SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Establishing trust in an updatable fTPM using remote attestation

Andreas Korb

SCHOOL OF COMPUTATION, INFORMATION AND TECHNOLOGY — INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Establishing trust in an updatable fTPM using remote attestation

Herstellung von Vertrauen in ein aktualisierbares fTPM durch Remote Attestierung

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

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

Munich, 15.12.2023 Andreas Korb

Acknowledgments

Abstract

Contents

1 Introduction

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

1.1 Motivation

Modern trust relationships, such as Zero Trust [1], require trustworthy platforms, which can reliably report their system state. In such models, trustworthiness can only be assumed after the platform configuration has been proved by all parties of the communication.

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

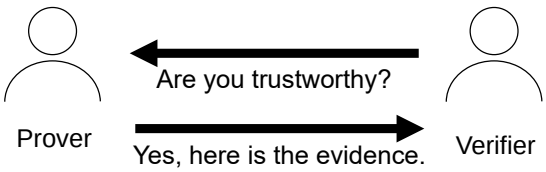


Figure 1.1: Simplified remote attestation process.

For example, this can be done with a Trusted Platform Module (TPM) on the prover's side. They rise in their deployments and importance, e.g., in 2013 the President's Council of Advisors on Science and Technology encourages the adoption of TPMs [2], and Microsoft publicized that they require a TPM module for Windows 11 in 2021 [3]. 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 [4].

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

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

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

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

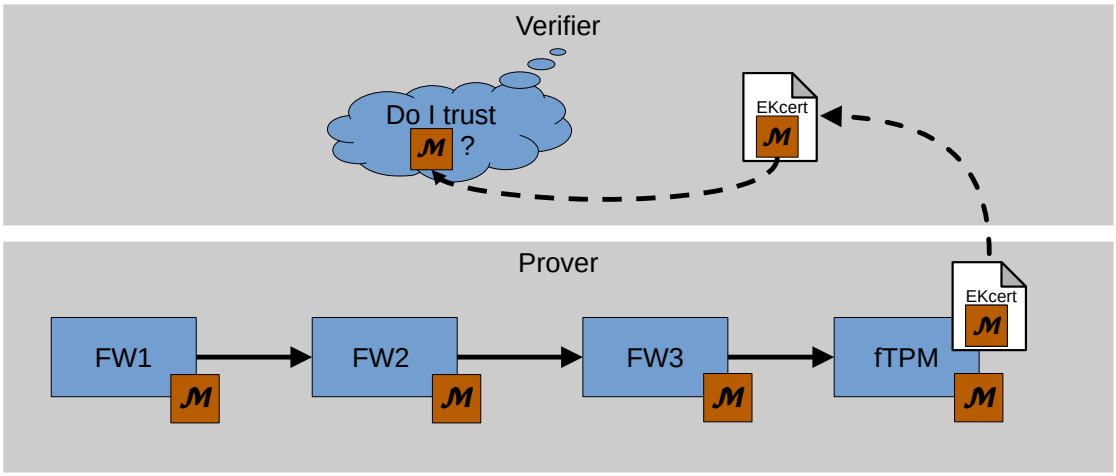


Figure 1.2: The naive process how a verifier establishes trust to an fTPM, which is in fact done by trusting its manufacturer. The brown markers indicate a manufacturer. The firmware and the fTPM were built by manufacturer $\mathcal{M}$, and the EK certificate indicates this manufacturer.

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

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

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

1.2 Goal

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

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

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

The research questions are:

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

1.3 Threat Model

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

For the network, we assume the Dolev-Yao attacker model [5]. That is, we consider an attacker who has the ability to perform any active or passive attacks on the network. The attacker may also have control over parts or the entire network, e.g., all routers, switches, and connections. However, attackers are limited in that they cannot control the end systems. They also cannot break cryptographic primitives, e.g., encryption, signing, and hashing.

1.4 Environment

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

1.5 Outline

2 Background

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

2.1 Trusted execution environment

One of the core security concepts of operating systems are the privilege levels of processes [6]. Thereby, processes are protected against other processes with the same or lower privilege level. However, they are not protected against more privileged processes [7]. This bears problems for example for cloud computing and edge computing. In cloud computing, other services, the hypervisor, or the cloud provider in general could potentially access sensitive data of the cloud tenant [8]. In edge computing, the edge applications deal with plain text data, while they are potentially running on insecure edge devices [9]. Hence, protection against more privileged processes is desired.

The Trusted execution environment (TEE) is a technology defined by GlobalPlatform[1] as an integrated hardware extension to processors. By that, the execution environment is separated into the Rich execution environment (REE) and the TEE by hardware. The REE runs commodity software, e.g., a Linux-based operating system with user applications. The TEE is an isolated tamper-resistant execution environment that guarantees the authenticity of the executed code, and the integrity of runtime states, e.g., memory [10]. Since a TEE is integrated into the processor, there is no separate chip required. Moreover, the TEE commonly follows the same user and kernel space separation as a rich OS. The kernel space is running a trusted OS, and the user space is running the trusted applications. It focuses on resisting software-based attacks generated in the REE, however, also protects against some hardware attacks [11].

Previous, mostly software-based technologies ensure confidentiality and integrity protection of data-in-transit and data-at-rest [12], while a TEE additionally protects data-in-use in hardware [12, 13].

Figure 2.1 illustrates the motivation. In the traditional architecture, i.e., without a TEE, if an attacker compromises the REE the full system is affected. With a TEE, the

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

attacker is limited to the REE, while the TEE continues to protect the secure assets, such as encryption keys. This results from the observation that the attack surface of a rich OS is much larger than that of a trusted OS, e.g., due to its network connectivity and the high dynamics of software installations, while the attack surface of a trusted OS is rather small and has tightly controlled interfaces.

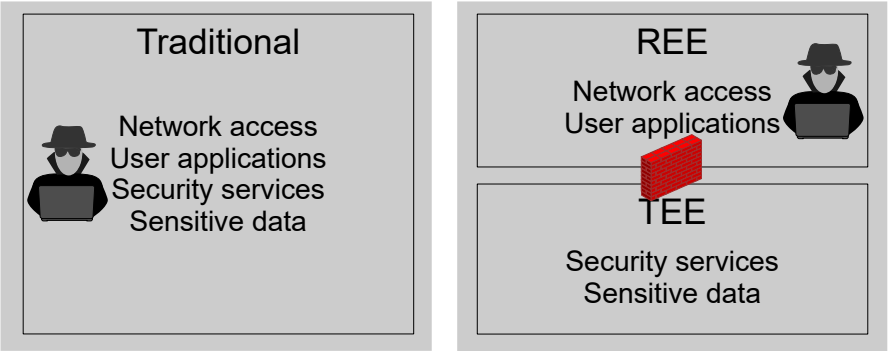


Figure 2.1: Comparison between a traditional architecture, and an architecture separating the REE and TEE. This illustrates the motivation of a TEE.

2.1.1 Arm TrustZone

One such TEE is ARM's TrustZone [14, 15]. 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 % [16], many of the approaches in this field of research use Arm TrustZone [17].

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

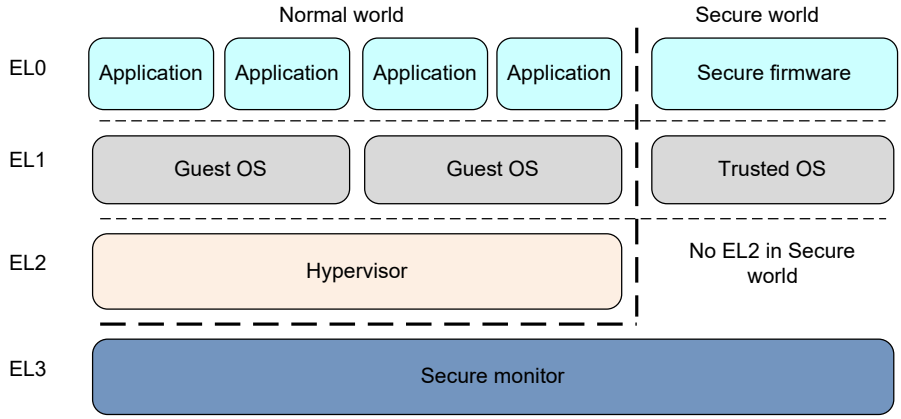


Figure 2.2: The architecture of Arm TrustZone for AArch64 [18]. The exception levels (EL) indicate the privilege levels.

these other technologies here. However, since our concept is not tied to ARM processors and can also be applied to others, they are mentioned for the sake of completeness.

2.2 Attestation

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

2.2.1 Local attestation

Local attestation is a procedure in which the state of a computer is measured, whereby the measurement result does not leave the computer but is used directly by a local component. One such example is a TPM that releases data, e.g., an encryption key, only when the computer is in a known state. This feature is known as sealing [20].

2.2.2 Remote attestation

In contrast to that, the measurement, usually called evidence, leaves the measuring machine, and is transmitted to a remote verifier for a remote attestation. This involves cryptographic primitives to establish trust into this evidence which is generally transmitted through an untrusted network.

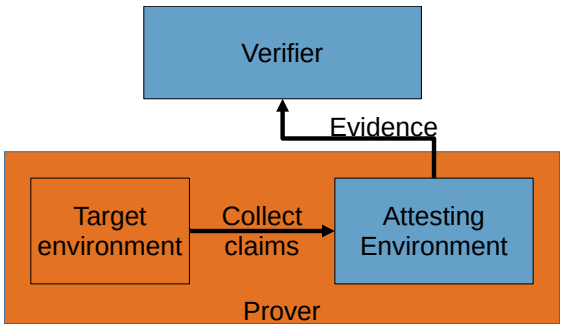


Figure 2.3: Data flow of remote attestation [21]. Initially, only the blue areas are trusted by the verifier. With the attestation, the verifier can choose to trust the target environment based on its measurements.

Remote attestation is a challenge-response protocol initiated by a remote attestor. Figure 2.3 depicts a simplified overview of the data flow of a remote attestation. The process is initiated by a remote trusted party (called "verifier") to verify that a target environment on the end-device (called "prover") has not been tampered with [22, 23]. This challenge contains a nonce, enforcing a fresh response. The response must be an evidence of the challenged system that it is trustworthy. To build that, an attesting environment on the prover device generally inspects the following properties of a program: (i) its code and data has been correctly loaded into memory for execution, and (ii) its data has not been maliciously modified at runtime.

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

It typically consists of two parts [24]. (i) The attestation, and (ii) the accompanying establishment of a secure channel. In this work, we focus on the first step.

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

2.3 Trusted Platform Module

The Trusted Computing Group (TCG)[2] published the first TPM specification (v1.2) in 2009 [25], and the most current specification (v2.0 Revision 01.59) ten years later in 2019 [20]. It describes a cryptographic coprocessor that increases trust in the host platform. Specifically, this means that the platform exhibits the expected behavior and that this behavior can be trusted. For that, the TPM maintains a separated state from the host platform, which enables the TPM to take measurements of the host platform. It is also a passive device, meaning it only does something when prompted. Table 2.1 summarizes the main features of TPMs.

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

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

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

The Platform Configuration Registers (PCRs) are the fundament for the remote system attestation. They are one-way registers, which values can never be written to an exact value, but only be extended. This operation is known as 'hash extend' [26]. Its design prohibits the removal of extensions, which would cause the TPM to forget a measurement, and the arbitrary writing of values, which would overwrite any previously conducted measurements. A PCR value holds a hash representing the platform state. Thereby, a remote verifier can request a so-called 'quote' from the TPM

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

on the host in question. A quote contains the hash of all requested PCR values and is digitally signed. Typically, a TPM contains 24 PCR registers[3], as defined as the minimum by [27], with the lower PCR values representing the system boot process and the higher ones representing the events after the kernel is booted [26]. The fixed length of the PCR values is important for the memory-constrained nature of TPMs [26].

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 [20]. For the sake of correctness, it should be noted that not every PCR is initialized with zero, as implied in the equation. For example, the TPM PC Client Platform specification [27] defines that PCRs 1–15 are initialized with all bits set to 0, while PCRs 17–22 are initialized with all bits set to 1.

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

TPM 1.2 is limited to SHA-1 hashes which are considered broken [28–30]. Although the SHA-1 uses in TPM 1.2 were analyzed to be not affected [31], cryptographic algorithms only become weaker over time [26]. In reaction, TPM 2.0 offers crypto-agility and allows newer algorithms such as SHA-256. In general, TPM 2.0 is more flexible, and is always turned on, while a TPM 1.2 needed to be turned on manually. Also, TPM 2.0 is more consistent across different implementations because of broader specifications. TPM 2.0 is the focused version nowadays, e.g., Microsoft recommends TPM 2.0 over TPM 1.2 because of security advantages [32], and also requires TPM 2.0 for Windows 11 with SHA-256 PCR registers [3].

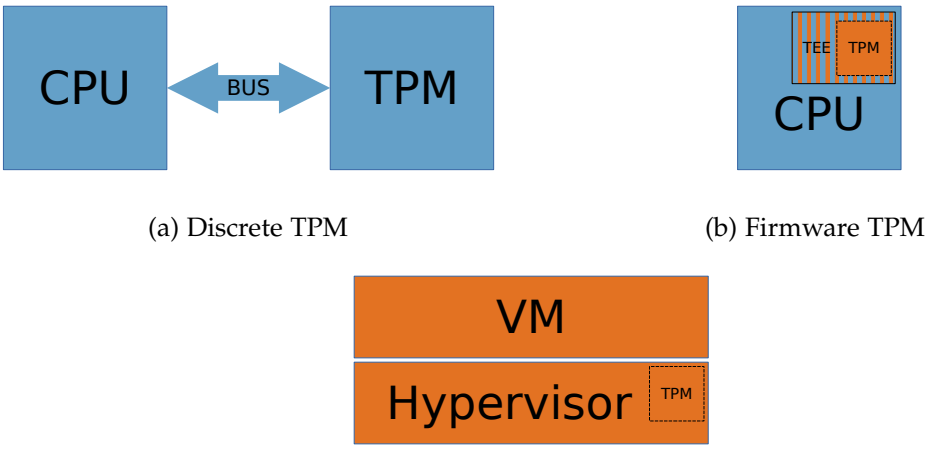
There are three types of TPMs, as illustrated in Figure 2.4. They all offer the same functionality, but with different security guarantees and performance characteristics.

2.3.1 Discrete TPM

This is the classical form of a TPM. It is a dedicated piece of hardware, connected to the CPU via a bus. It is designed and manufactured to be highly temper-resistant against hardware attacks. The TPM specifications [20, 27] do not demand a specific bus system, however, they define the interfaces between the TPM and the following bus systems: LPC, I^2C, and SPI.

The well-known 'TPM Reset Attack' was independently described in [33, 34]. It requires minimal hardware, precisely only a wire connecting the reset line of the LPC bus [35] to ground. This results in a reset signal for the TPM, yielding predictable values for the PCR registers. This allows an attacker to replay the measurement log

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



(a) Discrete TPM

(b) Firmware TPM

(c) Virtual TPM

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

of a benign boot process to achieve valid PCR values, even though a modified chain has been booted. Since TPM 1.2, TCG provides a mitigation specification for this reset attack [36], requiring the BIOS to overwrite sensitive data after each unexpected reset, preventing an attacker to gain a valid measurement log. However, this mitigation is still vulnerable to cold boot attacks [37, 38].

Winter and Dietrich [38] demonstrate a bus modification attack at TPMs integrated with the LPC bus or the I^2C bus. Their approach, labeled 'Active LPC frame hijacking', allows them to "lift" commands to a higher locality than the one they were originally sent with. This allows them to evolve the 'TPM Reset attack' from being only usable for S-RTM, to also D-RTM systems. They also introduce a new approach of circumventing the TPM's measurement feature. Instead of resetting the TPM as previously described [33, 34], they reset the main device, i.e., the users' device like a desktop PC while preventing the TPM from receiving the reset signal. This keeps the state of the TPM, e.g., the valid PCR values of the previous boot procedure, and the attacker can hijack the boot procedure triggered by the platform's reset and boot a malicious operating system or firmware, while the TPM still stores the old and valid PCRs. While its conceptually easier since the attacker does not need to know the measurement log since the valid PCR values are already in-place, it requires active manipulation of bus transmissions to shield the TPM from the reset signal.

Seunghun Han et al. [39] report two attacks on discrete TPMs to reset the PCR

registers. The first targets a gray area in the power management section of the TPM 2.0 specification. The TPM shall store its state into its non-volatile random access memory (NVRAM) before shutting down when the host platform goes to sleep, and restore it when it wakes up. However, the specification is missing a concrete description how to handle a lack of a stored state when waking up. Therefore, some implementations simply reset the state. Their second attack targets a DRTM, namely an implementation flaw in tboot [40], the most widely used measured boot environment used with Intel's Trusted Execution Technology. However, in their work, they found that some mutable function pointers are not measured, which allows attacks.

A time-based side-channel attack [41] 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 [42]. It is applicable to TPM 1.1 connected to an LPC bus. They observed that the data of some operations like unsealing are transmitted via the bus in plain text. Since TPM 1.2, however, the modules no longer send sensitive data unencrypted [38].

That invasive hardware attacks against dTPMs are possible was already shown by Tarnovsky in 2010 [43]. However, this requires a lot of time, knowledge and resources, i.e., hardware and money.

2.3.2 Firmware TPM

As seen in the previous section about discrete TPMs, the bus between the CPU and a TPM is a typical attack vector. An fTPM [44, 45] circumvents this by being directly executed by the CPU within a TEE, revealing no easily accessible bus. The trend is moving towards fTPMs, which can also been seen by the increasing efforts to bring an fTPM to the RISC-V processor family [46]. Also, since they require only a TEE which is mostly already available at currently used processors, they are cheaper for manufacturers.

As of now, a fTPM is strictly bound to the processor manufacturer, such that you can trust the underlying firmware as well which is provisioned by the manufacturer, e.g., Intel, too. For example, common implementations are the Intel® Platform Trust Technology (Intel PTT) [47], and AMD's Secure Processor (AMD-SP), which in fact is an ARM-based coprocessor on the die with Arm's TrustZone [48].

Running on the main processor, e.g., a fully-fledged Arm Cortex core, entails an advantage and a disadvantage. The disadvantage is that running on the same processor as the rest of the system means less isolation, while a dedicated TPM (dTPM) brings its own processor that is completely isolated from the main processor. The advantage, however, is that a main processor is generally much faster because dTPM processors

are weak [44, 49]. Raj et al. [44] and Cheng et al. [50] independently concluded that the firmware-based modules are generally much faster after comparing the performance of fTPMs and dTPMs.

In fact, there are more disadvantages. First, fTPMs cannot provide true RNG, since hardware is required for that [51]. Second, they are started later in the hosts' boot chain than a dTPM that is accessible from the beginning. This has the consequence that the hashes of the components booted before the fTPM have to be cached and later be forwarded to the fTPM as soon as it is available. Arm's Trusted Firmware-A[4], which is the Arm's reference implementation of software in the SW, protects this cached event log by keeping it in secure memory [52], i.e., memory which is only accessible in the SW. Last, fTPMs depend on more components for its security than single-component dTPMs, e.g., the TEE, and the boot chain.

Of course, there are also attacks against fTPMs. The previously mentioned side-channel attack [41] against dTPMs, can also be applied to fTPMs.

Jacob et al. [53] target proprietary AMD fTPMs by attacking their TEE, namely the AMD Secure Processor (AMD-SP). Thereby, they can expose the full internal state of the fTPM bypassing any authentication mechanisms. To do so, they leak the secret key from the BIOS flash chip which is used to derive the encryption and signature keys for the fTPMs non-volatile data. They achieve this by using a voltage fault injection that bypasses the authenticity check in the hosts' boot process and allows them to boot their own firmware component that leaks the required information.

Cfir Cohen from Google's cloud security team has uncovered an attack on fTPMs that run within AMD-SP [54]. They store a maliciously crafted payload — a certificate — on the fTPM and trigger a function with a stack-based overflow error that accesses this payload, giving them full control over the program counter.

2.3.3 Virtual TPM

A vTPM is a software-based TPM provided by a hypervisor for one of its managed virtual machines [55]. The vTPMs can be realized fully in software [55], or backed by dTPMs [56]. The hypervisor can provide a (theoretically) unlimited number of vTPMs. For the virtual machines it seems that they have access exclusive access to their own private TPM, even though all vTPMs are managed by the same hypervisor. A characteristic feature of virtual resources are their migration capabilities, i.e., they can be suspended and later continued on another machine. Virtual TPMs support this as well. Note the different security properties between vTPMs and dTPMs.

Because of the increasing popularity of cloud computing, the research of vTPMs focuses less on specific attacks, and more on reducing the trusted computing base, i.e.,

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

privacy-focused. The initially proposed design [55] has a large trusted base, e.g., the operating system and the hypervisor need to be trusted.

Wang et al. [57] 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 [12] describe a design named CoCoTPM where the hypervisor and the hosts' operating system do not need to be trusted as well. This is realized by establishing an integrity-protected secure channel with end-to-end encryption between the driver in the VM and the software TPM on the host.

Stateless ephemeral vTPMs [58] eliminate the need of manually establishing a secure channel by leveraging the confidential VM memory encryption provided by AMD's SEV-SNP, a variant of AMD secure encrypted virtualization (SEV) technology. Ephemeral vTPMs support the remote attestation of virtual machines. However, they intentionally do not support persistent storage to preclude exfiltration attacks on stored TPM state, which has the disadvantage that persistent keys or nonvolatile indexes cannot be stored.

2.4 Secure Boot and Measured Boot

When the system is started, the root component, e.g., from ROM, is executed. This subsequently launches the next component, and so forth. This boot structure is called the boot chain. Typically, the first component turns on the memory, the second stage initializes the platform, and finally, the last stage boots the operating system [59].

Secure Boot [60–62] 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 [63]. Hence, further defenses like Measured Boot have been designed.

Measured Boot [64] is a concept that is implemented in interplay with a TPM. It allows remote attestation to a later time. Just as with Secure Boot, each boot component hashes the subsequent component. However, instead of directly locally verifying the measured value, the hash value is passed to the TPM to extend a PCR value. As described in Section 2.3, these values can be used by a remote attestor to verify the state

of the software on the system. The goal is to detect manipulated system configurations.

Secure Boot and Measured Boot are often used in conjunction.

2.5 Device Identifier Composition Engine

DICE was originally proposed by Microsoft as part of their Robust Internet-of-Things (RIoT) architecture [65]. In 2017, the DICE specification was published by TCG [66], of which Microsoft is a member. Its purposes are to detect firmware tampering and enable device identification for a remote party, while its main attribute is its minimal hardware requirements.

DICE operates on a boot process layered into components [67]. 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 [68, 69]. The union of all security-relevant components of a device form its Trusted Computing Base (TCB). Their identities are called TCB Component Identifier (TCI). They are usually the hashes of the according firmware binary, but could also consist of a hardware product identifier.

The first component is the DICE itself, which consists of software and hardware. The DICE specification [70] states the three hardware requirements, the DICE layer

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

These requirements can be justified intuitively. (1) The UDS must be read-only and unique to the device to enable a base for long term identification. (2) The DICE layer reads and uses the UDS, and then needs to erase the UDS from memory while preventing other components from retrieving this secret during the power-on time. Otherwise, other entities can forge measurement or identification values. This lock mechanism can, for example, be realized with eFuses [70]. (3) Moreover, the misbehavior of the DICE layer cannot be detected since it is not preceded by anything that could measure it, i.e., it is the root of trust. Therefore, it must be immutable so that it remains in the trusted state in which the manufacturer provided it.

While the UDS is exclusive to the DICE layer, each other component retrieves a Compound Device Identifier (CDI) from its previous component. The CDI of each layer depends on two variables combined in a one-way function (OWF). (i) The own TCI, binding the CDI to the current layers' identity, i.e., the hash of itself, and (ii) the CDI

of the previous layer, making each CDI depending on the identities of all previous component identities. Therefore, if any component is modified, this reflects in the permutation of the CDIs of all subsequent components, as implied by Equation 2.2.

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

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

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

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

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

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

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

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

The next certificate is the DeviceID certificate. It is generated during device provisioning by the manufacturer, and signed by the manufacturers' private key. This links the DICE implementation to a manufacturer, which is important to retrieve the guarantees the manufacturer conducts for its DICE implementation to protect its UDS value, which is the hardware root of trust. This certificate also represents the long term identity of the device. The UDS alone cannot be used for this because it is kept strictly secret, whereas the DeviceID certificate is public. Since thereby the identification relies on asymmetric cryptography, the manufacturer does not need to maintain a database of UDS values, but only needs to keep and protect its private key.

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

continues the chain of trust, because if layer 0 changes, the DeviceIDs' private key also changes. This invalidates the signature of the next certificate for the public key in the DeviceID certificate.

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

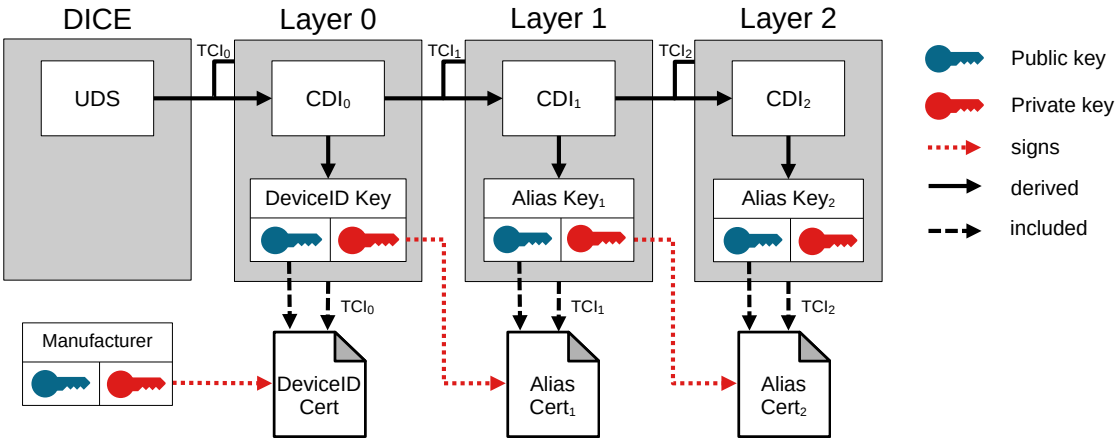


Figure 2.5: The generation of the CDIs and the certificates for each layer in a DICE architecture. Note that the diagram could be continued for an arbitrary number of layers.

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

To the best of our knowledge, DICE is so far considered a secure concept apart from physical attacks, only implementation problems can bear security problems [68, 71].

3 Related Work

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

One approach is to verify the code of fTPMs [72]. Here, the TPM 1.2 code is written in a functional programming language that enables automatic verification.

There exist efforts to improve the security of TPM by introducing the concept of hybrid TPMs [73, 74]. Kim and Kim [73] extend a hardware TPM with software support, which they name hTPM. This increases the defense of the TPM, e.g., circumventing side-channel attacks, and also enables more secure TPM functions, e.g., enabling true random number generation. Their hTPM implementation also shows significantly better performance due to the use of modern CPU features. Vice versa, Gross et al. [74] propose the reverse approach of backing an fTPM with hardware. While their implementation has similar properties to hTPM, it inherits some downsides of fTPMs. For example, their fTPM is still started later in the boot chain than a dTPM, which is not the case for hTPM. However, it is easier to update than hTPM since the lack of a dTPM, and the overall design is simpler.

4 Methodology

4.1 Terminology

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

In the original DICE release from Microsoft [65], the identifier of a component is called the Firmware ID (FWID). The TCG consortium later renamed it TCB Component Identifier (TCI). We believe this is to emphasize that the TCI does not necessarily have to be the hash of a firmware binary, but could also be, for example, the embedded ID of a hardware component. However, TCG has not fully implemented this terminology renaming. Their DICE Attestation Architecture [75] defines an X.509 extension that contains the TCIs. They continue to be referred to as FWIDs in the formal definition of this extension, while everywhere else they are referred to as TCIs. In personal correspondence with TCG, we have learned that this is due to backwards compatibility. The old term FWID is retained whenever it is used in something that is alive in the long term, like formal definitions, and the new term TCI in assets that can be updated more quickly, such as the specification text. Therefore, we will use the term TCI in this theoretical chapter, and in the implementation chapter (Chapter 5) we will use the term FWID, just as it is common practice at the TCG.

4.2 Architectural overview

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

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

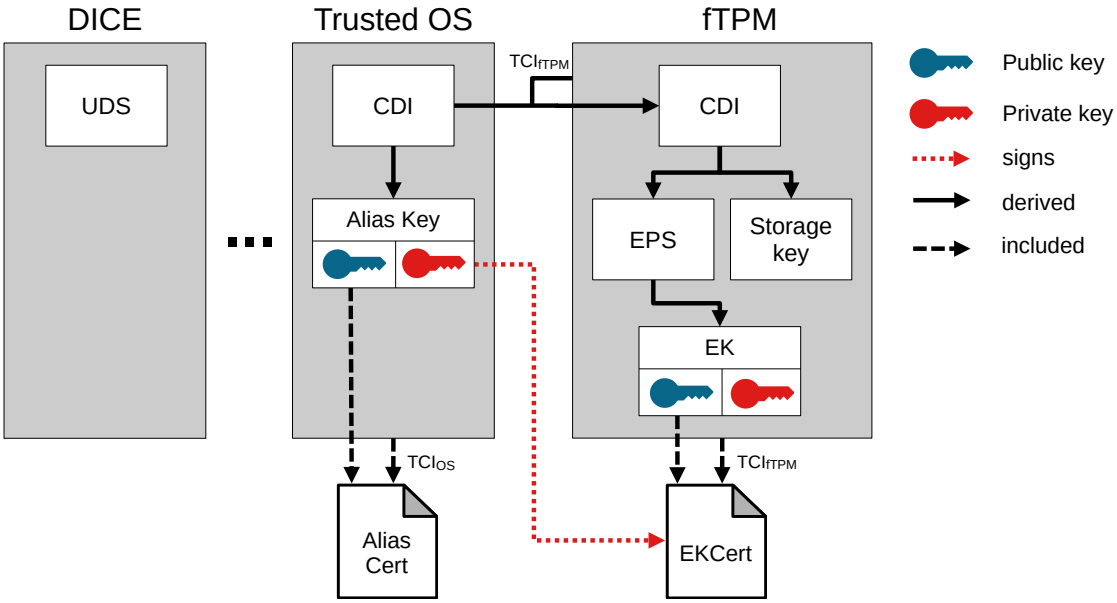


Figure 4.1: The architecture of our system.

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

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

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

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

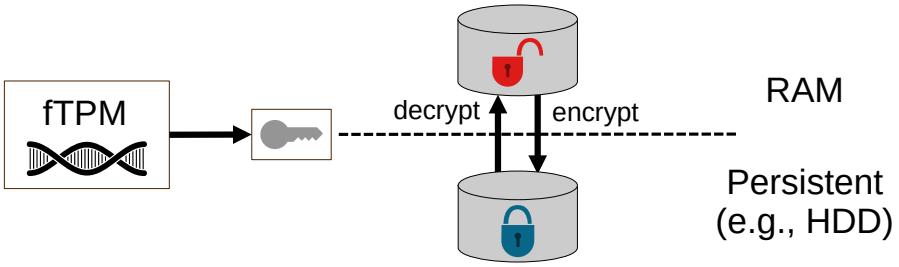


Figure 4.2: The fTPMs' storage is protected by a key derived from its identity.

is software-based and changes every time the fTPM or the underlying firmware is modified, without the host device changing. Instead, we use the DICE for this, which is hardware-based. Its DeviceID key, as the name suggests, represents the device identity. Note that the DeviceID contains the identity of layer 0 of the boot chain, i.e., the first mutable code. This can also be seen in Equation 2.3. For this reason, the DICE specification suggests keeping the first mutable code as small as possible so that it remains constant throughout the life of the device [67].

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

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

4.3 The identity of a firmware TPM

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

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

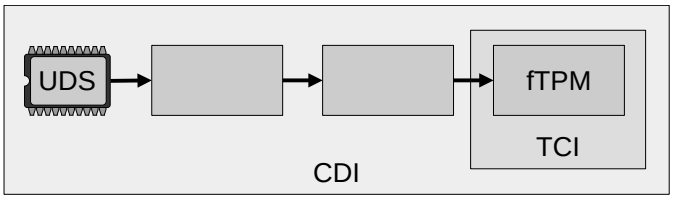


Figure 4.3: Visualizing the difference between the CDI and the TCI from the perspective of an fTPM component. The identity of the hardware is provided in the form of the UDS.

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

4.4 Attestation process

EKpriv = private portion of EK EKpub = public portion of EK EKcert = certificate with EKpub as subject

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 Does the verifier trust the prover's fTPM?

It is exactly the purpose of our system to make this question answerable for the verifier. First, the verifier needs to verify whether the signatures of the DICE certificate chain are valid, and that the root certificate from the manufacturer is considered trusted. This also involves checking if the manufacturer certificate or the Device ID certificate was revoked. Such an event might occur if the security of an old DICE implementation is broken, and the manufacturer wants to reflect this. Then, the verifier can 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.

$$C_{i,\,trusted} \coloneqq \bigwedge_{k=0}^{i} trusted(C_k) \tag{4.1}$$

After doing 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 higher level protocols where the prover is required to be in control of the private part of the EK which corresponds to the EK of the EK certificate, e.g., this is answered as part of whether the verifier can trust the quote. Hence, the verifier also needs to trust that the fTPM did not leak the EK

private key, as this would not only allow to replay a whole certificate chain, but even succeed the higher level protocols with the leaked EK private key. The verifier can derive whether it considers the EK private key 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.

Example security policies for a component

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.

4.4.2 Does the verifier trust the quote from the prover's fTPM?

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 sends a nonce, which the receiving prover includes 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, we use the fTPM's TCI embedded in the EKcert to allow the verifier to be ensured that the EK is restricted. For the EK's attributes to be represented the TCI, the template containing the EK's attributes must be generated in the firmware TPM's code. And, the verifier must only trust fTPM TCI's which have the restricted attribute of the EK baked in code.

4.5 Combining TPM and DICE infrastructure

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

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

As stated earlier, the EK certificate is where the DICE and the TPM infrastructure meet. So, this EK certificate needs to fulfill the requirements for an Alias certificate from the DICE specification [76], and the EK certificate requirements from the TPM specification [77]. Therefore, we need to ensure that these two specifications declare no conflicting requirements. DICE [76] 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 [78], and common fields that can occur in both specifications. In general, the EK certificate specification "does not preclude the use of other certificate extensions". The alias certificate specification leaves this undefined, i.e., it makes no statement whether this is permitted or prohibited. However, it is irrelevant for us, since the EK certificate specification does not define any own X.509 extensions. Some requirements depend on whether the measuring firmware has access to a secure clock. We assume the firmware to not having access to a secure clock, keeping the requirements low. 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 certificate authority (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.

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

Table 4.1: Comparing the requirements for an Alias and endorsement key certificate. If not otherwise stated, the source of these requirements are the previously noted specifications, i.e., [76, 77]. 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
Validity not before	Known time in recent past e.g., build time	–	No
Validity not after	No expiration	No expiration for non-user TPMs [27]	No
Authority Key Identifier	–	Must be present	No
Key Usage	keyCertSign unset	digitalSignature set	No
Certificate Policies	Local Attestation	At least one policy	No
Basic Constraints	–	Not a CA	No

not know the TCI part of the certificate. It also seems to be common practice, since all endorsement certificates we observed have an empty subject name. 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 [77]. It is assumed that the EK certificate is generated by the manufacturer who has this knowledge about the TPM. In our system, however, the DICE layer measuring the firmware TPM and ultimately generating the EK certificate does not know these values, as they are not constant and can change any time when the firmware TPM is exchanged. One possible solution is to keep these values in the metadata of the fTPM's binary, which the preceding layer can read and embed in the certificate. But this increases the complexity and the maintenance burden for the firmware TPM, which is usually not required since all this information (manufacturer, model, version) can be deduced from the TCI part of the certificate. Therefore, if a verifier trusts a TCI, he should also know which manufacturer, exact code and TPM specification it conforms to. Furthermore, the TCI is more accurate and reliable because it is an exact independent measurement of the firmware TPM rather than relying on information propagated by the manufacturer. For example, an underlying firmware component could alter the firmware TPM to pretend it is compliant with a newer specification (with potential security updates) than it actually is. This cannot happen with the TCI, 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 is due to the required out-of-band signing procedure of trusted applications before being deployed. This also with while system is shut down. This ensures that the TCI part of the EKcert generated at boot-time does not become obsolete, in other words, keeps representing the state of the currently running fTPM.

To protect against downgrade attacks: NV data is encrypted/integrity with AESP Encryption required to ensure confidentiality AESP used to also ensure integrity This is required since also the cipher text of the NV could be modified, which might change security critical information. Encryption-only would not prohibit that. As soon as

an integrity violation is detected, the fTPM is fully reset, effectively invalidating all previously stored data. Note that an attacker could thereby easily trigger a data loss. This has to avoided by integrating the good-practices with working with a TPM, which includes having secrets stored also elsewhere. This introduces storage and memory overhead. Processing overhead only slightly, since the data is already decrypted during start-time, which happens only once at boot time, and then later data is encrypted only while it is stored, which happens only ... Hence, there is no performance penalty during common uses of a TPM, e.g., key creation.

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

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

4.7 Privacy

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, e.g., with an ephemeral key.

Both is solved by introducing a new 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 attestation keys at any time, e.g., for each remote attestation process. This prevents cross-referencing signatures. 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 they certify an AK, the manufacturer of the TPM (or rooting DICE) which generated the AK will still be known to the verifier, since the verifier needs to know

the manufacturer's certificate. This can be a privacy-relevant information. Instead, we rely on a third-party Certificate Authority (CA), commonly referred to as privacy CA. Also, our EK is changed from a signing key to an encryption key, since signing with the EK can reveal the TPM's identity.

The privacy CA replaces the DICE manufacturer as the root of trust for the verifier. For this purpose, it 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 the only difference is that it is about the DICE manufacturer. 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 encrypted with EKpub, which can only be solved with AK. 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 [20] under "Attestation Key Identity Certification" and "Credential Protection."

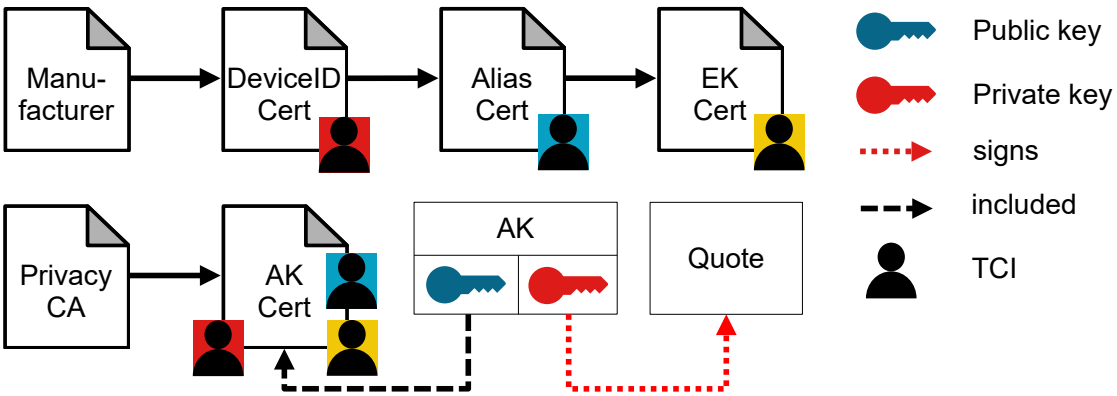


Figure 4.4: The adapted architecture integrating privacy.

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 and restricted.

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. It is guaranteed by the fact that the prover does not transmit any data derived from its UDS to the verifier. This protects the privacy of the prover.

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 [20]. 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 plain text 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 ID (FWID) will be used instead of the previously used term TCB Component Identifier (TCI).

5.1 Overview

We implement our system on ARM's Fixed Virtual Platform (FVP[1]) and the software infrastructure provided by OP-TEE for it. Therefore, the firmware components consist of ARM's TrustedFirmware-A (TF-A)[2]. However, their attestation is mocked, i.e., their Alias certificates are mocked, since TF-A does not implement DICE. The development efforts to implement that exceed the benefits, as also with mocked certificates up to the firmware the concept can be demonstrated. 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 instructions can be found on GitHub[3].

DICE The boot chain begins with the DICE hardware, which we mock as well, since ARM's FVP does not provide support for it. It is independent hardware, and therefore, neither part of the TEE nor the REE.

TF-A After reset, the CPU executes within the secure world. That is also the reason why machines that are unaware of the separation between secure world and normal world are running in the secure world, as they never modify the execution environment of the processor. This also ensures that TrustZone unaware systems have all expected privileges, which would be restricted in the normal world. 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 — EL3 Runtime Software — replaces the simple and rudimentary initialization performed by the AP Trusted ROM with more complete

[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`

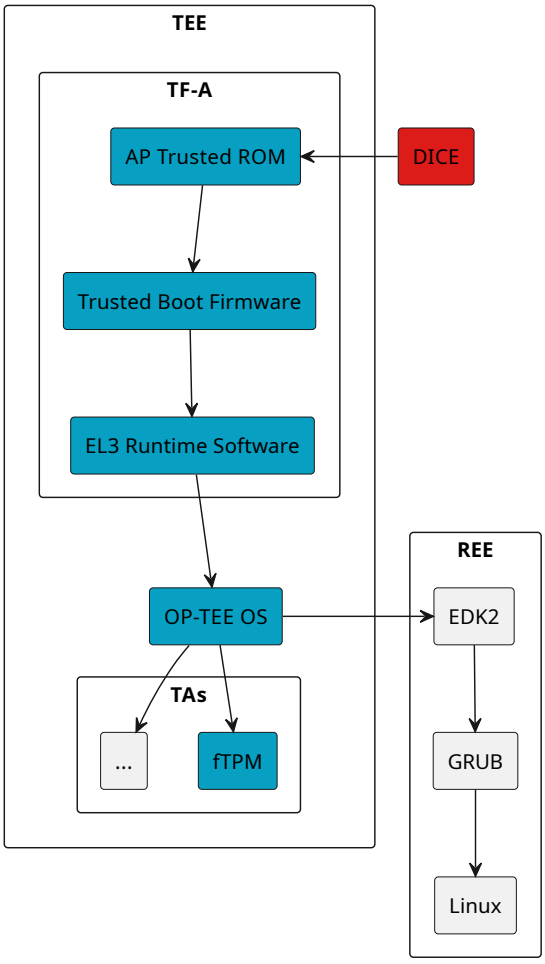


Figure 5.1: The boot chain of our system running in ARM's FVP. Blue: Represented by our yielding certificate chain. Red: Root-of-trust for verifier.

configurations by detecting the system topology, and enabling normal world software to function correctly. The EL3 Runtime Software also provides the monitor which conducts the context switches between the secure world and the normal world.

OP-TEE OS Eventually, the OP-TEE OS is loaded and executed. Just like an ordinary operating system, it initializes its functions offered to the user space of the secure world.

Trusted Applications TAs are not launched automatically. In fact, we are not aware of any functionality provided by the OP-TEE OS to register a TA to be started during the boot process. Instead, TAs are initialized the first time an application from the normal world wants to interact with it. Our TA in focus is the firmware TPM. A firmware TPM allows only 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 by serial buses like SPI. Most commonly, the only entity communicating with the fTPM will be a Linux kernel module, making it transparent to the user applications whether the TPM is implemented in firmware or hardware.

REE Last, the

5.2 Adaption of the Endorsement Key

The EK is a primary key, so it is derived from the Endorsement Primary Seed (EPS). There is also a default template defined by TCG [77], dictating the endorsement key to be a restricted encryption key, since it is privacy-sensitive. The default template is required to be able to reproduce the EK contained in the EK certificate so that the TPM can prove that this EK certificate corresponds to it by being able to generate the corresponding private key. The default template is not stored on the TPM, but it is part of the command triggering the key generation (`TPM2_CreatePrimary`). Therefore, the TPM itself does not need to know what the default template is, since it is always provided by TPM-capable software.

However, we want to use the EK as a signing key. The TPM specification provides a mechanism for this, where we need to store our custom EK template in a specific NV index within the TPM [77]. We use the values of the default EK template and only deviate from it when necessary in three aspects. We declare the EK as a (i) restricted signing key. This also requires to specify a (ii) signature scheme, and (iii) no inner symmetric key as required for signing keys since they are not allowed to have any children keys.

This template will be generated within the TPM after each manufacturer reset, so it will be preserved even after an identity change and a reset of the firmware TPM. However, the EK itself will change as the CDI changes, then the EPS and finally the EK. The attributes of the NV index are declared such that the template cannot be deleted [27]. This is possible by allowing to delete the NV index only when an unfulfillable policy is met.

The NV storage is not attested since it's "working data" and not "configuration data", and would be hard to attest since it's encrypted and the cipher text often changes since the IV is generated randomly on each store. However, the component right before the fTPM knows the fTPM's CDI, could generate the according symmetric key to decrypt the NV storage, and integrate the required NV values in the TCI. By baking the EK template generation in code which is already attested (and the code exists anyways), we prevent this complexity.

The "working data" is not attested, since this would restrict the functionality of the fTPM. So, the an NV index is with ordinary TPM commands, providing the required policy/authentication is fulfilled. We consider this configuration as out-of-scope for attestation, for the former mentioned reason. In other words, it could be that everyone can change the NV index, where the template is stored. And in the subsequent boot, we would create a EKcert for it, without anything. Therefore, during EKcert creation, generate the template in code (which is attested), and use this directly.

5.3 Prover

5.3.1 Normal World

5.3.2 Secure World

Measuring the fTPM

5.4 Attester

5.5 Times

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

5.6 Technical obstacles

We would have liked to use RSASSA-PSS which is formally proven to be secure over RSASSA-PKCS1-v1_5. RFC 8017 even requires RSASSA-PSS for new applications [79]. However, it is not fully supported by the tpm2-tools, yet[4].

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

6 Discussion

6.1 Assessment of the fulfillment of requirements

6.1.1 Security requirements

6.1.2 Attestation process requirements

TCG defines as part of their Trusted Attestation Protocol [80] the requirements for an attestation process to provide assurance to a verifier that it is (i) accurate, (ii) interpretable, and (iii) attributable.

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

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

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

6.2 Higher level protocols' compatibility

6.3 Implications of missing privacy

6.4 Hardware knowledge dependency

6.5 Hardware requirements DICE + fTPM vs TPM

6.6 Retrieve trustworthy TCIs

It boils down to whether the verifier can ensure that a specific component does not contain any relevant security holes. In an ideal scenario, the software manufacturers publicize the TCIs of their software which are considered as secure. However, as of today, this is not common practice.

6.6.1 closed source vs. open source

We expect the verifier to rely on external sources to collect its pool of trustworthy TCIs. The most prominent example is the observation of CVEs to see if specific versions of a software contain relevant security issues. For open source software, the verifier has the possibility to derive a security policy based on the source code.

Also, generally speaking, open source software receives more security audits, revealing security holes faster. With closed source firmware, the verifier is restricted to reverse-engineering (not necessarily doing that itself, e.g., it could also be done by security researchers), or relying on the developer to reliably publicize CVE's about its product.

Usually, this is easier for open source software, which is not a requirement, however.

6.7 Personal opinion about developed system

7 Future Work and Conclusion

7.1 Future Work

7.2 Conclusion

Abbreviations

TPM Trusted Platform Module

fTPM firmware TPM

dTPM dedicated TPM

DICE Device Identifier Composition Engine

TEE Trusted execution environment

REE Rich execution environment

PCR Platform Configuration Register

TCG Trusted Computing Group

NW Normal world

SW Secure world

List of Figures

List of Tables

Bibliography

[1] T. Stremlau. "A Trusted Secure Ecosystem Begins With Self-Protection." In: *ISACA Journal* 4 (July 2021).

[2] P. C. of Advisors on Science and Technology. *Immediate opportunities for strengthening the nation's cybersecurity*. Nov. 2013.

[3] Microsoft. *Windows 11 Minimum Hardware Requirements*. June 2021. URL: https://learn.microsoft.com/en-us/windows-hardware/design/minimum/minimum-hardware-requirements-overview.

[4] D. Sims. *Valorant's anti-cheat system requires TPM 2.0 and secure boot on Windows 11*. Sept. 2021. URL: https://www.techspot.com/news/91138-valorant-anti-cheat-system-requires-tpm-20-secure.html.

[5] D. Dolev and A. Yao. "On the security of public key protocols." In: *IEEE Transactions on Information Theory* 29.2 (Mar. 1983), pp. 198–208. DOI: 10.1109/tit.1983.1056650.

[6] T. A. Linden. "Operating System Structures to Support Security and Reliable Software." In: *ACM Computing Surveys* 8.4 (Dec. 1976), pp. 409–445. DOI: 10.1145/356678.356682.

[7] S. Bratus, P. C. Johnson, A. Ramaswamy, S. W. Smith, and M. E. Locasto. "The cake is a lie." In: *Proceedings of the 1st ACM workshop on Virtual machine security*. ACM, Nov. 2009. DOI: 10.1145/1655148.1655154.

[8] S. Nimgaonkar, S. Kotikela, and M. Gomathisankaran. "CTrust: A Framework for Secure and Trustworthy Application Execution in Cloud Computing." In: *2012 International Conference on Cyber Security*. IEEE, Dec. 2012. DOI: 10.1109/cybersecurity.2012.10.

[9] Z. Ning, J. Liao, F. Zhang, and W. Shi. "Preliminary Study of Trusted Execution Environments on Heterogeneous Edge Platforms." In: *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, Oct. 2018. DOI: 10.1109/sec.2018.00057.

[10] M. Sabt, M. Achemlal, and A. Bouabdallah. "Trusted Execution Environment: What It is, and What It is Not." In: *2015 IEEE Trustcom/BigDataSE/ISPA*. IEEE, Aug. 2015. DOI: 10.1109/trustcom.2015.357.

[11] GlobalPlatform. *TEE System Architecture Version 1.2*. Nov. 2018.

[12] J. Pecholt and S. Wessel. "CoCoTPM: Trusted Platform Modules for Virtual Machines in Confidential Computing Environments." In: *Proceedings of the 38th Annual Computer Security Applications Conference*. ACM, Dec. 2022. DOI: 10.1145/3564625.3564648.

[13] D. Lee. "Building Trusted Execution Environments." PhD thesis. EECS Department, University of California, Berkeley, May 2022. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2022/EECS-2022-96.html.

[14] ARM Limited. *ARM Security Technology - Building a Secure System using TrustZone Technology*. Issue C. 2009.

[15] B. Ngabonziza, D. Martin, A. Bailey, H. Cho, and S. Martin. "TrustZone Explained: Architectural Features and Use Cases." In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. IEEE, Nov. 2016. DOI: 10.1109/cic.2016.065.

[16] E. foundation. *IoT & Edge Developer Survey Report*. (Accessed July 2022). 2022. URL: https://outreach.eclipse.foundation/iot-edge-developer-2021.

[17] D. C. G. Valadares, N. C. Will, J. Caminha, M. B. Perkusich, A. Perkusich, and K. C. Gorgonio. "Systematic Literature Review on the Use of Trusted Execution Environments to Protect Cloud/Fog-Based Internet of Things Applications." In: *IEEE Access* 9 (2021), pp. 80953–80969. DOI: 10.1109/access.2021.3085524.

[18] Arm. *Security in an ARMv8 System*. 2017.

[19] M. Bartock, D. Dodson, M. Souppaya, D. Carroll, R. Masten, G. Scinta, P. Massis, H. Prafullchandra, J. Malnar, H. Singh, R. Ghandi, L. E. Storey, R. Yeluri, T. Shea, M. Dalton, R. Weber, K. Scarfone, A. Dukes, J. Haskins, C. Phoenix, and B. Swarts. *Trusted cloud :* tech. rep. Apr. 2022. DOI: 10.6028/nist.sp.1800-19.

[20] Trusted Computing Group. *Trusted Platform Module Library Specification*. Family "2.0", Level 00, Revision 01.59. May 2019.

[21] H. Birkholz, D. Thaler, M. Richardson, N. Smith, and W. Pan. *Remote ATtestation procedureS (RATS) Architecture*. RFC 9334. Jan. 2023. DOI: 10.17487/RFC9334. URL: https://www.rfc-editor.org/info/rfc9334.

[22] J. Ménétrey, C. Göttel, A. Khurshid, M. Pasin, P. Felber, V. Schiavoni, and S. Raza. "Attestation Mechanisms for Trusted Execution Environments Demystified." In: *Distributed Applications and Interoperable Systems*. Springer International Publishing, 2022, pp. 95–113. DOI: 10.1007/978-3-031-16092-9_7.

[23] G. Coker, J. Guttman, P. Loscocco, A. Herzog, J. Millen, B. O'Hanlon, J. Ramsdell, A. Segall, J. Sheehy, and B. Sniffen. "Principles of remote attestation." In: *International Journal of Information Security* 10.2 (Apr. 2011), pp. 63–81. DOI: 10.1007/s10207-011-0124-7.

[24] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. "Flicker." In: *ACM SIGOPS Operating Systems Review* 42.4 (Apr. 2008), pp. 315–328. DOI: 10.1145/1357010.1352625.

[25] ISO/IEC 11889:2009. *Trusted Platform Module*. Standard. Geneva, CH: International Organization for Standardization, May 2009.

[26] W. Arthur. *A Practical Guide to TPM 2.0 Using the New Trusted Platform Module in the New Age of Security. Using the New Trusted Platform Module in the New Age of Security*. Springer Nature, 2015, p. 392. ISBN: 9781430265849.

[27] Trusted Computing Group. *TCG PC Client Platform Firmware Profile Specification*. Level 00 Version 1.05 Revision 23. May 2021.

[28] V. Rijmen and E. Oswald. *Update on SHA-1*. Cryptology ePrint Archive, Paper 2005/010. https://eprint.iacr.org/2005/010. 2005. URL: https://eprint.iacr.org/2005/010.

[29] X. Wang, Y. L. Yin, and H. Yu. "Finding Collisions in the Full SHA-1." In: *Advances in Cryptology – CRYPTO 2005*. Springer Berlin Heidelberg, 2005, pp. 17–36. DOI: 10.1007/11535218_2.

[30] M. Stevens, E. Bursztein, P. Karpman, A. Albertini, and Y. Markov. "The First Collision for Full SHA-1." In: *Advances in Cryptology – CRYPTO 2017*. Springer International Publishing, 2017, pp. 570–596. DOI: 10.1007/978-3-319-63688-7_19.

[31] K. Goldman and S. Potter. *SHA-1 Uses in TPM v1.2*. Apr. 2010.

[32] Microsoft. *TPM recommendations*. Feb. 2023. URL: https://learn.microsoft.com/en-us/windows/security/information-protection/tpm/tpm-recommendations.

[33] B. Kauer. "OSLO: Improving the Security of Trusted Computing." In: *16th USENIX Security Symposium (USENIX Security 07)*. Boston, MA: USENIX Association, Aug. 2007. URL: https://www.usenix.org/conference/16th-usenix-security-symposium/oslo-improving-security-trusted-computing.

[34] E. R. Sparks. "A Security Assessment of Trusted Platform Modules." In: *Dartmouth College Undergraduate Theses* (2007).

[35] Intel. *Intel® Low Pin Count (LPC)*. Aug. 2002. URL: https://www.intel.com/content/dam/www/program/design/us/en/documents/low-pin-count-interface-specification.pdf.

[36] Trusted Computing Group. *TCG PC Client Platform Reset Attack Mitigation Specification*. Family "2.0", Version 1.10 Revision 17. Jan. 2019.

[37] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten. "Lest we remember." In: *Communications of the ACM* 52.5 (May 2009), pp. 91–98. DOI: 10.1145/1506409.1506429.

[38] J. Winter and K. Dietrich. "A hijacker's guide to communication interfaces of the trusted platform module." In: *Computers & Mathematics with Applications* 65.5 (Mar. 2013), pp. 748–761. DOI: 10.1016/j.camwa.2012.06.018.

[39] S. Han, W. Shin, J.-H. Park, and H. Kim. "A Bad Dream: Subverting Trusted Platform Module While You Are Sleeping." In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 1229–1246. ISBN: 978-1-939133-04-5. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/han.

[40] Intel. *Trusted Boot*. URL: https://www.sourceforge.net/projects/tboot.

[41] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger. *TPM-FAIL: TPM meets Timing and Lattice Attacks*. 2019. DOI: 10.48550/ARXIV.1911.05673.

[42] K. Kursawe, D. Schellekens, and B. Preneel. "Analyzing trusted platform communication." In: 2005.

[43] C. Tarnovsky. *Hacking the smartcard chip*. 2010. URL: http://www.blackhat.com/html/bh-dc-10/bh-dc-10-archives.html.

[44] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. *fTPM: A Firmware-based TPM 2.0 Implementation*. Tech. rep. MSR-TR-2015-84. Nov. 2015. URL: https://www.microsoft.com/en-us/research/publication/ftpm-a-firmware-based-tpm-2-0-implementation/.

[45] H. Raj, S. Saroiu, A. Wolman, R. Aigner, J. Cox, P. England, C. Fenner, K. Kinshumann, J. Loeser, D. Mattoon, M. Nystrom, D. Robinson, R. Spiger, S. Thom, and D. Wooten. "fTPM: A Software-Only Implementation of a TPM Chip." In: *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX: USENIX Association, Aug. 2016, pp. 841–856. ISBN: 978-1-931971-32-4. URL: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/raj.

[46] M. Boubakri, F. Chiatante, and B. Zouari. "Towards a firmware TPM on RISC-V." In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, Feb. 2021. DOI: 10.23919/date51398.2021.9474152.

[47] S. Domas. *11th Gen Intel® Core™ Processor Security*. Tech. rep. Intel®, 2021.

[48] F. Khalid and A. Masood. "Hardware-Assisted Isolation Technologies: Security Architecture and Vulnerability Analysis." In: *2020 International Conference on Cyber Warfare and Security (ICCWS)*. IEEE, Oct. 2020. DOI: 10.1109/iccws48432.2020.9292371.

[49] W. Goh and C. K. Yeo. "Teaching an Old TPM New Tricks: Repurposing for Identity-Based Signatures." In: *IEEE Security & Privacy* 11.5 (Sept. 2013), pp. 28–35. DOI: 10.1109/msp.2013.53.

[50] J. L. Cheng, K. C. Chen, H. W. Zhang, W. Y. Chen, and D. Y. Wu. "Emulating Trusted Platform Module 2.0 on Raspberry Pi 2." In: *International Journal of Security, Privacy and Trust Management* 9.3 (Aug. 2020), pp. 1–11. DOI: 10.5121/ijsptm.2020.9301.

[51] M. Stipčević and Ç. K. Koç. "True Random Number Generators." In: *Open Problems in Mathematics and Computational Science*. Springer International Publishing, 2014, pp. 275–315. DOI: 10.1007/978-3-319-10683-0_12.

[52] Arm Limited. *Interaction between Measured Boot and an fTPM (PoC)*. URL: https://trustedfirmware-a.readthedocs.io/en/latest/design_documents/measured_boot_poc.html.

[53] H. N. Jacob, C. Werling, R. Buhren, and J.-P. Seifert. *faulTPM: Exposing AMD fTPMs' Deepest Secrets*. 2023. DOI: 10.48550/ARXIV.2304.14717.

[54] C. Cohen. *AMD-PSP: fTPM Remote Code Execution via crafted EK certificate*. Jan. 2018. URL: https://seclists.org/fulldisclosure/2018/Jan/12.

[55] S. Berger, R. Caceres, K. A. Goldman, R. Perez, R. Sailer, and L. van Doorn. "vTPM: Virtualizing the Trusted Platform Module." In: *15th USENIX Security Symposium (USENIX Security 06)*. Vancouver, B.C. Canada: USENIX Association, July 2006. URL: https://www.usenix.org/conference/15th-usenix-security-symposium/vtpm-virtualizing-trusted-platform-module.

[56] D. Liu, J. Lee, J. Jang, S. Nepal, and J. Zic. "A Cloud Architecture of Virtual Trusted Platform Modules." In: *2010 IEEE/IFIP International Conference on Embedded and Ubiquitous Computing*. IEEE, Dec. 2010. DOI: 10.1109/euc.2010.125.

[57] J. Wang, C. Fan, J. Wang, Y. Cheng, Y. Zhang, W. Zhang, P. Liu, and H. Hu. *SvTPM: A Secure and Efficient vTPM in the Cloud*. 2019. DOI: 10.48550/ARXIV.1905.08493.

[58] V. Narayanan, C. Carvalho, A. Ruocco, G. Almási, J. Bottomley, M. Ye, T. Feldman-Fitzthum, D. Buono, H. Franke, and A. Burtsev. *Remote attestation of SEV-SNP confidential VMs using e-vTPMs*. 2023. DOI: 10.48550/ARXIV.2303.16463.

[59] J. Yao and V. Zimmer. *Building Secure Firmware*. Apress, 2020. DOI: 10.1007/978-1-4842-6106-4.

[60] J. Hendricks and L. van Doorn. "Secure bootstrap is not enough." In: *Proceedings of the 11th workshop on ACM SIGOPS European workshop*. ACM, Sept. 2004. DOI: 10.1145/1133572.1133600.

[61] UEFI Forum, Inc. *Unified Extensible Firmware Interface (UEFI) Specification Version 2.10*. Aug. 2022.

[62] J. Frazelle. "Securing the boot process." In: *Communications of the ACM* 63.3 (Feb. 2020), pp. 38–42. DOI: 10.1145/3379512.

[63] Z. Tao, A. Rastogi, N. Gupta, K. Vaswani, and A. V. Thakur. "DICE*: A Formally Verified Implementation of DICE Measured Boot." In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1091–1107. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/tao.

[64] Trusted Computing Group. *TCG EFI Platform Specification*. Version 1.22 Revision 15. Jan. 2014.

[65] P. England, A. Marochko, D. Mattoon, R. Spiger, S. Thom, and D. Wooten. *RIoT - A Foundation for Trust in the Internet of Things*. Tech. rep. MSR-TR-2016-18. Apr. 2016. URL: https://www.microsoft.com/en-us/research/publication/riot-a-foundation-for-trust-in-the-internet-of-things/.

[66] Trusted Computing Group. *TCG ANNOUNCES DICE ARCHITECTURE FOR SECURITY AND PRIVACY IN IOT AND EMBEDDED DEVICES*. Sept. 2017. URL: https://trustedcomputinggroup.org/tcg-announces-dice-architecture-security-privacy-iot-embedded-devices/.

[67] Trusted Computing Group. *DICE Layering Architecture*. Version 1.0 Revision 0.19. July 2020.

[68] S. Hristozov, M. Wettermann, and M. Huber. "A TOCTOU Attack on DICE Attestation." In: (2022). DOI: 10.48550/ARXIV.2201.11764.

[69] X. Carpent, K. Eldefrawy, N. Rattanavipanon, and G. Tsudik. "Temporal Consistency of Integrity-Ensuring Computations and Applications to Embedded Systems Security." In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. ACM, May 2018. DOI: 10.1145/3196494.3196526.

[70] Trusted Computing Group. *Hardware Requirements for a Device Identifier Composition Engine*. Level 00 Revision 78. Mar. 2018.

[71] L. Jäger and R. Petri. "DICE harder." In: *Proceedings of the 15th International Conference on Availability, Reliability and Security*. ACM, Aug. 2020. DOI: 10.1145/3407023.3407028.

[72] A. Mukhamedov, A. D. Gordon, and M. Ryan. "Towards a Verified Reference Implementation of a Trusted Platform Module." In: *Security Protocols XVII*. Springer Berlin Heidelberg, 2013, pp. 69–81. DOI: 10.1007/978-3-642-36213-2_11.

[73] Y. Kim and E. Kim. "hTPM." In: *Proceedings of the 1st ACM Workshop on Cyber-Security Arms Race*. ACM, Nov. 2019. DOI: 10.1145/3338511.3357348.

[74] M. Gross, K. Hohentanner, S. Wiehler, and G. Sigl. "Enhancing the Security of FPGA-SoCs via the Usage of ARM TrustZone and a Hybrid-TPM." In: *ACM Transactions on Reconfigurable Technology and Systems* 15.1 (Nov. 2021), pp. 1–26. DOI: 10.1145/3472959.

[75] Trusted Computing Group. *DICE Attestation Architecture*. Version 1.00 Revision 0.23. Mar. 2021.

[76] Trusted Computing Group. *DICE Certificate Profiles*. Version 1.0 Revision 0.01. July 2020.

[77] Trusted Computing Group. *TCG EK Credential Profile for TPM Family 2.0*. Version 2.5 Revision 2. Jan. 2022.

[78] S. O. Bradner. *Key words for use in RFCs to Indicate Requirement Levels*. RFC 2119. Mar. 1997. DOI: 10.17487/RFC2119. URL: https://www.rfc-editor.org/info/rfc2119.

[79] K. Moriarty, B. Kaliski, J. Jonsson, and A. Rusch. *PKCS #1: RSA Cryptography Specifications Version 2.2*. RFC 8017. Nov. 2016. DOI: 10.17487/RFC8017. URL: https://www.rfc-editor.org/info/rfc8017.

[80] Trusted Computing Group. *TCG Trusted Attestation Protocol (TAP) Information Model for TPM Families 1.2 and 2.0 and DICE Family 1.0*. Version 1.0 Revision 0.36. Sept. 2019.