SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Establishing trust in an updatable fTPM using remote attestation

Andreas Korb

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Establishing trust in an updatable fTPM using remote attestation

Herstellung von Vertrauen in ein aktualisierbares fTPM durch Remote Attestierung

Author:	Andreas Korb
Supervisor:	Prof. Claudia Eckert
Advisors:	Albert Stark, Johannes Wiesböck
Submission Date:	15.12.2023

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Munich, 15.12.2023 Andreas Korb

Acknowledgments

Abstract

Contents

1 Introduction

This chapter includes an explanation of the exact problem we are addressing, and why, a brief overview of our solution, and the attacks we are trying to fend off.

1.1 Motivation

Modern trust relationships, such as Zero Trust [**isaca2021**], require trustworthy platforms, which can reliably report their system state. In such models, trustworthiness can only be assumed after the platform configuration has been proved by all parties of the communication.

This is solved by remote attestation. In the simplest case, there is a prover and a verifier, as depicted in Figure 1.1. This requires the verifier to establish trust with software or hardware on the prover's machine that attests the remaining software running on the prover.

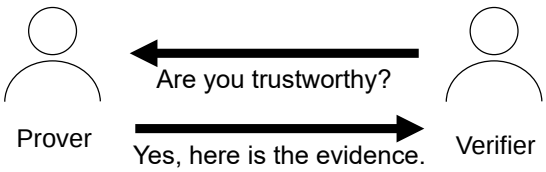


Figure 1.1: Simplified remote attestation process.

For example, this can be done with a Trusted Platform Module (TPM) on the prover's side. They rise in their deployments and importance, e.g., in 2013 the President's Council of Advisors on Science and Technology encourages the adoption of TPMs [**usa**], and Microsoft publicized that they require a TPM module for Windows 11 in 2021 [**win11req**]. They provide remote attestation mechanisms of system states, and their applications are still expanding beyond their traditional use-cases. For example, they are used in anti-cheat software for games [**valorant**].

A dedicated hardware TPM (dTPM) increases cost and hardware complexity - especially for embedded platforms. Through Trusted execution environments (TEEs), such as Arm TrustZone, a firmware TPM (fTPM) can be used to provide similar security guarantees as a dTPM chip.

For a dTPM, which consists of an independent hardware unit manufactured by a single manufacturer and is directly activated by power, it is sufficient to identify its manufacturer and understand his provided guarantees. In contrast, an fTPM runs atop other firmware components and is started later in the boot chain, making its security dependent on the underlying firmware stack. As a result, the fTPM can only be trusted if the entire underlying firmware stack is also trusted, since the underlying firmware could modify, i.e., compromise, the fTPM component which is then not detected.

However, while a TPM-compliant component provides an infrastructure with which trust in it can be established remotely, i.e., an endorsement certificate, the underlying firmware stack is not represented by this.

Currently, this is solved by the manufacturer providing not only the fTPM, but also the entire underlying firmware stack. Consequently, by establishing trust to the manufacturer of the fTPM, one can implicitly trust the underlying firmware as well by assuming they also originate from this manufacturer. This is possible since in the most general sense, one can derive from an endorsement certificate the endorser, i.e., manufacturer, and if the attester trusts the manufacturer and the guarantees he provides, trust is established to his provided components.

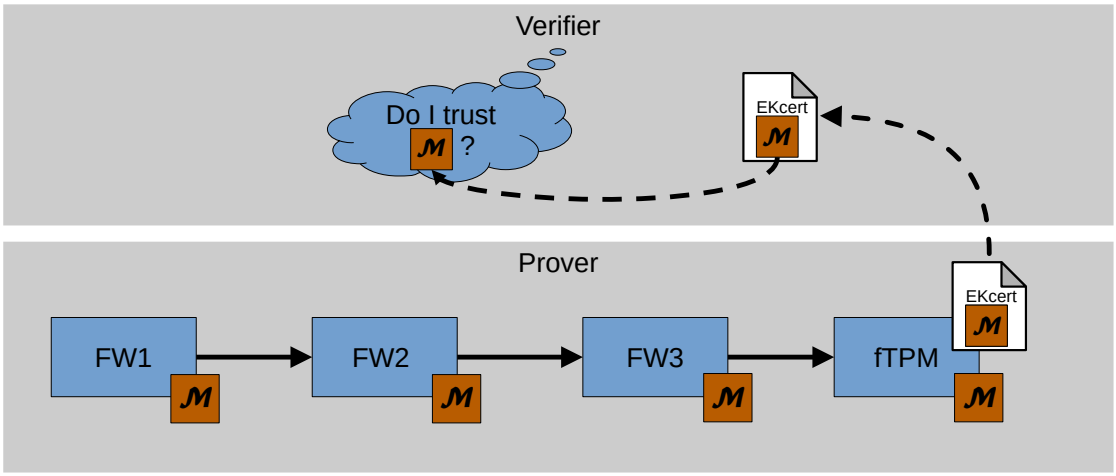


Figure 1.2: The naive process how a verifier establishes trust to an fTPM, which is in fact done by trusting its manufacturer. The brown markers indicate a manufacturer. The firmware and the fTPM were built by manufacturer $\mathcal{M}$, and the EK certificate indicates this manufacturer.

This process is illustrated in figure Figure 1.2. The provers' area shows its boot chain, the verifiers' area shows how it evaluates the trustworthiness against the provers' boot chain. The verifier trusts the entire firmware chain if he trusts the manufacturer of each

individual component. Note how the verifier must assume that the manufacturer of the firmware components is the same as the manufacturer of the fTPM. To the best of our knowledge, this is what manufacturers like Intel and AMD implement for their fTPMs, as confidence in their fTPMs is also only established through an EKcert.

In summary, with the current approach, the endorser, usually a CPU manufacturer, provides the firmware up to the fTPM and guarantees the firmware is not modifiable by untrusted parties. This enables to establish trust the other firmware components this manufacturer provided without knowing the firmware. This approach is limited, as with this mechanism, independent verifiers have to blindly trust the firmware manufacturer, which drastically limits trust relationships.

1.2 Goal

We establish independently verifiable fTPM stack, rooted in a hardware root of trust, that can be leveraged in a zero trust environment without requiring additional hardware or compromising on security. The goal of this approach is to break the requirement of the underlying firmware and the fTPM to originate from the same manufacturer, by providing the exact firmware component identities to the verifier, such that it can decide for itself whether they are trustworthy without relying on the manufacturer.

One mechanism enabling firmware attestation is the Device Identifier Composition Engine (DICE), focusing on resource-constrained devices. Although this mechanism shifts trust from the firmware provider to the hardware provider by allowing firmware attestation through a hardware root of trust, the exclusive use of this integrated solution is unsuitable for large dynamic systems, for example Linux based devices. Nevertheless, the advantage is that the identity of each component of the firmware boot chain is represented.

We propose a hybrid solution, combining the advantages of DICE and fTPMs, yielding an independently verifiable certificate chain representing the boot chain up to and including the fTPM. This enables a verifier to establish trust in an fTPM if the underlying firmware is benign as well and thus, providing a way to independently assess the properties of the fTPM.

The research questions are:

- What constitutes the identity of an fTPM?
- How to combine the DICE and TPM infrastructure?
- How to manage an fTPM's persistent data securely?
- How to enable privacy for the attestation mechanism?

1.3 Threat Model

The main threat is the modification of the binary of the fTPM before or during boot. For example, by exchanging the SD card storing the binary. However, we assume that the fTPM cannot be modified by malicious parties after the boot process regardless of whether the fTPM is benign or compromised because we trust the TEE environment. Out-of-scope are hardware attacks, side-channel attacks, control-flow attacks, and Denial of Service attacks.

For the network, we assume the Dolev-Yao attacker model [**Dolev1983**]. That is, we consider an attacker who has the ability to perform any active or passive attacks on the network. The attacker may also have control over parts or the entire network, e.g., all routers, switches, and connections. However, attackers are limited in that they cannot control the end systems. They also cannot break cryptographic primitives, e.g., encryption, signing, and hashing.

1.4 Environment

This work was created at the 'Fraunhofer-Institut für Angewandte und Integrierte Sicherheit AISEC' in Garching. It is part of the 'Fraunhofer Society for the Promotion of Applied Research e. V.', which is an organization distributed over Europe with main focus on applied research. In the roughly 35 years of its existence, it rose to become the largest research institute in Europe with around 30,000 employees.

1.5 Outline

2 Background

This chapter discusses the relevant background knowledge required to understand the remainder of this work.

2.1 Trusted execution environment

One of the core security concepts of operating systems are the privilege levels of processes [**Linden1976**]. Thereby, processes are protected against other processes with the same or lower privilege level. However, they are not protected against more privileged processes [**Bratus2009**]. This bears problems for example for cloud computing and edge computing. In cloud computing, other services, the hypervisor, or the cloud provider in general could potentially access sensitive data of the cloud tenant [**Nimgaonkar2012**]. In edge computing, the edge applications deal with plain text data, while they are potentially running on insecure edge devices [**Ning2018**]. Hence, protection against more privileged processes is desired.

The Trusted execution environment (TEE) is a technology defined by GlobalPlatform[1] as an integrated hardware extension to processors. By that, the execution environment is separated into the Rich execution environment (REE) and the TEE by hardware. The REE runs commodity software, e.g., a Linux-based operating system with user applications. The TEE is an isolated tamper-resistant execution environment that guarantees the authenticity of the executed code, and the integrity of runtime states, e.g., memory [**Sabt2015**]. Since a TEE is integrated into the processor, there is no separate chip required. Moreover, the TEE commonly follows the same user and kernel space separation as a rich OS. The kernel space is running a trusted OS, and the user space is running the trusted applications. It focuses on resisting software-based attacks generated in the REE, however, also protects against some hardware attacks [**GPSysArch**].

Previous, mostly software-based technologies ensure confidentiality and integrity protection of data-in-transit and data-at-rest [**Pecholt2022**], while a TEE additionally protects data-in-use in hardware [**Pecholt2022**, **Lee:EECS-2022-96**].

[1] `https://globalplatform.org/`

Figure 2.1 illustrates the motivation. In the traditional architecture, i.e., without a TEE, if an attacker compromises the REE the full system is affected. With a TEE, the attacker is limited to the REE, while the TEE continues to protect the secure assets, such as encryption keys. This results from the observation that the attack surface of a rich OS is much larger than that of a trusted OS, e.g., due to its network connectivity and the high dynamics of software installations, while the attack surface of a trusted OS is rather small and has tightly controlled interfaces.

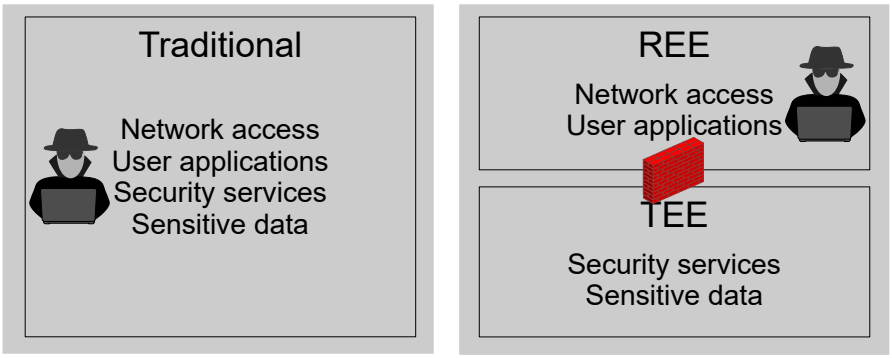


Figure 2.1: Comparison between a traditional architecture, and an architecture separating the REE and TEE. This illustrates the motivation of a TEE.

2.1.1 Arm TrustZone

One such TEE is ARM's TrustZone [**ARM09**, **Ngabonziza2016**]. It partitions all software and hardware resources of the containing system into the Normal world (NW) and the Secure world (SW), as shown in Figure 2.2. The secure monitor is triggered by the dedicated instruction Secure Monitor Call (SMC), which then manages the context switches between the NW and the SW. While the SW can access the resources of the SW and the NW, the NW is restricted to its own assigned resources. Since ARM is the dominant processor architectures for IoT devices with a market share of 86 % [**eclipse**], many of the approaches in this field of research use Arm TrustZone [**Valadares2021**].

Our implementation also leverages TrustZone to enable the execution and the remote attestation of an fTPM.

2.1.2 Further TEE technologies

Other TEE technologies are Intel Software Guard Extensions (SGX), and AMD Secure Encrypted Virtualization (SEV), in the future also Intel Trusted Domain Extensions (TDX), and ARM Confidential Computing Architecture (CCA). Since we focus on

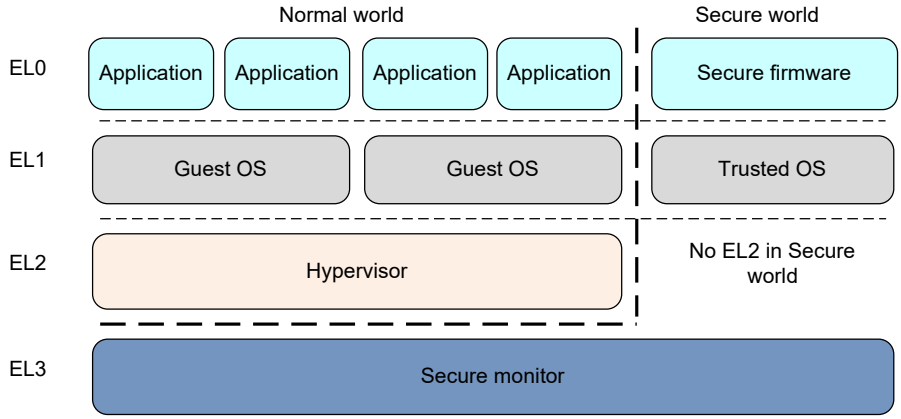


Figure 2.2: The architecture of Arm TrustZone for AArch64 [**TZArch**]. The exception levels (EL) indicate the privilege levels.

the implementation of our concept with ARM TrustZone, we do not go into detail about these other technologies here. However, since our concept is not tied to ARM processors and can also be applied to others, they are mentioned for the sake of completeness.

2.2 Attestation

According to NIST SP 1800-19B [**Bartock2022**] an attestation is "the process of providing a digital signature for a set of measurements securely stored in hardware, and then having the requester validate the signature and the set of measurements." Specifically in our context, attestation is a mechanism for software to prove its identity. In the following, the two types are discussed.

2.2.1 Local attestation

Local attestation is a procedure in which the state of a computer is measured, whereby the measurement result does not leave the computer but is used directly by a local component. One such example is a TPM that releases data, e.g., an encryption key, only when the computer is in a known state. This feature is known as sealing [**tpm**].

2.2.2 Remote attestation

In contrast to that, the measurement, usually called evidence, leaves the measuring machine, and is transmitted to a remote verifier for a remote attestation. This in-

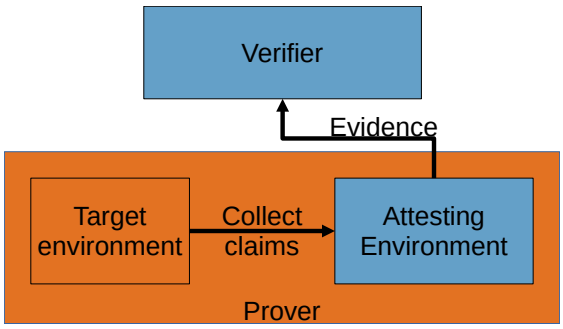


Figure 2.3: Data flow of remote attestation [**rfc9334**]. Initially, only the blue areas are trusted by the verifier. With the attestation, the verifier can choose to trust the target environment based on its measurements.

volves cryptographic primitives to establish trust into this evidence which is generally transmitted through an untrusted network.

Remote attestation is a challenge-response protocol initiated by a remote attestor. Figure 2.3 depicts a simplified overview of the data flow of a remote attestation. The process is initiated by a remote trusted party (called "verifier") to verify that a target environment on the end-device (called "prover") has not been tampered with [**Menetrey2022**, **Coker2011**]. This challenge contains a nonce, enforcing a fresh response. The response must be an evidence of the challenged system that it is trustworthy. To build that, an attesting environment on the prover device generally inspects the following properties of a program: (i) its code and data has been correctly loaded into memory for execution, and (ii) its data has not been maliciously modified at runtime.

The attesting environment acts a root of trust for measurements (SRTM). It is required on the prover because at least one trusted component is necessary to conduct trusted measurement of the prover. In many cases, the SRTM is running within the TEE, since thereby it is better protected from attacks.

It typically consists of two parts [**McCune2008**]. (i) The attestation, and (ii) the accompanying establishment of a secure channel. In this work, we focus on the first step.

A remote attestation procedure can also be divided into two categories, implicit and explicit attestation. In implicit attestation, the state of the verifier is implicitly inferred from the fact that the verifier has control over a signing key that is only accessible in a known state. With explicit attestation, the state of the device is explicitly described in the evidence.

2.3 Trusted Platform Module

The Trusted Computing Group (TCG)[2] published the first TPM specification (v1.2) in 2009 [**ISO11889**], and the most current specification (v2.0 Revision 01.59) ten years later in 2019 [**tpm**]. It describes a cryptographic coprocessor that increases trust in the host platform. Specifically, this means that the platform exhibits the expected behavior and that this behavior can be trusted. For that, the TPM maintains a separated state from the host platform, which enables the TPM to take measurements of the host platform. It is also a passive device, meaning it only does something when prompted. Table 2.1 summarizes the main features of TPMs.

Table 2.1: TPM main features and exemplary use-cases.

Feature	Use-case
Device identification	Identify a machine before granting it access to resources
Key Storage	Store encryption keys securely
Random Number Generator	Seed the key generation algorithms
Platform Configuration Registers	Store measurements of system components

Each key generated by a TPM is derived from two parameters, a source of object entropy and a key template. The object entropy for a primary key comes from the according primary seed, for a derived key from the parent key. A template contains metadata of the key, such as the type of key, e.g., RSA, and the object attributes. The attributes assign for example whether the key can be used for encryption or signing. Keys can also be restricted, which restricts the key to be used with data generated by the TPM. For signing keys, this prohibits the TPM to sign data with it which seems to be generated by the TPM, but are actually not. This prevents an attacker from asking the TPM to sign an artificially constructed quote. And restricted encryption keys can only be used to encrypt TPM generated data like keys.

The Platform Configuration Registers (PCRs) are the fundament for the remote system attestation. They are one-way registers, which values can never be written to an exact value, but only be extended. This operation is known as 'hash extend' [**Arthur2015**]. Its design prohibits the removal of extensions, which would cause the TPM to forget a measurement, and the arbitrary writing of values, which would overwrite any previously conducted measurements. A PCR value holds a hash representing the platform state. Thereby, a remote verifier can request a so-called 'quote' from the TPM

[2] `https://trustedcomputinggroup.org/`

on the host in question. A quote contains the hash of all requested PCR values and is digitally signed. Typically, a TPM contains 24 PCR registers[3], as defined as the minimum by [**tcgPcClient**], with the lower PCR values representing the system boot process and the higher ones representing the events after the kernel is booted [**Arthur2015**]. The fixed length of the PCR values is important for the memory-constrained nature of TPMs [**Arthur2015**].

The PCR value at index i can only be modified, i.e., extended, by adding together the currently contained hash value and the new hash, as depicted in Equation 2.1 [**tpm**]. For the sake of correctness, it should be noted that not every PCR is initialized with zero, as implied in the equation. For example, the TPM PC Client Platform specification [**tcgPcClient**] defines that PCRs 1–15 are initialized with all bits set to 0, while PCRs 17–22 are initialized with all bits set to 1.

$$PCR(i)_{t=0} \coloneqq 0, \quad PCR(i)_{t+1} \coloneqq hash(PCR(i)_t \parallel new\ value) \tag{2.1}$$

TPM 1.2 is limited to SHA-1 hashes which are considered broken [**cryptoeprint:2005/010**, **Wang2005**, **Stevens2017**]. Although the SHA-1 uses in TPM 1.2 were analyzed to be not affected [**sha1tpm12**], cryptographic algorithms only become weaker over time [**Arthur2015**]. In reaction, TPM 2.0 offers crypto-agility and allows newer algorithms such as SHA-256. In general, TPM 2.0 is more flexible, and is always turned on, while a TPM 1.2 needed to be turned on manually. Also, TPM 2.0 is more consistent across different implementations because of broader specifications. TPM 2.0 is the focused version nowadays, e.g., Microsoft recommends TPM 2.0 over TPM 1.2 because of security advantages [**micrec**], and also requires TPM 2.0 for Windows 11 with SHA-256 PCR registers [**win11req**].

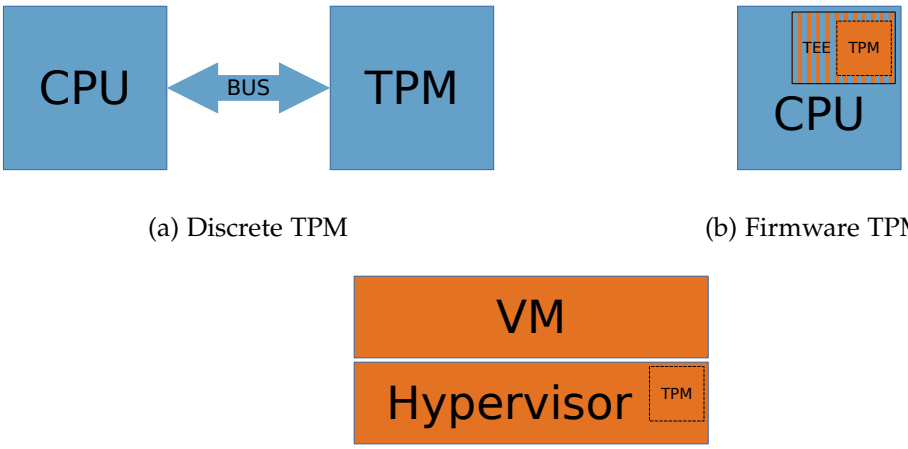
There are three types of TPMs, as illustrated in Figure 2.4. They all offer the same functionality, but with different security guarantees and performance characteristics.

2.3.1 Discrete TPM

This is the classical form of a TPM. It is a dedicated piece of hardware, connected to the CPU via a bus. It is designed and manufactured to be highly temper-resistant against hardware attacks. The TPM specifications [**tpm**, **tcgPcClient**] do not demand a specific bus system, however, they define the interfaces between the TPM and the following bus systems: LPC, I^2C, and SPI.

The well-known 'TPM Reset Attack' was independently described in [**kauerBernhard**, **sparks2007**]. It requires minimal hardware, precisely only a wire connecting the reset

[3]Note that we are aware that 'PCR register' is a Redundant Acronym Syndrome, but we have chosen to leave it as such for clarity, as 'PC register' can be associated with other meanings.



(a) Discrete TPM

(b) Firmware TPM

(c) Virtual TPM

Figure 2.4: Schematic illustration of the different TPM types in their pure form. Blue: Hardware, Orange: Software.

line of the LPC bus [**lpc**] to ground. This results in a reset signal for the TPM, yielding predictable values for the PCR registers. This allows an attacker to replay the measurement log of a benign boot process to achieve valid PCR values, even though a modified chain has been booted. Since TPM 1.2, TCG provides a mitigation specification for this reset attack [**tcgResetFix**], requiring the BIOS to overwrite sensitive data after each unexpected reset, preventing an attacker to gain a valid measurement log. However, this mitigation is still vulnerable to cold boot attacks [**Halderman2009**, **Winter2013**].

Winter and Dietrich [**Winter2013**] demonstrate a bus modification attack at TPMs integrated with the LPC bus or the I^2C bus. Their approach, labeled 'Active LPC frame hijacking', allows them to "lift" commands to a higher locality than the one they were originally sent with. This allows them to evolve the 'TPM Reset attack' from being only usable for S-RTM, to also D-RTM systems. They also introduce a new approach of circumventing the TPM's measurement feature. Instead of resetting the TPM as previously described [**kauerBernhard**, **sparks2007**], they reset the main device, i.e., the users' device like a desktop PC while preventing the TPM from receiving the reset signal. This keeps the state of the TPM, e.g., the valid PCR values of the previous boot procedure, and the attacker can hijack the boot procedure triggered by the platform's reset and boot a malicious operating system or firmware, while the TPM still stores the old and valid PCRs. While its conceptually easier since the attacker does not need to

know the measurement log since the valid PCR values are already in-place, it requires active manipulation of bus transmissions to shield the TPM from the reset signal.

Seunghun Han et al. [**aBadDream**] report two attacks on discrete TPMs to reset the PCR registers. The first targets a gray area in the power management section of the TPM 2.0 specification. The TPM shall store its state into its non-volatile random access memory (NVRAM) before shutting down when the host platform goes to sleep, and restore it when it wakes up. However, the specification is missing a concrete description how to handle a lack of a stored state when waking up. Therefore, some implementations simply reset the state. Their second attack targets a DRTM, namely an implementation flaw in tboot [**tboot**], the most widely used measured boot environment used with Intel's Trusted Execution Technology. However, in their work, they found that some mutable function pointers are not measured, which allows attacks.

A time-based side-channel attack [**Moghimi2019**] during signature generation based on elliptic curves allows an attacker to recover 256-bit private keys for ECDSA and ECSchnorr signatures.

A passive sniffing attack is shown in [**Kursawe2005AnalyzingTP**]. It is applicable to TPM 1.1 connected to an LPC bus. They observed that the data of some operations like unsealing are transmitted via the bus in plain text. Since TPM 1.2, however, the modules no longer send sensitive data unencrypted [**Winter2013**].

That invasive hardware attacks against dTPMs are possible was already shown by Tarnovsky in 2010 [**tarnovsky**]. However, this requires a lot of time, knowledge and resources, i.e., hardware and money.

2.3.2 Firmware TPM

As seen in the previous section about discrete TPMs, the bus between the CPU and a TPM is a typical attack vector. An fTPM [**Raj2015**, **197213**] circumvents this by being directly executed by the CPU within a TEE, revealing no easily accessible bus. The trend is moving towards fTPMs, which can also been seen by the increasing efforts to bring an fTPM to the RISC-V processor family [**Boubakri2021**]. Also, since they require only a TEE which is mostly already available at currently used processors, they are cheaper for manufacturers.

As of now, a fTPM is strictly bound to the processor manufacturer, such that you can trust the underlying firmware as well which is provisioned by the manufacturer, e.g., Intel, too. For example, common implementations are the Intel® Platform Trust Technology (Intel PTT) [**intelProcessorSecurity**], and AMD's Secure Processor (AMD-SP), which in fact is an ARM-based coprocessor on the die with Arm's TrustZone [**Khalid2020**].

Running on the main processor, e.g., a fully-fledged Arm Cortex core, entails an

advantage and a disadvantage. The disadvantage is that running on the same processor as the rest of the system means less isolation, while a dedicated TPM (dTPM) brings its own processor that is completely isolated from the main processor. The advantage, however, is that a main processor is generally much faster because dTPM processors are weak [**Goh2013**, **Raj2015**]. Raj et al. [**Raj2015**] and Cheng et al. [**Cheng2020**] independently concluded that the firmware-based modules are generally much faster after comparing the performance of fTPMs and dTPMs.

In fact, there are more disadvantages. First, fTPMs cannot provide true RNG, since hardware is required for that [**Stipcevic2014**]. Second, they are started later in the hosts' boot chain than a dTPM that is accessible from the beginning. This has the consequence that the hashes of the components booted before the fTPM have to be cached and later be forwarded to the fTPM as soon as it is available. Arm's Trusted Firmware-A[4], which is the Arm's reference implementation of software in the SW, protects this cached event log by keeping it in secure memory [**tf-a-measured-boot**], i.e., memory which is only accessible in the SW. Last, fTPMs depend on more components for its security than single-component dTPMs, e.g., the TEE, and the boot chain.

Of course, there are also attacks against fTPMs. The previously mentioned side-channel attack [**Moghimi2019**] against dTPMs, can also be applied to fTPMs.

Jacob et al. [**Jacob2023**] target proprietary AMD fTPMs by attacking their TEE, namely the AMD Secure Processor (AMD-SP). Thereby, they can expose the full internal state of the fTPM bypassing any authentication mechanisms. To do so, they leak the secret key from the BIOS flash chip which is used to derive the encryption and signature keys for the fTPMs non-volatile data. They achieve this by using a voltage fault injection that bypasses the authenticity check in the hosts' boot process and allows them to boot their own firmware component that leaks the required information.

Cfir Cohen from Google's cloud security team has uncovered an attack on fTPMs that run within AMD-SP [**cohen**]. They store a maliciously crafted payload – a certificate – on the fTPM and trigger a function with a stack-based overflow error that accesses this payload, giving them full control over the program counter.

2.3.3 Virtual TPM

A vTPM is a software-based TPM provided by a hypervisor for one of its managed virtual machines [**268868**]. The vTPMs can be realized fully in software [**268868**], or backed by dTPMs [**Liu2010**]. The hypervisor can provide a (theoretically) unlimited number of vTPMs. For the virtual machines it seems that they have access exclusive access to their own private TPM, even though all vTPMs are managed by the same hypervisor. A characteristic feature of virtual resources are their migration capabilities,

[4] `https://www.trustedfirmware.org/`

i.e., they can be suspended and later continued on another machine. vTPMs support this as well. Note the different security properties between vTPMs and dTPMs.

Because of the increasing popularity of cloud computing, the research of vTPMs focuses less on specific attacks, and more on reducing the trusted computing base, i.e., privacy-focused. The initially proposed design [**268868**] has a large trusted base, e.g., the operating system and the hypervisor need to be trusted.

Wang et al. [**Wang2019**] bring the vTPM into the TEE, namely Intel SGX, essentially creating an fTPM and vTPM hybrid. They launch each vTPM in a private hardware-protected enclave. This reduces the trusted computing base to the individual enclaves and SGX itself, enabling the host operating system and hypervisor to be untrusted.

Pecholt and Wessel [**Pecholt2022**] describe a design named CoCoTPM where the hypervisor and the hosts' operating system do not need to be trusted as well. This is realized by establishing an integrity-protected secure channel with end-to-end encryption between the driver in the VM and the software TPM on the host.

Stateless ephemeral vTPMs [**Narayanan2023**] eliminate the need of manually establishing a secure channel by leveraging the confidential VM memory encryption provided by AMD's SEV-SNP, a variant of AMD secure encrypted virtualization (SEV) technology. Ephemeral vTPMs support the remote attestation of virtual machines. However, they intentionally do not support persistent storage to preclude exfiltration attacks on stored TPM state, which has the disadvantage that persistent keys or nonvolatile indexes cannot be stored.

2.4 Secure Boot and Measured Boot

When the system is started, the root component, e.g., from ROM, is executed. This subsequently launches the next component, and so forth. This boot structure is called the boot chain. Typically, the first component turns on the memory, the second stage initializes the platform, and finally, the last stage boots the operating system [**Yao2020**].

Secure Boot [**Hendricks2004**, **UEFI**, **Frazelle2020**] is verifying components of the boot chain directly at boot-time. For that, the boot component is equipped with a public key. With that, they verify the digital signature of the respective subsequent component, before handing over the execution. This ensures the authenticity of the boot components. Alternatively, merely the hashes of the components can be measured and compared with known values, which only ensures integrity and not authenticity. The first boot component, usually stored in ROM, needs to be trusted without verification, i.e., it acts as the root of trust. However, Secure Boot does not prevent downgrade attacks, since only the authenticity, but not the concrete versions of boot components are verified [**272306**]. Hence, further defenses like Measured Boot have been designed.

Measured Boot [**tcgMeasuredBoot**] is a concept that is implemented in interplay with a TPM. It allows remote attestation to a later time. Just as with Secure Boot, each boot component hashes the subsequent component. However, instead of directly locally verifying the measured value, the hash value is passed to the TPM to extend a PCR value. As described in Section 2.3, these values can be used by a remote attestor to verify the state of the software on the system. The goal is to detect manipulated system configurations.

Secure Boot and Measured Boot are often used in conjunction.

2.5 Device Identifier Composition Engine

DICE was originally proposed by Microsoft as part of their Robust Internet-of-Things (RIoT) architecture [**England2016**]. In 2017, the DICE specification was published by TCG [**tcg-microsoft-tpm**], of which Microsoft is a member. Its purposes are to detect firmware tampering and enable device identification for a remote party, while its main attribute is its minimal hardware requirements.

DICE operates on a boot process layered into components [**dice-layering-arch**]. Later components are typically more feature rich and complex than earlier ones. Each component is measured prior becoming active by the preceding component. Great care must be taken to ensure that the identity of the measured and thereafter executed component is consistent [**Hristozov2022**, **Carpent2018**]. The union of all security-relevant components of a device form its Trusted Computing Base (TCB). Their identities are called TCB Component Identifier (TCI). They are usually the hashes of the according firmware binary, but could also consist of a hardware product identifier.

The first component is the DICE itself, which consists of software and hardware. The DICE specification [**dice-hardware-reqs**] states the three hardware requirements, the DICE layer

1. has to store a read-only Unique Device Secret (UDS),
2. has exclusive access to the UDS,
3. and is immutable.

These requirements can be justified intuitively. (1) The UDS must be read-only and unique to the device to enable a base for long term identification. (2) The DICE layer reads and uses the UDS, and then needs to erase the UDS from memory while preventing other components from retrieving this secret during the power-on time. Otherwise, other entities can forge measurement or identification values. This lock mechanism can, for example, be realized with eFuses [**dice-hardware-reqs**]. (3) Moreover, the

misbehavior of the DICE layer cannot be detected since it is not preceded by anything that could measure it, i.e., it is the root of trust. Therefore, it must be immutable so that it remains in the trusted state in which the manufacturer provided it.

While the UDS is exclusive to the DICE layer, each other component retrieves a Compound Device Identifier (CDI) from its previous component. The CDI of each layer depends on two variables combined in a one-way function (OWF). (i) The own TCI, binding the CDI to the current layers' identity, i.e., the hash of itself, and (ii) the CDI of the previous layer, making each CDI depending on the identities of all previous component identities. Therefore, if any component is modified, this reflects in the permutation of the CDIs of all subsequent components, as implied by Equation 2.2.

Just as the DICE layer must ensure to have exclusive access to the UDS, each later layer must ensure to have exclusive access to its CDI. The layers can derive further secrets using their CDI as a seed, such as the Device ID key pair (Equation 2.3) or an alias key pair (Equation 2.4).

$$CDI_n = \underbrace{UDS \circ TCI_0}_{CDI_0} \circ \ldots \circ TCI_n \tag{2.2}$$

$$DeviceID\_KeyPair = KDF(CDI_0) \tag{2.3}$$

$$Alias\_KeyPair_n = KDF(CDI_n) \tag{2.4}$$

where $a \circ b = OWF(a, b)$, $\circ$ being a left-associated operator, and KDF being a key derivation function.

The end result of a boot process using DICE is a certificate chain, which can be also seen in Figure 2.5 and Equation 2.5. Here, each certificate represents a layer by embedding the identity of the layer, i.e., its TCI. The certificate chain is built up step by step during the boot process, with the first two certificates (manufacturer certificate and DeviceID certificate) being static and stored on the device, and the remaining certificates (Alias certificates) being generated during boot time.

$$Manufacturer\ Cert \rightarrow DeviceID\ Cert \rightarrow Alias\ Cert\ [\rightarrow Alias\ Cert]^* \tag{2.5}$$

The chains' root is the certificate of the DICE manufacturer. It is either provided by the device or can be retrieved from the manufacturer itself, e.g., via its website.

The next certificate is the DeviceID certificate. It is generated during device provisioning by the manufacturer, and signed by the manufacturers' private key. This links the DICE implementation to a manufacturer, which is important to retrieve the guarantees the manufacturer conducts for its DICE implementation to protect its UDS value, which is the hardware root of trust. This certificate also represents the long term identity of the device. The UDS alone cannot be used for this because it is kept strictly

secret, whereas the DeviceID certificate is public. Since thereby the identification relies on asymmetric cryptography, the manufacturer does not need to maintain a database of UDS values, but only needs to keep and protect its private key.

While the DeviceIDs' public key is publicly stored on the device as part of the DeviceID certificate, the corresponding private key still must be generated during the boot process. Only if the matching private key is generated, which is only the case if the identity of layer 0 did not change, the certificate chain can be continued. This continues the chain of trust, because if layer 0 changes, the DeviceIDs' private key also changes. This invalidates the signature of the next certificate for the public key in the DeviceID certificate.

The remaining alias certificates in the chain each represent the identity of a layer. Recall that a measurement is always conducted from the previous component. Hence, the previous component also has to create the AliasCert, since the just measured component is not trusted to do that. Each DICE layer must only assume the lower layers to be trustworthy. Measured TCIs are persisted in alias certificates signed with the private key of the measuring layer.

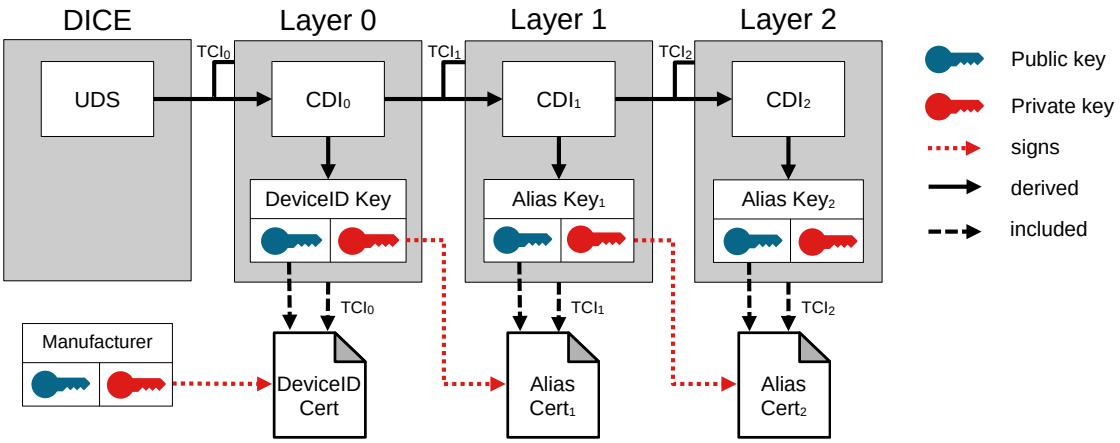


Figure 2.5: The generation of the CDIs and the certificates for each layer in a DICE architecture. Note that the diagram could be continued for an arbitrary number of layers.

The generation of the CDI values and certificates are shown in Figure 2.5. There, the CDIs and certificates are drawn where they are eventually used or what they represent, not where they are created. E.g., CDI_n and Alias $Cert_n$ are both created by layer $n-1$.

To the best of our knowledge, DICE is so far considered a secure concept apart from physical attacks, only implementation problems can bear security problems [**Jaeger2020**, **Hristozov2022**].

3 Related Work

In the following, we describe defense mechanisms for fTPMs that can be seen as complementary to our approach. They all have in common that they offer no way for a third party to ensure that the hardened fTPM is actually running on the device under test, which is exactly what our work aims to cover.

One approach is to verify the code of fTPMs [**Mukhamedov2013**]. Here, the TPM 1.2 code is written in a functional programming language that enables automatic verification.

There exist efforts to improve the security of TPM by introducing the concept of hybrid TPMs [**Kim2019**, **Gross2021**]. Kim and Kim [**Kim2019**] extend a hardware TPM with software support, which they name hTPM. This increases the defense of the TPM, e.g., circumventing side-channel attacks, and also enables more secure TPM functions, e.g., enabling true random number generation. Their hTPM implementation also shows significantly better performance due to the use of modern CPU features. Vice versa, Gross et al. [**Gross2021**] propose the reverse approach of backing an fTPM with hardware. While their implementation has similar properties to hTPM, it inherits some downsides of fTPMs. For example, their fTPM is still started later in the boot chain than a dTPM, which is not the case for hTPM. However, it is easier to update than hTPM since the lack of a dTPM, and the overall design is simpler.

4 Methodology

4.1 Terminology

Before we dive into technical explanations, we want to clear some potential terminology confusion.

In the original DICE release from Microsoft **[England2016]**, the identifier of a component is called the Firmware ID (FWID). The TCG consortium later renamed it TCB Component Identifier (TCI). We believe this is to emphasize that the TCI does not necessarily have to be the hash of a firmware binary, but could also be, for example, the embedded ID of a hardware component. However, TCG has not fully implemented this terminology renaming. Their DICE Attestation Architecture [**TCGAttestation2021**] defines an X.509 extension that contains the TCIs. They continue to be referred to as FWIDs in the formal definition of this extension, while everywhere else they are referred to as TCIs. In personal correspondence with TCG, we have learned that this is due to backwards compatibility. The old term FWID is retained whenever it is used in something that is alive in the long term, like formal definitions, and the new term TCI in assets that can be updated more quickly, such as the specification text. Therefore, we will use the term TCI in this theoretical chapter, and in the implementation chapter (Chapter 5) we will use the term FWID, just as it is common practice at the TCG.

4.2 Architectural overview

The architecture of our proposed and later implemented system is illustrated in Figure 4.1. As you might notice, it is similar to our overview picture of DICE (Figure 2.5). This is to be expected, since our system uses DICE. We leverage the DICE as our static root of trust for measurements (S-RTM). Static in this context means that it uses the trusted state that a device has at the always same point in time, here after switching on, for further measurements. This is in contrast to a dynamic RTM (D-RTM), which is able to do this at any time, e.g., Intel SGX.

The boot process continues from here in the usual DICE manner until the firmware TPM is reached. The component that measures the fTPM is usually the trusted operating system running in EL1 in the secure world, as seen in Figure 2.2. Like any other DICE

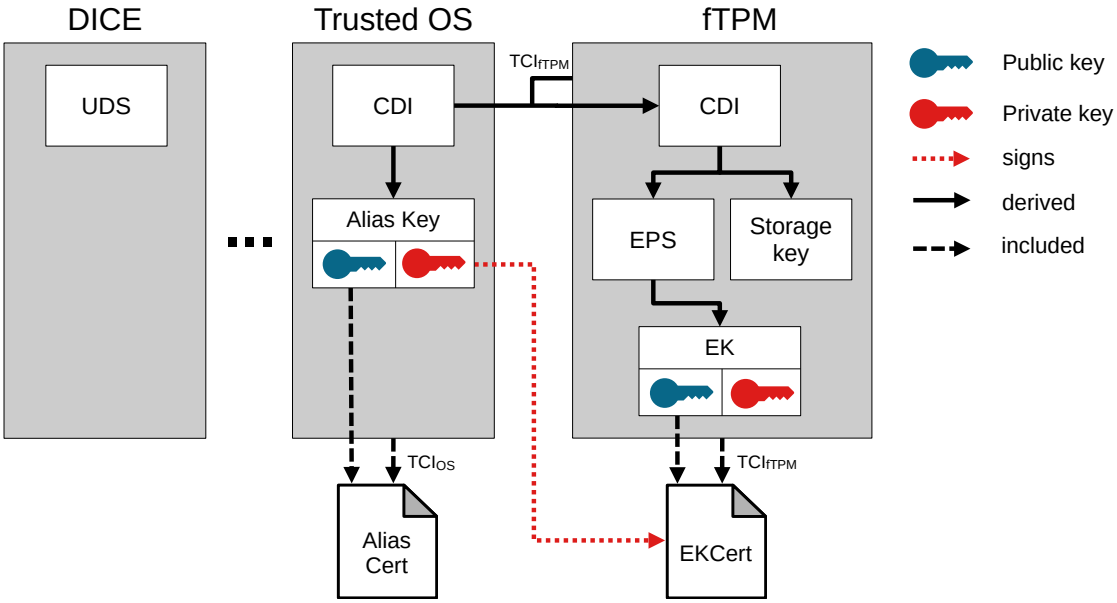


Figure 4.1: The architecture of our system.

component, the fTPM receives its secret Compound Device Identifier (CDI) from its predecessor layer. Recall that the CDI is tied to the identity of the fTPM including the entire underlying firmware stack and the UDS. Two values are derived from the CDI.

First, the storage key is generated. This is a symmetric encryption key that is used to encrypt the fTPM storage space in RAM before it is written to a persistent storage space such as a hard disk drive (HDD). At no time does the HDD see plain text data (Figure 4.2). Since the storage key is derived from the CDI, the old storage data is inaccessible if the identity of the TPM changes, e.g., due to a TPM modification or an update of a previous firmware component, which is equivalent to a full manufacturer reset. This enables the property that an fTPM storage must never be accessible to another TPM.

Then, the primary endorsement seed (EPS) is generated. It is the seed that is used to generate the primary endorsement key (EK). A primary key in the sense of the TPM means that it has no parent key, but a parent seed, here the EPS. The indirection of generating the CDI via the EPS instead of generating the EK directly from the CDI is introduced because the code of fTPMs can be hardcoded to use the EPS during EK generation. And we want our system to require as few modifications to TPM code as possible.

The EK of a dedicated TPM represents the long-term identity of the device as long as the TPM is not soldered away. Our EK does not do this because a firmware TPM

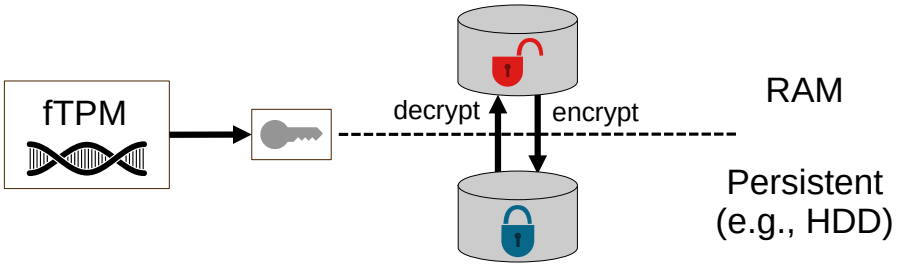


Figure 4.2: The fTPMs' storage is protected by a key derived from its identity.

is software-based and changes every time the fTPM or the underlying firmware is modified, without the host device changing. Instead, we use the DICE for this, which is hardware-based. Its DeviceID key, as the name suggests, represents the device identity. Note that the DeviceID contains the identity of layer 0 of the boot chain, i.e., the first mutable code. This can also be seen in Equation 2.3. For this reason, the DICE specification suggests keeping the first mutable code as small as possible so that it remains constant throughout the life of the device [**dice-layering-arch**].

By default, the EK is a restricted encryption key. It is not used for signing because the resulting signatures may reveal the TPMs' identity. Therefore, the usual procedure is to create a short-lived attestation key (AK) on the TPM. The verifier communicates the public part of the AK to a third party privacy CA who ensures that the holder of the AK is the same as the holder of the EK. The privacy CA then issues an attestation certificate confirming that the AK originates from a genuine TPM, without revealing the identity of the TPM. Finally, the AK can be used by the TPM to sign attestation evidences in a private fashion. We waive this separation and use the EK directly for attestations to avoid the need for a third party CA.

The DICE and the TPM infrastructure intersect at the EK certificate. From the DICE's point of view it is an alias certificate, from the TPM's point of view it is the EK certificate.

4.3 The identity of a firmware TPM

DICE offers two identities for each component. The TCI and the CDI. As shown in Equation 2.2, the CDI of a component changes when (i) the identity of the hardware changes, i.e., the UDS, (ii) the identity of a preceding component changes, or (iii) the component itself changes. In contrast, the TCI is the identity of a single component, considered in isolation, usually the hash of its binary. It only changes when the component itself is changed, regardless of the hardware or preceding components. So,

while a CDI should be statistically unique since it is derived from a UDS with this property, a given TCI can be found on many devices if they contain exactly the same software component. Note that the TCI is part of the CDI, as shown in Figure 4.3.

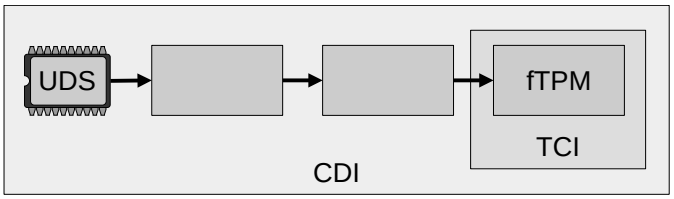


Figure 4.3: The visualization of the difference between the CDI and the TCI. The identity of the hardware is provided in the form of the UDS.

Hence, we decided to derive the identity of an fTPM, i.e., its EK, ultimately from the CDI. This binds the identity of the fTPM to any security-relevant component preceding it.

The TCI (and thus the CDI too) must also represent security-relevant configurations. Compilation flag configurations are already inherently part of the TCI, as they are embedded in the final binary and are therefore automatically measured as part of the measurement of the binary. Of more interest are the configurations that are not part of the binary. They are usually provided in well-known formats such as `json` or `xml`. However, Microsoft's fTPM reference implementation does not contain such configurations, which simplifies the TCI generation of our fTPM by limiting it to the measurement of the fTPM binary data. The TPM's storage cannot be part of its identity as it changes during runtime after each data write, e.g., storing an arbitrary key. This is a problem because DICE only runs during the boot time in which the identity of the fTPM is measured. The identity of the fTPM must not change afterwards, otherwise the identity reported by DICE and the actual identity of the fTPM would be inconsistent. We also do not want to restrict the permissible values of the working data of an fTPM.

4.4 Attestation process

$$trusted(C_i) := \bigwedge_{k=0}^{i} trusted(C_k) \tag{4.1}$$

We use an explicit attestation procedure. This makes it sufficient for the verifier to know the trusted TCI, whereas implicit attestation would require a database of known public keys corresponding to a trusted TCI. And since each public key for a device is unique, as it is ultimately derived from the device's unique UDS, the verifier would

need to know the mapping between public keys and the corresponding TCI for each device, which we consider unrealistic. Also, this is a hindrance as the verifier should be able to trust an unknown device by trusting the DICE manufacturer.

4.5 Combining TPM and DICE infrastructure

The result of our DICE boot process is a certificate chain, starting with the manufacturer certificate and DeviceID certificate, and ending with the EK certificate. In between can be an arbitrary number of Alias certificates for "ordinary" firmware components (see Equation 4.2).

$$Manufacturer\ Cert \rightarrow DeviceID\ Cert\ [\rightarrow Alias\ Cert]^* \rightarrow EKCert \tag{4.2}$$

As stated earlier, the EK certificate is where the DICE and the TPM infrastructure meet. So, this EK certificate needs to fulfill the requirements for an Alias certificate from the DICE specification [**DICE_certs**], and the EK certificate requirements from the TPM specification [**tcg-ek**]. Therefore, we need to ensure that these two specifications declare no conflicting requirements. The DICE Certificate Profiles [**DICE_certs**] defines the requirements for various certificate types. Our certificate is an attestation certificate in this specification.

We only consider restrictions for the X.509 fields that are absolute requirements, i.e. declared as "MUST" according to RFC 2119 [**Bradner1997**], and common fields that can occur in both specifications. In general, the EK certificate specification "does not preclude the use of other certificate extensions". The alias certificate specification leaves this undefined, i.e., it makes no statement whether this is permitted or prohibited. However, it is irrelevant for us, since the EK certificate specification does not define any own X.509 fields. Some requirements depend on whether the measuring firmware has access to a secure clock. We assume the firmware to not having access to a secure clock, keeping the requirements low. Another property which can affect the requirement of our certificate is its further usage. Our EK certificate is a leaf of the certificate chain. Therefore, we do not consider requirements for a certificate representing a certificate authority (CA) signing further certificates.

We present the result of our comparison in Table 4.1. In summary, there are mostly no conflicts since both certificates expect the same value, both requirements can be satisfied with one value, or only one of the certificates dictates a restriction for a specific field.

The only conflict is in the subject name. An Alias certificate must either identify the TCB class (general) or instance (specific), an EK certificate allows only a value uniquely identifying the TPM (specific) or empty otherwise. A general term like "fTPM"

Table 4.1: Comparing the requirements for an Alias and endorsement key certificate. If not otherwise stated, the source of these requirements are the previously noted specifications, i.e., [**DICE_certs**, **tcg-ek**].

Field	Alias Cert	EKCert	Conflict
Version	3	3	No
Subject name	identify TCB class or instance	Uniquely identify TPM or empty	Yes
Issuer name	embedded CA issuing the certificate	entity that vouches that TPM is genuine	No
Validity not before	Known time in recent past e.g., build time	–	No
Validity not after	No expiration	No expiration for non-user TPMs [**tcgPcClient**]	No
Basic Constraints	–	Not a CA	No
Authority Key Identifier	–	Must be present	No
Key Usage	keyCertSign unset	digitalSignature set	No
Policy OIDs	Local Attestation Policy OID	–	No

is prohibited by the EK certificate specification. The only common denominator is a unique identifier. However, that is already part of the TCI embedded in our EK certificate. We chose to favor the EK certificate specification here, and leave the subject name empty. This ensures that the EK certificate is also as expected for systems that do not know our solution and do not know the TCI part of the certificate. It also seems to be common practice, since all endorsement certificates we observed have an empty subject name.

The Subject Alternative Name extension is required to contain the TPM Manufacturer, model, and version by the EK certificate specification [**tcg-ek**]. It is assumed that the EK certificate is generated by the manufacturer who has this knowledge about the TPM. In our system, however, the DICE layer measuring the firmware TPM and ultimately generating the EK certificate does not know these values, as they are not constant and can change any time when the firmware TPM is exchanged. One possible solution is to keep these values in the metadata of the fTPM's binary, which the preceding layer can read and embed in the certificate. But this increases the complexity and the maintenance burden for the firmware TPM, which is usually not required since all this information (manufacturer, model, version) can be deduced from the TCI part of the certificate. Therefore, if a verifier trusts a TCI, he should also know which manufacturer, exact code and TPM specification it conforms to. Furthermore, the TCI is more accurate and reliable because it is an exact independent measurement of the firmware TPM rather than relying on information propagated by the manufacturer. For example, an underlying firmware component could alter the firmware TPM to pretend it is compliant with a newer specification (with potential security updates) than it actually is. This cannot happen with the TCI that is part of the certificate chain, as this malicious firmware component would be detected.

4.6 Updating the fTPM

We consider it as critical that the fTPM is updatable. This is due to the history of fTPMs showing vulnerabilities which have been patched consequently.

Our fTPM can be only updated with the system shut down. This is due to the required out-of-band signing procedure of trusted applications before being deployed. This also with while system is shut down. This ensures that the TCI part of the EKcert generated at boot-time does not become obsolete, in other words, keeps representing the state of the currently running fTPM.

To protect against downgrade attacks: NV data is encrypted/integrity with AESP Encryption required to ensure confidentiality AESP used to also ensure integrity This is required since also the cipher text of the NV could be modified, which might change

security critical information. Encryption-only would not prohibit that. As soon as an integrity violation is detected, the fTPM is fully reset, effectively invalidating all previously stored data. Note that an attacker could thereby easily trigger a data loss. This has to avoided by integrating the good-practices with working with a TPM, which includes having secrets stored also elsewhere. This introduces storage and memory overhead. Processing overhead only slightly, since the data is already decrypted during start-time, which happens only once at boot time, and then later data is encrypted only while it is stored, which happens only ... Hence, there is no performance penalty during common uses of a TPM, e.g., key creation.

This might seem redundant because of the storage protection of for example TrustZone, but this does not protect against downgrade attacks. With our approach, the access to the data is bound to the exact identity of the fTPM including all underlying firmware.

So, our protection additionally protects data-at-rest, while the data-at-use is protected by the TEE's secure memory, i.e., the memory isolation from the normal world.

4.7 Privacy

In general, endorsement keys represent device identities and are therefore privacy-sensitive. Our EK takes on the role of an attestation key, which has the task of signing attestation evidences. Although our EK does not represent the device identity, it does represent the identity of the shorter lived fTPMs. This makes our EK also privacy-sensitive, since its generated signatures can be cross-referenced and then traced back to the according EK. Our approach has the advantage that we do not need a trusted third party, but trust the manufacturer of our trust anchor directly (DICE). In Section 4.7, an extension of our system is presented with privacy in mind.

5 Implementation

As explained in Section 4.1, from now on the term Firmware ID (FWID) will be used instead of the previously used term TCB Component Identifier (TCI).

5.1 Adaption of the Endorsement Key

The EK is a primary key, so it is derived from the Endorsement Primary Seed (EPS). There is also a default template defined by TCG [**tcg-ek**], dictating the endorsement key to be a restricted encryption key, since it is privacy-sensitive. The default template is required to be able to reproduce the EK contained in the EK certificate so that the TPM can prove that this EK certificate corresponds to it by being able to generate the corresponding private key. The default template is not stored on the TPM, but it is part of the command triggering the key generation (`TPM2_CreatePrimary`). Therefore, the TPM itself does not need to know what the default template is, since it is always provided by TPM-capable software.

However, we want to use the EK as a signing key. The TPM specification provides a mechanism for this, where we need to store our custom EK template in a specific NV index within the TPM [**tcg-ek**]. We use the values of the default EK template and only deviate from it when necessary in three aspects. We declare the EK as a (i) restricted signing key. This also requires to specify a (ii) signature scheme, and (iii) no inner symmetric key as required for signing keys since they are not allowed to have any children keys.

This template will be generated within the TPM after each manufacturer reset, so it will be preserved even after an identity change and a reset of the firmware TPM. However, the EK itself will change as the CDI changes, then the EPS and finally the EK. The attributes of the NV index are declared such that the template cannot be deleted [**tcgPcClient**]. This is possible by allowing to delete the NV index only when an unfulfillable policy is met.

The creation of the template is done in code of the fTPM. This is a convenient feature, as the template is thus also part of the TCI of the fTPM. And since a verifier is free to decide whether to trust a particular TCI, he will most likely only trust a TCI if it contains that the TPM generates a restricted EK. However, this is also possible with a generic TPM with a little more effort with the `TPM2_Certify` command.

5.2 Prover

5.2.1 Normal World

5.2.2 Secure World

Measuring the fTPM

5.3 Attester

5.4 Times

Usually, the notBefore time of Alias certs should be build time, and notAfter should be infinite. Network-enable our FVP, but heavy setup for simple demonstration of our system. Could malfunction sometimes because of time dependencies. To keep our system easily understandable and simple to access a demonstration, we use fixed times.

5.5 Technical obstacles

We would have liked to use RSASSA-PSS which is formally proven to be secure over RSASSA-PKCS1-v1_5. RFC 8017 even requires RSASSA-PSS for new applications [**Moriarty2016**]. However, it is not fully supported by the tpm2-tools, yet[1].

[1] `https://github.com/tpm2-software/tpm2-tools/issues/3283`

6 Discussion

6.1 Assessment of the fulfillment of requirements

6.1.1 Security requirements

6.1.2 Attestation process requirements

TCG defines as part of their Trusted Attestation Protocol [**tap**] the requirements for an attestation process to provide assurance to a verifier that it is (i) accurate, (ii) interpretable, and (iii) attributable.

(i) Accurate attestation data represents the actual state of the device. This includes freshness, i.e., the data is not replayed and does not represent an old, outdated state of the device.

(ii) Intuitively, the data must be interpretable by the verifier. In other words, the verifier must be able to derive a decision about the trustworthiness of the prover based on the attestation data.

(iii) It must be possible to assign the attestation data to a specific device, i.e., it must be verifiable that the attestation data originates from the prover.

6.2 Higher level protocols' compatibility

6.3 Implications of missing privacy

6.4 Hardware knowledge dependency

6.5 Hardware requirements DICE + fTPM vs TPM

6.6 Requires reproducible builds

6.7 Personal opinion about developed system

7 Future Work and Conclusion

7.1 Future Work

7.2 Conclusion

Abbreviations

TPM Trusted Platform Module

fTPM firmware TPM

dTPM dedicated TPM

DICE Device Identifier Composition Engine

TEE Trusted execution environment

REE Rich execution environment

PCR Platform Configuration Register

TCG Trusted Computing Group

NW Normal world

SW Secure world

List of Figures

List of Tables