Trusted Platform Modules: Introduction and Shortcomings

BZA course project
April 2024

Ondřej Ondryáš
Faculty of Information Technology
Brno University of Technology
Brno, Czech Republic
Email: xondry02@stud.fit.vut.cz

***Abstract*—This article gives an introductory overview of the Trusted Platform Module (TPM), a cryptographic device that enables trust in computing platforms, allowing the determination of expected behaviour and protection of the system integrity, while providing various security-related features, such as the generation and storage of cryptographic keys or remote attestation. It presents the rationale for its development and use, showing concrete examples of its deployment. A brief example of software implementation utilising the TPM is also shown. In the end, it discusses various problems in the use of TPMs, attack vectors, and showcases some of the previously discovered vulnerabilities in TPMs.**

I. Trusted computing

Trusted computing is one of the many concepts in cybersecurity focusing on the protection of data and systems against external threats. It involves integrating security mechanisms within computing platforms, enabling them to act reliably and predictably using built-in trust elements. These mechanisms allow users to assess the trustworthiness of a computing platform based on its ability to perform expected functions securely and consistently. Trusted computing aims not only to secure systems from attacks but also to ensure integrity in them, that is, to provide means of assertion that they operate exactly as intended [1].

The concept of *trust* in trusted computing is often defined by the predictability and verifiability of a platform's behaviour. Trust is established through mechanisms able to demonstrate that a platform acts in expected ways [1], [2]. For example, a security policy may require that a device boots securely, only executes software that has been approved and has not been compromised, or that stored data is protected against unauthorised access. A trusted environment must be able to guarantee that such a policy is fulfilled.

An important term is *Trusted Computing Base* (TCB). It refers to a collection of all the system resources (hardware and software) that are responsible for maintaining its security policy [2]. The TCB is basically the core set of elements that must be trusted in order to provide security [3]. Any component that is part of the TCB must be protected from unauthorised modification or interference, as any breach of the TCB could compromise the security of the whole system. A TCB must be able to prevent itself from being compromised by any hardware or software that is not part of the TCB [2].

The Trusted Computing Group (TCG) is an international non-profit organisation formed in 2003 by major technology companies to develop, define, and promote open standards for secure computing [4]. One of the group's key contributions has been the development of the *Trusted Platform Module* (TPM), a component fundamental to the establishment of a TCB, allowing to determine if the TCB has been compromised [2]. The specification was also made into the ISO/IEC standard 11889:2015.

A. Trust and trustworthiness

It is important to note the difference between trust and trustworthiness. [2] explicitly says that trust is "meant to convey an expectation of behaviour". Then it adds that predictability in behaviour does not necessarily imply that the behaviour is worthy of trust. In contrast, [1] argues, perhaps in a more general sense, that "something is trustworthy if its behaviour is predictable". Looking at other sources, NIST uses several definitions of trustworthiness, one of them being [5]: "worthy of being trusted to fulfil whatever critical requirements may be needed for a particular component, subsystem, system, [...], or other entity." Although there is no single exact definition of trustworthiness, one can argue that it is not an inherent property of a system but rather an attribute given to a system by the entity that uses it, based on various aspects.

II. The TPM

The TPM specification [2] defines a physical component separated from the host system that aids in securing machines in several ways. It provides a set of features to the platform (e.g. a PC) that help it establish a TCB, and, by extension, the security of the platform. They ensure that critical security operations are isolated from the main CPU, making it significantly harder for adversaries to manipulate or bypass security mechanisms. TPMs operate on the principle of providing a hardware *root of trust*: by reposing one's trust in the TPM and its vendor, one can transitively trust other functions of the system, given that the TPM enforces a certain trust policy before executing them [2].

The basic security functions provided by the TPM are [3], [6]:

- generating quality random numbers,
- device authentication: a TPM can be used for secure device authentication using its unique per-device RSA or ECC key,
- device attestation: a TPM can ensure platform integrity by taking and storing security measurements of software state,
- generating, storing, and limiting the use of cryptographic keys:
  - key (secret) binding/wrapping: generated keys may be encrypted (wrapped) using the TPM's secret key that never leaves the TPM, effectively binding the new key to that TPM,
  - key (secret) sealing: secrets can be sealed with a specific device state and only when a device satisfies the sealed state condition, a TPM allows data decryption,
- encryption and decryption or digital signing.

By handling cryptographic operations directly within the TPM, the risk of sensitive data being exposed to malware is significantly reduced.

TPMs allow for remote attestation: they can create a cryptographically secure (signed) report of the software state of a system, including a report of the boot process, which can then be sent to a remote server to prove the system is running known and trusted software before accessing resources. This is particularly useful in scenarios that require verification of device integrity in a networked environment, such as when connecting device to a protected network.

A. Realisations of the TPM

There are several ways the TPM specification may be implemented. The main ones are [7], [8]:

- discrete TPM (dTPM) is implemented as a standalone hardware module and typically provides the highest level of security using tamper resistance;
- integrated TPM (iTPM) is a hardware chip integrated into the package of another chip such as the CPU – this provides similar level of security but usually not tamper resistance;
- firmware TPM (fTPM) is a software-based implementation running in a protected (trusted) execution environment of the main CPU (such as AMD TrustZone or Intel SGX) – while this can largely improve performance, it makes the TPM dependent on many additional aspects, hence extending the attack surface.

Additionally, software TPMs (that is, TPM emulators) are used for testing and prototyping. In cloud computing, virtual TPMs may be used. For example, Microsoft provides vTPMs for its Azure confidental VMs: the virtual machines can access a virtual device that behaves according to the TPM specification. In turn, the hypervisor uses hardware features, AMD Secure Encrypted Virtualization and Secure Memory Encryption, to ensure that the vTPM instance is isolated both from the host and from all the other VMs [9]. Attempts have been made to create hybrid solutions to compensate the low performance or low agility in fixing vulnerability problems [6].

B. Examples of usage

One common way of using the TPM is through cryptographic service providers. These are basically abstraction layers that provide a common interface for cryptographic operations to applications. Their significant application is hardware-aided key management. PKCS #11 is a widely known platform-independent API for cryptographic tokens [10]; and a TPM implementation is available[1]. This allows for broader compatibility with existing applications; however, it only provides a small subset of the TPM's capabilities.

TPMs are extensively used to facilitate full-disk encryption, such as Microsoft's BitLocker [11]. BitLocker binds the encryption to the hardware of the device. The TPM stores the encryption keys in a way that prevents them from being exported, ensuring that the keys cannot be easily compromised or copied. The encryption process begins by generating a full volume encryption key, which is itself encrypted by a volume master key and stored in an unencrypted portion of the disk. To unlock the drive, the integrity of the system is first verified by the TPM based on measurements of the software and firmware that were taken when BitLocker was activated. If the integrity checks are met, the TPM releases the BitLocker keys to the system, allowing the operating system to start and the drive to be decrypted during use.

The TPM's integration into Windows extends to other modern security solutions like Windows Hello for Business and virtual smart cards [12]. Windows Hello for Business replaces traditional password-based authentication with stronger two-factor authentication methods, utilising a TPM to protect the cryptographic keys associated with user credentials. To some extend, the TPM can be used as a virtual smart card, mimicking the security benefits of physical smart cards with the convenience of being implemented in the user's computer. This feature leverages the TPM's dictionary attack protection to secure the private keys and certificates used for user authentication on Windows devices.

Measured Boot and, by extension, remote attestation are also enabled by TPM in Windows [12]. Measured Boot records a comprehensive log of the boot process into the TPM, which includes verifying the integrity of each component of the boot sequence. This process ensures that any anomalies or tampering attempts are detected and recorded, providing a reliable basis for assessing the trustworthiness of the system at start-up.

C. History

The first widely employed version was TPM 1.1b, released in 2003. It introduced the core concepts such as the Platform Configuration Registers used to maintain the integrity of a boot

[1] https://github.com/tpm2-software/tpm2-pkcs11

sequence. It provided RSA key generation, secure storage, and device health attestation. However, TPM 1.1b was limited by hardware incompatibilities and lacked robust defences against both logical and physical attacks. Most importantly, there was no protection against dictionary attacks [3].

TPM 1.2 was first announced in 2004 [13] and was continuously developed until 2009 [3]. It addressed many of these shortcomings by standardising interfaces and introducing protections against dictionary attacks. It also added new features such as direct anonymous attestation to help privacy concerns. Additionally, TPM 1.2 newly incorporated nonvolatile RAM to preserve certificates critical for establising the device identity by the vendor. It also introduced Certified Migratable Keys to allow secure key migration under specific conditions. This version also integrated a timestamp mechanism for signing operations to enhance transaction security [3].

A major flaw in the 1.x family was that it had been designed around a fixed set of algorithms. This became clear around 2005 after the first significant attack on SHA-1 had been published. The TCG began to work on a complete overhaul of the specification, introducing flexible structures capable of using various cryptographic algorithms. TPM 2.0 gave the architects the opportunity to discard the original design with its flaws and start over. Thus, a new generation was developed, introducing complete algorithm independence, as well as a new, unified, flexible but complex authorisation framework [3].

III. Architecture and features

The TPM 2.0 is designed to address these issues [3]:

- identification of devices,
- secure generation of keys (provided by a hardware RNG),
- secure storage (and cahging) of keys,
- maintaining a non-volatile store of secrets and certificates (even when the main disk is wiped),
- device health attestation (cryptographically secure, i.e., a compromised system cannot report it is healthy),
- algorithm agility (ability to change algorithms without revisiting the specification),
- unified authorization,
- quick key loading,
- flexible management of resources in the TPM.

A TPM *entity* is an item that can be directly referenced (using a handle). The term *object* is also used, but it refers to a specific subset of entities: keys and data [2].

Hierarchies

TPM is structured around several hierarchies, which are logical structures that help organise and control access to keys and other entities. The most notable hierarchies are the Storage Hierarchy, the Endorsement Hierarchy, and the Platform Hierarchy. Each of these hierarchies serves different purposes, ranging from secure storage of encryption keys to platform integrity verification.

PCRs

Platform Configuration Registers (PCRs) are special-purpose registers in TPM that capture cryptographic hashes of system and software state to ensure the integrity of the platform. These registers are updated and extended with new measurements as the system boots and software components are loaded, making them essential for the TPM's integrity checking processes [2], [3].

PCRs are designed to be tamper-evident; they do not allow the stored value to be directly overwritten but instead use a specific method to update their content called the *extend* operation. This operation is a fundamental aspect of how TPM ensures system integrity [3].

The input to the extend operation is a value (usually a digest) which represents a measurement of a system component or configuration. This measurement could be the hash of a bootloader, kernel, or even application binaries and configuration files.

The existing value in the PCR is concatenated with the new hash value. This result is then hashed again:

$$PCR_{new} = \text{Hash}(PCR_{old} \mid\mid value)$$

The extend operation is used primarily during the boot process to measure and record the integrity of each component as it is loaded. For example, as the BIOS starts, it measures itself and extends this measurement into PCR 0. When the bootloader is loaded, the BIOS measures the bootloader code before executing it and extends this measurement into PCR 1.

This process is repeated for various system components. This recorded sequence of hash values in the PCRs provides a verifiable chain of trust. Any alteration in the software or firmware components would result in a different hash value, altering the final PCR value. The TPM is able to output a signed vector of PCR values called an *attestation* or *quote*. PCRs can also be used in an authorisation policy to restrict the use of other objects; for example, a key may be configured so that it can only be accessed if the system is in an expected state [3].

A. NVRAM

Non-volatile (NVRAM) indexes are another critical component of TPMs. They are essentially secure storage areas within the TPM that hold data, such as configuration settings, which need to persist securely across system reboots or changes (such as disk replacement). One use of NVRAM is for architecturally defined data or fields defined in the TPM specification. This includes authorisation values, seeds and proofs, counters, a clock, etc. The NV memory can hold structured data that have been made persistent, such as a key, and even user-defined unstructured data. Access to these indexes can be controlled through authorisation mechanisms [3].

B. Keys

The TPM can both generate and import externally generated keys. It supports both asymmetric and symmetric keys, signing, decryption or storage. Each key has individual security

controls, which can include a password, an authorisation policy, restrictions on duplication, and limits on its use. Keys can be both certified and used to certify other keys. Keys are always placed into one of the three key hierarchies (listed above) under the control of different security roles, and each can form trees of keys [3].

A TPM is a memory-constrained device so it cannot hold many keys in its memory (a typical TPM may have 5–10 key slots [3]). For most keys, it acts rather as a cache: when a key is creates, it is wrapped with (encrypted by) a parent key. This is returned to the caller, which is responsible for storing the key. When the TPM is to use the key, the application must load it into the TPM [3]. Note that the caller cannot access the actual key as it is encrypted by a secret that the TPM will never give expose.

IV. Using the TPM

The TPM is a passive device where all communication is initiated by the host device. The request-response protocol, the data structures used, and their binary encoding are explained in great detail in the TPM specification [2], which is actually designed so that C header files can be generated from it [3]. The specification also captures the workings of each command using C code, so there is no ambiguity as to how a TPM should operate [3].

Although one could manually craft the TPM command bytestream and parse the responses, it is clear that such an approach would not be viable for the developers. There are several ways of using the TPM from applications. The baseline approach is to use the *TPM Software Stack* [14], another standard published by the TCG that includes various levels of abstraction.

A prominent, community-driven open-source C implementation of the stack specification is available under the name of *tpm2-software*[2]. In addition to the TSS implementation called *tpm2-tss*, this community also provides command-line tools and other interfaces and supplementary software.

Other implementations of TPM stacks are available. IBM develops its own *ibmtss* library[3], though it is not API-compatible with TCG TSS. Microsoft has its *TSS.MSR* stack[4] available for .NET, C++, Java, JavaScript and Python; however, at the time of writing, the project does not seem particularly maintained.

The stack is shown in Figure 1. At the bottom, there is the device driver that is used to send and receive data from the actual device. Data is provided by the Access Broker and Resource Manager. The AB is used to synchronize multi-process access to a single TPM instance. The RM could be likened to a virtual memory manager in an OS: Because TPMs have limited memory and cannot store all the objects that applications work with, the RM swaps TPM objects between the host's memory and the TPM. It has to keep track of the resources and their identifiers, translating between virtual and physical object handles. The TAB and RM do not have to be a single component, but this is the usual implementation [3]. The RM does not have to be used, applications can manage the contexts manually [14]. For example, on Linux, a TPM is normally visible as two block devices: `/dev/tpm0` can be used for direct access, while `/dev/tpmrm0` represents the TPM managed by a RM.

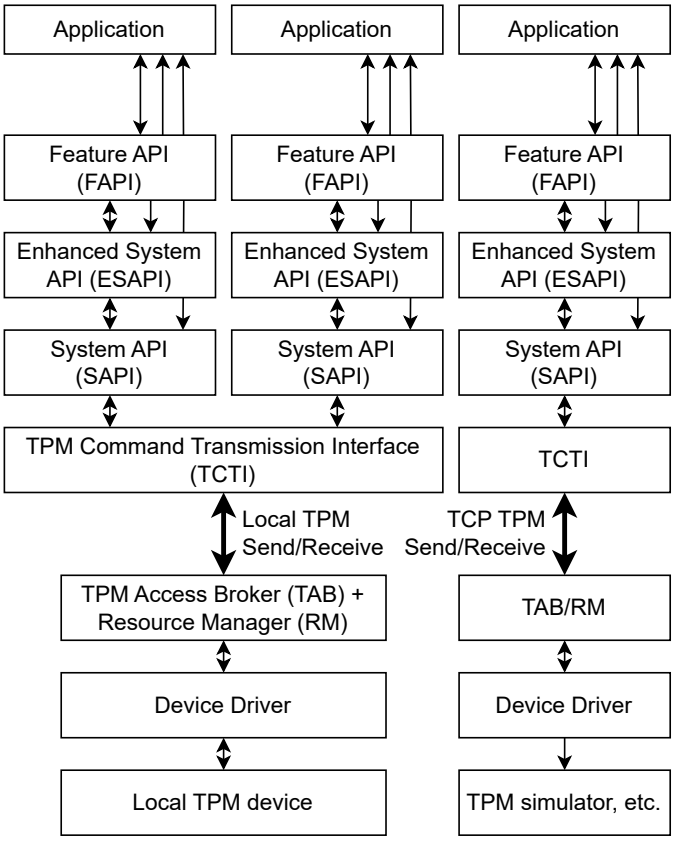

Fig. 1. The TPM Software Stack, designed to isolate TPM application programmers from the low level details of interfacing to the TPM [14].

The TCTI handles communication to and from the lower layers of the stack. For example, different interfaces are required for local HW TPMs, firmware TPMs, or virtual TPMs. The System API is a layer that provides low-level access to all the functionality of a TPM. This interface is aimed at expert applications. The Enhanced System API sits directly above the SAPI, providing easier access to cryptographic functions, context, and object management. It is significantly less complex than the SAPI but still requires a deep understanding of TPMs [14].

The Feature API provides a higher level abstraction. It was designed with the hope that 80% programs that use TPMs could be written with it. Its use requires cooperation with the user and the OS: it uses configuration files – profiles to define the set of algorithms and other details. Keys are exposed with a reduced set of attributes. Other limitations are listed in the FAPI specification [15].

[2] https://tpm2-software.github.io/
[3] https://github.com/kgoldman/ibmtss
[4] https://github.com/microsoft/TSS.MSR

A. Examples

The examples in Listings 1 and 2 show how the GetRandom function of the TPM can be used using the Enhanced System API and the Feature API. This trivial functionality was chosen to demonstrate some of the differences between the APIs.

Starting from the top: an ESAPI context may be initialised using pointers to existing transmission interface pointers; in FAPI, this is controlled by the profile. However, FAPI must always be provisioned before usage – this process configures certain keys and policies for use with FAPI and must be done once (typically per-user).

The call to the GetRandom command follows. In the ESAPI code, the command accepts three "session handles". Each TPM command can use up to three session handles, used to identify a session (used to maintain state and authorisations between commands). The RNG command may accept a session in certain cases, although none is used here. In FAPI, the user cannot manage TPM sessions as they are handled internally by the layer. Also note that FAPI uses the standard `uint8_t` type, whereas ESAPI uses a custom typedef for the output buffer. The cleanup process is similar in both examples.

```
ESYS_CONTEXT *context = NULL;
TSS2_RC result = Esys_Initialize(&context,
    NULL /* ptr to existing TCTI context */,
    NULL /* specification of ABI version */);

TPM2B_DIGEST *bytes = NULL;
result = Esys_GetRandom(context,
    ESYS_TR_NONE /* unused session handle */,
    ESYS_TR_NONE /* unused session handle */,
    ESYS_TR_NONE /* unused session handle */,
    8 /* bytes requested */, &bytes);

// Print
for (size_t i = 0; i < bytes->size; i++)
    printf("%02x", bytes->buffer[i]);

// Free heap memory allocated by Esys
Esys_Free(bytes);
// Closes the context allocated
// by Esys_Initialize
Esys_Finalize(&context);
```

Listing 1: Generating random numbers in C using ESAPI. Error handling is omitted but should be used after each Esys command. Detailed parameter descriptions are available in Chapters 6.1, 6.7 and 6.2 of [16] and in Chapter 16.1 of [17].

Listing 3 shows an example of generation and usage of a password-protected signing key. Observe that virtually no key parameters (such as algorithm type) are passed in the function: this is because they are controlled by the current profile. Also note the use of string paths identifying the key. FAPI uses custom path references to objects [15]:

```
<Profile name> / <Hierarchy> / (<Parent
        object> /)* <Object>
```

In this case, profile is omitted (the user's default is used), the object is stored in the Storage hierarchy (HS), it's parent is the Storage Primary Key (SRK) and the key is called MyKey.

```
FAPI_CONTEXT *context = NULL;
TSS2_RC result = Fapi_Initialize(&context,
    NULL /* must be null */);

// One-time provisioning is necessary with FAPI
// but may (should) be done externally
result = Fapi_Provision(context,
    // Authentication values
    NULL, NULL, NULL);

uint8_t *bytes = NULL;
result = Fapi_GetRandom(context,
    8 /* bytes requested */, &bytes);

// Do sth. with bytes
// Free the allocated byte array and the context
Fapi_Free(bytes);
Fapi_Finalize(&context);
```

Listing 2: Generating random numbers in C using the Feature API. Error handling is omitted but should be used after each Fapi command. Detailed parameter descriptions are available in Chapters 5.1, 5.3, 3.4 and 4.2 of [15].

```
uint8_t digestToSign[32] = {0xde, ..., 0xef};
uint8_t *signature; size_t signatureSize = 0;
FAPI_CONTEXT *context = NULL;
TSS2_RC res = Fapi_Initialize(&context, NULL);

res = Fapi_CreateKey(context,
    /* path to the key in hierarchy */
    "/H_S/SRK/MyKey",
    /* signing key; no dict attack protection */
    "sign,noDa",
    NULL /* policy path */,
    "mypassword" /* authentication value */);

res = Fapi_Sign(context, "/HS/SRK/MyKey",
    NULL /* default padding */,
    digestToSign, sizeof(digestToSign),
    &signature, &signatureSize,
    NULL, NULL /* output cert. and pubkey */);

// Cleanup
```

Listing 3: An example of generating a key and signing a digest using the Feature API. Error handling is again omitted. Detailed parameter descriptions are available in Chapters 6.1 and 6.2 of [15].

See that when creating the signature, one does not have to authenticate with the password associated with the previously created key. This shows the internal session management of FAPI. A complete example of achieving the same behaviour in ESAPI would take up at least one typeset column of code.

B. Usability

In spite of attempts to ease the usage of TPMs, e.g, by introducing the very abstract Feature API, it is still surprisingly difficult to use the TPM in real-world applications. Even though a TPM is present in virtually every modern PC [8], its adoption in software that could effectively take advantage of this technology seems low [3]. The issue seems to lie in the overwhelming complexity of the specifications and features.

[3] notes that one reader of the specification has claimed that TPM provides "security through incomprehensibility." Al-

though the specification attempts to describe the functionality as clearly as possible, it is focused on how a TPM should work, not how it should be used. Furthermore, the information is scattered across four parts of the TPM specification and up to eight documents that make up the TSS specification.

A qualitative case study by Rao et al. from 2022 [18] also concludes that the TPM library APIs are not developer-friendly. They identified areas where the APIs contain usability and even security pitfalls. They present several recommendations to the library API developers, such as:

- include background information,
- provide code snippets for common use cases,
- improve entry-level documentation,
- provide developer-friendly error messages and concise output messages,
- promote secure cryptographic primitives,
- and more.

Not only do the discovered issues show that the state of the TPM ecosystem hinders itself from further penetration of common software; it also provides ground for questioning the security of software that does use it.

V. Security problems in TPMs

TPMs are designed to enhance the security of computing platforms. However, despite their critical role in secure computing, they are not without vulnerabilities. Numerous attacks have been shown targeting concrete TPM devices, implementations (Sections VI and VII demonstrate two such attacks), or even flaws in the specification[5]. Each successful recovery of a private key highlights the critical need for cryptographic implementations in TPMs to adhere to strict operational standards that are resilient against such side-channel attacks. The discovered attacks show the fragility of assuming that TPMs, by their very nature, are immune to all forms of cryptographic exploitation.

As with all hardware security devices, TPMs can be susceptible to physical tampering and side-channel attacks, such as power analysis or EM analysis. In security-critical applications, dTPMs should be used, as they are expected to be designed in a tamper-resistant manner and protected against side-channel attacks. However, the fact that these devices are separated from the platform presents another threat: Many attacks have been tried targeting the data bus between a TPM and a CPU [8]. For example, [19] presents a serial bus interposer called TPM Genie that can undermine many of the purposes of the TPM, such as measured boot or the RNG.

Although iTPMs and fTPMs cannot be attacked in this way, they lose the tamper resistance property, making way for other class of intrusions. For example, an attack based on voltage glitching targeting AMD's secure processing abilities was shown in [20] to successfully compromise encryption keys. The authors state that TPM-based full-disk encryption may actually be less secure than TPM-less protection based only on a solid passphrase.

[5]For example, see CVE-2023-1017 and CVE-2023-1018.

A. Certified implementations

This widespread adoption of TPMs necessitated formal certification processes, such as those based on the Common Criteria and FIPS standards. Despite these standards, numerous certified TPMs, even those achieving high certification levels like CC EAL4+, have been found to be vulnerable to a variety of attacks that could lead to private key disclosure [8]. These vulnerabilities include the ROCA or TPM-Fail attack (described below).

Reflecting on these issues, it is clear that CC and FIPS certifications, while essential, do not necessarily guarantee the security of TPMs. The certification processes often focus on compliance with a set of predefined criteria that may not encompass all possible vulnerabilities or real-world attack vectors. In the long term, a re-evaluation of the certification processes, is needed. These processes must evolve to include more rigorous tests for side-channel attack resistance, Regular security audits and updates based on the latest research should become a standard part of the lifecycle management of TPMs.

VI. The TPM-Fail attack

The TPM-Fail attack described in [21] exploits vulnerabilities in the timing characteristics of the cryptographic operations conducted by TPMs that support elliptic curve cryptography, particularly in the ECDSA and ECSchnorr signature schemes. They originate in the variation in execution times during cryptographic processes, which are influenced by secret-dependent conditions. Essentially, the time it takes to generate cryptographic signatures varies based on the values of the cryptographic keys and nonces involved, introducing an exploitable side channel. These variations are sufficient to employ advanced lattice-based techniques to systematically deduce and recover 256-bit private keys. Several TPM implementations that exhibit this flaw were discovered, including dTPMs certified at Common Criteria EAL4+ and fTPMs in Intel CPUs. Furthermore, the authors showed that the attack could also be successfully performed over network, specifically targeting a remote VPN service that uses a TPM for handshake authentication.

A. Phases of the attack

The attack unfolds through a three-phase process. The first phase involves profiling the TPM by collecting detailed timing data while generating cryptographic signatures using *known* private keys. This step is critical as it establishes data on how signature generation times correlate with different nonce values (temporary values used once). During this phase, the attacker measures how long the TPM takes to sign digital messages and records this alongside the signature data. For the vulnerable TPM implementations discussed in the paper, signing a message with a nonce that had more leading zero bits showed to typically take less time.

In the second phase, the adversary collects signatures and their corresponding timing data from the TPM without prior knowledge of the private keys. The goal is to identify signature operations that exhibit certain *bias* in the used nonces. By

observing and comparing these timings with the previously collected baseline, the attacker can pinpoint operations where the execution time deviates in a manner indicative of a vulnerable nonce.

The final phase leverages lattice-based cryptanalysis to recover the private keys. This advanced mathematical technique is used to analyse the biased nonces identified in the previous phase. The attacker uses the signatures with biased nonces, combining them with the timing data, to form a lattice of possible nonce values. Its structure is based on the specific observed variations in timing, and through various computational techniques, it is possible to deduce the private key from these correlations. This step involves solving what is known as the Hidden Number Problem (HNP), where the hidden number (in this case, parts of the private key) is inferred from the information leakage caused by the biased nonces.

B. Effectiveness and impacts

The effectiveness of this recovery mechanism depends on the precision of the timing data collected and the specific targeted device. The paper shows three models: a local user with system-level privileges that can run exact measurement software; a local user measuring the responses only using userspace means; and a remote adversary who can only measure round-trip timings in remote TPM-based operations.

In the first case, the key could be successfully recovered after collecting about 10,000 signatures from Intel's fTPM, which took about 27 minutes; or about 40,000 signatures from STM's dTPM, which took about 80 minutes. In the second case, the estimated time to collect enough data was about 160 minutes. In the third case, when a remote VPN service was targeted, enough samples could be collected in a little over 5 hours.

C. Countermeasures

If the user does not have direct (kernel-level) access to the TPM, a software-based countermeasure can be used. The OS can add a delay to the TPM interface to ensure that all commands are executed in constant time. However, this requires a precise estimation of an upper bound for the execution time for these operations. This is not trivial, since the execution times vary among different TPMs. For network-based attacks, IDS systems may be used, but again they do not offer complete protection. Such solutions should be treated as a stopgap rather than a true fix, as they do not address the underlying vulnerability.

These days, this specific issue was solved by vendors by adjusting the firmware of the TPMs. However, not all devices have necessarily received the update. Even so, the existence of such vulnerability shows a class of possible attack methods that are yet to be discovered in other implementations.

VII. The ROCA attack

The Return of Coppersmith's Attack (ROCA) first described in [22] is a significant vulnerability discovered in the RSA key generation process used in devices produced by Infineon Technologies AG, a major semiconductor manufacturer and vendor of smartcards or TPMs. This flaw affects the way the RSA modulus N is constructed, leading to reduced randomness in the generation of RSA primes, making these keys susceptible to computationally feasible factorisation.

RSA keys require the generation of two large prime numbers (p and q). However, the method used by Infineon's RSA implementation led to the primes being generated in a predictable pattern, significantly reducing their randomness. Only by analysis of RSA keys exported from the manufacturer's cards and tokens, the authors managed to identify that all RSA primes have the following form:

$$p = kM + (65537^a \bmod M)$$

where M is a known primorial (product of the first n successive primes); k and a are unknown and the generated keys vary only in their values. The most important property of the keys is that the size of M is large and almost comparable to the size of the prime p. Since M is large, the sizes of k and a are small. This significantly reduces the entropy; for example, in 512-bit keys, 256-bit prime p is used and the a and k values only represent 99 bits of entropy, reducing the complexity from 2^{256} to 2^{99}.

RSA keys generated from this algorithm can be efficiently recognised with a negligible error. An RSA public key carries the public modulus N. If the modulus was generated by an Infineon device, it is of the form:

$$N = (kM + 655373^a \bmod M) \cdot (lM + 65537^b \bmod M)$$

for $a, b, k, l \in \mathbb{Z}$. That implies $N \equiv 65537^{a+b} \bmod M$, so the existence of the discrete logarithm $a + b = log_{65537} N \pmod M$ serves as a strong fingerprint of such keys. Due to other mathematical properties of N, the value of this logarithm can be found within microseconds.

The authors show how Coppersmith's algorithm, first shown in 1996 [23], can be used to factorise N. This algorithm was originally developed to factorise a modulus when high bits of one of the prime factors were known. The attack uses lattice reduction techniques, most notably the Lenstra–Lenstra–Lovász (LLL) algorithm, to achieve this goal. In the ROCA attack, the algorithm is actually used as a parametrised blackbox. The paper then gives an algorithm that transforms the original problem accordingly using the properties of N.

A. Effectiveness and impacts

The ROCA vulnerability affects the security of RSA keys across various bit lengths. The time and cost of factorising a 1024-bit was verified to be about 97 CPU days (considering a single CPU core from 2014) with costs around $40–$80, while a 2048-bit key was estimated to take about 140.8 CPU years, with costs reaching up to $40,000. These costs make it feasible, especially for high-value targets, to break the encryption in a practical time frame if significant computational resources are dedicated. Interestingly, the difficulty of factorising RSA keys does not strictly scale with key length, as some longer

keys might be easier to factorise due to the specifics of their generation process.

For TPMs, the implications of this vulnerability are equally severe. The vulnerability allows for the potential factorisation of TPM-resident keys, notably those at 2048 bits, which are used to secure critical operations including Microsoft BitLocker's encryption keys. Although the exact number of in-use TPMs with this vulnerability cannot be determined, [22] and [8] suggest that they are probably still widely present among consumer devices.

VIII. Conclusion

One of the major contributions of TPMs is their ability to bring a higher level of assurance to certain security processes both in corporate and end-user environments. For example, the use of TPMs in scenarios like disk encryption, as seen with Microsoft's BitLocker, showcases how TPMs can effectively secure data at rest by ensuring the encryption keys are bound to the hardware and not easily extractable, in a way that does not affect the user's day-to-day life.

However, the adoption and practical implementation of TPMs also face several issues. The complexity of the technology often acts as a barrier to its widespread use. Intricate details in specifications and the neccessity of a deep understanding the functionalities can deter developers and enterprises from integrating TPMs into their systems. This complexity not only affects usability but also can introduce errors in implementation, potentially creating new security vulnerabilities.

This article presented an introduction to the Trusted Platform Module ecosystem. Hopefully, it provided insight into the rationale for using the platform, as well as a starting point for developers that may want to explore how it could help their security-concerned software. Some issues faced by TPMs and their users were discussed, showing that TPMs are not immune against attacks and thus cannot be considered a miraculous solution of all our problems. The existence of various vulnerabilities highlights a critical aspect of security devices: the need for ongoing updates, patches, and security assessments.

References

[1] G. Proudler, L. Chen, and C. Dalton, *Trusted Computing Platforms*. Springer International Publishing, Jan. 2015.

[2] Trusted Computing Group, "Trusted Platform Module Library Specification, Part 1: Architecture, Family 2.0," Trusted Computing Group, Specification, Mar. 2024, Level 00, Revision 01.83. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-1-Architecture.pdf

[3] W. Arthur and D. Challener, *A practical guide to TPM 2.0*, 1st ed. New York, NY: Apress, Jan. 2015.

[4] Trusted Computing Group. (2024) About TCG. Accessed: 2024-04-02. [Online]. Available: https://trustedcomputinggroup.org/about/

[5] R. Ross, V. Pillitteri, R. Graubart, D. Bodeau, and R. McQuaid, "Developing cyber-resilient systems: A systems security engineering approach," National Institute of Standards and Technology (U.S.), Tech. Rep., Dec. 2021. [Online]. Available: http://dx.doi.org/10.6028/NIST.SP.800-160v2r1

[6] Y. Kim and E. Kim, "htpm: Hybrid implementation of trusted platform module," in *Proceedings of the 1st ACM Workshop on Workshop on Cyber-Security Arms Race*, ser. CYSARM'19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–10. [Online]. Available: https://doi.org/10.1145/3338511.3357348

[7] Trusted Computing Group. (2015) Trusted platform module TPM 2.0: A brief introduction. Accessed: 2024-04-02. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-A-Brief-Introduction.pdf

[8] P. Svenda, A. Dufka, M. Broz, R. Lacko, T. Jaros, D. Zatovic, and J. Pospisil, "TPMScan: A wide-scale study of security-relevant properties of TPM 2.0 chips," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, vol. 2024, no. 2, pp. 714–734, 03 2024. [Online]. Available: https://tches.iacr.org/index.php/TCHES/article/view/11444

[9] Microsoft. (2023, Aug.) Virtual TPMs in Azure Confidential VMs. Accessed: 2024-04-02. [Online]. Available: https://learn.microsoft.com/en-us/azure/confidential-computing/virtual-tpms-in-azure-confidential-vm

[10] OASIS, "PKCS #11 Cryptographic Token Interface Base Specification Version 2.40," OASIS, OASIS Standard, Apr. 2015, edited by Susan Gleeson and Chris Zimman. [Online]. Available: https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html

[11] Microsoft. (2023, Jul.) BitLocker countermeasures. Accessed: 2024-04-03. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/operating-system-security/data-protection/bitlocker/countermeasures

[12] ——. (2023, Nov.) How Windows uses the Trusted Platform Module. Accessed: 2024-04-03. [Online]. Available: https://learn.microsoft.com/en-us/windows/security/hardware-security/tpm/how-windows-uses-the-tpm

[13] Trusted Computing Group. (2011, February) Trusted computing group TCG timeline. Accessed: 2024-04-02. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TCG-Timeline_rev-Feb-2011.pdf

[14] ——, "TCG TSS 2.0 Overview and Common Structures Specification," Trusted Computing Group, Specification, Sep. 2021, Version 1.0, Revision 10. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf

[15] ——, "TCG Feature API (FAPI) Specification," Trusted Computing Group, Specification, Jun. 2020, Version 0.94, Revision 09. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_FAPI_v0p94_r09_pub.pdf

[16] ——, "TCG Enhanced System API (ESAPI) Specification," Trusted Computing Group, Specification, Oct. 2021, Version 1.00, Revision 14. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TSS_ESAPI_v1p0_r14_pub10012021.pdf

[17] ——, "Trusted Platform Module Library Specification, Part 3: Commands, Family 2.0," Trusted Computing Group, Specification, Jan. 2024, Level 00, Revision 01.83. [Online]. Available: https://trustedcomputinggroup.org/wp-content/uploads/TPM-2.0-1.83-Part-3-Commands.pdf

[18] S. P. Rao, G. Limonta, and J. Lindqvist, "Usability and security of trusted platform module (TPM) library APIs," in *Eighteenth Symposium on Usable Privacy and Security (SOUPS 2022)*. Boston, MA: USENIX Association, Aug. 2022, pp. 213–232. [Online]. Available: https://www.usenix.org/conference/soups2022/presentation/rao

[19] J. Boone, "TPM genie: Inteposer attacks against the Trusted Platform Module serial bus," NCC Group, Mar. 2018. [Online]. Available: https://github.com/nccgroup/TPMGenie/blob/master/docs/NCC_Group_Jeremy_Boone_TPM_Genie_Whitepaper.pdf

[20] H. N. Jacob, C. Werling, R. Buhren, and J.-P. Seifert, "faultpm: Exposing amd ftpms' deepest secrets," 2023.

[21] D. Moghimi, B. Sunar, T. Eisenbarth, and N. Heninger, "TPM-FAIL: TPM meets timing and lattice attacks," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2057–2073. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/moghimi-tpm

[22] M. Nemec, M. Sys, P. Svenda, D. Klinec, and V. Matyas, "The Return of Coppersmith's Attack: Practical Factorization of Widely Used RSA Moduli," in *24th ACM Conference on Computer and Communications Security (CCS'2017)*. ACM, 2017, pp. 1631–1648.

[23] D. Coppersmith, "Finding a small root of a bivariate integer equation; factoring with high bits known," in *Advances in Cryptology — EUROCRYPT '96*, U. Maurer, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 178–189.