bytes(&mut seed).is_err() {...}
let padding = Oaep::new_with_label::<Sha256, &str>("IDENTITY\0");
let Ok(enc_seed) = ek.encrypt(&mut thread_rng(), padding, seed.as_slice()) else{...};
```

Using the key derivation function described above, a key is generated using the seed, the name of the object whose residency is to be checked as well as the label "STORAGE".

```
let Ok(sym_key) = Self::tpm2_kdfa(
 SHA256_ALG_ID,
 seed.clone().to_vec(),
 "STORAGE".as_bytes().to_vec(),
 akname.clone(),
 Vec::new(),
 bits
) else {...};
```

Using “Cipher Feedback Mode” (CFB) this key is then used together with the associated symmetric encryption scheme (AES in our case) to encrypt the credential. The target TPM will be able to retrieve the plaintext credential by decrypting the seed and regenerating the symmetric key:

```
let mut enc_identity = cv.serialize();
Aes128CfbEnc::new(sym_key.as_slice().into(), &iv.into()).encrypt(&mut enc_identity);
```

Serializing *enc_identity* returns its binary buffer preceded by the buffers length. Next, an HMAC is calculated over the encrypted identity (the encrypted credential). To this end, another key is generated again using the seed but this time using the label “INTEGRITY”:

```
let Ok(hmac_key) = Self::tpm2_kdfa(
  SHA256_ALG_ID,
  seed.clone().to_vec(),
  "INTEGRITY".as_bytes().to_vec(),
  Vec::new(),
  Vec::new(),
  256
) else {...};
```

Using this key, the HMAC is calculated:

```
let Ok(mut hmac) = Hmac::<Sha256>::new_from_slice(hmac_key.as_slice()) else{...};
let mut outer_hmac: Vec<u8> = enc_identity.clone();
outer_hmac.append(&mut akname.clone());
hmac.update(&outer_hmac);
let outer_hmac = hmac.finalize().into_bytes().to_vec();
```

This concludes the credential generation process as described in the specification. Unfortunately, the different parts need to be arranged in a specific way to be correctly interpreted by the *activateCredential()* TPM call on the sensor node. The problem is that the header that needs to be specified is poorly documented. Luckily **tpm2-tools**[42] implements the functionality, which means its source code contains the correct structure. The project’s repository contains the necessary information: 4 magic bytes (“badcc0de”) followed by a 4 byte integer depicting the version (“1”).

5.4.3 Endorsement Certificate Check

To deal with certificates, the *openssl* crate was used to implement a function called *check_cert(cert:X509, intermediate_certs: Stack<X509>, trusted_root_certs: Option<Vec<X509»)*, which takes the certificate to check, all the necessary intermediate certificates to complete this check as well as optional trusted certificates to avoid having to alter the developer machines certificate store. First the function creates a new *X509Store*:

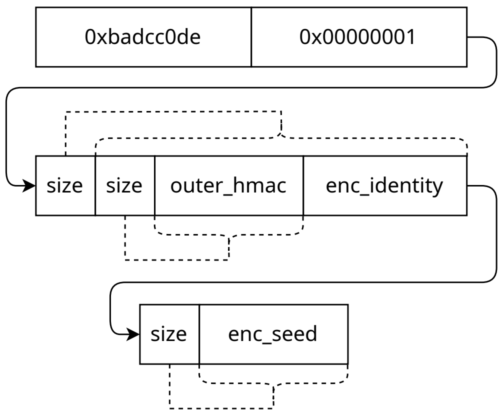


Figure 5.6: Structure of the final output of MakeCredential(). (sizes are 2 bytes, big-endian order)

```
let mut builder = match X509StoreBuilder::new(){
 Ok(cert_store_builder) => cert_store_builder,
 Err(err_stack) => return Err(...)
};
match builder.set_default_paths(){
 Ok(_) => {}
 Err(err_stack) => return Err(...)
}
```

Calling the *set_default_paths()* function on the *X509StoreBuilder* automatically retrieves the certificates from the system certificate store. The optional root certificates are also added:

```
if let Some(certs) = trusted_root_certs {
 for root_certificate in certs {
  match builder.add_cert(root_certificate){
   Ok(_) => {}
   Err(err) => {
    info!("Could not add root certificate: {}", err);
   }
  }
 }
}
```

Another (omitted) helper function, called *retrieve_intermediate_certs(cert: X509)* automatically tries to retrieve all the certificates needed to verify the chain. It does so by reading the respective *authority info* fields. This is just a placeholder as the plain HTTP connection in use would obviously break the security provided by Tor. In future instances, these certificates need to be provided by some other party, and passed to the *check_cert(...)* function. The *X509Store* with the trusted certificates, the certificate to check as well as the intermediate ones are then passed to a *X509StoreContext* and checked:

```
 match context.init(
  &store,
  cert.as_ref(),
  intermediate_certs.as_ref(),
  |ctx| {ctx.verify_cert()}
 ){
  Ok(ret) => Ok(ret),
  Err(err_stack) => Err(...)
 }
```

The verification validates the hierarchical relation between the certificates and makes sure the chain ends in one that's in the store of trusted ones. If that is the case, the trust in the initial certificate was successfully established.

5.5 Sensor

In its current stage, the Digidow sensor consists of 3 components:

1. The "face-lib" tasked with detecting and recognizing faces.
2. The "sensor-lib" responsible for providing the REST interface to connect to the sensor as well as managing registrations.
3. The "sensor" itself, tying all the functionality together.

Since registrations are initiated by the PIA and attestations are optional (especially when viewed from the perspective of the sensor), the registration logic should not be altered. The sensor just needs to make sure to provide a handler providing the necessary attestation proof upon request. The component handling requests towards the sensor is the "sensor-lib" which is thus a prime target for the necessary modifications/additions. Before attestation requests can be served though, the sensor needs to be able to "talk" to its TPM.

5.5.1 Reaching the Sensor

To enable the PIA to reach the sensor, the two previously described HTTP endpoints need to be defined. Since HTTP connections are traditionally stateless, the sensor needs to store the attestation parameters, until the PIA sends the second request. For this, the sensor holds a map of all currently running attestations in its shared object:

```
attestations: HashMap<String, Attestation>
pub fn add_attestation(&mut self, host:String, attest:Attestation) {
 self.attestations.insert(host, attest);
}
pub fn retrieve_attestation(&mut self, host:&String) -> Option<Attestation>{
 self.attestations.remove(host)
}
```

The first endpoint (*attest_init*) is used to initiate an attestation with the sensor:

```
(&Method::POST, "/v1/attest_init") => {
 let body_bytes = request.into_body().collect().await.unwrap().to_bytes();
 let body_str = String::from_utf8(body_bytes.to_vec()).unwrap();
 let attest = match Attestation::new(body_str) {
  Ok(attest) => {attest},
  Err(error) => {...}
 };
 let (resp,akpub) = match Attestation::create_credential_data(&attest){
  Ok(resp) => resp,
  Err(error) => {...}
 };
 reg.lock().unwrap().add_attestation(akpub,attest);
 Ok(Response::new(resp.into()))
}
```

It initializes the TPM structures necessary (AK, EK and EK cert), packages them up as a JSON object, stores the attestation structure in the shared object and returns the result to the PIA. The second endpoint (*attest_finalize*) retrieves the corresponding attestation structure from the shared object and creates the quote based on the received credential. The quote together with the individual PCR values and IMA log are then returned, again as a JSON object:

```
(&Method::POST, "/v1/attest_finalize") => {
 let body_bytes = request.into_body().collect().await.unwrap().to_bytes();
 let body_str = String::from_utf8(body_bytes.to_vec()).unwrap();
 let host = request.uri().host().unwrap().to_string();
 let Ok(body): Result<AttestFinalizeRequest, serde_json::Error> =
  serde_json::from_str(&body_str) else {...};
 let Some(attest) =
  reg.lock().unwrap().retrieve_attestation(&body.akpub) else{...};
 let quote = match Attestation::create_quote(attest, body.credential, host){
  Ok(quote) => quote,
  Err(error) => {...}
 };
 Ok(Response::new(quote.into()))
}
```

One important aspect here is that using the host address provided by the verifier to avoid the “replay” attack described in section 4.2 will not work if the correctness of the host field is not enforced by the underlying HTTP stack which one cannot rely on! The reasoning for this arguably very poor solution is the time at which this bug was discovered, and the related rework required within the sensor binary to provide the host address from its own internal state.

5.5.2 Interacting with the TPM

As already described in section 2.3.4, the TPM specification presents a few options on how to interact with the TPM chip. While calling other programs in the form of the “tpm2-tools” from within Rust would generally be possible, dealing with errors and the system dependency of having the programs installed is not acceptable. Fortunately, there is already a wrapper library called "tss_esapi"[32] available, granting access to the functions of the "libtss2-esys"[45] interface directly from within Rust. This crate does most of the heavy lifting, not only providing the library to Rust programs but also implementing abstractions for more complex tasks. Besides the library itself, its developers also provide several examples, showcasing the capabilities of the library. One of these examples "certify.rs"[32] already includes much of the functionality required for attestation, which is why some of the TPM related code is based on it.

For the crate to know where the TPM is, an environment variable called *TCTI* is set to the TPM device path:

```
TCTI=device:/dev/tpmrm0
```

5.5.3 Generating Reference Data

Before it makes sense to run an attestation, reference data needs to be created. If the provided proofs cannot be compared to a known set of values, one cannot derive a decision whether a system is in a known state. Creating reference data includes the following steps:

- Retrieve current PCR values
- Retrieve current IMA log state
- Retrieve additional information about the sensor system (e.g. software and hardware state)
- Arrange all of the above into a JSON file
- Create evidence that the derived information stems from a known trustworthy source
- Distribute the reference data for PIAs to use in their attestation

The additional information on hard- and software is currently just stored in a file called *Spec.json*. This could just as easily be automatically generated by running a set of commands to retrieve the system configuration, like *lshw* to print hardware information, *lsusb* to list usb devices or *uname -a* to get information about the kernel. How this is done is very much dependent on the specific sensor in question, its operating system, and its architecture. The only important thing is that it is called *Spec.json* and that it is not malformed. In the ideal case, this information would be retrieved directly in the sensor binary, every time a new set of reference data is requested. This way, one makes sure that the reference data always contains the newest information. The following structure is used to manage the reference data around the attestation process:

```
pub struct LogEntry {
 pub pcr_slots: IndexMap<u32, String>,
 pub ima_log: IndexMap<String, Vec<String>>,
 pub software_version: IndexMap<String,String>,
 pub hardware_specification: IndexMap<String,String>,
}
```

Reading the IMA log works exactly the same as if the log was read during attestation. The most important thing here is, that the PCR values are read before the IMA log, so that the digest in PCR 10 can be derived. Retrieving the reference data as a quote, meaning signed by an attestation key, does not make much sense, since the residency of the signing key could not be proved without a credential interaction.

5.5.4 Generating the Attestation Quote

To assemble the quote we need the following components:

- An attestation key capable of signing the quote
- The PCR selection to be included in the quote's digest
- A vector containing qualifying data
- The signature scheme to be used

Most of the attestation related code was added to the *sensor-lib* via a new source file called *attestation.rs*. It holds the structure *Attestation* and the first thing that happens upon a new attestation request is, that a new object of this structure is created by calling the structure's *new()* function:

```
pub struct Attestation{
 callback: String,
 pcrselect: PcrSelectionList,
 context: Context,
 _ek_alg: AsymmetricAlgorithmSelection,
 hash_alg: HashingAlgorithm,
 _sig_alg: SignatureSchemeAlgorithm,
 ek_cert: Vec<u8>,
 ek_public: Public,
 ek_context: TpmsContext,
 ak_public: Public,
 ak_context: TpmsContext,
}
```

It holds information like the PCRs to be added to the quote, the context of the TPM device in use, the algorithms in use during the attestation as well as information on the public portions of the endorsement and the attestation key, together with their respective saved contexts. When calling the *new()* function, the algorithms are set to default values, the endorsement key is read, a new attestation key is generated and the PCR list is parsed. After this structure is initialized, the information necessary for the PIA to generate a credential is assembled as a JSON structure and returned in the form of an HTTP reply:

```
let body = json!({
 "ekcert": BASE64_STANDARD.encode(attest.ek_cert.clone()),
 "ekpub": BASE64_STANDARD.encode(ek_pub),
 "akpub": BASE64_STANDARD.encode(ak_pub),
}).to_string();
```

Now, the PIA generates the credential and issues an *attest_finalize* request, as outlined before. When this arrives at the sensor, it holds all the necessary information to generate the attestation response. To begin with, the context for both EK and AK, stored in the shared object is restored:

```
 //load ek and ak context from attestation object and retrieve new handles
 let Ok(ek_handle) = context.context_load(attest.ek_context) else{
  return Err(...);
 };
 let Ok(ak_handle) = context.context_load(attest.ak_context) else{
  return Err(...);
 };
```

Next, the secret within the credential needs to be retrieved. For this, two authorizations are needed: one for the attestation key and one for the endorsement key. Correspondingly two separate authentication sessions are started:

```
let Ok(Some(ak_auth_session)) = context.start_auth_session(
 None,
 None,
 None,
 SessionType::Hmac,
 SymmetricDefinition::AES_128_CFB,
 attest.hash_alg,
) else{
 return Err(...);
};

let Ok(Some(endorsement_auth_session)) = context.start_auth_session(
 None,
 None,
 None,
 SessionType::Policy,
 SymmetricDefinition::AES_128_CFB,
 attest.hash_alg,
) else{
 return Err(...);
};
```

For authorizing the access to the attestation key, an HMAC session is enough. Accessing the endorsement hierarchy requires a policy session, which means that the corresponding assertion functions need to be called as well:

```
 if context.execute_with_nullauth_session(|ctx| {
  ctx.policy_secret(
  PolicySession::try_from(endorsement_auth_session).unwrap(),
  AuthHandle::Endorsement,
  Default::default(),
  Default::default(),
  Default::default(),
  None,
  )
 }).is_err() {...}
```

Per default, the secrets necessary to access the endorsement key as well as the attestation key are empty. As soon as the necessary authorization sessions are in place, the credential retrieved from the PIA needs to be parsed. The structure (depicted in figure 5.6) is stripped of the magic value as well as the version and split into two blocks, one containing the *enc_identity* with its corresponding *outer_hmac* and the *enc_seed*. These together with the sessions can then be passed to the *activate_credential()* wrapper function:

```
let credential = credential_blob[...].to_vec();
let encrypted_secret = credential_blob[...].to_vec();
let Ok(credential) = context.execute_with_sessions(
 (Some(ak_auth_session), Some(endorsement_auth_session), None),
  |ctx| {
   ctx.activate_credential(
    ak_handle.into(),
    ek_handle.into(),
    IdObject::try_from(credential)?,
    EncryptedSecret::try_from(encrypted_secret)?,
   )
  }
) else{return Err(...);};
```

Afterwards, the authorization sessions are dropped, and a new one is created to run the creation of the quote. Again, an HMAC session is enough as only the attestation key needs to be accessed and reading PCR values does not require any authorization. The quote allows for the inclusion of an additional byte vector, the so called *qualifying_data*. This is added to the quote structure within the TPM and consequently signed together with the PCR digest. In this implementation, it is used for the freshness, as well as the sensor identity checks as outlined in the threat model in chapter 4. Since the field is only allowed to be as long as the hash values produced by the corresponding PCR bank (SHA256 = 32 Bytes), the two vectors are concatenated and the checksum (SHA256 hash in this case) over the result is used:

```
let mut qualifying_data_digest = Sha256::new();
qualifying_data_digest.update(credential.to_vec());
qualifying_data_digest.update(onion_address.into_bytes());
let qualifying_data = qualifying_data_digest.finalize().to_vec();
```

After that, the quote can be generated:

```
let Ok(attestation_quote) = context.execute_with_session(Some(session), |ctx| {
 ctx.quote(
 KeyHandle::from(ak_handle),
 Data::try_from(qualifying_data.clone())?,
 tss_esapi::structures::SignatureScheme::RsaSsa {
   hash_scheme: HashScheme::new(attest.hash_alg)
 } ,
 attest.pcrselect.clone()
 )
}) else{return Err(...);};
let (quote,signature) = attestation_quote;
```

And then the individual PCR values:

```
 let Ok(pcr_values) = tss_esapi::abstraction::pcr::read_all(
  &mut context, attest.pcrselect.clone()
 )else {...};
```

An important thing to keep in mind is, that in the time between creating the quote and reading the PCR values, PCR 10 could have been extended by an IMA measurement. The function is not atomic and as such, there needs to be a check, whether the digest inside the quote matches the PCR values. If they don't, the process needs to be repeated, calling for the following loop:

```
 let (quote, signature, pcr_values) = loop{

  //Read operations ...

  //Calculate expected pcr digest
  let mut quote_digest_check = Sha256::new();
  for digest in pcr_bank {
   quote_digest_check.update(digest.1.to_vec());
  }
  let expected_digest = quote_digest_check.finalize().to_vec();

  match quote.clone().attested() {
   AttestInfo::Quote{info} => {
    if info.pcr_digest().to_vec().eq(&expected_digest) {
     break (quote,signature,pcr_values);
    }
   }
   _ => {return Err(...)}
  }
 };
```

After the PCR values and their digest are correctly read, the IMA log is procured from */sys/kernel/security/ima/binary_runtime_measurements*. Here, the timing of the read is not as important, as long as it is done after the PCRs were read. If the log contains an entry that was not yet extended into PCR 10 they can simply be discarded by the PIA while checking the log, while still producing the correct value. The read binary representation of the file, as well as the byte vectors for the quote and its accompanying signature are encoded as BASE64 and added to the following JSON structure.

```
 let body = json!({
  "quote": BASE64_STANDARD.encode(quote_bytes),
  "signature": BASE64_STANDARD.encode(signature_bytes),
  "pcr_slots": pcr_slots,
  "ima_log": BASE64_STANDARD.encode(ima_log),
 });
```

This JSON string is then returned to the PIA as the reply for the *attest_finalize* request, concluding the sensor's involvement in the attestation procedure. Using that, the PIA can derive a trust decision as outlined in chapter 5.4 and based on the outcome, invoke the sensor's registration endpoint.

Chapter 6

Testing

The system used for testing is a Raspberry Pi 4 with 4 GB of RAM. It is connected to the institute's internal network and thus shares the university's internet uplink. The sensor binary in use is the version resulting from commit *182c2fef252b488bb6564338995f05b50eb5dc8d*[1]. The Linux kernel in use is version *6.12.38*, built from (the Raspberry Pi foundation) source [12] resulting from commit *9c09b75242960117155712f41ce540df2e3cd63c*[2] using the configuration described in 5.2.3. The U-Boot binary was built from its respective repository [34] with the state resulting from commit *1c250e444ad3b15315ee8b0fcb3fc3acc26449e2*[3] based on the configuration described in section 5.2.2.

6.1 A Note on Building

The Raspberry Pi 4, in the configuration used for testing, is not capable of compiling the sensor binary itself, as it is seemingly running out of memory. A potential solution is to use or increase the size of a swap file/partition, though the build speed still does not lend itself to quick iteration (>1h build time). Since the sensor is built using *Nix*, a Pi 5 was used as a remote build host. When building the sensor binary on Pi 5, the part that takes the longest is compiling the sensor's dependencies, clocking in at approximately 50 minutes. An unfortunate side effect of building via Nix is that a small code change, like inserting timing measurements, triggers Nix to rebuild all dependencies as it deems them as one unit called "sensor-deps". This obviously drastically increases the time it takes to test code changes on the sensor.

The rest of these sections describes the necessary configuration changes to use the Raspberry Pi 5 as a remote builder, based on information provided at [10] and [6].

6.1.1 On the Builder (Pi 5)

First, the builder defines the features it supports, which is important if requested by a party initiating the build job (change in */etc/nix/nix.conf*):

```
system-features = nixos-test benchmark big-parallel kvm aarch64-linux
```

[1]https://git.ins.jku.at/proj/digidow/sensor/-/commit/182c2fef252b488bb6564338995f05b50eb5dc8d
[2]https://github.com/raspberrypi/linux/commit/9c09b75242960117155712f41ce540df2e3cd63c
[3]https://github.com/u-boot/u-boot/commit/1c250e444ad3b15315ee8b0fcb3fc3acc26449e2

After that, the user designated for building, needs to be added to the set of Linux users trusted by Nix (change in */etc/nix/nix.conf*):

```
trusted-users = sensor
```

Lastly, the PATH variable needs to be exported once an SSH connection is started, to allow the system user SSH logged into to access the Nix binaries. To this end, the contents of the PATH variable are added to the ssh server configuration (*/etc/ssh/sshd_conf*):

```
SetEnv PATH=/home/sensor/.cargo/bin:/home/sensor/.nix-profile/bin-
:/nix/var/nix/profiles/default/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin-
:/usr/bin:/sbin:/bin:/usr/local/games:/usr/games
```

The contents of the PATH variable can be extracted from the build host using:

```
echo $PATH
```

6.1.2 On the Target (Pi 4)

First, Nix needs to be made aware of the fact, that there are build systems available within the network. To this end, the following line is added to */etc/nix/nix.conf*:

```
builders = @/etc/nix/machines
```

The *machines* file contains one entry per build host, similar to the following:

```
sensor@192.168.240.89 aarch64-linux /home/user/.ssh/id_rsa 4 1
```

From left to right the entry contains: the user as well as the IP of the build host, the architecture which the build host builds for, the location of the key used to authenticate, the maximum number of jobs the builder can serve as well a speed factor. Lastly, to make sure that Nix can access the build system, a public key has to be registered with its ssh server. This can be done using the following command:

```
ssh-copy-id sensor@192.168.240.89
```

This will copy the key */home/user/.ssh/id_rsa*. For security reasons one might want to generate a key for use with the build host exclusively.

6.2 Attestation Cost

6.2.1 Approach

In this section, the total time spent running the actual logic of the attestation is measured. For this purpose, the Rust built-in *Instant* structure is used, which can be used to measure the time that passed between the *now()* and *elapsed()* call. In terms of timings the following measurements were taken:

- The overall attestation time, from its start until it is accepted.
- The round trip time for both the *attest_init* and the *attest_finalize* requests.
- The time it takes the sensor locally to serve the *attest_init* and the *attest_finalize* request respectively. This allows the calculation of the delay incurred by the Tor network.
- The time it takes to interact with the transparency log.
- Access times for the various TPM requests at play (creating EK, activating credential, creating the quote, reading PCR values)

These timings should give a good understanding of the performance impact the attestation entails.

6.2.2 Results

The measurements were taken a total of 10 times with the following timings being the respective average. Table 6.1 shows the results of these measurements.

In the current setup, the attestation takes on average 32 s with a comparatively low deviation of 4.44 s. The combination of transparency log interaction, *attest_init* and *attest_finalize* interactions on average account for 99.4% of the overall attestation time. The delay inflicted by the Tor network is not always constant as it relies on the current state of the network and the circuit nodes that are chosen for a given connection. Building the circuit also incurrs a cost, which can be seen when comparing the delay for the *attest_init* request to that of the *attest_finalize* request, the latter of which is much lower (6.86 s vs 1.04 s). Applying this logic to the round trip time of the reference checks, a large portion of the very high deviation of 6.52 s can be simply explained by Tor circuit creation (which the first interaction with the transparency log would obviously

Table 6.1: Measurement results for the attestation procedure

Measure	**Average**	**Stdev**
Overall attestation time	32.36s	4.44s
attest_init round trip time	13.07s	3.74s
attest_finalize round trip time	6.05s	0.8s
attest_init sensor execution time	6.22s	83ms
attest_finalize sensor execution time	5.01s	92ms
attest_init Tor delay	6.86s	3.75s
attest_finalize Tor delay	1.04s	0.75s
Complete transparency log interaction	13.06s	6.52s
TPM EK creation	495.6ms	15.55ms
TPM Activate credential	689.76ms	18.85ms
TPM quote creation	506.19ms	12.92ms
TPM PCR value retrieval	540.49ms	16.82ms

also be exerted to).

The code that is running locally on the sensor, has a very low deviation of 82.6 and 91.9 milliseconds respectively. It is noteworthy however, that the Raspberry Pi SoC is thermally limited, due to the face recognition models running on the CPU rather than an AI accelerator. This could have an impact when the system is running for extended amounts of time. On a Raspberry Pi, thermal throttling can be checked using the following command:

```
user@sensor:~/sensor $ vcgencmd get_throttled
throttled=0xe0000
```

As per a blog post [8] on the official Raspberry Pi forum, bit 17 and 18 of that result show that, on the one hand, the SoC frequency was capped and, on the other hand, the system is actively thermal throttling.

In terms of time cost, the 4 observed TPM invocations are rather close together with a general low deviation. On average, the sensor is spending a total of 558 milliseconds per TPM operation. Given that 3 of them are necessary at minimum, it is to be expected that at least 1.67 s on TPM cost are spent on every attestation.

Chapter 7

Conclusion

This thesis presents a possible implementation of a remote attestation procedure within the CDL Digidow. It demonstrates the necessary techniques and software components, and integrates seamlessly with the existing registration process. During the implementation of the practical parts, a number of limitations as well as avenues for future work manifested themselves, which are to be addressed in the following.

7.1 Limitations

7.1.1 Performance

Section 6.2 surfaced clear shortcomings of the attestation procedure in terms of timing/cost. To a certain degree, this is due to the Tor network. This is not limited to the cost of building up a circuit to access a hidden service, but also to occasional stability problems, that manifest themselves in reset connections:

```
[2025-10-15T12:43:39.995Z INFO  sensor_lib::rest] Could not send credential request:
tor: tor operation timed out: Failed to obtain hidden service circuit to […]tad.onion
2025-10-15T12:42:48.546Z ERROR [pia_core::net::client] Could not fetch sensors: tor:
 tor operation timed out: Failed to obtain hidden service circuit to ???jyd.onion
```

Above, one can see two such fails on the sensor as well as the PIA side respectively. These failed connections cost a significant amount of time.

The minimum TPM cost of 1.67 seconds is also a problem, as this means that no matter how fast the network connections or the sensor binary run, it is unrealistic to serve multiple users in parallel while not breaking real time constraints.

Saving up on round trips is one of the main ways one can deal with these problems. A promising solution would be to directly distribute the endorsement key, the endorsement certificate and an attestation key. That way, the verifier can generate the credential directly, presenting all the needed information in one go. Also, the sensor could use a single attestation key instead of recreating them on every request. Since this would cut down TPM interactions as well, the sensor code would likely also run faster.

7.1.2 Bootloader and Chain of Trust

As already mentioned in the prior chapters, the Raspberry Pi bootloaders are not suited for creating a robust chain of trust. In the ideal case, one would rely on the first measurement to be done by immutable code read from ROM. This does not happen on the Pi, bloating its *Root of Trust for Measurement* by including the first, second and even third stage bootloader with the latter two being changeable. This obviously begs the question of why it is reasonable to develop an attestation procedure for the Pi at all, prompting a set of justifications:

- The Pi is hosting a prototype face recognition binary and can thus be considered a prototype itself. It is not expected to be deployed as-is but rather using hardware with better support for TPM measurements from software in ROM. In short, it is a stand-in for a more secure platform, that was chosen because it is easy to develop for.
- While it does not provide all the potential security guarantees provided by more sophisticated platforms, the implemented attestation still increases the cost for an attacker dramatically, as altering the bootloader without having physical access and without triggering a suspicious IMA measurement is not trivial.

7.1.3 IMA Log

There are some issues that were encountered when working with the IMA log files.

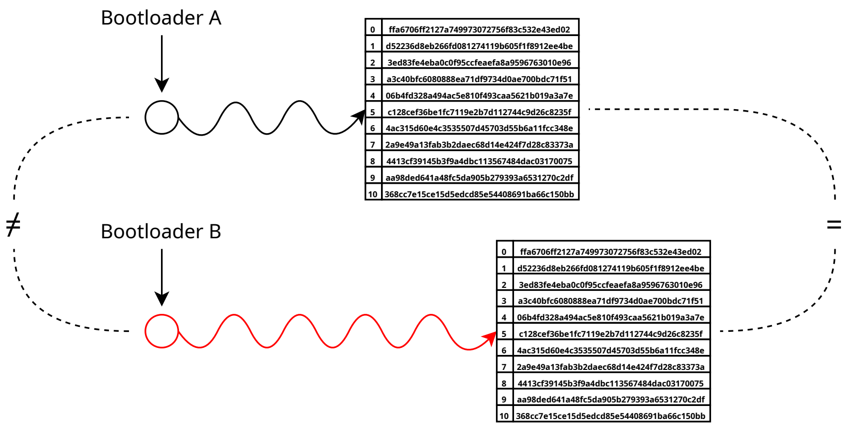
Log size

A problem which was already identified previously in the thesis of Michael Preisach is, that depending on the uptime of the sensor, the IMA log could potentially grow to a size of "about 40MB"[33]. At the moment, this problem is alleviated by increasing the allowed request size in the Rocket configuration. This of course is far from a proper fix.

The IMA documentation proposes a potential solution to this, via "incremental attestations" [22]. The idea is to "incrementally" check the IMA log with recurring attestations, while not including the parts that were already checked. In theory at some point the IMA log will settle down, as no new accesses on files that have not already been measured will take place, leading to fewer and fewer changes. This does not alleviate the first, large initial check, since many measurements happen early after startup, but it at least mitigates the effects of a large log later during operation. For this to work, one of the communicating parties has to store the value of PCR 10 at the point of the last attestation, so that PCR 10 can still be evaluated for consistency.

Policy

As alluded to earlier, the choice of IMA policy caused quite a bit of problems. Choosing the policy too tightly, the coverage of the attestation tends to be



0	ffa6706ff2127a749973072756f83c532e43ed02
1	d52236d8eb266fd081274119b605f1f8912ee4be
2	3ed83fe4eba0c0f95ccfeaefa8a9596763010e96
3	a3c40bfc6080888ea71df9734d0ae700bdc71f51
4	06b4fd328a494ac5e810f493caa5621b019a3a7e
5	c128cef36be1fc7119e2b7d112744c9d26c8235f
6	4ac315d60e4c3535507d45703d55b6a11fcc348e
7	2a9e49a13fab3b2daec68d14e424f7d28c83373a
8	4413cf39145b3f9a4dbc113567484dac03170075
9	aa98ded641a48fc5da905b279393a6531270c2df
10	368cc7e15ce15d5edcd85e54408691ba66c150bb

0	ffa6706ff2127a749973072756f83c532e43ed02
1	d52236d8eb266fd081274119b605f1f8912ee4be
2	3ed83fe4eba0c0f95ccfeaefa8a9596763010e96
3	a3c40bfc6080888ea71df9734d0ae700bdc71f51
4	06b4fd328a494ac5e810f493caa5621b019a3a7e
5	c128cef36be1fc7119e2b7d112744c9d26c8235f
6	4ac315d60e4c3535507d45703d55b6a11fcc348e
7	2a9e49a13fab3b2daec68d14e424f7d28c83373a
8	4413cf39145b3f9a4dbc113567484dac03170075
9	aa98ded641a48fc5da905b279393a6531270c2df
10	368cc7e15ce15d5edcd85e54408691ba66c150bb

Figure 7.1: Two distinct execution thread leading to the same measurement.

miniscule and the derived security gain is too. Choosing it looser and a lot of files are measured, causing a set of reference data to break quickly due to the measurement of a file which was not part of the initial measurement.

7.2 Optimizations and Future Work

7.2.1 Platform Endorsement

A broader problem with Digidow sensors being built upon commodity hardware is, that there is no form of platform endorsement, like the platform certificates introduced in section 2.3.3. There is thus no guarantee, that the S-RTM was really executed, leading to the problem depicted in figure 7.1.

These are two distinct execution threads leading to identical measurements. The verifier only observes the PCR-values. From its perspective, these two systems are thus the same. One has to assume that there was some sort of "chain of trust", that at some point more trustworthy code measured less trustworthy code to make it accountable. The reality however is, that one cannot know how the platform arrived at a measurement, if its origin is uncertain. And this is not necessarily an exclusive problem for the Raspberry Pi, but might also be a problem for a more general set of open platforms. If the platform manufacturer (which is not necessarily the same as the TPM manufacturer) does not provide endorsement, that the initial code runs and cannot trivially be interrupted, the whole measurement has to essentially be drawn into question.

In regard of the solution presented in this thesis, this has two main consequences:

- There needs to be an architectural component introduced into Digidow to do the platform endorsement of sensors. This could either be a completely new entity or added to the responsibilities of an existing one. This has another consequence:

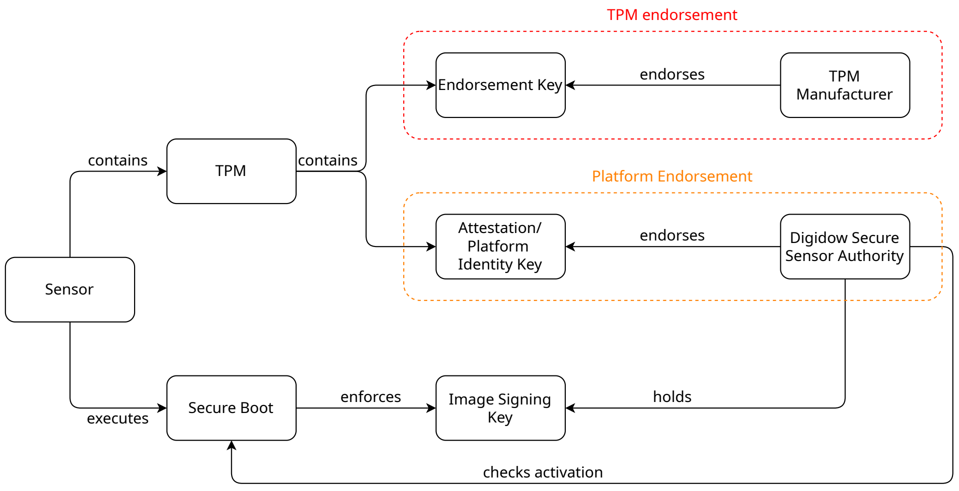


Figure 7.2: Proposed architecture to allow for platform endorsement.

- In the context of remote attestation, sensors need to be treated as individuals as opposed to types. There needs to be a platform certificate and ID for every sensor node. When using UEFI to run measured boot, PCR values will also be different for every sensor (e.g. PCR 1 measuring UEFI partition). For the manufacturer to be able to provide reference material for some type of device, it would thus be necessary to omit some of the PCRs containing device specific measurement. This further strengthens the need to treat sensors on an individual basis.
- The Raspberry Pis need to be configured in a way that ensures that the S-RTM runs all the time. This could for example be done by utilizing the Pi's secure boot feature, which locks the device down to only boot signed images. Exclusively signing bootloaders with enabled TPM support effectively ensures, that the measurement happens, as there is no other way to start the device otherwise.

Figure 7.2 gives an overview over the proposed architectural changes.

Each sensor gets its own “Platform Identity Key”, which is secured by its TPM. The presence of this key can be proven via the directives described in previous sections. The “Digidow Secure Sensor Authority” then checks that secure boot is enabled and if it is, issues a certificate for the platform ID. This way sensors gain an additional source of endorsement, which proves to the verifiers, that the provided evidence, originates from a valid S-RTM.

7.2.2 Measuring Sensor Directory

One of the major findings of this thesis was, how it is not reasonable, that every PIA attests to every sensor it wants to register to. This would create a high number of attestation requests on the sensors, quickly overwhelming what a typical TPM is capable of in terms of performance. Furthermore, the attestation performance itself is too low for PIAs to register to sensors quickly. It therefore

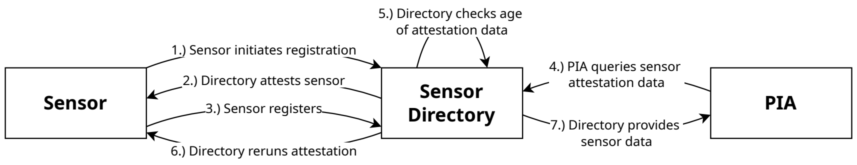

5.) Directory checks age of attestation data
1.) Sensor initiates registration
2.) Directory attests sensor
3.) Sensor registers
6.) Directory reruns attestation
Sensor
Sensor Directory
4.) PIA queries sensor attestation data
7.) Directory provides sensor data
PIA

Figure 7.3: Remote attestation run by the sensor directory.

makes sense to shift the attestation away from the PIA to a more centralized position, detached from the registration process.
The one entity which lends itself best under these requirements is the sensor directory. Figure 7.3 shows a proposal of how such an attestation could work.

1. The sensor is trying to enter the Digidow system by initiating the registration process with the/a sensor directory,
2. Then the sensor directory attests the sensor, making sure it is in a trustworthy state.
3. If the sensor could convince the directory of its trustworthiness, it successfully registers to the directory.
4. At some point a PIA queries the directory for this specific sensor, also requesting the attestation data for that sensor.
5. Since the sensor state potentially changed after being attested the last time, the sensor directory has to ensure that it is still in a trusted state. For that it randomly generates a time frame from a previously defined interval, for which the attestation is valid. This time is chosen at random, to prevent a potential attacker on the sensor directory from knowing when to expect the next attestation, making it dangerous to run an attack at any given point in time, because there might not be enough time to remove the evidence.
6. If the attestation is not valid anymore, the directory reruns the process.
7. The necessary information is assembled and returned to the PIA, which can now use the attestation data to derive the attestation decision normally. Having the PIA evaluate the attestation itself takes load from the sensor directory and allows PIAs to decide for their own attestation policy.

The result of this process is obviously, that the PIA has to put a lot more trust into the sensor directory than before. Another problem is, that sensors need to be part of a sensor directory to use attestation which imposes limitations on out-of-band sensor management. It will however alleviate the performance problems described above and keep sensors secure from denial-of-service attacks by using a separate, hidden attestation service address only known to the directory.

7.2.3 Algorithm Support

The attestation procedure is currently limited to:

- RSA2048 for asymmetric encryption
- AES128 for symmetric encryption

- SHA256 for hashing
- RSASSA for signing

Since TPMs also support elliptic curve cryptography, which would allow for shorter keys, it would make sense to extend the support to these as well. In general, a broader support for different crypto schemes would be desirable.

7.2.4 NixOS

Quite a bit of time during the creation of this thesis was allocated to getting NixOS running on the Raspberry Pi. Ultimately, these efforts did not produce anything of use, not least because of the slow build times in emulated aarch64 environments. NixOS offers vast potential in terms of how the sensor platforms software is built and deployed with current initiatives within the institute aiming to provide:

- Reproducible builds using Nix
- Complete aarch64 cross compiling on more readily available x86 build hosts

If and when these endeavors will bear fruits is subject of future developments; but if they do, there is a lot of potential for NixOS to improve the sensor's software stack and ultimately making it more predictable and thus more secure.

7.2.5 Attestable Update

An important addition to the presented thesis is the ability to update the system without having to remeasure everything. Schemes like this already exist, e.g. [50].

Bibliography

[1] Sigurd Frej Joel Jørgensen Ankergård, Edlira Dushku, and Nicola Dragoni. 2021. State-of-the-Art Software-Based Remote Attestation: Opportunities and Open Issues for Internet of Things. *Sensors*, 21, 5. https://www.mdpi.com/1424-8220/21/5/1598.

[2] Alexander Sprogø Banks, Marek Kisiel, and Philip Korsholm. 2021. Remote Attestation: A Literature Review. (2021). DOI: 2105.02466. arXiv: 2105.02466.

[3] Henk Birkholz, Dave Thaler, Michael Richardson, Ned Smith, and Wei Pan. 2023. Remote ATtestation procedureS (RATS) Architecture. RFC 9334. (January 2023). DOI: 10.17487/RFC9334.

[4] George Coker, Joshua Guttman, Peter Loscocco, Amy Herzog, Jonathan Millen, Brian O'Hanlon, John Ramsdell, Ariel Segall, Justin Sheehy, and Brian Sniffen. 2011. Principles of remote attestation. In *International Journal of Information Security*. Springer, Bedford, MA, USA, (June 2011). DOI: 10.1007/s10207-011-0124-7.

[5] European Commission. 2025. EU Digital Identity Wallet. Retrieved 11/26/2025 from https://ec.europa.eu/digital-building-blocks/sites/spaces/EUDIGITALIDENTITYWALLET/pages/694487738/EU+Digital+Identity+Wallet+Home.

[6] Nix Community. 2025. Distributed build. Retrieved 11/12/2025 from https://wiki.nixos.org/wiki/Distributed_build.

[7] Dmitry Kasatkin, "mzohar". 2023. Sourceforge IMA Wiki. Retrieved 08/01/2025 from https://sourceforge.net/p/linux-ima/wiki/Home/.

[8] "dom" - Raspberry Pi Engineer. 2025. Raspbian Jessie linux 4.4.9 Severe Performance Degradation. Retrieved 10/17/2025 from https://forums.raspberrypi.com/viewtopic.php?f=63&t=147781&start=50#p972790.

[9] NixOS Foundation. 2025. Nix and NixOS | Declarative builds and deployments. Retrieved 12/02/2025 from https://nixos.org/.

[10] NixOS Foundation. 2025. Remote Builds. Retrieved 10/15/2025 from https://nix.dev/manual/nix/2.28/advanced-topics/distributed-builds.html.

[11] NixOS Foundation. 2025. Single-user installation. Retrieved 12/10/2025 from https://nixos.org/download/.

[12] Raspberry Pi Foundation. 2025. Kernel source tree for Raspberry Pi. Retrieved 10/15/2025 from https://github.com/raspberrypi/linux.

[13] Raspberry Pi Foundation. 2025. Raspberry Pi software. Retrieved 12/10/2025 from https://www.raspberrypi.com/software/.

[14] Raspberry Pi Foundation. 2024. The Linux kernel. (June 2024). Retrieved 08/06/2025 from https://www.raspberrypi.com/documentation/computers/linux_kernel.html.

[15] Aurélien Francillon, Quan Nguyen, Kasper B. Rasmussen, and Gene Tsudik. 2014. A minimalist approach to Remote Attestation. In *2014 Design, Automation and Test in Europe Conference and Exhibition*. IEEE, Germany, pp. 1–6. DOI: 10.7873/DATE.2014.257.

[16] Google. 2025. Android Verified Boot 2.0. Retrieved 12/04/2025 from https://android.googlesource.com/platform/external/avb/+/android16-qpr2-release/README.md.

[17] Google. 2025. Hardware-backed Keystore. Retrieved 10/09/2025 from https://source.android.com/docs/security/features/keystore.

[18] Google. 2025. Key and ID attestation. Retrieved 10/09/2025 from https://source.android.com/docs/security/features/keystore/attestation.

[19] Johannes Holland. 2022. TPM 2.0 in U-Boot on Raspberry Pi 4. (September 2022). Retrieved 08/06/2025 from https://github.com/joholl/rpi4-uboot-tpm.

[20] Philip Howard. 2025. The Raspberry Pi GPIO pinout guide. Retrieved 11/12/2025 from https://pinout.xyz/.

[21] IBM. 2023. IMA Documentation. Retrieved 07/10/2025 from https://ima-doc.readthedocs.io/en/latest/index.html.

[22] IBM. 2023. IMA Event Log - IMA Log Verification. Retrieved 10/15/2025 from https://ima-doc.readthedocs.io/en/latest/event-log-format.html#ima-attestation-recommendations.

[23] Dell Technologies Inc. 2025. Secured Component Verification. Retrieved 11/13/2025 from https://www.delltechnologies.com/asset/en-us/solutions/business-solutions/technical-support/secured-component-verification-datasheet.pdf.

[24] HP Inc. 2025. HP Platform Certificate Download Package. Retrieved 11/13/2025 from https://enterprisesecurity.hp.com/s/article/HP-Platform-Certificate-Download-Package?language=en_US.

[25] Esa Jääskelä. 2024. Raspberry Pi 4, LetsTrust TPM and Yocto. Retrieved 07/09/2025 from https://ejaaskel.dev/raspberry-pi-4-letstrust-tpm-and-yocto/.

[26] William A. Johnson, Sheikh Ghafoor, and Stacy Prowell. 2021. A Taxonomy and Review of Remote Attestation Schemes in Embedded Systems. *IEEE Access*, 9, 142390–142410. DOI: 10.1109/ACCESS.2021.3119220.

[27] Itachi Lab. 2023. U-Boot on Raspberry. (2023). Retrieved 12/10/2025 from https://itachilab.github.io/u-boot-on-raspberry/u-boot-on-raspberry.html.

[28] Ben Laurie, Adam Langley, and Emilia Kasper. 2013. Certificate Transparency. RFC 6962. (June 2013). DOI: 10.17487/RFC6962.

[29] Michael Roland and Tobias Höller and René Mayrhofer. 2023. Digitale Identitäten in der physischen Welt: Eine Abwägung von Privatsphäreschutz und Praktikabilität. article. (January 2023). DOI: 10.1365/s40702-023-00949-1.

[30] NIST. 2022. NIST Retires SHA-1 Cryptographic Algorithm. Retrieved 10/02/2025 from https://www.nist.gov/news-events/news/2022/12/nist-retires-sha-1-cryptographic-algorithm.

[31] Unique Identification Authority of India. 2025. Aadhaar. Retrieved 11/26/2025 from https://www.uidai.gov.in/.

[32] Parsec. 2024. TSS 2.0 Rust Wrapper over Enhanced System API. Version 7.6.0. (December 2024). Retrieved 09/25/2025 from https://docs.rs/tss-esapi/latest/tss_esapi/.

[33] Michael Preisach. 2022. System Integrity and Attestation for Biometric Sensors. thesis. linz, (January 2022).

[34] Das U-Boot Project. 2025. "Das U-Boot" Source Tree. Retrieved 10/15/2025 from https://github.com/u-boot/u-boot.

[35] Das U-Boot Project. 2025. Measured Boot. Retrieved 10/16/2025 from https://docs.u-boot.org/en/latest/usage/measured_boot.html.

[36] Roberto Román, Rosario Arjona, and Iluminada Baturone. 2023. A lightweight remote attestation using PUFs and hash-based signatures for low-end IoT devices. *Future Generation Computer Systems*, 148, 425–435. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.06.008. https://www.sciencedirect.com/science/article/pii/S0167739X23002236.

[37] The rust-random Developers. 2025. The Rust Rand Book. Retrieved 12/04/2025 from https://rust-random.github.io/book/.

[38] Josef Scharinger. Lectures on Biometric Identification. Johannes Kepler University Linz - Austria, (2024).

[39] Joshua Schiffman and Monty Wiseman. 2024. Supply Chain Assurance Using TCG Technology. (January 2024). Retrieved 11/13/2025 from https://csrc.nist.gov/csrc/media/Presentations/2024/supply-chain-assurance-using-tcg-technology/images-media/supply-chain-assurance-using-tcg-technology.pdf.

[40] Hailun Tan, Wen Hu, and Sanjay Jha. 2015. A remote attestation protocol with Trusted Platform Modules (TPMs) in wireless sensor networks. *Security and Communication Networks*, 8, 13, 2171–2188. DOI: 10.1002/sec.1162. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sec.1162.

[41] tpm2-software community. 2024. OSS implementation of the TCG TPM2 Software Stack (TSS2). Version 4.1.3. (May 2024). Retrieved 09/25/2025 from https://github.com/tpm2-software/tpm2-tss.

[42] tpm2-software community. 2024. tpm2-tools. (April 2024). Retrieved 09/25/2025 from https://github.com/tpm2-software/tpm2-tools.

[43] Trusted Computing Group (TCG). 2024. TCG Platform Certificate Profile. Version 2.0. (July 2024). Retrieved 11/13/2025 from https://trustedcomputinggroup.org/wp-content/uploads/TCG-Platform-Certificate-Profile-Version-2.0-Revision-39.pdf.

[44] Trusted Computing Group (TCG). 2020. TCG Server Management Domain Firmware Profile Specification. Version 1.00. (December 2020). Retrieved 11/06/2025 from https://trustedcomputinggroup.org/wp-content/uploads/TCG_ServerManagDomainFWProfile_r1p00_pub.pdf.

[45] Trusted Computing Group (TCG). 2020. TCG TSS 2.0 Enhanced System API (ESAPI) Specification. Version 1.00. (May 2020). Retrieved 09/25/2025 from https://trustedcomputinggroup.org/wp-content/uploads/DICE-Attestation-Architecture-v1.2_pub.pdf.

[46] Trusted Computing Group (TCG). 2025. Trusted Platform Module 2.0 Library Part 1: Architecture. Version 184. (March 2025). Retrieved 07/09/2025 from https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-1-Version-184_pub.pdf.

[47] Trusted Computing Group (TCG). 2025. Trusted Platform Module 2.0 Library Part 2: Structures. Version 184. (March 2025). Retrieved 07/09/2025 from https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-2-Version-184_pub.pdf.

[48] Trusted Computing Group (TCG). 2025. Trusted Platform Module 2.0 Library Part 3: Commands. Version 184. (March 2025). Retrieved 07/09/2025 from https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-3-Version-184_pub.pdf.

[49] Trusted Computing Group (TCG). 2024. Trusted Platform Module Library Part 1: Architecture. Version 1.83. (January 2024). Retrieved 07/31/2025 from https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-1-Architecture.pdf.

[50] Ahmad B. Usman and Mikael Asplund. 2024. Remote Attestation with Software Updates in Embedded Systems. In *2024 IEEE Conference on Communications and Network Security (CNS)*. IEEE, Taipei - Taiwan, (October 2024), pp. 1–6. DOI: 10.1109/CNS62487.2024.10735526.

[51] WenXin Leong, Artem Yushev. 2022. Remote Attestation Optiga TPM. (February 2022). Retrieved 08/20/2025 from https://github.com/Infineon/remote-attestation-optiga-tpm.

[52] Lee Wilson. 2013. The TCG Dynamic Root for Trusted Measurement. (June 2013). Retrieved 11/13/2025 from https://trustedcomputinggroup.org/wp-content/uploads/DRTM-Specification-Overview_June2013.pdf.