

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

I would like to take this opportunity to thank all those who have supported me in the preparation of this thesis. My thanks go to my supervisor Prof. Dr. Claudia Eckert, who gave me the opportunity to write this thesis. I would also like to express my special thanks to my advisors Albert Stark and Johannes Wiesböck, who patiently stood by my side and answered all my questions. Without their help and support, this thesis would not have been possible, and I am sure they have helped me greatly to start my journey in the field of cyber security. Last but not least, I would like to thank all my friends who have been patient and understanding during the writing of this thesis and who have sacrificed their time to proofread my work.

Abstract

Zero Trust is a cybersecurity paradigm in which a network, e.g., an enterprise network, is considered compromised. Therefore, each device of every service request must be verified before the request is served. This is made possible by remote attestation, which is enabled by TPMs, for example. For that, they authenticate themselves by propagating their so-called endorsement certificate naming their manufacturer, who guarantees they conform to the TPM specification to ensure their security properties. While this approach is sufficient for hardware TPMs as they are standalone chips, for firmware TPMs (fTPM) it is assumed that the manufacturer of the fTPM is the same as that of all firmware components that were booted before the fTPM. This is due to the fact that any previous firmware component can compromise the fTPM during loading. The verifier in the remote attestation procedure lacks the ability of verifying the entire boot chain up to the fTPM. We propose a remote attestation system that provides the verifier with this possibility. To compensate for the lack of a hardware root of trust of an fTPM compared to a hardware TPM, we introduce DICE as the hardware root of trust. The verifier only needs to trust the manufacturer of DICE, while every firmware component beyond is explicitly attested by passing their identities to the verifier. The three benefits of our solution are that (i) the manufacturer of the fTPM and its preceding firmware components can be independent of each other, (ii) detection of modification of fTPM by remote verifier, and (iii) protection of the fTPM's data-at-rest. DICE measures the first firmware component, which is then repeated up to the fTPM. These measurements are forwarded to the remote verifier, which can then detect potentially malicious changes to each measured component. The fTPM's data-at-rest is protected by binding it to the identity of the fTPM. This means that the data of an fTPM is only accessible to the fTPM that created it as long as its identity does not change, which makes downgrade attacks and changes to the fTPM less attractive to attackers.

Contents

Acknowledgments	iii
Abstract	v
1 Introduction	1
1.1 Motivation	1
1.2 Goal	3
1.3 Threat Model	4
1.4 Security goals	4
1.5 Environment	5
1.6 Outline	5
2 Background	7
2.1 Trusted execution environment	7
2.1.1 Arm TrustZone	8
2.1.2 Further TEE technologies	8
2.2 Remote Attestation	8
2.3 Trusted Platform Module	10
2.3.1 Discrete TPM	12
2.3.2 Firmware TPM	13
2.3.3 Virtual TPM	14
2.4 Secure Boot and Measured Boot	14
2.5 Device Identifier Composition Engine	15
3 Related Work	19
3.1 Attacks on TPMs	19
3.1.1 Dedicated TPM	19
3.1.2 Firmware TPM	20
3.2 Hardening of TPMs	21
3.2.1 Firmware TPMs	21
3.2.2 Virtual TPMs	21
3.3 Attestation schemes of TPMs	22
4 Methodology	23
4.1 Terminology	23
4.2 Architectural overview	23
4.2.1 Storage key	24

4.2.2	EPS and EK	25
4.3	The identity of a firmware TPM	26
4.4	Attestation process	27
4.4.1	Verifier establishes trust to the prover's fTPM	27
4.4.2	Verifier establishes trust to the prover's quote	28
4.5	Combining TPM and DICE infrastructure	29
4.6	Updating the fTPM	31
4.7	Privacy	32
5	Implementation	35
5.1	Overview	35
5.2	Boot chain	35
5.3	Firmware TPM initialization	38
5.4	Firmware TPM attestation	40
5.5	Creating and storing our EK template and certificate	42
5.6	Times in certificates and systems	43
5.7	Implementing encrypted storage	44
5.8	Isolating storage of fTPM in OP-TEE	44
5.9	Technical obstacles	45
5.9.1	tpm2-tools	45
5.9.2	OP-TEE	45
5.9.3	Firmware TPM TA	46
6	Discussion	47
6.1	Assessment of the fulfillment of requirements	47
6.1.1	Security goals	47
6.1.2	Attestation process requirements	48
6.2	Implications of openly propagating system state	49
6.3	Build pool of trusted TCIs	49
6.3.1	Closed source vs. Open source	49
6.4	Hardware requirements	50
6.5	Proving the fTPM runs in the TEE	50
6.6	Caveats of attesting the fTPM's identity	51
7	Future Work and Conclusion	53
7.1	Future Work	53
7.2	Conclusion	53
	Abbreviations	55
	List of Figures	57
	List of Tables	59

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, trust is no longer implicitly assumed, e.g., by the fact that a device is located within the boundaries of a company. Instead, each device is considered compromised until proven otherwise on a per-request basis for resource (e.g., printers) and data access [Rose2020].

This is solved by remote attestation. In the simplest case, there is a prover and a verifier, as depicted in Figure 1.1. The challenge is that the verifier observes nothing but bytes from the prover, and while a benign prover will tell the truth about its state, a compromised prover will lie about its state and claim a trustworthy one. Therefore, the verifier must establish trust to a helper component on the prover's side. This component must be independent of the rest of the prover's machine and immutable, so that it can be trusted by trusting its manufacturer. The manufacturer is commonly propagated by certificates stored on such helper components and signed by the corresponding manufacturer. Consequently, the component attests the state of the prover's machine, from which the verifier can deduce whether the prover can be considered trustworthy.



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 (key) certificate (EKcert), 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.

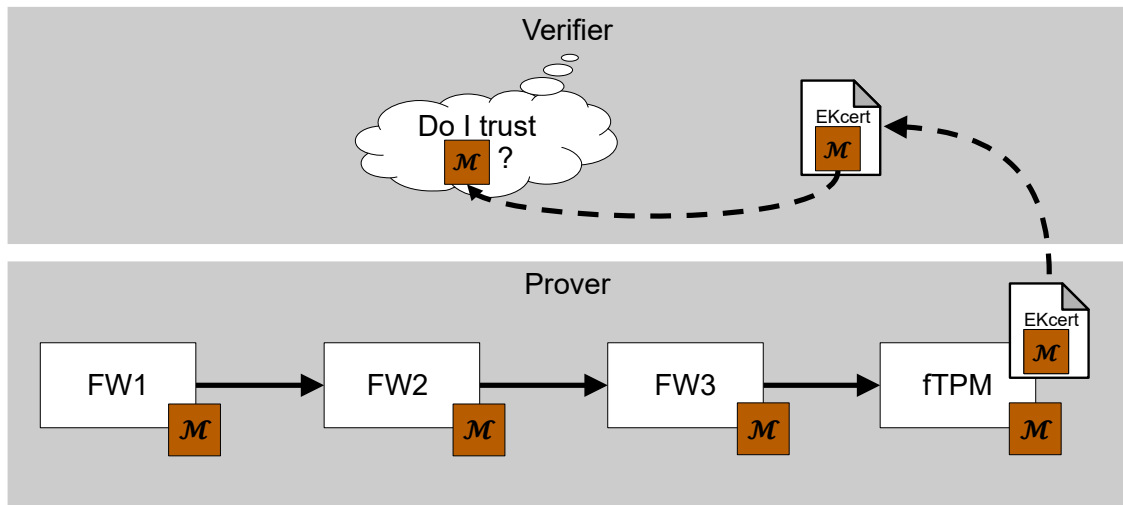


Figure 1.2: The naive process how a verifier establishes trust to an fTPM, which is in fact done by trusting its manufacturer. The brown markers indicate a manufacturer. The firmware (FW) and the fTPM were built by manufacturer \mathcal{M} , and the EK certificate indicates this manufacturer.

This process is illustrated in figure Figure 1.2. The prover's box shows its boot chain, the verifier's box shows how it evaluates the trustworthiness against the prover's 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 trust to 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 an independently verifiable fTPM stack, rooted in a hardware root of trust, that can be leveraged in a zero trust environment with little hardware requirements and no 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 their manufacturer. Instead, it is sufficient to trust the independent manufacturer of the hardware root of trust, which in contrast requires no assumptions.

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 we aim to answer are listed below.

- 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 this attestation mechanism?

1.3 Threat Model

The attacker we are interested in is able to replace the fTPM or one of its predecessor components. Therefore, there is a risk that a remote party trusts a firmware TPM that is in fact not trustworthy. For example, an attacker could install a malicious update of a relevant firmware component on the target device. However, we assume that the attacker is only able to do this before or during the boot process of the device, but not afterwards. Hardware attacks, side-channel attacks, control-flow attacks, and denial-of-service attacks are out-of-scope.

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. Last, they cannot break cryptographic primitives, e.g., encryption, signing, and hashing.

1.4 Security goals

In this section, we want to formally describe the security goals of our solution so that we can later briefly discuss whether and how we have achieved the corresponding objectives.

- **Compromised fTPM cannot fake its identity**

A compromised fTPM must not be able to lie to the verifier about its identity. It is sufficient for a lie to be recognized, and the verifier can consequently classify the verifier's fTPM as untrustworthy.

- **Small root of trust**

A root of trust of small size, e.g., in the means of code size, hardware size, and complexity, yields a small attack surface [Singaravelu2006]. This is due to the fact that a small component yields less potential implementation errors, and its approaches to guarantee specific security properties are more manageable.

- **Isolation of fTPM storage**

Data must only be accessible or modifiable within the boundaries of the TPM access controls, i.e., the TPM commands defined by its specification [tpm20]. This includes the protection against other trusted applications running in the same TEE.

- **Protect fTPM data from downgrade attack of the fTPM**

The current storage of the fTPM should be sealed to the identity of the fTPM's identity, such that when the fTPM is modified, e.g., by a downgrade attack, even the fTPM cannot access its old data anymore.

- **Privacy of remote attestation process**

The verifier should be able to establish trust to an fTPM without having to know the identity of the fTPM, i.e., its EK.

1.5 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.

1.6 Outline

In the background, we provide knowledge that is necessary to understand the rest of the work. Afterwards, we discuss related work, i.e., attacks on TPMs to further motivate this work, approaches to hardening TPMs, and work that enables remote attestation similar to ours. Under methodology, we explain the concept of our solution and subsequently present our proof-of-concept implementation. Finally, we discuss our design and implementation, rounded off by future work and the conclusion.

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 could potentially access sensitive data of the cloud tenant [Nimgaonkar2012]. In edge computing, the edge applications deal with plaintext 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 GlobalPlatform¹ 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 (TAs). 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].

Figure 2.1 illustrates the motivation of a TEE. 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

¹<https://globalplatform.org/>

and the high dynamics of software installations, while the attack surface of a trusted OS is rather small and has tightly controlled interfaces.

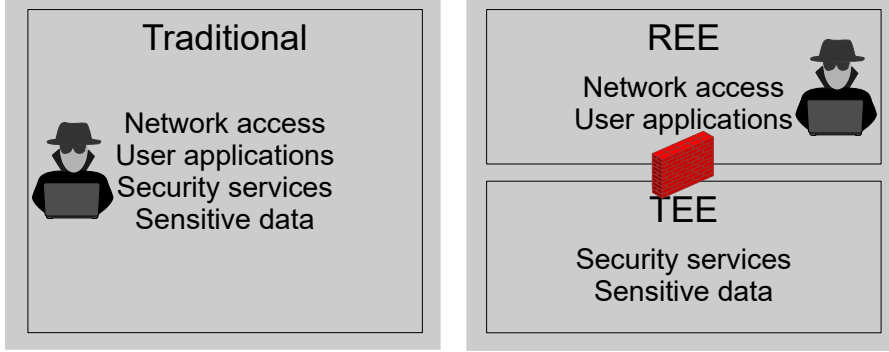


Figure 2.1: Comparison between a traditional architecture, and an architecture separating the REE and TEE. This illustrates the motivation of a TEE.

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 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 Remote 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

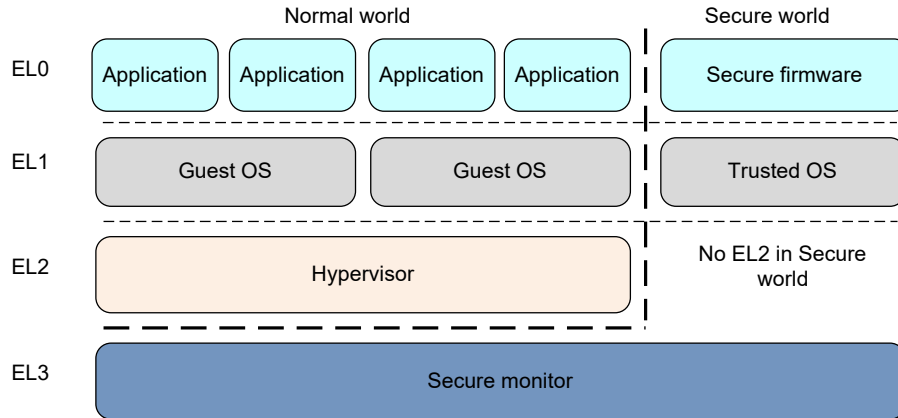


Figure 2.2: The architecture of Arm TrustZone for AArch64 [TZArch]. The exception levels (EL) indicate the privilege levels.

having the requester validate the signature and the set of measurements.”

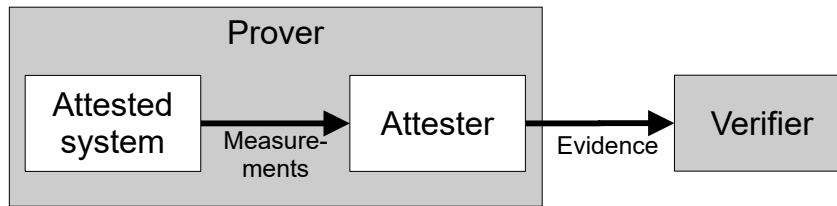


Figure 2.3: Data flow of attestation data for a remote attestation [rfc9334].

Remote attestation is a challenge-response protocol initiated by a remote party (called verifier) to verify that a target environment, i.e., the attested system, on the end-device (called prover) has not been tampered with [Menetrey2022, Coker2011]. Figure 2.3 depicts a simplified overview of its data flow.

In a typical remote attestation procedure, the attester measures the attested system during its boot process. A verifier initiates the remote attestation protocol by sending a challenge with a nonce to the prover. The prover then signs the measurement data combined with the nonce, with the final data structure usually referred to as the evidence. The nonce forces a fresh response and thus prevents replay attacks on the evidence. The evidence is then transmitted to the remote verifier, where it is evaluated.

The attester acts here as the Root of Trust for Measurements (RTM) for the verifier. Most importantly, it must be isolated from the attested system such that it cannot be compromised from the attested system. The verifier must be able to build trust to the attester through another channel, e.g., by getting to know the attester’s manufacturer and the security properties that they guarantee for their attester. It is required on the prover because the verifier requires a trusted component on the prover’s system whose statements it can trust.

Remote attestation commonly consists of two steps [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 [**dice-layering-arch**]. In implicit attestation, the state of the prover is implicitly inferred from the fact that it has control over a signing key that is only accessible if it is in a known state. That is, the sole ability to create the evidence defines the properties of trustworthiness. With explicit attestation, the state of the device is explicitly described in the evidence. The solution we propose carries out an explicit attestation.

2.3 Trusted Platform Module

The Trusted Computing Group (TCG)² published the first TPM specification (v1.1) in August 2000 [**tpm11**], and the most current specification to date (v2.0 Revision 01.59) 19 years later in November 2019 [**tpm20**]. It describes a cryptographic coprocessor that increases trust in the host platform. Specifically, this means that the TPM 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 according short explanations.

Feature	Explanation
Device identification	Identify a machine, e.g., before granting it access to resources
True Random Number Generator	Seed key generation algorithms
Key Storage	Store secret keys
Platform Configuration Registers	Store measurements of system components
Sealing	Bind access to data to state of host system i.e., specific PCR values

Each key created and stored on a TPM is part of one of four hierarchies. The following list shows the official names of the hierarchies as defined in the TPM specification [**tpm20**], some alternative names behind them in brackets, as they are sometimes referred to, and the intended use-case of the hierarchies [**Arthur2015**].

- **Storage hierarchy (owner hierarchy)**

This is the hierarchy mainly used by the end user of a TPM, e.g., to store SSH keys.

²<https://trustedcomputinggroup.org/>

- **Endorsement hierarchy (privacy hierarchy)**
Therein are stored privacy-sensitive keys, most importantly the EK.
- **Platform hierarchy**
It is intended to be used by early boot code like the UEFI. For example, the UEFI can store its configurations under this hierarchy.
- **Null hierarchy (ephemeral hierarchy)**
Used for temporary keys, e.g., when the TPM is being used as a cryptographic coprocessor. Keys in this hierarchy are discarded when the TPM is restarted.

The hierarchies are separated in order to provide actors with finely granular access to the TPM. For example, a privacy administrator can only have access to the endorsement hierarchy, while the end user only has access to the owner hierarchy. They possess different behaviors as well. The Null hierarchy, for example, cannot be restricted, while the Platform hierarchy is unlocked each time the TPM is restarted, i.e., has an empty password, and is intended to be locked by the early boot code by setting a password only known to the early boot code.

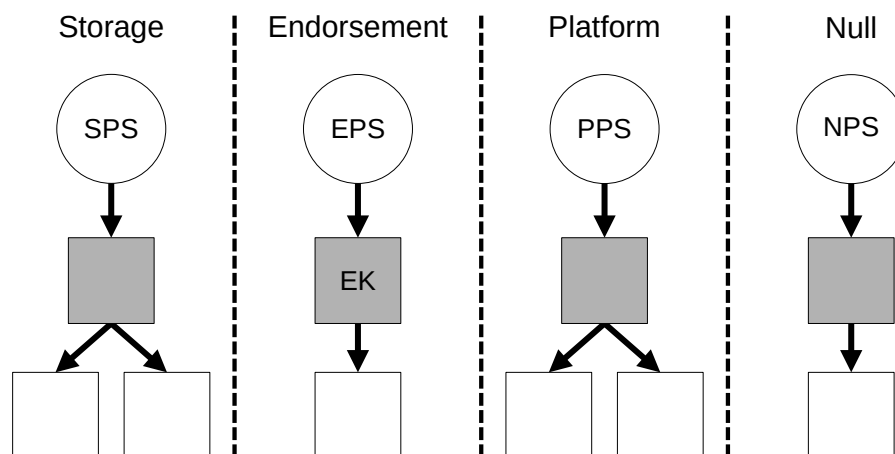


Figure 2.4: The source of entropy of the keys generated in a TPM for each hierarchy. Circle: primary seed, e.g., the storage primary seed (SPS), gray rectangle: primary key, e.g., the endorsement key (EK), white rectangle: ordinary key.

A TPM derives keys 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 an ordinary key from the parent key, as depicted in Figure 2.4. Note that the number of children and the hierarchy depth in this figure are exemplary and there are no technical restrictions for this in the TPM specification.

A template contains metadata of the key, such as the type of key, e.g., RSA, and the object attributes, e.g., 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 [tpm20]. For signing keys, this prohibits the TPM to sign data that seems to be

generated by the TPM, but is actually not. This prevents an attacker from asking the TPM to sign an artificially constructed quote. Analogously, restricted encryption keys can only be used to encrypt data generated by the TPM, e.g., other keys created on the TPM.

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 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, 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 [tpm20]. For the sake of correctness, it should be noted that not every PCR is initialized with zero, as stated 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.

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

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. 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 2.5. 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. They have their own processor, memory, and storage so that they are completely isolated from the host system and can only be accessed via the bus system. The TPM specifications [tpm20, tcgPcClient] do not demand a specific bus system, however, they define the interfaces between the TPM and the following bus systems: LPC, I²C, and SPI.

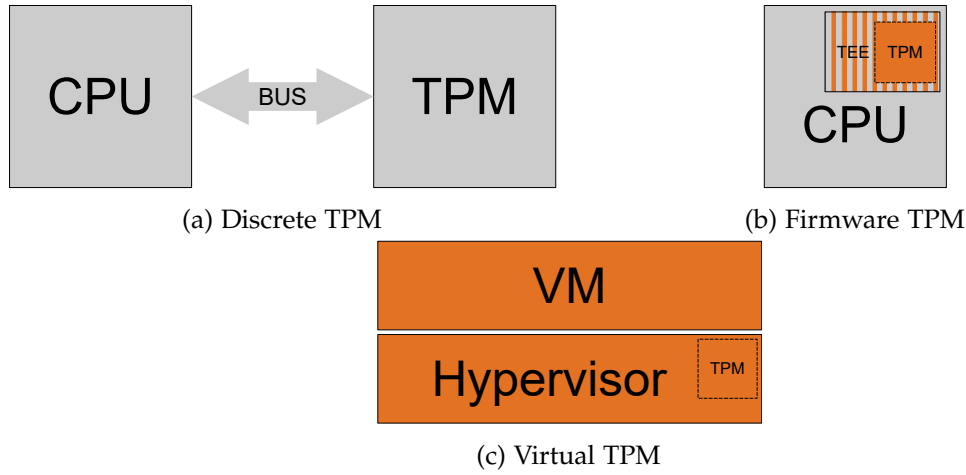


Figure 2.5: Schematic illustration of the different TPM types in their pure form. Grey: Hardware, Orange: Software.

2.3.2 Firmware TPM

An fTPM [Raj2015, 197213] is executed directly by the host CPU within a TEE. This isolates it from the REE, which means that even the operating system in the REE cannot arbitrarily access the memory of the fTPM. The REE can only send commands to the fTPM via controlled interfaces. These commands are piggybacked by calls from the REE to the fTPM in the TEE. The trend is moving towards fTPMs, which can also be seen by the increasing efforts to bring an fTPM to the RISC-V processor family [Boubakri2021]. Common implementations are the Intel® Platform Trust Technology (Intel PTT) [intelProcessorSecurity], and AMD’s Secure Processor (AMD-SP), with the latter being 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 advantages and disadvantages. A disadvantage is that running on the same processor as the rest of the system means less isolation from the remaining system. Furthermore, they are started later in the host’s boot chain than a dedicated TPM (dTPM) that is accessible from the beginning. This has the consequence that the measurements 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-A³—which is the Arm’s reference implementation of software in the TEE—protects this cached event log by keeping it in secure memory [tf-a-measured-boot], i.e., memory which is only accessible in the TEE. Last, fTPMs depend on more components for its security than single-component dTPMs, e.g., the hardware-provided isolation between the REE and the TEE, and the boot chain.

However, since they require only a TEE which is mostly already available at currently used processors, they are cheaper for manufacturers as they require less extra hardware.

³<https://www.trustedfirmware.org/>

In addition, TPM processors are weak [Goh2013, Raj2015]. Raj et al. [Raj2015] and Cheng et al. [Cheng2020] independently concluded that firmware TPMs executed on main processors are generally much faster than dTPMs. fTPMs are also a viable option for adding TPM functionality to older devices through software updates rather than hardware replacements, which is particularly valuable in times like the recent chip supply shortage [Voas2021, casper2021].

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 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, i.e., they can be suspended and later continued on another machine. Virtual TPMs support this as well. Note that a vTPM does not inherently have its own security properties, but that these depend entirely on the vTPM implementation of the hypervisor.

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 verifier 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 low hardware requirements.

DICE operates on a boot process layered into components [dice-layering-arch]. Each layer must only assume the lower layers to be trustworthy. 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 to prevent time-of-check to time-of-use attacks (TOCTTOU) [Hristozov2022, Carpent2018]. The union of all security-relevant components of a device form its Trusted Computing Base (TCB). Their individual identities are called TCB Component Identifier (TCI), which are usually the hashes of the according firmware binary, but could also consist of a hardware product identifier. The TCI must also represent security-relevant configurations of a component. Compilation flag configurations are already inherently part of the TCI, as they affect the final binary and are therefore automatically measured as part of the measuring it. There are also configurations that are not part of the binary, which are usually provided in well-known formats such as json or xml. They must be measured in conjunction with the corresponding binary file or represented by two separate TCIs.

The DICE layer is the first to be booted. Its specification [dice-hardware-reqs] states 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 provide a basis for long-term identification and derivation of the device's own secrets. (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 later 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 layer's 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 no subsequent layer has 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).

$$CDI_n = \underbrace{UDS \circ TCI_0}_{CDI_0} \circ \dots \circ TCI_n \quad (2.2)$$

$$DeviceID_KeyPair = KDF(CDI_0) \quad (2.3)$$

$$Alias_KeyPair_n = KDF(CDI_n) \quad (2.4)$$

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 2.6 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 alias certificates being generated during boot time.

$$Manufacturer\ Cert \rightarrow DeviceID\ Cert \rightarrow Alias\ Cert [\rightarrow Alias\ Cert]^* \quad (2.5)$$

The chain's 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, and is typically self-signed.

The next certificate is the DeviceID certificate. It is generated during device provisioning by the manufacturer, and signed by the manufacturer's 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 DeviceID's 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 can be generated, which is only possible

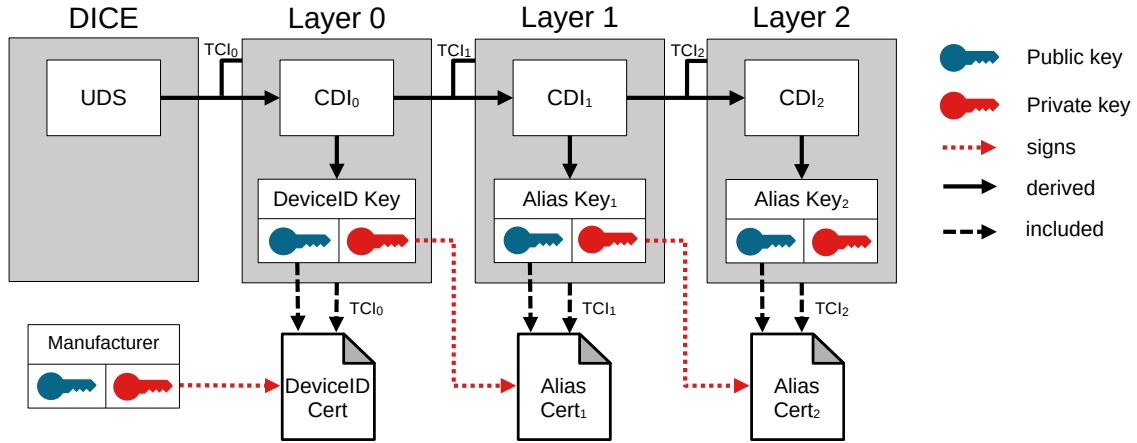


Figure 2.6: The generation of the CDIs and the certificates for each layer in a DICE architecture. Note that the diagram could be continued for an arbitrary number of layers.

if the identity of layer 0 did not change, the certificate chain can be continued.

The remaining alias certificates in the chain each represent the identity of a layer. They contain the measured TCI of layer n and are signed with the private key of the measuring layer $n - 1$.

The generation of the CDI values and certificates are shown in Figure 2.6. 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 $AliasCert_n$ are both created by layer $n - 1$, but passed to and used by layer n .

For remote attestation, the prover forwards this DICE certificate chain to the verifier. The verifier must understand the manufacturer part of the DeviceID certificate from DICE and decide whether to trust it. If this is the case and the signatures of the certificate chain can be verified with the corresponding public keys, the verifier can derive the running software of the prover from the TCIs part of the certificates. Finally, he can decide whether he considers this list of TCIs to be trustworthy.

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

3.1 Attacks on TPMs

3.1.1 Dedicated TPM

The well-known ‘TPM Reset Attack’ on TPM 1.1 was described independently in [kauerBernhard, sparks2007], whereby the state of the host computer and the TPM are decoupled. It requires minimal hardware, precisely only a wire connecting the reset line of the LPC bus [lpc] to ground. The TPM understands this as a reset signal, yielding predictable values for the PCR registers. This allows an attacker perform a boot process with malicious components, later resetting the PCR values to a known value with the reset attack, and then replay the measurements of a benign boot process. This not only spoofs the attestation process, but also allows the attacker to access secrets stored on the TPM, which is sealed to the benign state of the host machine. TCG mitigated this problem by introducing localities with TPM 1.2 by restricting the extension of specific PCRs to special hardware modes that are no longer accessible in the later boot process [tpmResetMitigation].

Winter and Dietrich [Winter2013] demonstrate a bus modification attack at TPMs integrated with the LPC bus or the I²C 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 Static RTM (S-RTM), to also Dynamic RTM (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, e.g., a PC, while preventing the TPM from receiving the reset signal. This keeps the state of the TPM, i.e., the PCR values of the previous benign 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 benign PCRs. While its conceptually easier since the attacker does not need to know the measurement log since the benign 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. If the host platform goes into sleep mode, it can send a command to the TPM demanding it to store its current state including its PCRs in its non-volatile random access memory (NVRAM). When the host platform wakes up again, it can request that the saved state be restored with a corresponding command. However, the

specification lacks a concrete description of the behavior if the TPM has not saved any state before going to sleep, but still receives the command to restore its “saved” state when waking up. It merely states that the TPM implementation is expected “to take corrective action”. Hence, some implementations simply reset the TPM which resets the PCRs as well. This also applies to the latest version of the TPM specification at the time of writing [tpm20]. Their second attack targets a D-RTM, namely an implementation flaw in tboot¹, a widely used measured boot environment used with Intel’s Trusted Execution Technology. They found that some mutable function pointers are not measured, which enables 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 plaintext. 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.

3.1.2 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 circumvents this by being directly executed by the CPU within a TEE, revealing no easily accessible bus.

However, 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 host’s boot process and allows them to boot their own firmware component that leaks the required information.

Cfir Cohen from the Google Cloud Security team also targets AMD’s fTPM running AMD-SP (formerly known as AMD-PSP) [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. However, this bug seems to be limited to AMD’s fTPM implementation and does not appear in TCG’s reference code.

¹<https://www.sourceforge.net/projects/tboot>

3.2 Hardening of TPMs

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.

3.2.1 Firmware TPMs

One approach is to formally verify the code of fTPMs towards specific security properties. Mukhamedov et al. [Mukhamedov2013] write portions of the TPM 1.2 code in a functional programming language—namely F#—that enables automatic verification.

Raj et al. [Raj2015] call for hardware entropy for a secure fTPM implementation, but do not elaborate on how this can be achieved. Kim and Kim [Kim2019] propose an abstraction layer on top of a fTPM and a dTPM—the hybrid TPM (hTPM). It makes it possible to switch between the hardware and software solution as required. They use the hTPM to combine the advantages of an fTPM and a dTPM, e.g., by making the dTPM the source for the hardware entropy of the fTPM. In addition, the hTPM performs significantly better in software mode than in hardware mode due to the use of modern CPU features. However, all this comes at the cost of increasing complexity.

In contrast, Gross et al. [Gross2021] propose backing an fTPM with hardware without requiring a dTPM. For that, they provide cryptographic and entropy support through hardware. However, their implementation inherits the downsides of fTPMs which are not related to their hardware support, but to their software nature. 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.

3.2.2 Virtual TPMs

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 host's 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.

3.3 Attestation schemes of TPMs

The SMART attestation mechanism proposed by Defrawy et al. [EURECOM+3536] establishes a D-RTM with little hardware requirements. In fact, their hardware requirements and overall approach are similar to DICE, i.e., they require a ROM containing a key (corresponding to DICE’s UDS), and that can only be accessed by SMART. Their secret key is directly used to sign attestation data, while for DICE the UDS acts as entropy to derive individual secrets from for each firmware component. This allows SMART to establish a D-RTM, in contrast to the S-RTM provided by DICE. However, sealing data to the identity of a firmware component is not possible with SMART. Hence, while the sole remote attestation of an fTPM is enabled with SMART, it cannot bind the fTPM’s storage to the fTPM’s identity.

Jäger, Petri, and Fuchs [Jaeger2017] describe how the remote attestation procedure described in the DICE specification can be put into practice by discussing implementation options. However, they complement our work by evaluating how to implement DICE. Jäger and Petri continue their work later [Jaeger2020] because they observed a limitation in their initial implementation, allowing to jump into DICE code possibly leaking the UDS. Lorych and Jäger carried on exploring the design space of DICE [Lorych2022] later on. As with SMART, the goal of all these publications is not to attest an fTPM and therefore do not describe how to combine the infrastructure of DICE and fTPMs.

Ménétrey et al. [Menetrey2022] discuss attestation mechanisms for TEEs. Our fTPM is also running in a TEE, and can thereby be attested with this approach. However, it assumes that the entire TEE is trusted

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 identifier (FWID). The TCG consortium later renamed it 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 adopted 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 machine-readable 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. This ensures that the explanation of our implementation better matches the actual code, where FWID is the term as it is used by automatically generated code.

Occasionally, the name of an asymmetric key is suffixed with ‘priv’, ‘pub’ or ‘cert’ to designate the private part, the public part, or the certificate, respectively. For example, EKpriv refers to the private portion of EK. And EKcert corresponds to the certificate with EKpub as its subject key.

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 2.6). This is to be expected, since our system leverages DICE as our 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 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

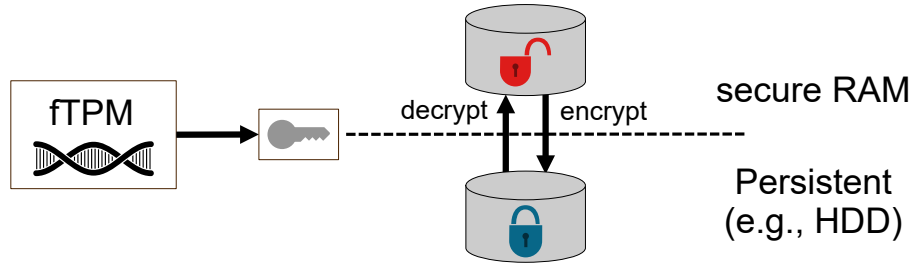


Figure 4.2: The fTPM's storage is protected by a key derived from its identity.

a particular fTPM and can exploit a vulnerability of an earlier version of that fTPM, it is not possible to replace the fTPM with its old version, as neither the attacker nor the fTPM itself will be able to decrypt the previous data.

However, it does not protect against the isolated downgrade of the fTPM itself or solely its data. When the fTPM is downgraded, as previously described, the data is reset. Nevertheless, new data generated by the downgraded TPM might still be leakable by vulnerabilities of the downgraded fTPM. Our storage key also does not protect the fTPM data from a rollback attack, i.e., the freshness of the fTPM data is not guaranteed. This attack can be attractive for malicious actors to reset the try count of PINs to work around the lockout mechanism of the TPM. Another example is to restore the data wherein a secret was stored, however, not yet protected by a PIN. The protection against the rollback of fTPM data or the fTPM itself can be achieved by storing them in a Replay Protected Memory Block (RPMB) partition [eMMC, UFS]. For this reason, an RPMB partition is part of Microsoft's hardware requirements for a firmware TPM [Raj2015].

Only the TEE can write to the RPMB (authenticated write). And the TEE can ensure that received data really originates from the RPMB (authenticated read). Each command is unique due to a nonce (for read operations) or a write counter (for write operations), which prevents replay attacks. The secure channel to the RPMB is established by a secret key shared between the TEE and the RPMB. Hence, every component within the TEE can arbitrarily access and modify the data on the RPMB, and must therefore be trusted. This is different to our approach with the storage key, whereby we bind the access to the data to the identity of the fTPM, i.e., we trust only the identity of the fTPM instead of the entire TEE. In other words, we seal the data of the fTPM with the identity of the fTPM instead of allowing the entire TEE to access the data at any time. However, RPMB and our storage key are orthogonal and can be used in conjunction.

4.2.2 EPS and EK

Then, the Endorsement Primary Seed (EPS) is generated based on the CDI. 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 EK from the EPS via the CDI instead of generating it 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 CDI must also be removed from memory as quickly as possible to reduce the time frame in which leaks are possible, but the EPS must be accessible to the fTPM throughout its entire runtime. It is therefore good practice to extract long-term secrets from the short-term secret CDI and then quickly delete the CDI from memory.

The EK of a dedicated TPM represents the long-term identity of its host device as long as the TPM is not soldered or plugged away. Our EK does not do this because a firmware TPM 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 by default because the resulting signatures may reveal the TPM’s identity. We deviate from that by creating the EK as a restricted signing key. While this breaks privacy of the prover, it has the advantage of not requiring a third party Certificate authority (CA). More details and an extension to our system introducing privacy is provided in Section 4.7.

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, i.e., only for case (iii). 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 a component’s TCI is part of its CDI, as shown in Figure 4.3.

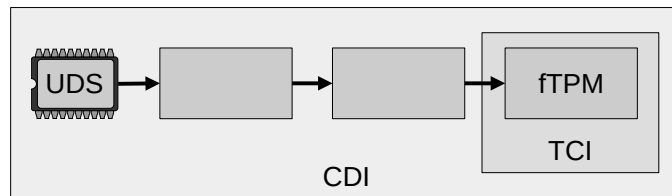


Figure 4.3: Visualizing the difference between the CDI and the TCI from the perspective of an fTPM component. The identity of the hardware is provided in the form of the UDS.

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 rationale for this is that once a previous component has changed, it is unknown whether it has gone into a benign or malicious state. And if it is malicious, it could change the firmware TPM and thereby access its sensitive material such as private keys. Although this is later recognized by the verifier during remote attestation, sensitive data could still be leaked. If the identity of the firmware TPM is extended to everything prior to the fTPM, its data is no longer accessible as soon as something preceding it changes. This is due to the derivation of the storage key from the CDI. And this behavior should also be reflected in the EK, so that the storage key and the EK should always change together or not change at all.

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, which makes its measurement as part of the TCI pointless.

4.4 Attestation process

We use an explicit attestation procedure. This makes it sufficient for the verifier to know its trusted TCIs, whereas implicit attestation would require a database of trusted Alias public keys representing a trusted component. And since each Alias public key is device unique, as it roots in the device's unique UDS, the verifier would need to know all Alias public keys for each component on every device he trusts, which we consider unrealistic. Also, this would be a hindrance as the verifier should be able to establish trust into an unknown device by trusting the DICE manufacturer, and knowing the identities of trustworthy firmware.

4.4.1 Verifier establishes trust to the prover's fTPM

First, the verifier retrieves the DICE certificate chain generated by our solution from the prover. He must then check whether the signatures of the certificate chain are valid and whether he considers the DICE implementation to be trustworthy by knowing its manufacturer. This also involves checking if the DeviceID certificate issued by the manufacturer was revoked. Such an event might occur if the security of an old DICE implementation is broken, and the manufacturer wants to reflect this.

At this point, the verifier must traverse the certificate chain starting from the root and verify whether each component represented by a certificate is trusted, by checking the embedded TCI. An untrusted component must be assumed to lie about its conducted measurements. For example, it can modify the subsequent component without reflecting this in the measurement of the TCI. Consequently, as soon as a component is untrusted,

all subsequent components have to be considered untrusted as well. This behavior is also represented mathematically in Equation 4.1. Therefore, trusting the fTPM requires to trust all underlying firmware components.

$$C_{i, \text{trusted}} := \bigwedge_{k=0}^i \text{trusted}(C_k) \quad (4.1)$$

The *trusted* function of Equation 4.1 checks the information provided by a certificate C against security policies defined by the verifier. In the following, we would like to present some example policies for improving comprehensibility for the reader.

- We generally do not trust the component's manufacturer and therefore, do not trust the component.
- The component is up-to-date, and there are no known vulnerabilities and therefore, we trust the component.
- The component is outdated, but all updates are only functional instead of security-relevant, so we still trust it.
- We do not know the TCI of the component. We follow a 'Deny by default' policy. Therefore, we do not trust the component.

After doing all this, the verifier can only state that "I trust the certificate chain and the components of *some* machine represented by it." This is due to the possibility of any actor to simply replay the certificate chain. But we need to promote the *some* to *the machine I communicate with*. This is solved in subsequent protocols where the prover is challenged to be in control of EKpriv which corresponds to the EKpub of the EK certificate. For example, by verifying the subsequently retrieved quote. Although this is not explicitly part of our proposed solution, we nevertheless describe it for better understanding and provide a comprehensive explanation of the entire attestation process.

4.4.2 Verifier establishes trust to the prover's quote

For that, the verifier needs to know that the quote was signed with the restricted EKpriv corresponding to the EKpub in EKcert, and that the fTPM is in control of EKpriv. The verifier also needs to trust that the fTPM did not leak EKpriv, as this would not only allow to replay a whole certificate chain, but even succeed the subsequent protocols with the leaked EKpriv. The verifier can derive whether it considers the EKpriv of the fTPM as not leaked based on the fTPM's TCI value, as one requirement of trusting a TCI must be whether the verifier considers the component to keep its security guarantees.

To trigger the protocol, the verifier sends a nonce to the prover, which includes it in the quote request sent to the firmware TPM, i.e., TPM2_Quote. The nonce prevents replay attacks. The verifier also needs to know that the EK is restricted. Otherwise, a

compromised prover could generate quotes representing arbitrary states which do not represent the prover's actual state.

Usually, this is ensured by the manufacturer in that he only signs the EKcert for a restricted EK. Of course, this cannot be applied to our solution, as our EKcert is dynamically created without a TPM manufacturer asserting specific attributes of the EK. Instead, the verifier needs to derive from the fTPM's TCI embedded in the EKcert that the EK is restricted. For the EK's attributes to be represented in the TCI, the template containing the EK's attributes must be generated in the firmware TPM's code. This ensures that the TCI also represents the template.

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).

$$\text{Manufacturer Cert} \rightarrow \text{DeviceID Cert} [\rightarrow \text{Alias Cert}]^* \rightarrow \text{EK Cert} \quad (4.2)$$

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. So, this 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. DICE [**DICE_certs**] defines the requirements for various certificate types. Our certificate is referred to as an attestation certificate in their specification.

We only consider restrictions for the X.509 fields that are absolute requirements, i.e., declared as "MUST" according to RFC 2119 [**Bradner1997**]. 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 extensions. The requirements about the certificate's validity depend on whether the measuring firmware has access to a secure real-time clock (SRTC) containing the absolute physical time. We assume the firmware to not having access to an SRTC, keeping the requirements low. The restrictions of our certificate also depend on its further usage. It is a leaf of the certificate chain. Therefore, we do not consider requirements for a certificate representing a CA signing further certificates.

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

Table 4.1: Comparing the requirements for an Alias and EK certificate. If not otherwise stated, the source of these requirements are the previously noted specifications, i.e., [DICE_certs, tcg-ek]. The upper half contains basic certificate fields, and the lower half certificate extensions.

Field	Alias Cert	EK Cert	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
Subject Alternative Name	—	TPM details	No
Validity (not before)	known time in recent past e.g., build time	—	No
Validity (not after)	no expiration	no expiration [tcgPcClient]	No
Authority Key Identifier	—	must be present	No
Key Usage	not to verify signatures of certificates	verify signatures other than those on certificates	No
Certificate Policies	Local Attestation	at least one policy	No
Basic Constraints	—	not a CA	No

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. So, a general term like “fTPM” is prohibited by the EK certificate specification, and an empty subject by the DICE 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. An empty subject names also appears to be common practice in EK certificates, as this is the subject name chosen for all EK certificates we observed. This should not be regarded as representable, however, since the sample size is three.

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 embedded by the TPM’s manufacturer. For example, an underlying firmware component could change these details embedded in the firmware TPM to pretend that it is compliant with a newer specification with potential security updates than it actually is. This cannot happen with the TCI, which is part of the certificate chain, as this malicious firmware component would be detected as long as it is not the first DICE layer. To still fulfill the EK certificate specification, we suggest to use general terms. For example, the manufacturer could be defined as “DICE”, the model as “FW”, and the version as TPM 2 compliant, whereby the minor version is not specified, i.e., zero.

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 ensures that the TCI part of the EKcert generated at boot-time does not become obsolete, in other words, keeps representing the identity of the currently running fTPM.

The code of the fTPM is replaced during an update. The fTPM therefore retrieves a new CDI and then a new storage key. The old data can therefore no longer be accessed, which effectively leads to a manufacturer reset.

This mechanism is common practice as this is also described in the manuals for TPM upgrades by Lenovo [**LenovoTpmUpgrade**] and Intel [**intelTpmUpgrade**]. It is underpinned by the importance of pausing BitLocker before upgrading a TPM due to its upcoming data loss [**BitlockerTpmUpgrade**]. Thereby, BitLocker's encryption key is temporarily stored in plaintext on the hard drive, which is consequently restored on the TPM after its update.

Apart from the associated loss of data, there is no other obstacle to updating the fTPM. Updating in this sense even means replacing, e.g., with the fTPM of another manufacturer. As it is explicitly measured, it can be replaced at will without changing the manufacturer of the DICE or TEE.

4.7 Privacy

First, we elaborate what reveals the identity of the device when conducting a remote attestation, and then suggest a modification to our architecture to integrate privacy.

In an ordinary remote attestation process with our system as described in Section 4.4, the verifier retrieves the certificate chain and a TPM quote. After the root certificate, the certificate chain is continued with the DeviceID certificate, whose subject key provides the long-term identity of the device. It then continues with alias certificates, each of which contains subject keys that represent the identity of the hardware and the firmware components that have already been executed. The closing EK certificate of the chain contains the key representing the identity of the fTPM. The quote's signature is also relevant to the prover's privacy, as the signature is generated with a unique EKpriv. Consequently, all these keys have to be hidden to preserve privacy, and the signature of the quote must be generated with a key that does not represent a long-term identity.

Both is solved by introducing a new signing key—the Attestation key (AK)—which is used for signing the quote. It is created by the prover's TPM, and is an ephemeral key. Hence, the prover can generate any number of AKs at any time, e.g., for each remote attestation process. This prevents the correlation of signatures, i.e., the proof that multiple signatures originate from the same TPM. However, the AK also has to be certified to originate from an authentic TPM, just as the EK. In contrast to certifying EKs, manufacturers cannot be called upon to vouch for AKs. This is because if a manufacturer is referenced in the certificate of an AK, it is disclosed to the verifier, which means that privacy-relevant information is revealed. Instead, we rely on a third-party CA, commonly referred to as privacy CA.

The privacy CA replaces the DICE manufacturer as the root of trust for the verifier. For this purpose, this CA first retrieves the original certificate chain and an AK from the prover. In a pure TPM system, the privacy CA must verify that the EKcert represents an authentic TPM by verifying the certificate chain and retrieving the manufacturer from

the EKcert. In our system it is about the DICE manufacturer referenced by the DeviceID certificate. The privacy CA must then ensure that the AK provided by the prover comes from the same TPM as the EK of the just retrieved EKcert. In short, this works by the privacy CA generating a challenge constructed with AKpub and encrypted with EKpub, which can only be solved by an entity who has control over AKpriv and EKpriv. This process also allows the privacy CA to verify that the AK is restricted. The procedure is described in more detail in the TPM specification [tpm20] under “Attestation Key Identity Certification” and “Credential Protection.”

In addition, this privacy extension removes the need for a custom template of EK, resetting it from a signing key to an encryption key, as signing with the EK can reveal the identity of the TPM.

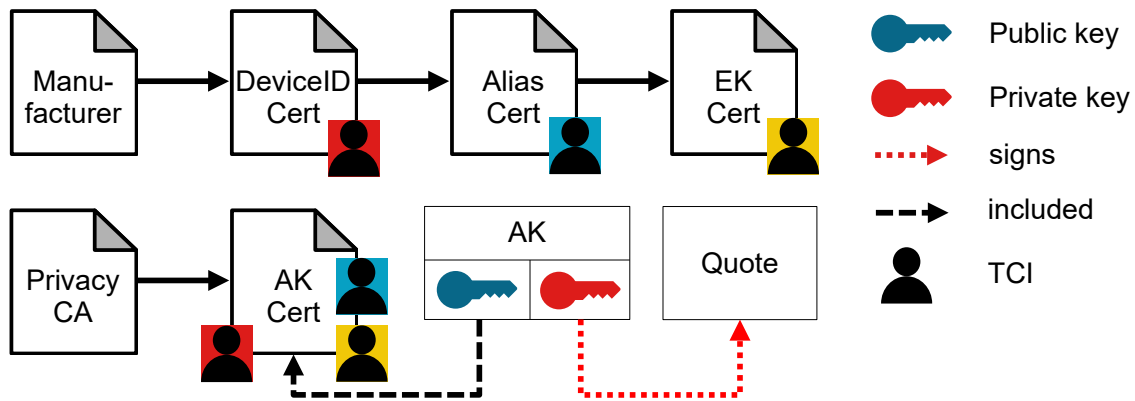


Figure 4.4: The adapted architecture integrating privacy.

We also need to transfer the DICE information from the old certificate chain to the new private one. After performing the formerly described actions, the privacy CA creates a certificate vouching for that the AK was generated by an authentic TPM, hiding which exact TPM, even the manufacturer. The privacy CA also copies each TCI of the old certificate chain into the AK certificate, as depicted in Figure 4.4. The yielding private certificate chain is returned to the prover, who can forward it to a verifier without revealing its identity.

Whether the verifier trusts the firmware TPM is conducted in the same way as described in Subsection 4.4.1, with the only difference is that the TCIs are all embedded into a single certificate, i.e., the AK certificate. The verifier must also be able to trust the prover’s quote as explained in Subsection 4.4.2. For that, he must trust the issuer of the AK certificate that he has verified the properties of the AK, i.e., that it stored on an authentic TPM. However, the privacy CA does not need to check whether AK is restricted. This can be carried out by the verifier itself, as it continues to receive the TCI of the fTPM.

Ultimately, this adapted privacy architecture changes the statement a verifier can make from “I am communicating with this authentic TPM in control of this EK”, to “I am communicating with some authentic TPM in control of this AK.” In other words, the

verifier does not know exactly which TPM they are communicating with. The privacy of the prover is guaranteed by the fact that the prover does not transmit any data derived from its UDS to the verifier.

Nevertheless, privacy is not fully ensured because of the TCI values revealed to the verifier. The order and values of the TCIs of an AK certificate might be sufficient to identify that it communicated with this particular prover before. This can only be prevented by transferring the evaluation of the TCIs from the verifier to another entity, e.g., the privacy CA. One possible realization is for the privacy CA to add to its policy that it will only certify AKs if they originate from a prover it considers trustworthy based on the TCIs it provided. However, this makes the verifier dependent on another entity that decides whether the TCIs provided are trustworthy or not, which we consider undesirable.

We also want to highlight that the AK does not need to be a child of the EK, it does not even have to be part of the endorsement hierarchy. In fact, it should not be part of the endorsement or platform hierarchy, since the TPM behaves differently when signing a quote depending on the hierarchy of the signing key [tpm20]. If the signing key is part of the endorsement or platform hierarchy, the TPM assumes that privacy is irrelevant and embeds the TPM's firmware version, reset count, and restart count in plaintext in its generated quote. This tuple might identify the TPM. If the key is part of another hierarchy, this data is obfuscated by adding random offsets to each value, which is desirable if privacy is a concern.

5 Implementation

As explained in Section 4.1, from now on the term Firmware identifier (FWID) will be used instead of the previously used term TCB Component Identifier (TCI). Also keep in mind that while TEE and REE are the technology independent terms, we mainly use Secure world (SW) and Normal world (NW) here because of our implementation with Arm’s TrustZone.

5.1 Overview

We run our implementation on Arm’s Fixed Virtual Platform (FVP)¹ which is a complete simulation of the Armv8-A architecture including TrustZone.

To do this, we use the software infrastructure provided by OP-TEE for various platforms, including FVP. OP-TEE uses the TrustedFirmware-A (TF-A)² package from Arm as firmware boot components. However, we mock their attestation, i.e., their Alias certificates are statically compiled into the binaries instead of being dynamically generated, as TF-A and FVP do not implement DICE. The development efforts to implement that exceeds the benefits, as the concept can also be demonstrated with mocked certificates. For that, only the certificates up to the OP-TEE OS are mocked, including OP-TEE OS’s private key to sign the subsequent alias certificate, which is our EKcert.

Our implementation with compilation and running instructions can be found on GitHub.³

5.2 Boot chain

The boot process is depicted in Figure 5.1. DICE is the root of trust, because incorrect behavior remains undetected and would jeopardize the security of our attestation process.

DICE Theoretically, the boot chain begins with the DICE hardware, but this is not included in FVP. Therefore, we simply assume its presence by mocking the first few certificates of the yielding certificate chain. Furthermore, it is independent hardware, and therefore, neither part of the TEE nor the REE.

¹<https://developer.arm.com/Tools%20and%20Software/Fixed%20Virtual%20Platforms>

²<https://www.trustedfirmware.org/projects/tf-a/>

³<https://github.com/akorb/master-thesis-meta>

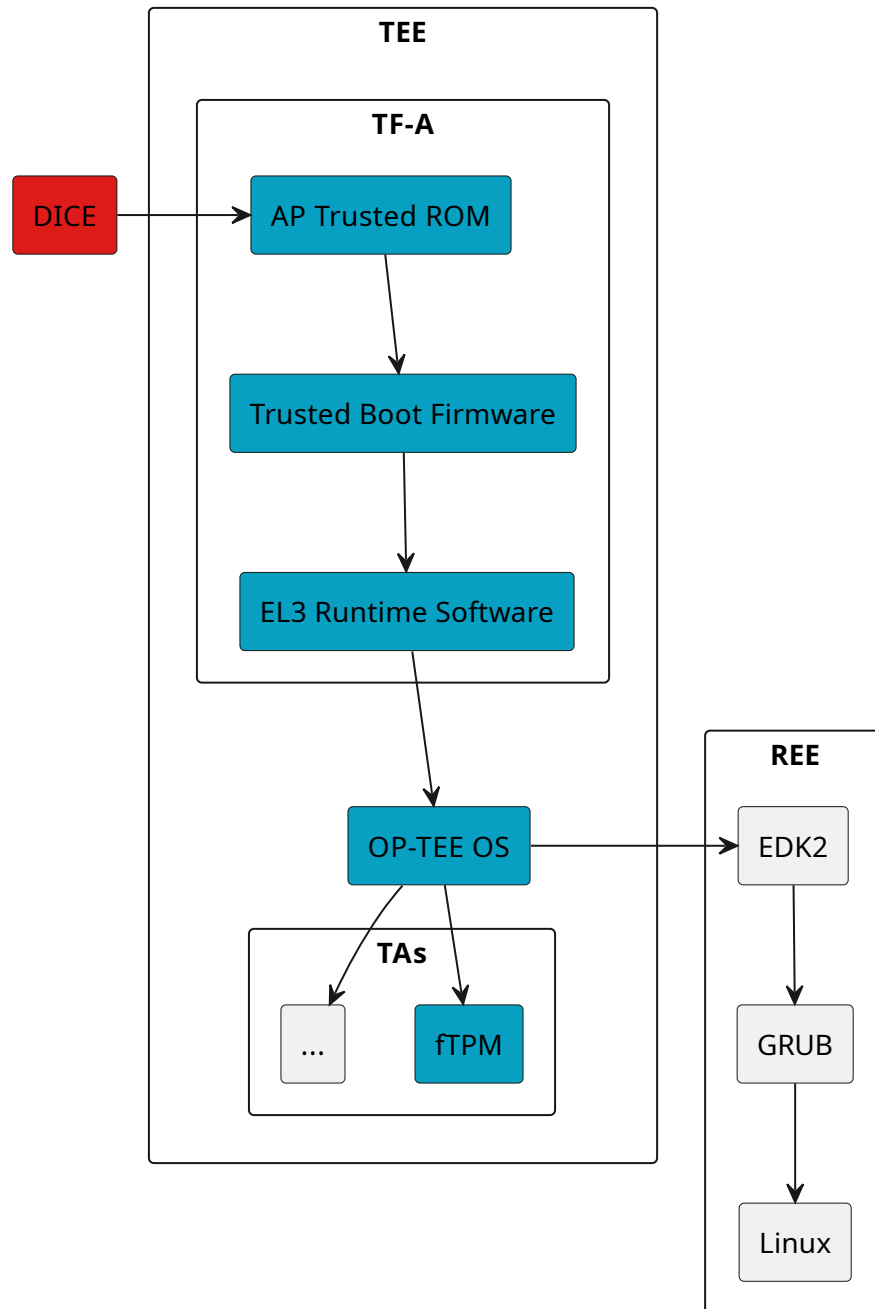


Figure 5.1: The boot chain of our system running in Arm's FVP. Blue: Represented by our yielding certificate chain. Red: Root of trust for verifier, and also just assumed to be present.

TF-A After reset, the CPU executes within the SW. That is also the reason why boot software that is unaware of the separation between SW and NW are running in the SW, as they never modify the execution environment of the processor. This ensures that such systems have all expected privileges, which would be restricted in the NW. The Application Processor (AP) Trusted ROM sets up the platform-specific exception vectors. The Trusted Boot Firmware enables the MMU, performs the platform security setup, and other tasks. The final component of TF-A—the EL3 Runtime Software—replaces the simple and rudimentary initialization performed by the AP Trusted ROM with more complete configurations by detecting the system topology, and enabling NW software to function correctly. It also provides the monitor which conducts the context switches between the SW and the NW. More complete and detailed information can be found in the TF-A documentation.⁴

OP-TEE OS Just like an ordinary OS, OP-TEE OS⁵ initializes its functions offered to the user space of the SW, i.e., the Trusted applications (TAs).

Trusted Applications Our TA in focus is the firmware TPM. We use the reference code⁶ by Microsoft which implements a TPM, and the stub code, which provides and implements the interfaces required to be a TA of OP-TEE OS. The combination of the TPM code with the OP-TEE interfaces results in a fTPM. This fTPM only allows a single connection at any time, i.e., it prohibits concurrent access as this could lead to inconsistent states. This also mirrors hardware TPMs, which are usually attached via serial buses like SPI to the processor. Typically, the only entity that communicates with the fTPM is a Linux kernel module⁷, so it is transparent to the user applications whether the TPM is implemented in firmware or hardware. Note that TAs are not started automatically. In fact, we are not aware of any function provided by OP-TEE OS to register a TA to be started during the boot process. Instead, TAs are initialized the first time someone wants to interact with them.

EDK II TianoCore EDK II⁸ is the first component launched in the NW. It is a reference implementation of UEFI [UEFI] by Intel.

GRUB The GNU GRand Unified Bootloader⁹ is a bootloader which is responsible for loading and transferring control to OS kernel software.

Linux The final component to boot is the Linux¹⁰ operating system.

⁴<https://trustedfirmware-a.readthedocs.io/en/latest/design/firmware-design.html>

⁵https://github.com/OP-TEE/optee_os

⁶<https://github.com/microsoft/ms-tpm-20-ref/>

⁷https://docs.kernel.org/security/tpm/tpm_ftpm_tee.html

⁸<https://github.com/tianocore/edk2>

⁹<https://www.gnu.org/software/grub/>

¹⁰<https://www.kernel.org/>

5.3 Firmware TPM initialization

Initialization begins with the derivation of all secrets from the CDI. Note that we mocked the CDI that would be passed from OP-TEE OS to the fTPM in practice. However, OP-TEE OS does not implement DICE.

We use the Mbed TLS library¹¹ providing cryptographic primitives for the derivation of the secrets, and also to build X.509 certificates. Mbed TLS is already part of OP-TEE, and its functionality is modular and allows certain functionality to be activated or deactivated at a fine granular level. Since the target machines are embedded devices with limited resources, the user should only activate the functions that he really needs. We therefore had to activate some functions.

The formulas which derive secrets directly from the CDI (Equation 5.1, Equation 5.2) use a keyed-hash message authentication code (HMAC) function. This is inspired by the CDI derivation proposed in the DICE hardware requirements specification [**dice-hardware-reqs**]. Actually, it proposes two functions to combine information to a new secret—a simple hash function, and a HMAC function. It also recommends the HMAC function which calculation takes a little more time, but protects the CDI with twice the level than the simple hash function. This is backed by Jäger et al. [**Jaeger2017**], and NIST SP 800–57 [**Barker2020**]. We declare the inner hash function used by the HMAC according to the required data size of the secret. For example, for storage encryption, we use AES-128, and therefore, use the MD5 function to retrieve a key with a sufficient size. Note that while MD5 is considered broken, HMAC in conjunction with MD5 is not [**Bellare2006**]. The HMAC functions are seeded with a fixed character string that describes the purpose of the output secret.

$$K_{storage} = \text{HMAC}_{MD5}(\text{CDI}, \text{'DATA STORAGE KEY'}) \quad (5.1)$$

$$EPS = \text{HMAC}_{SHA512}(\text{CDI}, \text{'ENDORSEMENT PRIMARY SEED'}) \quad (5.2)$$

$$EK = \text{KDF}(EPS, EK_{template}) \quad (5.3)$$

We must ensure we retrieve exactly the same EK as the TPM would generate by a TPM2_CreatePrimary request with our EK template. Therefore, we use the TPM internal functions to generate the EK. The EK consists of a private and a public portion. The private part never leaves the TPM, and the public portion is forwarded to OP-TEE's attestation PTA to be used as the subject key for EKcert, as shown by Figure 5.2. A pseudo TA (PTA) provides the same interfaces as an ordinary TA, but runs in kernel mode within OP-TEE OS instead of in user mode. Therefore, it has more privileges than an ordinary TA. The attestation PTA requires these privileges in order to read the memory of the calling TA, i.e., the fTPM TA, which is processed into the FWID that is finally embedded in the EKcert. The attestation PTA hashes the memory pages of the calling TA that are constant, i.e., executable or read-only data. Also, Microsoft's fTPM reference implementation does not contain separate configuration files, which simplifies the TCI generation of our fTPM by limiting it to the measurement of the fTPM itself.

¹¹<https://mbed-tls.readthedocs.io/en/latest/>

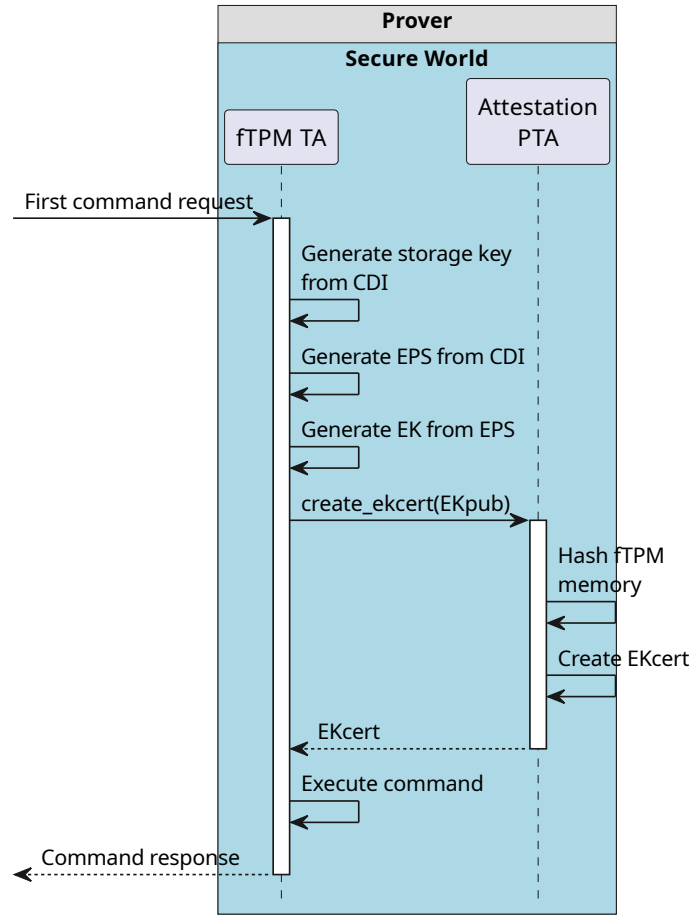


Figure 5.2: A UML sequence diagram describing the initialization of our firmware TPM.

The attestation PTA signs EKcert with OP-TEE’s private alias key. This key is mocked in our implementation. Note that the fTPM has never access to the private alias key of OP-TEE OS, so it cannot fake its alias certificate/EKcert.

Another approach to measure the fTPM is to simply extract the hash embedded in its TA binary header. A TA is an ELF binary wrapped in an OP-TEE specific format, which header contains a signature over all metadata and the payload, i.e., the ELF executable. While it is simpler and faster, it does not attest a TA’s dynamically linked libraries. Although the reference firmware TPM does not link dynamically to any library, we use the attestation of the memory data to be more future-proof.

To embed the FWID into the EKcert, we use an X.509 extension defined by the DICE Attestation Architecture [TCGAttestation2021]—the TCB Info Evidence Extension. It is defined formally in the ASN.1 description language. Consequently, we use ASN1c¹² to translate this formal definition to C code. In particular, we use a self-compiled version

¹²<https://github.com/vlm/asn1c>

of ASN1c, since its last official release dates back to 2017, whereby its generated C code throws various warnings with a modern C compiler. While it allows multiple FWIDs to be embedded into an X.509 certificate, our implementation only stores one into each alias certificate. However, the privacy focused architecture explained in Section 4.7 could leverage the possibility to store multiple FWIDs in the extension. Each FWID consists of an identifier of the hash algorithm used, and the according digest. We use SHA-256.

The resulting certificate generated by the attestation PTA is returned to the firmware TPM, which also has access to all preceding certificates. In our implementation, these preceding certificates are simply mocked.

5.4 Firmware TPM attestation

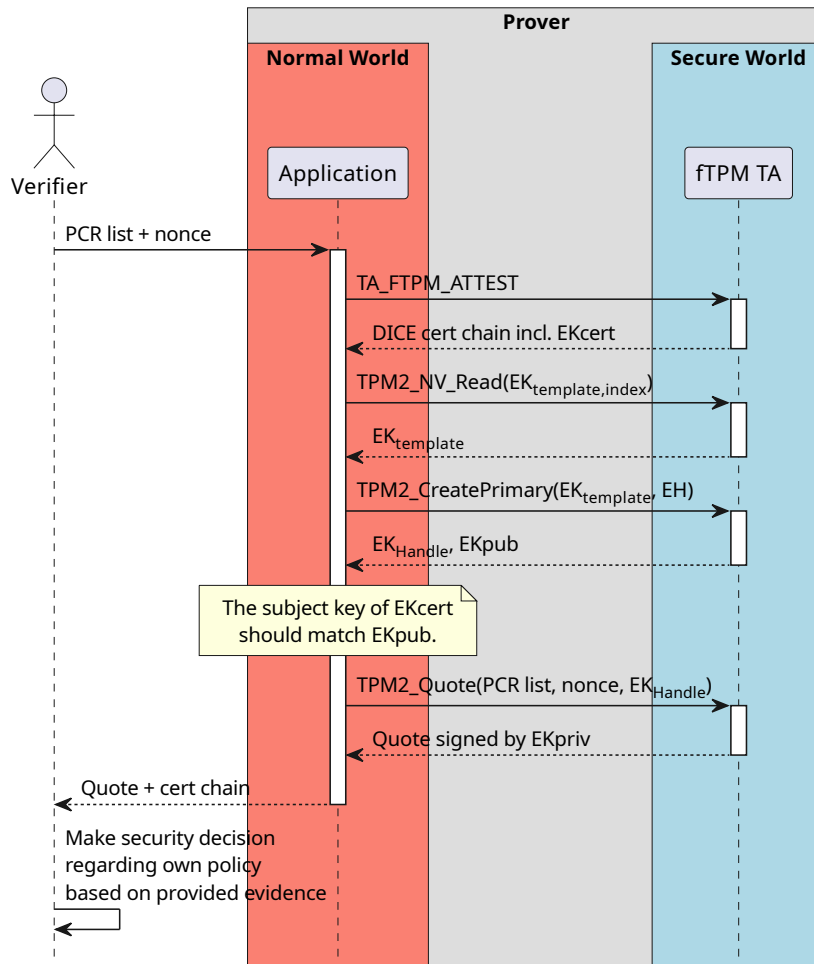


Figure 5.3: A UML sequence diagram describing the attestation of our firmware TPM.

We added a new TA command called `TA_FTPM_ATTEST` to the fTPM to obtain the

entire certificate chain from any application. Normally, this command is issued by the application that performs the prover part of the remote attestation. We would like to emphasize that we do not refer to a new TPM command that would imply an extension of the TPM specification, but a new TA command that is intercepted by the OP-TEE stub code of the fTPM and processed without the involvement of the TPM-specific core code.

Recall that we earlier described that the fTPM TA only ever allows a single connection to it, which is usually a Linux module that provides the `/dev/tpm0` and `/dev/tpmrm0` nodes to communicate with the fTPM. We have therefore implemented the prover's side of the remote attestation with the ability to unload this Linux module before issuing `TA_FTPM_ATTEST`, and then load the Linux module again.

The prover's user space application starts with issuing the `TA_FTPM_ATTEST` command to the firmware TPM, as shown by Figure 5.3. Consequently, it receives the certificate chain created by DICE with the EKcert as leaf certificate.

Afterwards, the prover wants the fTPM to create the EK including its private portion. To do this, it first reads the EK template from the non-volatile (NV) storage of the fTPM, which was used earlier to create the EK part of the EKcert during the initialization process of the fTPM.

Consequently, it then sends the template that has just been retrieved back to the TPM as part of the command `TPM2_CreatePrimary`. Since this command creates a primary key, no parent has to be specified as the primary seed and the specified hierarchy is used instead. So, we specify the endorsement hierarchy (EH), which makes the TPM use the EPS derived from the CDI to generate the EK. The TPM returns a handle to the EK just created, which is an integer, as well as the public part of the EK. The private part of the EK is not returned, as this never leaves the TPM in plaintext.

Eventually, the prover wants to create a quote to establish trust to the prover's system state in the NW, and to prove that it is in control of the EKpriv. So, it issues the command `tpm2_quote` to the TPM, specifying the PCR registers requested by the verifier, the verifier supplied nonce, and the handle to the EK just generated.

The prover's application handling the remote attestation is now in possession of the certificate chain, and a quote. It transmits both to the verifier. The verifier extracts the EKpub which is the subject key of the EKcert. With that, it can verify the digital signature of the quote. The ability of creating a quote with a fresh nonce proves the control of EKpriv by the fTPM. Therefore, the verifier can trust that the certificate chain does not have been replayed, and it represents the device it communicates with.

In our implementation of the prover we do not send the TPM commands to the TPM ourselves, but use `tpm2_createek -t` and `tpm2_quote` from the `tpm2-tools`.¹³ Nevertheless, it executes these commands behind the scenes. We also use `tpm2_checkquote` in the verifier's implementation to check the signature of the quote, and to ensure that the nonce in the quote matches the nonce that the verifier has previously generated.

Note that the implementation just described assumes that the quote returned by `TPM2_Quote` contains the values of the PCR registers. While this was the case with

¹³<https://github.com/tpm2-software/tpm2-tools>

TPM 1.2, this changed with version 2.0. Instead, the resulting quote only contains a hash of the values of the requested PCR registers. The corresponding plaintext PCR values are transmitted unprotected to the verifier. After validating the quote, the verifier can check whether the hash of the plaintext PCR values computed by itself matches the PCR hash from the quote. If this is the case, it can trust them. This is also done by the tool `tpm2_checkquote`. We have omitted it from the previous explanation and Figure 5.3 in order to focus on the important aspects. For the sake of completeness and correctness, we mention it here anyway.

As a small note, we made sure that the property `tpmGeneratedEPS` of our fTPM is set to 1 as it indicates that the EPS was generated by the TPM [tpm20], which is the case in our implementation as it is derived from the CDI within the TPM.

5.5 Creating and storing our EK template and certificate

The EK is a primary key, so it is derived from the EPS, and also a key template. The template must be provided as part of the command `TPM2_CreatePrimary` to the TPM. The TPM can contain a template for an EK key, which shall be the template with which the EK part of the EKcert was created. This is necessary 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.

A TPM can save an EK template, but is not obliged to do so. Then, the default template defined by TCG [tcg-ek] is used, dictating the EK to be a restricted encryption key, since it is privacy-sensitive, and that it must be an RSA 2048 key. As this is fitting for most TPM's, the definition of the default template removes the burden of most TPM's to provide the template. It is then the duty of the entity communicating with the TPM to provide the default EK template, since it is unknown to the TPM itself. Though, we extended our firmware TPM to be able to generate the default and our custom EK template within code, such that it can create the EK which it forwards to the OP-TEE OS to create the EK certificate.

We want to use the EK as a signing key to sign attestation data, i.e., a quote. Hence, we need to deviate from the default EK template, and also provide our template within the NV of our fTPM. We start with the default EK template and only modify the fields when necessary. In total, we needed to change three aspects. We declare the EK as a (i) restricted signing key. This also requires to specify a (ii) signature scheme where we selected RSASSA-PKCS1-v1_5 [Jonsson2003] with SHA-256, and (iii) no inner symmetric key as required by the TPM specification [tpm20] for signing keys since they are not allowed to have any children keys, as a signing key cannot encrypt its children for storage outside the TPM.

This template will be generated within the TPM after each manufacturer reset, so it will be preserved even after an identity change and a subsequent reset of the fTPM. Thus, it is always present and stored in the NV index `0x01c00004` as defined by the TPM EK specification [tcg-ek] for RSA 2048 EK templates. We set the attributes for this

NV index as defined by the TPM PC Client specification [tcgPcClient]. Thereby, the EK template in the NV can only be written or deleted if a specific policy is fulfilled. However, the policy is empty, which can never be fulfilled. This results in a non-deletable EK template. We also store the EKcert in the NV index 0x01c00002 as defined by the TCG EK Credential Profile [tcg-ek] with the same attributes as the respective template, so that it also cannot be deleted. The template and certificate do not contain any sensitive material and can hence be read by anyone using the command TPM2_NV_Read.

Nevertheless, our fTPM does not retrieve the EK template from the NV index to create the EK to subsequently forward it to the OP-TEE OS to generate the EK certificate. This would entail an indirection via the NV storage, which is not attested. Instead, we explicitly use the EK template generated by code, which is attested via the fTPM's TCI. The NV storage is not attested as it is working data and not configuration data. Hence, the fTPM's TCI would not only represent the fTPM's behavior, but also its stored data. Since a TPM can store arbitrary data, this would explode the amount of TCI values that the verifier needs to know in order to derive the trustworthiness.

5.6 Times in certificates and systems

In Table 4.1 we presented the restrictions the alias and the EK certificate specifications conduct on the valid times of the certificates. Thereby, the period of validity of our certificates must be from a known time in the recent past, e.g., the component's build time, without expiration date.

We initially aimed for using the build time for the start of the validity period. However, this turned out to be infeasible, since the guest machine in the FVP does not have internet access out of the box, and FVP also does not provide an option to use the time of the host. While the host generates the mocked certificates, the FVP must continue the certificate chain and also check whether the current time falls within the validity period of the resulting certificates. Therefore, the time of the host and the FVP guest would need to be synchronized manually, requiring unnecessary engineering effort. Instead, we fix the times of the guest machine and the certificate's validity periods to pre-determined values. This ensures that the FVP does not have to be configured for internet access which keeps the effort to launch the demonstration low, and results in a higher stability of the demonstration system.

We use 2023-07-25 00:00:00 as the start period for the certificates. This date has no further meaning and was chosen arbitrarily at some point during development. As required in the specifications, we indicate that the certificates do not have a well-defined expiration date, which is indicated by 9999-12-31 23:59:59 [Boeyen2008].

The time of the guest machine is automatically set to 2023-08-02 11:46 with the Linux tool date at startup. As before, this date has no further meaning. It is only important that this date is after the start period of the certificates.

5.7 Implementing encrypted storage

Overall, the reference TPM's memory size has a size of 16,896 bytes, separated in a NV storage of 16,384 bytes, and the admin space of 512 bytes. The admin space is reserved to persist defined data like the TPM's attributes, e.g., `tpmGeneratedEPS`. The NV storage can store arbitrary data, e.g., the EK template, certificate, or an encryption key. For example, Microsoft's BitLocker stores the encryption key for hard disk encryption in the NV storage.

The OP-TEE OS manages storage in blocks. It is only possible to write entire blocks, not partial blocks. Therefore, little changes can be expensive to persist if the according block is big. Hence, the OP-TEE specific code of the reference fTPM splits the memory size of 16,896 bytes in 512 bytes blocks resulting in $16,896 \div 512 = 33$ blocks.

We consult the recommendations of the BSI [[bsi-key-recommendations](#)] to determine the encryption method. Therefore, we use AES-128 in GCM mode to protect the data's confidentiality and integrity with an initialization vector (IV) and tags of length 96 bits. These are the recommended minimum sizes that we have chosen in order to spare resources. There is one IV and tag for each block which results in a storage overhead of 792 bytes as shown in Equation 5.4.

$$\frac{33 \times (96 + 96) \text{ bits}}{8} = 792 \text{ bytes} \quad (5.4)$$

The IVs are randomized on every write event. On any mismatch between the stored tag and the tag produced during decryption, the firmware TPM is reset.

The data is loaded from the hard disk and then decrypted only at startup time. And it is written only during shutdown of the TPM or if a command modifies the TPM's storage. As data is mainly written to the TPM during the provisioning time and only read during daily use, we expect that the impact on performance will be small.

5.8 Isolating storage of fTPM in OP-TEE

The data of the individual TAs must be saved somewhere permanently. The OP-TEE OS stores them in the REE file system. Hence, this data must be protected. OP-TEE encrypts the secure storage files before sending them to the REE using a pseudo-randomly generated key—the File Encryption Key (FEK). It is stored in the metadata of the corresponding file, encrypted with a key unique to each TA—the TA Storage Key (TSK). It is derived from the Secure Storage Key (SSK), which is unique to the device.

$$TSK = \text{HMAC}_{\text{SHA256}}(\text{SSK}, \text{TA}_{\text{UUID}}) \quad (5.5)$$

$$\text{FEK} = \text{decrypt}_{\text{TSK}}(\text{file}_{\text{metadata}}) \quad (5.6)$$

$$\text{file}_{\text{plain}} = \text{decrypt}_{\text{FEK}}(\text{file}_{\text{cipher}}) \quad (5.7)$$

Equation 5.5 shows that the TSK depends only on the SSK and the TA's UUID. As said, the SSK is same for the whole device and hence for every TA, and the TA_{UUID}

is public. A close look reveals that other TA's on the same device could therefore use the publicly available UUID to generate the fTPM's TSK and consequently, decrypt the FEKs of the fTPM's files and access private data of the fTPM. However, this is prevented by our integration of an additional storage key, which encrypts the data before it is sent to the OP-TEE OS.

The TEE specification offers an ultimate solution for that on the conception level of the trusted OS without requiring a manual encryption step in the TA itself. It is defined in the TEE Management Framework [**GP_ManagementFramework**], which offers the possibility of grouping TAs into domains and subdomains, whereby only TAs in the same domain or subdomain can potentially decrypt each other's data. However, as of time of writing, this is not implemented in OP-TEE OS yet.

5.9 Technical obstacles

5.9.1 tpm2-tools

As mentioned earlier, we use the `tpm2_checkquote` tool to verify the prover's quote on the verifier's side. However, this tool fails to perform an important check to ensure that the quote was generated by the TPM and not externally. This poses a major security risk. We have reported this to the authors of the `tpm2-tools` via the recommended channel.¹⁴

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.¹⁵

The tool `tpm2_createek` did not adhere to the TPM specification when it created the EK with a template from the NV storage of the TPM. A template always contains a nonce, but it can be overwritten by storing another nonce in a specifically defined NV index. We did not want to deviate from the standard nonce, which is a buffer of 256 bytes all set to 0. Therefore, we did not write a nonce to the NV index, and only the EK template, which conforms to the TPM specification [**tcg-ek**]. However, `tpm2_createek` expected a template *and* a nonce to be present. We ended up writing an empty nonce in the according NV index of the firmware TPM with the sole aim of circumventing this problem. Later, the maintainers of the `tpm2-tools` fixed it.¹⁶ Then, it turned out that this tool has another bug that caused the address of the nonce to be used instead of the actual nonce. We fixed this issue.¹⁷

5.9.2 OP-TEE

Initially, it failed to follow the official documentation to create the complete software ecosystem with the reference firmware TPM enabled. Eventually, we determined the

¹⁴<https://github.com/tpm2-software/tpm2-tools/security/advisories/GHSA-5495-c38w-gr6f>

¹⁵<https://github.com/tpm2-software/tpm2-tools/issues/3283>

¹⁶<https://github.com/tpm2-software/tpm2-tools/issues/3278>

¹⁷<https://github.com/tpm2-software/tpm2-tools/pull/3280>

problem and fixed it.¹⁸

It was important for us to also test whether our system behaves as expected if the CDI changes. For that, we needed to persist the storage of the firmware TPM between launches of the FVP to modify the CDI during its downtime. However, the FVP version required to accomplish this turned out to freeze when the necessary functionality was activated. It took extensive debugging efforts to find a workaround.¹⁹

The OP-TEE OS provides a libc library which implements only subset of the C standard library. Its authors copy code of the newlib²⁰ to their libc on-demand when required. Unfortunately, the code generated by the asnlc project expects functions not part of OP-TEE's libc. Fortunately, these functions were not critical and could be removed manually in a reasonable amount of time.

5.9.3 Firmware TPM TA

We have enabled a compilation option offered by the fTPM TA which appears not to have been thoroughly tested. When the option is enabled, the TPM uses code written specifically for the OP-TEE platform to generate the EPS. This resulted in a crash of the fTPM during startup. We fixed the bug and opened a pull request for the fix to be merged upstream.²¹

Eventually, we did not use this code. Nevertheless, this bug was found in the context of this thesis, and hence, we want to mention it here.

¹⁸https://github.com/OP-TEE/optee_os/issues/6111

¹⁹https://github.com/OP-TEE/optee_os/issues/6162

²⁰<https://sourceware.org/newlib/>

²¹<https://github.com/microsoft/ms-tpm-20-ref/pull/98>

6 Discussion

6.1 Assessment of the fulfillment of requirements

6.1.1 Security goals

Here, we return to the security goals defined in the introduction and describe how we fulfil them.

- **Compromised fTPM cannot fake its identity**

The predecessor component of the fTPM, i.e., the trusted OS, performs the measurement of the fTPM, embeds it in the alias certificate representing the fTPM and signs it with its private alias key. As long as the trusted OS behaves correctly, the fTPM has no access to the trusted OS's private alias key. Therefore, the alias certificate representing the fTPM can be trusted if the TCI of the trusted OS is trustworthy since it presents the trusted OS's behavior, as then the fTPM cannot fake the certificate containing its identity. Actually, all TCIs of the preceding components must be trusted for that. In general, each component of the fTPM firmware stack must not have access to the private alias key of the component preceding it.

- **Small root of trust**

We base the trust in our system on the design and guarantees of the DICE architecture. Its root of trust consists of the UDS and its protection mechanisms. The UDS may only be read once per operating time of the implementing device and only by DICE. As the actual secret is a single integer value that represents a statistically unique value, we consider the root of trust to be small.

- **Isolation of fTPM storage**

The problem originates from the fact that the File Encryption Key (FEK) for the data of a TA is derived by a TEE-wide secret and the TA's UUID, with the latter being public. Hence, any TA can potentially derive the FEK of any other TA within the same TEE. However, they cannot know the other TA's CDI, since TA's cannot measure each other. Consequently, this prevents other TA's from generating the fTPM's storage key of our solution. If the trusted OS or any other preceding component is modified, e.g., to leak the fTPM's data, the CDI of the fTPM changes, consequently its storage key and its old data cannot be accessed anymore.

- **Protect fTPM data from downgrade attack of the fTPM**

The binding of the TPM data to the identity of the fTPM described for the storage

isolation also protects against downgrade attacks. Whether an fTPM is updated, downgraded or otherwise changed with malicious intent is reflected in its CDI, so that the old storage key cannot be restored.

- **Privacy of remote attestation process**

We have proposed an extension to our system to ensure the privacy of the prover during the remote attestation process. For this purpose, we introduce a proxy, the so-called privacy CA, between the verifier and the prover. In this process, the prover sends its certificate chain to the privacy CA, which must verify that the prover is benign, and then packages the information in a way that preserves privacy by leaving behind information that reveals the prover's identity to the verifier. For example, the privacy CA must verify that the verifier's DICE implementation is authentic with its DeviceID certificate and that the fTPM is actually running within the TEE. This requires information about the prover's hardware, which must not be disclosed to the verifier if data protection is a concern for the prover. If privacy CA honors its promise not to disclose the prover's identity and the verifier trusts the privacy CA, the prover's privacy is protected.

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.

Accurate Accurate attestation data represents the actual identity of the device's firmware. This includes freshness, i.e., the data is not replayed and does not represent an old, outdated state of the device. While our system ensures freshness not alone, it is established by the subsequent protocol, e.g., retrieving a quote, as long as a verifier-supplied nonce is involved.

Interpretable 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. Our system ensures that by propagating the TCIs as part of the publicly specified TCB Info Evidence extension. Also, the TCIs constitute of a hash, and the identifier of the used hash algorithm. Both are well-known concepts easily understandable for a verifier.

Attributable 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. Just like accuracy, this is also solved by the protocol that follows the attestation of the fTPM, since such protocols usually involve signing of attestation data, usually a quote. The quote must be signed with EKpriv corresponding to the EKpub in the EKcert. And EKpriv can only be

created by a device that has the TCIs represented by the previously obtained certificate chain and the according UDS, which is secret and unique for the respective prover.

6.2 Implications of openly propagating system state

An attacker could use the TCIs to find the exact version of the running software and match this to known vulnerabilities [rfc9334].

To counteract this, the certificate chain can be transferred from the prover to the verifier via TCP/TLS, which guarantees the confidentiality of the TCI part of the certificates from eavesdroppers. However, a malicious verifier could initiate the entire attestation process, which is then the TLS endpoint that receives the certificates. This can be circumvented by authenticating the verifier. However, this means that the prover only shares its attestation data with certain verifiers. This is a trade-off that must be weighed up by the implementers of our solution.

6.3 Build pool of trusted TCIs

In our proposed solution, the verifiers must know the TCIs they trust. However, these reference values must be obtained from somewhere. This is a general problem of TPM (PCRs) and DICE (TCIs) attestation and not specific to our system. In an ideal scenario, the software developers of boot components publish the TCIs and the corresponding security attributes. The developers would also need to keep these listings up to date, e.g., if a vulnerability was discovered in a particular version to declare it insecure. The verifiers can then collect these TCIs, possibly filter them again with their own security policies, and save them as their trusted TCI pool. To date, however, we are not aware of this being common practice.

Alternatively, verifiers can also create their own trusted TCI pool. This could be a possible solution if a verifier expects a manageable number of possible firmware on the prover. This is the case, for example, if the verifier provisions the devices that will later be the provers themselves.

6.3.1 Closed source vs. Open source

It is counter-intuitive for the acquisition of TCIs that it is irrelevant whether the software is closed or open source. In fact, it can be more difficult with open source software. This is because the calculation of TCIs is based on the binary code and not the source code, since the binary code is eventually loaded in the memory of a device and executed. While closed-source software must always be provided as a binary file, open-source software may only be distributed in the form of its source code. The binary file must then be generated by the device provisioner itself. If its build environment does not support reproducible builds [Lamb2022], there may be many binaries and conversely

many TCIs that differ only in irrelevant details, e.g., embedded timestamps. This is unlikely to appear with closed source software, as there is only a single distributor.

However, closed source software restricts the criteria that can be used to decide whether a TCI is trustworthy or not. The trustworthiness of open source software can be derived independently of its source code, and also its exact change history if an older version is evaluated. With closed source software, you have to rely on the developer of the closed-source software to communicate openly if security problems occur.

6.4 Hardware requirements

A major advantage of firmware TPMs over dedicated TPMs besides the better performance is that they involve less hardware components for the final system. The question at hand is whether the hardware requirements of an fTPM with the addition of the hardware requirements of our solution undermine this advantage.

At the introduction of fTPMs by Raj et al. from Microsoft [Raj2015], they require a TEE, storage hardware supporting a Replay Protected Memory Block (RPMB) partition, a secure world hardware fuse, and a secure entropy source for the integration of a secure fTPM. Thereof, a hardware fuse, a secure entropy source, and a TEE are commonly part of the processor without the need for additional hardware. An RPMB partition is required to protect the fTPM's storage from access outside the TEE, and prevent rollback attacks. Rollback attacks of the fTPM's data or the fTPM itself are not prevented by our introduction of the storage key. So, we also recommend using a trusted OS implementing secure storage on a RPMB partition, which involves the additional hardware requirement of an RPMB which is not necessary for dedicated TPMs. However, storage has to present anyway, and a RPMB partition is supported by storage technologies commonly used in embedded devices, e.g., eMMC [eMMC] and UFS [UFS].

Our solution also involves the hardware requirements of DICE. The purpose of its hardware requirements is to protect the UDS. According to Lorych and Jäger [Lorych2022], this is usually implemented in latches which block access to a specific memory area after a bit has been set; such latches are commonly part of processors. They name the STM32G0 based on the Arm Cortex-M0+ or the STM32L4 based on the Arm Cortex-M4 as examples. For a system to be DICE-compatible, it is therefore usually sufficient for the processor to support it without the need for additional external hardware.

So, we are not cancelling out the advantage of less additional hardware components over a dedicated TPM, and instead we add restriction in the choice of processors and storage type.

6.5 Proving the fTPM runs in the TEE

It is important for the verifier to know that the fTPM runs within the TEE and not the REE. Otherwise, the fTPM would not be isolated from the components within the REE, which usually offer a large attack surface because they communicate with the Internet,

for example. The remote attestation process we propose enables this by passing the processor's name of the prover within the DeviceID certificate to the verifier. The verifier can then understand, e.g., from the hardware's manual, that the hardware starts in the TEE and can derive from the TCIs part of the certificate chain when the system switches to the REE during the boot process. It can therefore understand that the fTPM was launched and is running in the TEE.

For our proposed architecture with privacy in mind, this check has to be conducted by the privacy CA. That is due to that the verifier must not know details of the prover's hardware, since this might reveal its identity.

6.6 Caveats of attesting the fTPM's identity

The identity of a software component does not contain its current state. While the identity of a software can be derived from its binary file assuming it is statically linked, its state is derived from a snapshot of its memory during runtime.

We attest the identity of the fTPM and not its state. This means that we do not detect attacks that occur after the startup process, i.e., during its runtime.

7 Future Work and Conclusion

7.1 Future Work

The logical consequence is the implementation of our solution on real hardware instead of in a simulation environment such as FVP. This allows the interaction with the hardware to be verified, especially with an RPMB partition that requires hardware support. In addition, the impact of our solution on the performance of the system can then be measured in practice. This is especially interesting since we integrate another storage encryption.

Although it makes sense to show our solution on Arm hardware first, we are, as mentioned, not limited to it. Therefore, a future work is to concretize the description of our implementation for TEE technologies other than Arm, e.g. Intel SGX.

The system we propose can also be transferred from the DICE architecture to other technologies that also perform firmware measurements. A new technological framework that generalizes DICE is Caliptra [**caliptra**]. It is based on the concept of DICE, but is not limited to it.

As mentioned in Section 4.2, RPMB's rollback protection only protects against attacks from outside the TEE. We would like to establish a design that tightens the rollback protection from the trust of the entire TEE to the identity of the fTPM. This can probably be achieved by encrypting the metadata stored on the RPMB with the storage key derived from the identity of the fTPM, i.e., its CDI, before sending it to the RPMB. This must be implemented in the trusted operating system and not in the fTPM TA, as the trusted operating system normally manages the metadata.

Furthermore, our solution does not protect against runtime attacks on the fTPM. In general, TAs in a TEE are not resistant to security problems caused by programming errors, e.g., buffer overflow attacks. Therefore, remote attestation of the control flow integrity of the fTPM may be a desired function. Displaying the current state of the fTPM instead of its identity at boot time would be a useful extension to our solution.

7.2 Conclusion

In this work, we have proposed a novel remote attestation scheme to establish trust in a firmware TPM. fTPMs cannot be trusted based on their isolated identity alone, as their underlying software components are also security relevant, unlike dedicated TPMs which are a separate chip. We therefore use the DICE as a hardware root of trust and measure each component during the boot process up to the fTPM. The verifier can thus

learn the measurements of the corresponding components and decide whether they are classified as trustworthy. These measurements are transmitted from the prover to the verifier in the form of certificates. Since it is in the nature of certificates that they are not considered secret and can therefore be easily replayed by malicious provers, the verifier must ensure that the certificates correspond to the device he is communicating with. This is not directly ensured by our system, but should be part of the attestation protocol that runs atop of our system, e.g., the fTPM attesting the state of the system with a quote. To do this, the prover must prove that he has the private key that corresponds to the public key part of the certificate that describes the firmware TPM. These keys are unique for the identity of the device (the UDS) and the identities of the individual components (the TCIs) and therefore cannot be generated by other, potentially malicious, verifiers. We concluded the presentation of our system with an explanation of a proof-of-concept implementation and a discussion of the feasibility, caveats and limitations of the system.

We believe that our system is an important step towards the independence of the manufacturer of the firmware TPM and its upstream software, whereas today verifiers trust a single manufacturer who is assumed to have provided all these components. Our system also creates a hardware root of trust for the firmware TPM which, as the name suggests, cannot be provided by the fTPM as it is a software component. In contrast, dedicated TPMs are capable of acting as a hardware root of trust. Our system closes this gap.

Abbreviations

CA Certificate authority

RTM Root of Trust for Measurements

S-RTM Static RTM

D-RTM Dynamic RTM

TPM Trusted Platform Module

fTPM firmware TPM

dTPM dedicated TPM

TCG Trusted Computing Group

PCR Platform Configuration Register

EPS Endorsement Primary Seed

EK Endorsement key

AK Attestation key

DICE Device Identifier Composition Engine

TCB Trusted Computing Base

TCI TCB Component Identifier

FWID Firmware identifier

UDS Unique Device Secret

CDI Compound Device Identifier

TEE Trusted execution environment

REE Rich execution environment

NW Normal world

SW Secure world

TA Trusted application

List of Figures

1.1	Simplified remote attestation process.	1
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 (FW) and the fTPM were built by manufacturer \mathcal{M} , and the EK certificate indicates this manufacturer. . . .	2
2.1	Comparison between a traditional architecture, and an architecture separating the REE and TEE. This illustrates the motivation of a TEE.	8
2.2	The architecture of Arm TrustZone for AArch64 [TZArch]. The exception levels (EL) indicate the privilege levels.	9
2.3	Data flow of attestation data for a remote attestation [rfc9334].	9
2.4	The source of entropy of the keys generated in a TPM for each hierarchy. Circle: primary seed, e.g., the storage primary seed (SPS), gray rectangle: primary key, e.g., the endorsement key (EK), white rectangle: ordinary key.	11
2.5	Schematic illustration of the different TPM types in their pure form. Grey: Hardware, Orange: Software.	13
2.6	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.	17
4.1	The architecture of our system.	24
4.2	The fTPM's storage is protected by a key derived from its identity.	25
4.3	Visualizing the difference between the CDI and the TCI from the perspective of an fTPM component. The identity of the hardware is provided in the form of the UDS.	26
4.4	The adapted architecture integrating privacy.	33
5.1	The boot chain of our system running in Arm's FVP. Blue: Represented by our yielding certificate chain. Red: Root of trust for verifier, and also just assumed to be present.	36
5.2	A UML sequence diagram describing the initialization of our firmware TPM.	39
5.3	A UML sequence diagram describing the attestation of our firmware TPM.	40

List of Tables

2.1	TPM features	10
4.1	Certificate comparison	30