POLITECNICO DI TORINO

Master's Degree in Cybersecurity

Master's Degree Thesis

Enhancing RISC-V Security: Implementing TPM Functionality in a gem5 Simulation environment

Supervisors

Prof. Alessandro SAVINO

Prof. Stefano DI CARLO

Prof. Sadia SHAMAS

Candidate

Rosanna LANDI

Abstract

Security in RISC-V architectures requires hardware solutions that can guarantee reliability, integrity, and data protection. In this scenario Trusted Platform Module (TPM) and Physical Unclonable Functions (PUFs) play a primary role in the context of hardware security. Trusted Platform Module (TPM) is a dedicated security chip consolidated as building blocks to obtain a trusted root point, offering a standardized set of features that cover cryptographic key generation and protection, platform integrity measurements, and remote attestation. PUFs are unique cryptographic primitives that exploit intrinsic variations in the physical characteristics of circuits to generate answers that are distinct and cannot be replicated. The objective of this thesis is to develop a simulation TPM model for RISC-V architecture integrated into gem5. The research was carried out in two phases: first, the implementation of key functionalities of the TPM 2.0 model according to the TCG specifications; second, the integration of PUFs into the TPM. In particular, the TPM has been extended to manage the forwarding phase of a challenge towards the PUF, which produces a response based on its intrinsic characteristics. The results of the simulations show that the developed framework allows faithful reproduction of the operations of the TPM and can be used to explore the benefits of combining it with a PUF in RISC-V environment, increasing the security by reducing the dependence on static secrets. Moreover, the choice of gem5 as the basic platform makes the tool accessible and flexible, allowing for experiments without the need for dedicated physical hardware. The primary contribution of this work is the creation of an open-source simulation environment for the TPM model on RISC-V. Future prospects include extending TPM 2.0 specifications, testing distributed authentication protocols in cloud and IoT contexts, and analyzing model resilience against advanced attacks.

Acknowledgements

Un ringraziamento speciale al mio Coordinatore, il Prof. Alessandro Savino, la cui costante disponibilità è stata fondamentale per la realizzazione di questo lavoro.
A mio padre che mi ha insegnato a sognare e ad ottenere con fermezza, determinazione e orgoglio ciò che il mio cuore desidera. Sei il mio esempio e la mia ancora, sempre.
A mia madre, che crede in me, gioisce per me, e sa sempre essere fiera di me, più di chiunque altro. Sei la mia forza.
Alle mie sorelle, non sapete quante volte io mi sia nutrita del vostro coraggio, che tutti hanno scambiato per mio. Perché tutte le volte in cui ridiamo insieme, mi fa credere che nella vita non sarò mai sola. Siete la mia casa.
Un ringraziamento particolare va ai miei nonni, pilastri della mia vita, per l'amore incondizionato, il sostegno instancabile e gli insegnamenti che mi hanno reso la persona che sono oggi.
A Manuel, alla mia metà, a te che dai nome ad ogni mio sforzo: futuro. Ti ringrazio per la pazienza con cui hai atteso questo traguardo insieme a me.
Alla mia famiglia, e alle persone che mi vogliono bene e credono in me. Grazie per avermi amato incondizionatamente, a prescindere da questo traguardo.
Ai miei veri amici. Non siamo fatti per stare da soli ma nemmeno con chiunque. Siete la dimostrazione che l'amicizia è credere l'uno nell'altro senza bisogno di parole. Grazie per le risate, che hanno reso onore ad ogni singola ora di studio.

Table of Contents

List of Tables

List of Figures

Acronyms

TPM
Trusted Platform Module

PUF
Physical Unclonable Function

SRAM
Static Random Access Memory

FS
Full System

SE
System Emulation

FIFO
First In First Out

TIS
TPM Interface Specification standard

CBR
Command Response Buffer

PCR
Platform Configuration Registers

TCG
Trusted Computing Group

TCB

Trusted Computing Base

RoT

Root of Trust

CRTM

Core Root of Trust for Measurement

RTM

Root of Trust for Measurement

RTS

Root of Trast for storage

RTR

Root of Trust for Report

HMAC

Hash-based Message Authentication Code

IP

Intellectual Properties

IoT

Internet of Things

Chapter 1

Introduction

In the context of modern computing architectures, hardware security has become an essential element in ensuring the reliability, integrity, and protection of sensitive data. The RISC-V architecture, thanks to its open-source and modular Instruction Set Architecture (ISA), has fostered a strong innovative impulse, finding rapid diffusion in areas such as the Internet of Things (IoT) and embedded systems.
In particular, in these scenarios — characterized by physically exposed devices and limited computational resources — software security alone is not sufficient. Therefore, it's essential to establish a root of trust at the hardware level. However, unlike proprietary ISAs like ARM or x86, the RISC-V ecosystem does not natively offer standardized hardware security mechanisms that serve as a trusted foundation. This lack can introduce potential vulnerabilities, demanding the development of dedicated solutions to ensure system integrity from the very early stages of startup.

To address this need, the work focuses on the integration of two fundamental technologies. the Trusted Platform Module (TPM) and Physical Unclonable Functions (PUF). The TPM is defined by the Trusted Computing Group (TCG) and designed to provide cryptographic and hardware-based protection, isolating security operations from the rest of the system. Key features provided by TPM 2.0 include:

- Protected Cryptographic Key Generation and Storage.
- Device authentication using a single key (e.g. RSA).
- Measurement of Platform Integrity (Integrity Measurement) during the boot process using special registers called PCRs (Platform Configuration Registers).

Thanks to these features, TPM is the basis for implementing secure boot and remote attestation, providing cryptographic proof of system health.

PUFs are hardware cryptographic primitives that exploit the intrinsic physical variability of integrated circuits, resulting from microscopic manufacturing differences, to generate unique and non-replicable secrets. A PUF acts as an "unclonable physical function": given an input stress (challenge), it produces a unique output fingerprint (response) for that specific circuit. This "silicon fingerprint" can be converted into a cryptographic key of the device without the need to store it in non-volatile memory. The key is regenerated on the fly whenever needed, eliminating the risk associated with storing static secrets.

By combining the potential of TPM and PUFs, it is possible to design hardware security solutions more robust than using each technology individually. On the one hand, TPM provides a consolidated and standardized set of features for the creation, management and safe use of cryptographic keys, as well as for system integrity verification and outward attestation. On the other hand, the TPM can exploit PUFs to generate features-dependent secrets that are intrinsically unique to each device. In this way, the TPM could delegate the generation of cryptographic keys or authentication tokens to the PUF, avoiding having to rely on immutable secret saved in memory. The union of TPM and PUF aims to increase the overall security of the system, reducing the dependence on static secrets and making it harder for an attacker to clone or compromise the hardware root of trust.

Currently, there is a lack of a unified open-source simulation platform that allows studying and testing the integration of a TPM with PUFs on RISC-V architecture. Although there are software emulators for TPM or hardware implementations of PUF on FPGAs, there is no integrated framework available for the combined analysis of these components in a full RISC-V environment.
This framework would be extremely useful both for the research community (which could evaluate new security ideas without resorting to expensive dedicated hardware) and for educational and experimental purposes. Furthermore, with the growing adoption of RISC-V in areas such as the Internet of Things (IoT) and edge or cloud systems, it becomes strategic to have trusted computing mechanisms adaptable to these open contexts. An environment flexible and accessible simulation would allow to explore, for example, secure protocols boot and remote attestation in RISC-V-based IoT sensor networks, or to evaluate the use of TPM in virtual RISC-V cloud infrastructures, all without proprietary hardware constraints.

To address these challenges, gem5 was chosen as a development platform and simulation. Gem5 is an established modular, open-source and fullsystem-level architectural simulator, widely used in both the academic and industrial worlds for research on the architecture of computing systems. It supports several ISAs (including RISC-V) and allows to model custom hardware components within

complete systems, offering a balance between accuracy (at the cycle or instruction level) and configuration flexibility. The choice of gem5 also ensures high reproducibility of experiments: being open-source software, the developed model can be shared with the scientific community, which can replicate the tests, verify the results and possibly further extend the platform. In summary, gem5 provides the ideal foundation for building a simulated environment that faithfully reproduces behavior of a RISC-V system equipped with TPM and PUF, allowing interactions to be observed in detail hardware-software in security scenarios.

The present thesis is therefore divided into two main phases of work. In the first phase, it was designed and implemented a model of TPM 2.0 compliant with TCG specifications, integrating it into the RISC-V architecture simulated on gem5. This involved the development of the fundamental functions of the TPM according to the standard: authentication mechanisms have been implemented (both with password and HMAC, according to the authorization policies envisaged by TPM 2.0), the interfaces of FIFO and CRB (Command Response Buffer) communication for sending and receiving commands, the registers of configuration and integrity measurement (the PCRs, Platform Configuration Registers), as well as the management of TPM internal non-volatile memory (NV memory) for secure data storage persistent such as activation keys, monotonic counters, policies, etc. This stage required a deepening analysis of the TCG-defined TPM architecture and careful integration into the simulator: the TPM is implemented as a peripheral of the simulated RISC-V system, interfaced with the processor via a bus (as a memory-mapped device). The second phase of the work consists in the integration of a PUF module into the TPM. At this stage, the TPM was extended so that it could interact with the simulated PUF. In practice, a forwarding procedure has been implemented: TPM sends a challenge to the PUF module, which calculates the response based on its unique physical properties and returns it to the TPM. This integration also involved defining an interface between TPM and PUF in the context of gem5: it has been assumed that the PUF is mapped as a device to which the TPM can send commands.

The main outcome of this study is the creation of an open-source simulation environment (based on gem5) combining TPM and PUF over RISC-V architecture. This is, as far as is known, the first framework of this kind available to the community. It allows to faithfully reproduce the behaviour of a real TPM, however enriched by the presence of a PUF, to be able to experience the benefits of the combination of TPM with PUF. In addition to reproducing known scenarios, the environment of simulation also facilitates the exploration of new ideas: thanks to the flexible nature of gem5, they could implement and test various types of PUFs (e.g. PUF arbiter, ring oscillator PUF, PUF based on SRAMs, etc.)

within the same TPM context to compare their behaviour, or evaluate the impact of parameters such as challenge/response length on safety and security performance.

The structure of the thesis reflects these objectives: after this general introduction to the problem (chapter 1), chapter 2 presents the context theoretical and the state of the art in hardware security, describing RISC-V architectures, principles of trusted computing, the operation of TPMs and concepts related to PUFs, as well as an analysis of existing solutions and related research. Chapter 3 details the design and Implementation of the TPM 2.0 model in gem5: the internal architecture of the TPM and its integration with PUF is described, the design choices made, and the features supported, with particular attention to adherence to the TCG standard. Finally, in Chapter 5, the results obtained are discussed and analyzed. The thesis concludes with the Conclusions where the benefits introduced by the TPM model (e.g. in terms of increased security and flexibility) are explored, highlight the current limitations of the solution and outline future prospects and possible developments of work.

Chapter 2

State of Art

2.1 Trusted Platform Module (TPM)

2.1.1 Definition and Historical Context

The rapid expansion of computer systems connected to the network expanded the attack surface, introducing crucial challenges for data reliability and confidentiality. In this scenario, the introduction of persistent malware and threats operating at a lower level than the operating system represent a critical vulnerability. The compromise of a device's firmware, for example, allows malicious actors to achieve deep persistence that is difficult to detect by software tools, jeopardizing the integrity and confidentiality of information from the initial stages of booting. Such type of threats have made it clear that software countermeasures alone are insufficient. As a result, protecting data integrity and confidentiality has become a necessary requirement which cannot be ensured software-type solutions only. The need has emerged to introduce hardware-level security mechanisms that can act as Root of Trusts, developing robust cryptographic and authentication capabilities that operate independently of the operating system and application processes.

The first practical implementations of computer security based on hardware components date back to the '60s, with the Multics system (1964) often cited as a pioneering example. Within CPUs each program runs at a protection layer (Protection Ring) controlled by Hardware and which has a predefined set of privileges, in which programs can only execute a certain set of machine instructions [1]. These ideas led to the definition of Trusted Computing Base (TCB), which is the set of all hardware, firmware, and software components critical for the security of a system. TCB represents the "trusted" basis of calculation on which security rests: if the basic components of the hardware or software were compromised, the entire system would be affected [2]. On the base of these principles, at the end of the

'90s, the leading IT companies created an initiative called Trusted Computing, focused on the introduction of security mechanisms at the hardware level. In 1999, the Trusted Computing Platform Alliance (TCPA) consortium was founded, then evolved in 2003 into the current Trusted Computing Group (TCG), with the aim of standardizing hardware security solutions in personal computers [3]. In this context, the TPM was born, a dedicated security chip that implements cryptographic and integrity attestation capabilities at the platform level, providing a hardware anchor for trust in the system.

According to Trusted Computing Group, *"TPM (Trusted Platform Module) is a computer chip (microcontroller) that can securely store artifacts used to authenticate the platform (your PC or laptop)*" [4, p. 1]. This dedicated security chip plays a crucial role in safeguarding a system by enabling platform integrity measurements, generating and securely storing cryptographic keys, and acting as a root of trust. It ensures system security from a hardware perspective without depending on the operating system or applications. The TPM is defined and standardized by the Trusted Computing Group (TCG). As a root of trust, the TPM provides a reliable foundation for higher-level security services, including verifying if the system is running in a trusted state, and ensuring an isolated environment for cryptographic operations. The design of TPM includes several main parts: a random number generator and engines for cryptographic operations using both symmetric and asymmetric algorithms. It also has secure non-volatile memory for storing keys and configuration registers, known as Platform Configuration Registers (PCRs). With these components, the TPM enables secure boot, disk encryption (such as BitLocker), and remote attestation.

The earliest version, TPM 1.1 (2003), introduces basic features like key generation and saled storage. TPM 1.2 (2005) achieves the strongest cryptographic capabilities by supporting RSA and SHA-1, with this version, there has been a greater diffusion of the use of TPM in enterprise PCs. A fundamental change occurred with the introduction of TPM 2.0 in 2014, which supports a greater number of hash and asymmetric algorithms (such as elliptic curve cryptography and SHA-256) along with support for the use of symmetric ciphers; amplify the authorization methods allowing authorization with clear-text passwords and Hash Message Authentication Code (HMAC) and targeted a wider range of devices, including mobile platforms and embedded devices. In 2021, Microsoft made the introduction of TPM mandatory in Windows 11 ensures protection against malware and firmware attacks. Nowadays over 2 billion devices integrate TPM. [4] [5]

2.1.2 Root of Trust

According to TCG "a Root of Trust (RoT) is a component that performs one or more security-specific functions, such as measurement, storage, reporting, verification, and/or update. A RoT is trusted always to behave in the expected manner, because its misbehavior cannot be detected (such as by measurement) by attestation or observation"[6, p. 9].
The TCG requires three Root of Trust in a platform to define it as trusted [7]:

- Root of Trust for Measurement (RTM) is the initial element in the chain of trust. First, the Core Root of Trust for Measurement (CRTM) initiates the process by measuring the code to be loaded. Then, the RTM computes its hash and saves it in a secure storage such as Platform Configuration Register (PCRs). Finally, this process spreads the chain of trust, with each component measuring the next, ensuring a reliable sequence of trusted components. This role is usually performed by the CPU, which is controlled by the CRTM. The CRTM is the first set of instructions to execute during boot and it represents the starting point from which the chain of trust takes shape. Starting from CRTM, each component measures the following component, gradually spreading the chain of trust.

- Root of Trust for Storage (RTS) ensures the secure storage of sensitive data, such as cryptographic keys, hashes, passwords, and certificates. The TPM can act as an RTS, ensuring that sensitive assets and data can be accessed only by trusted hardware and software.

- Root of Trust for Reporting (RTR) generates cryptographic attestations by creating a digitally signed hash of the specific PCR values stored within the TPM. The RTR must verify that the integrity measurements taken by the RTM and stored in the TPM's PCRs match the expected reference values for a trusted system state.

The main functions of the Root of Trust are [8]:

- Secure Boot verifies the integrity and authenticity of the firmware before executing it. Secure boot is used to construct a chain of trust that goes from hardware components to the application layers. This can be used to prevent the injection of malicious code during the boot.

- Secure Storage provides a protected environment for storing sensitive data by isolating the information from the entire system, thereby preventing both physical and remote attacks.

- Secure Attestation is used to verify the identity of users or devices to guarantee access only to trusted entities. It can be also used to generates ephemeral keys,

which are essential for protecting conversations and safeguarding information from tampering and eavesdropping.

- Secure Update prevents injection of malicious code via compromised update, by verifying the authenticity and integrity of updates before installation.
- Cryptographic Acceleration used to perform encryption, decryption and signature faster than software alone.
- Secure Debug restricts access to debugging features giving the possibility to perform these features only to authorized users. This prevents that the attacker extracts sensitive information by exploiting debug interface.

2.1.3 Architecture of TPM

The TPM can be implemented in multiple forms but typically it is realized as discrete chip welded into device's motherboard. Modern implementation integrate TPM in the CPU or chipset. It is a passive device this means that it perform only action after a command is received. The architecture of the TPM include various components [9] [10].

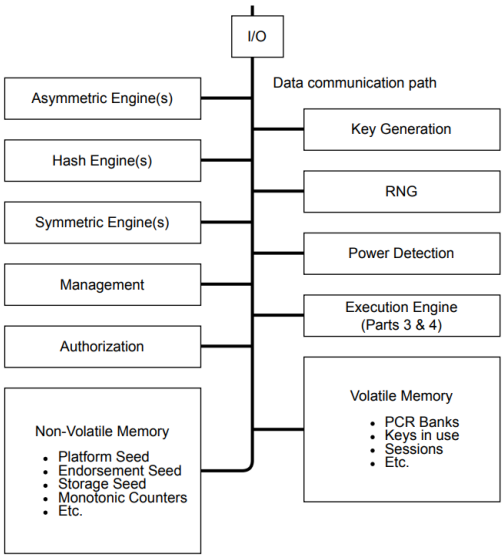

Figure 2.1: Architecture of TPM – diagram taken from Trusted Platform Module 2.0 Library Part 1[9, p. 39].

The TPM architecture is formed by a set of cryptographic functions, which are used to set up the basis of its operations. These functions include asymmetrical encryption, symmetrical encryption, hashing and digital signatures. The integration of

these algorithms allows the TPM to perform the key generation, the storage of the keys and the encryption of safe data in a tampering resistant way. Asymmetrical encryption, in particular through RSA or ECC algorithms, allows the key exchange and safe digital signatures, thus guaranteeing non -repudiation and authentication. Symmetrical encryption, such as AES, is used for rapid and efficient encryption of data, contributing to the confidentiality of the sensitive information processed by the TPM. Hashing algorithms, in particular SHA-256, play an essential role in the functions of verification of the integrity of the TPM. They are used to create unique digital fingerprints for software and firmware components. By generating hash values, that correspond to the measurements of the software states, the TPM can ensure that the start -up process has not been compromised. This ability is crucial to establish a "trusted root" within the processing environments, since the TPM can certify the integrity of the systems in which it is distributed.
Another component is the non volatile memory (NV memory) used to store keys and certificates. There are also special memory register called platform configuration registers (PCRs), which are volatile memory that stores the hashes of the measurements of the system components.
The random number generator (RNG) is used to provides randomness for cryptography operations.
An important TPM's element is the execution engine that permits to execute commands within the TPM in a secure way.
The I/O interface help to communicate with the external world.
TPM exposes its functionalities through a set of standard software interfaces. The TPM Software Stack (TSS), defined by TCG, provides a layered architecture that allows application and operating system to interact with TPM without knowing hardware-specific details. TSS is organized in more levels each one with a different level of abstraction [11].

- Feature API (FAPI) high-level interface to perform common operations like keys creation and attestation.
- Enhanced System API (ESAPI) mid-level of abstraction that permits a broader spectrum of operations.
- System API (SAPI) low-level interface that corresponds to the commands defined in the TCG specific of TPM 2.0.
- Resource Manager handles TPM when more applications would like to use it simultaneously.

This layered design makes TPM more flexible allowing applications to communicate directly with API, while those who need advanced features can use the levels closest to the hardware. An example of implementation is the open source project tpm2-tss used in Linux. [12]

TPM can be implemented in three different way [13]. Discrete TPM is an autonomous physical chip welded in the motherboard, this type of TPM offers the maximum level of security and they are used in most critical system. Integrated TPM is also an hardware solution that is integrated into one or more semiconductor packages together and separated from other components. Firmware TPM is a software solution, that is executed in a protected environment such as a Trusted Execution Environment (TEE), within CPU or System-on-Chip (SoC). Software TPM it's an emulator to simulate TPM without hardware protection. It's the less secure type, in fact is used during the development and testing phase to simulate the product.
In the following scheme it's possible to observe the main differences between each type of implementation.

TPM TYPE	SECURITY LEVEL	CHARACTERISTICS	ADVANTAGES	TYPICAL APPLICATIONS
Discrete (dTPM)	Very High	Dedicated chip, tamper-resistant hardware	Strong isolation, certified security	Critical systems, servers
Integrated (iTPM)	High	Integrated into chipset or SoC	No extra chip, cost efficient	Gateways, enterprise PCs
Firmware (fTPM)	Medium	Runs in CPU's Trusted Execution Environment (TEE)	Flexible, widely available	Laptops, consumer devices
Software (sTPM)	Low	Pure software emulation	Useful for testing and prototyping	Development, research
Virtual (vTPM)	Medium-High	Managed by hypervisor	Supports cloud scalability, flexible deployment	Cloud, virtual machines

Figure 2.2: TPM's types

2.1.4 Key functionalities of TPM

Key generation and Management

The key generation is one of the fundamental functionalities performed from the Trusted Platform Module. Cryptography is the art of modifying information to make them unreadable without knowing the key. This allow people to exchange information in a way that prevents others from reading it. This process is called encryption and can be performed through the use of symmetric encryption or asymmetric encryption, both require the use of a key. As stated before thanks to its integrated cryptographic engine, the TPM is able to create asymmetric, symmetric

and ephemeral keys in an isolated and protected environment with respect to the operating system. The TPM guarantees that the key are produced through a random number generator and saved in a secure storage.
Trusted Platform Modules (TPMs) store secret values called seeds that remain within the module and persist after system reboots. These seeds facilitate deterministic key generation, eliminating the need to directly store cryptographic keys. The process of key generation follows TCG's standards and can generate different types of key, each with a specific purpose. [14] [15]

- Endorsement Key (EK) is a persistent asymmetric key, installed in the TPM during the production phase. This key represents the root of trust for operation of attestation. The Endorsement Key (EK) is a public/private key-pair. The private key is generated within the TPM and is never revealed outside. EK uniquely identify TPM and other functions that cannot be modified. It can be observed that much of the trust related to the TPM comes from the characteristics assigned to the EK.

- Storage Root Key (SRK) is created when the TPM is initialized. As part of the key hierarchy, it serves as the root of the storage structure. The Storage Root Key protects other keys by encrypting them before they are stored outside the TPM. This means that only the TPM that originally created these keys can later reload and use them, making the system security more robust and trustworthy.

- Attestation Identity Keys (AIK) are used to attest, allowing TPM to sign quotes about the integrity of the platform. They are used to avoid direct exposure of the EK. Typically, AIKs are certified by a Privacy Certificate Authority (CA) or through Direct Anonymous Attestation (DAA) protocols.

- Ephemeral and session keys are temporary keys generated by the TPM for the duration of a session. These types of keys are ephemeral, meaning that they are deleted when the session is closed.

The following figure illustrates the structure of the key hierarchy in the Trusted Platform Module (TPM).

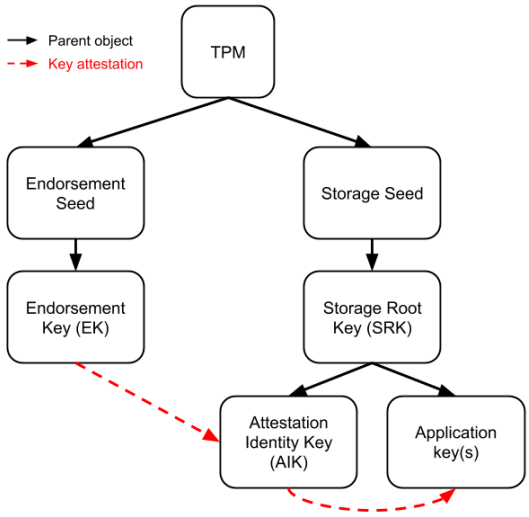

Figure 2.3: Key Hierarchy - diagram taken from Eric Chiang posts [14, p. 1]

Secure Boot and Firmware Integrity

The execution environment for code must be strictly controlled. Secure Boot refers to the process of establishing a verifiable and secure initial system state. Ensuring integrity involves storing the signature of the boot sequence in the Trusted Platform Module (TPM) and verifying that this signature is valid during each boot. The system cannot proceed without confirmation of the boot sequence by the TPM. Standard secure boot methods are used to construct a chain of trust that goes from hardware components to the application layers. This approach can be replicated across several stages, creating a chain of trust by dividing the boot process into distinct levels. [10]

Figure 2.4: Secure Boot

A first protected bootloader A, stored in a secure memory, verifies the integrity and authenticity of a second bootloader B. The second bootloader, B, verifies the integrity and authenticity of the Operating System kernel and the Firmware. Typically, the first bootloader, A, cannot be modified and is trusted, whereas the second bootloader, B, can be updated.

Remote Attestation

Remote attestation constitutes a core function of the Trusted Platform Module (TPM). This process allows an external verifier, known as a challenger, to acquire trustworthy evidence regarding the integrity and configuration of a remote platform. This function is accomplished by utilizing measurements stored within the TPM. The primary objective of attestation is to verify that the system, whether hardware or software, operates only authorized software and firmware, and remains free from malicious code. During the attestation process, two distinct roles are defined: the Attester, which refers to the entity that contains the TPM and seeks to demonstrate its integrity, and the Verifier, which aims to assess the integrity of the Attester. The process for performing remote attestation is the following [16]:

1. The Verifier sends a challenge to the Attester used to avoid reply attacks, plus a command that tells the platform to read the values stored in the Platform Configuration Registers (PCRs) and report them along with a signature. This report is commonly referred as attestation report or quote. Since the report values could be forged, the report must be signed using the Attestation Key (AK) that is certified by the Endorsement Key (EK) or a Privacy CA. This signature serves to confirm that the measurements are authentic and originate from a specific TPM, since the EK is unique to each device.
   The TPM manufacturer issues a certificate of identity for the AIK, signed by the EK or a Privacy CA, so that the Verifier knows that the AIK belongs to a genuine TPM without exposing the EK key itself.

2. The Attester signs the measurements and challenge using the AK private key and sends them to the Verifier.

3. The Verifier validates the signature thanks to the AK public key and checks the received measurements against reference measurements (known as good values).

4. If the Verifier determines that the platform's configuration is secure, the Verifier will grant an authentication key to the platform, indicating a valid attribute.

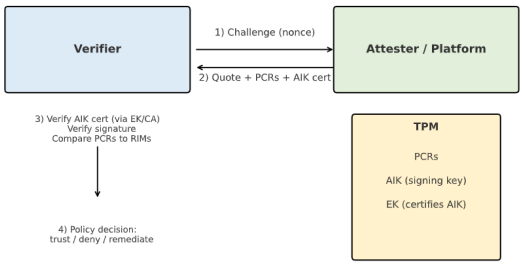

Figure 2.5: Simplified workflow of Remote Attestation

Remote attestation is more than a single process and can take several forms, each with its own practical factors. The main difference is between static attestation, which uses measurements from the boot process stored in Platform Configuration Registers (PCRs), and dynamic attestation, which starts after boot by a trusted component like a hypervisor to check the platform's current state. Static attestation allows to verify that the system is in a safe state during the boot, while dynamic attestation allows for checks runtime vulnerabilities that could not be founded during the initial boot attestation. [17]
Even if remote attestation brings security advantages, it also comes with practical challenges. It's essential to keep updating Reference Integrity Measurements (RIMs), since old or missing values can cause false negatives. In large environments like enterprises or cloud systems, scalability is an issue because many devices may need to be attested simultaneously. [18]

Integrity of Data

Binding is a fundamental function of the Trusted Platform Module (TPM), which ensures data protection. This process involves encrypting information in a way that only the device that generated the keys, or to which the keys are associated, can decrypt the data. In fact, the data are restricted to the specific hardware and cannot be accessed from an unauthorized device. The binding is particularly important for sensitive files, like cryptographic keys or digital certificates. This process strengthens the trustworthiness of the system by binding information to the TPM, which acts as the root of trust.
In the binding process, the data to protect are encrypted with TPM's blind key, which is a unique RSA key derived from a storage key. Since the key is TPM-specific, it cannot be used outside, and the data can be decrypted only on a platform that has that TPM.
Another important process that ensures the integrity of the data is the sealing process. Unlike the binding process, which is a standard encryption procedure, sealing also relies on the system's current state as recorded in the Platform Configuration

Registers (PCRs). The Trusted Platform Module (TPM) prevents decryption unless the PCR values match those stored during encryption. Consequently, data is accessible only when the machine boots in the identical security state. [15]

Protected Location

One of the fundamental security properties of the Trusted Platform Module (TPM) is the ability to offer secure environments for managing sensitive data and cryptographic operations. In the specification, a distinction is made between protected location and shielded location. A protected location ensures that data, such as private keys, cannot be read or extracted directly from the TPM, thus offering passive protection even when such data is stored externally, as long as it remains encrypted. A shielded location, on the other hand, implies an active protection context: sensitive data can only be used within the TPM via protected capabilities (functionalities that can be executed only in the TPM), ensuring that critical information never leaves the secure confines of the device. Together, these two functions form the core of the TPM's ability to protect keys and operations from unauthorized access or manipulation. [7]

Authorization

The authorization process involves verifying who can access specific functionalities and information within the TPM. TPM supports different authorization mechanisms [9]:

- Password-Based Authorization is the simplest method, which involves sharing a password between the TPM and the user or software that requires access.
- HMAC-Based Authorization is a method that can help prevent reply attacks by utilizing a nonce, specifically one for the TPM and another for the caller.
- Policy-Based Authorization defines complex constraints through TPM policy. Some of these constraints could include: PCR values, the signature of a specific entity, counters, or temporal limits.

Support to authentication protocols

The Trusted Platform Module (TPM) is not only a module for key encryption and protection, but also provides fundamental mechanisms for authentication protocols. Thanks to its architecture based on asymmetric keys and a hardware root of trust, TPM enables strengthening authentication processes at both local and remote levels. [19]

- Key-based authentication: The TPM generates and stores cryptographic keys that can be used in authentication protocols, such as RSA and ECC, ensuring that private keys never leave the device.
- Challenge-response: The TPM can respond to cryptographic challenges proposed by a server or verification entity, demonstrating possession of a key without revealing it.
- User credential protection: TPM can be integrated with authentication protocols such as Kerberos, TLS, or virtual smart cards, securely storing session keys and certificates.
- Single Sign-On (SSO) and two-factor authentication: combining TPM with PIN, password, or biometric data results in a secure hardware factor that strengthens the entire login process.
- Remote Attestation.

2.1.5 Real-world applications of TPM

Security in Windows 11

With the introduction of Windows 11, Microsoft made support for TPM 2.0 mandatory, recognizing its central role as the trusted hardware root for operating system protection. TPM facilitates different security features, in particular BitLocker and Windows Hello.

BitLocker utilizes TPM to securely safeguard disk encryption keys, ensuring data remains inaccessible if the drive is removed or the system is tampered. BitLocker, a complete disk encryption function implemented in Microsoft Windows, is based on a multi -level encryption protocol to safeguard sensitive data. At the center of its framework are two critical keys: the integral volume encryption button (FVEK) and the volume key (VMK). The FVEK is a symmetrical key generated specifically to encrypt the data on the volumes, while the VMK acts as an intermediary that encrypts the FVEK. This double key approach is the basis of BitLocker's security architecture, facilitating secure encryption and decryption processes. At the time of the start of the system or access to the user, the VMK is decrypted using various authentication mechanisms, allowing access to FVEK and thus allowing the decryption of the stored data. The use of the VMK allows BitLocker to provide further security levels, ensuring that even if the FVEK is compromised, the encrypted data would remain safe unless the VMK is possible to access. The VMK can be protected using different authentication methods, including password-based access, recovery keys and hardware-based safety via the reliable platform module (TPM).

The importance of TPM becomes particularly obvious in the management of VMK. When using BitLocker, the VMK is safe in the TPM chip. This hardware security module guarantees that the VMK cannot be extracted or compromised by unauthorized software, thus retaining the secret of FVEK. BitLocker can works without a Trusted Platform Module (TPM); however, with the use of TPM, it validates the integrity of boot and system files before decrypting a protected volume. [10] [20]

In collaboration with BitLocker, Windows Hello serves as an advanced authentication method which improves the user experience by providing secure access without the need for traditional passwords. Using TPM, Windows Hello facilitates the management of biometric data, the firm storage of fingerprints or facial recognition data in the module. This prevents exposure of authentication data sensitive to potentially vulnerable operating environments, considerably reducing the risks associated with systems based on passwords. The integration of TPM into Windows Hello allows the generation of unique cryptographic keys linked to the authenticated user; These keys are essential for operations such as secure connection processes, unlock efforts of devices and even integration with online services requiring user identification. [21]

Could and Virtual Environment

Together with the evolution of cloud computing, the concept of a virtual trust platform module (VTPM) has become a fundamental innovation. The VTPM replicates the central functionalities of a physical TPM, but operates within virtualized environments. Its implementation in cloud infrastructure is particularly significant, since it allows multiple tenants to safely share the underlying hardware while maintaining their different security parameters. The VTPM generates, stores and manages virtualized instances of keys and credentials in a way that aligns with multiple cloud services. This is generally achieved through optimized emulation based on software of TPM functionalities, allowing a safe virtualization of computing. Given the growing adoption of TPM and VTPM in cloud and virtual environments, an exhaustive analysis of its implications is essential, particularly in relation to safety, performance and compatibility challenges. Security remains a main consideration, since the deployment of TPM or VTPM does not automatically guarantee complete protection against all forms of cyber threats. Potential risks include attacks against hypervisors infrastructure and vulnerabilities introduced through erroneous configuration of the VM that require comprehensive risk assessments and mitigation strategies. Performance implications arise from computational overload associated with cryptographic operations that can influence the capacity for response and general efficiency of cloud services. Since TPM operations can incur latency due to cryptographic processing requirements, especially in high demand environments,

an examination of how to balance security benefits with performance compensation is essential. In addition, compatibility challenges is a significant barrier in the implementation of TPM and VTPM within cloud due to heterogeneity of hardware and software platforms, and the variations of TPM specifications with different manufacturers. In addition, existing virtualization technologies must guarantee integration with physical and virtual TPM implementations to avoid interruptions of the service. To address these compatibility challenges it is important the collaboration between cloud suppliers, hardware manufacturers and software developers to create standardized solutions and guidelines to facilitate the adoption of TPM in the cloud.

Internet of Things (IoT)

IoT devices spread through various sectors, including medical care, transport and industrial automation, its interconnected nature raises important security concerns related to the integrity of data, confidentiality and user privacy. Rapid diffusion of these devices appear not to be in parallel with adequate safety measures, this makes IoT networks vulnerable to various cyber threats, including unauthorized access, data tempering, and distributed denial of service attacks (DDOS). Therefore, ensuring security within the IoT ecosystem is essential not only to protect confidential information, but also to maintain user confidence and to guarantee the reliability of services that depend on millions of users around the world. The TPM could play an important role in improving the security features of IoT. It can facilitates authentication of the device, ensuring that only legitimate devices can access to the network. This is particularly crucial in IoT environments, where if unauthorized devices obtain access can lead to alarming security violations. In addition, TPM allows safe communication channels through the key establishment and management, safeguarding the data transmitted between IoT devices and their associated applications. In addition, the capacity of TPM to verify the integrity of the system, including the integrity of the firmware and the software, acts as a significant element against malware and other malicious exploits designed to compromise device performance or obtain unauthorized access to sensitive data. In addition, the integration of TPM into IoT devices improves the general resilience of the IoT ecosystem by providing a basis for establishing trust not only within individual devices but also throughout the network. Even if there are a lot of advantages of the introduction of TPM in IoT, several challenges prevent its adoption. One of the main challenges is the lack of standardization between various TPM implementations. Given the variety of IoT devices, manufacturers can implement TPM technology in different way, this could make communication and cooperation between devices difficult. In addition, scalability restrictions and resources on low power devices represent significant barriers to TPM implementation. Many

IoT devices are designed to operate with minimal energy consumption and low computational power, making the embedding of additional hardware modules difficult. Consequently, the integration of TPM technology into IoT devices requires innovative designs and optimizations, to make sure that safety improvements do not obstruct the efficiency of IoT. [22]

2.1.6 Vulnerabilities

This subchapter examines the vulnerabilities that can be found in TPM technology and the challenges associated with its adoption. Understanding these weaknesses is crucial to assessing the validity of TPM as a solution to ensure data integrity and confidentiality. The main challenges associated with the adoption of TPM arise from the complexity of its specification and the consequent extension of the attack surface. The TPM 2.0 reference library is multifaceted and difficult to manage without risk: an obvious example is the CVE-2023-1017 and CVE-2023-1018 vulnerabilities, which affected billions of devices due to parsing errors and buffer overflows in the library itself.[23]

Architectural and Implementation Vulnerabilities

In discrete TPMs (dTPMs) communication occurs on external buses, such as SPIs or LPCs, susceptible to interception by an adversary with physical access; in contrast, in TPM firmware versions (fTPM or PTT) the module is integrated into the CPU or chipset, reducing physical exposure but increasing the attack surface through side and shared channels. The operational complexity related to the use of PCR registers for measured boot and for remote attestation is another issues. While they make it possible to validate the integrity of the start-up chain, they also require strict management policies and the constant updating of reference baselines, without the system risks false positives or, worse, the impossibility of detecting real compromises. [24]
A further challenge concerns the quality of the random number generator provided by the TPM, whose compliance with the NIST SP 800-90 and FIPS 140-3 standards is essential, but not always uniformly verifiable in different devices [25].

Firmware vulnerabilities and Software Stack Vulnerabilities

Firmware vulnerabilities represent another significant risk vector within TPM technology. The firmware operates in a privileged mode and has wide control of the system. Firmware violations can lead to unauthorized access or manipulation of cryptographic functions performed by the TPM. An important problem issued when the firmware is outdated or without patches. If the firmware has known weaknesses, attackers can exploit them by sending malicious code or commands to

the TPM.
In addition, insecure communication protocols are another type of firmware vulnerability. TPM often communicates safely with other devices, but if these communication protocols are weak, attackers can intercept data that are transferred. Poorly secured firmware can lead to data leaks and unauthorized access to confidential information.
Finally, implementation errors in libraries and the software stack, as demonstrated by the 2023 vulnerabilities, confirm that the robustness of the TPM depends not only on the hardware but also on the quality of the code that handles its operations. [26]

Physical and Side-Channel Attacks

Hardware defects in TPM, although less frequently reported than software vulnerabilities, carry critical risks. The most relevant attack is the side-channel attack, in particular through the analysis of module response times. TPM-FAIL has shown how both dTPMs and fTPMs can reveal enough information to reconstruct ECDSA keys through timing measurements, with a few thousand observations. [27]
These attacks are often aimed at cryptographic operations used by the TPM, analyzing time information, energy consumption, or electromagnetic emissions. If successfully run, an adversary could recover cryptographic keys or confidential data, defeating the central purpose of the TPM. An example of a side channel attack is the well-documented "electromagnetic side-channel attacks," which allows attackers to capture and rebuild sensitive signals emitted by the TPM, revealing private keys. [28]
Added to this are cases of traffic interception on external buses in systems with dTPM. Research has shown how it is possible to capture BitLocker unlock keys using economical tools for sniffing SPI or LPC traffic. [29]
Another physical vulnerability is related to failure injection techniques. In this method, attackers try to induce errors in the TPM using different physical means, such as voltage peaks or temperature changes. When normal TPM operations are interrupted, confidential information may be exposed or compromised. [30]
Finally, TPM-only configurations for disk protection, in addition to being vulnerable to such techniques, also lend themselves to more classic bypasses such as cold-boot attacks, which exploit the remanence of data in RAM, and DMA attacks. [29]

Virtual TPM Vulnerabilities

Virtual Trusted Platform Modules (vTPMs), widely used in cloud environments to provide a virtual “root of trust”, introduce additional layers of complexity and attack surface. While offering flexibility and functions analogous to a physical

TPM, vTPMs lose the physical security properties inherent in a dedicated chip and depend entirely on the integrity of the host environment: if the hypervisor, virtual machine monitor, or cloud control plane is compromised, all the guarantees offered by the vTPM are severely weakened.
In such scenarios an attacker can exploit techniques such as VM escape, privilege escalation, or VMM compromise to directly manipulate the vTPM service —such as extracting keys from it or altering cryptographic operations— since trust is delegated to the host's software. The literature and practical implementations also show more "ordinary" class vulnerabilities: NVRAM replacement, rollbacks, vTPM backing file attacks, and communication channel control attacks between guests and vTPMs, all of which are possible when the vTPM is not protected by an isolated hardware environment (e.g., TEE). For this reason, many recent proposals (e.g., SvTPM) recommend encapsulating the vTPM logic in a Trusted Execution Environment or anchoring the vTPM to trusted hardware resources: these countermeasures reduce the reliance on hypervisor correctness and mitigate control attacks and key theft, while maintaining the benefits of virtualization.[31]

2.2 Physical Unclonable Function (PUF)

2.2.1 Definition

In recent years, the panorama of IT security has undergone significant transformations, which require advanced techniques to strengthen the integrity and confidentiality of data. A promising solution that has emerged in this domain is the concept of physically unclonable functions (PUF). PUFs are unique cryptographic primitives that exploit intrinsic variations in the physical characteristics of circuits to generate answers that are distinct and cannot be replicated. A PUF operates using the non-ideals found within the hardware components, such as the variations of the properties of silicon, the inconsistencies of production and the differences deriving from environmental influences. These factors culminate in the creation of a response that is not only distinctive for each instance of a PUF but also resistant to cloning efforts. The fundamental advantage of PUF technology lies in its ability to produce a unique output based on these physical characteristics, thus facilitating safe authentication and key generation processes that are based on the hardware itself. In the context of authentication, the PUFs present significant progress with respect to traditional cryptographic keys. Systems based on conventional key, although effective, are often vulnerable to various forms of attack, including key theft, replica and unauthorized access. On the contrary, the PUFs offer a solid alternative, since the authentication process does not depend on a static key, which can be intercepted or duplicated. Instead, the PUFs generate a reconstruct each time a unique response based on the physical status of the device at the time of

the query . [32] [33]

2.2.2 Property

PUFs exploit the inherent manufacturing variations that are within silicon hardware to create unique digital identifiers or digital fingerprints for each device. The silicon fingerprint is generated through inherent variations of the process that occur during the manufacture of semiconductors. These variations occur thank to factors such as temperature fluctuations, voltage differences and material inconsistencies. Consequently, each chips manufactured, even if they have the same production line, will have negligible discrepancies, which will lead to unique identification characteristics. The silicon fingerprint acts as a digital signature for hardware, promoting a robust identification that is practically impossible to replicate. These functions are evaluated based on several parameters that contribute to their robustness and effectiveness. [33]

- Uniqueness of PUFS is essential; Each instance is generated through random physical characteristics that arise from the manufacturing process, ensuring that even identical devices exhibit different responses. This uniqueness property is essential to distinguish one unit from another in a multitude of applications, to perform safe authentication and generation of cryptographic keys. A device can be uniquely identified through its set of challenge-response pair (CPR).

- Despite its unique characteristics, it is essential that the PUF produce consistent results when they are submitted to the same entry, which allows reliable performance over time. This aspect guarantees that legitimate users can recover the same response to interacting with the PUF, reinforcing their role in applications where authentication is vital. The ability to reproduce identical results in specified conditions is essential for confidence in hardware security mechanisms.

- The unpredictability of the results is vital to ensure several attacks, including those derived from attempts on cloning or reverse engineering of the device. PUF generates results that are not easy to predict, which fortifies their effectiveness as a security measure. This unpredictability element ensures that even if an adversary has an instance of a PUF, it cannot simply extrapolate or derive the responses of other identical units.

- Unclonability further consolidates the importance of PUF in the field of security devices. The physical variation inherent between different iterations of the same hardware makes it practically impossible to clone a precision PUF, also for manufacturer. This feature serves as a formidable barrier against

replication attacks aimed at avoiding security characteristics. Unclonability ensures that even with access to the same manufacturing conditions, intrinsic discrepancies in unique physical characteristics prevent the generation of equivalent PUF outputs.

- Evaluability means a computation of the response, given the challenge, which is possible within the strict timing, area, power, energy and cost budget. So, the response must be easy to compute.

- One-Wayness means that given a random response of that instance, there isn't an efficient inversion algorithm on the PUF instance, that finds a challenge that would produce a response equal or close to the given response.

PUFs can be classified according to the size of Challenge-Response Pairs [33].

- Weak PUF are defined by their limited production capacity, generally capable of producing a singular response for each unique challenge presented. They are mostly used for key storage. The CPR access must be restricted, otherwise the attacker can try to predict the unknown CPR. Example of weak PUF is the SRAM-based PUF.

- Strong PUF are equipped with a significantly larger output potential, allowing more responses corresponding to a single challenge. This improvement allows the creation of resilient authentication systems. A portion of the CPR set can be public, because is impossible to predict the unknown CPR from them. Example of strong PUF are Arbiter and Ring Oscillator design.

PUFs are classified according to their composition in three different types: non-electronic, electronic and silicon-based.
Non-electronic PUFs mainly use physical attributes of their substrates to create intricate patterns that are inherently exclusive to each filament. They are based on non-electronic technologies or material, for example, the random fiber structure of a sheet of paper.
On the contrary, electronic PUFs integrate active components that interact with electronic systems to produce responses to external stimuli. This interaction allows advanced characteristics as dynamic real-time verification processes, which significantly improve the effectiveness of security measures.
Silicon-based PUFs represent another critical innovation within this domain. These PUFs take advantage of the properties of silicon, the random micro-variations that are created during manufacturing, to develop unique complex but efficient identifiers that are not only reliable but also scalable with the social demand for security improvements. [33]

There is another crucial distinction between intrinsic and non-intrinsic PUFs. The intrinsic PUFs exploit the microscopic variations in the semiconductive themselves, allowing unique fingerprint capacities without the need for additional components. However, non-intrinsic PUFs require the integration of additional element to generate distinctive responses. [33]

2.2.3 Types of PUFs

PUFs can be categorized in various classes based on their underlying operational principles. Delay-based PUFs, exploit variations in propagation delay in circuit elements. On the other hand, memory-based PUFs depend on unpredictable variations in memory elements, such as SRAM cells or DRAM cells. Mixed signal PUFs combine aspects of delay and memory approaches, usually using components such as oscillators or capacitors to generate exclusive signatures under different environmental conditions.

Arbiter PUF

PUF Arbiters are explicitly classified as non-clonable physical functions based on the delay, in which the essence of their uniqueness derives from these unpredictable delays of the path. The architecture of an arbiter PUF consists on two parallel rows of multiplexer and an input signal, that is splitted in two identical paths, one on each row. The sub-paths taken from the input signal is decided from the binary challenge. The fundamental construction embraces the concept that small differences in propagation delays through the device can produce significantly varied production. The response is given by an arbiter, typically a D-Latch Flip Flop, that captures which of the two signals arrives first. [33]

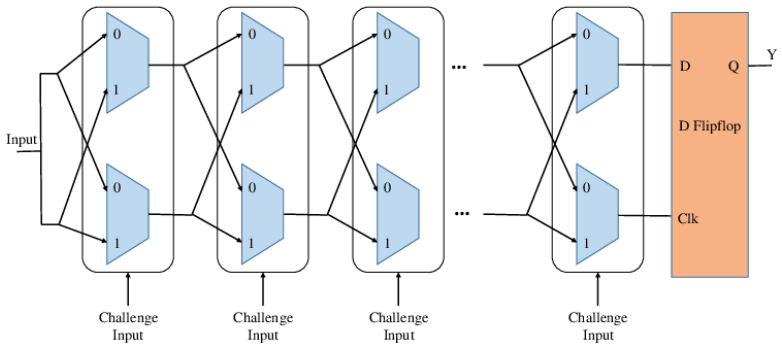

Figure 2.6: Arbiter PUF Design - figure taken from [34, p. 1]

The process of generation of unique responses within PUF Arbiter is mainly attributed to the variations of the delays of the devices and associated parameters, which can be sectioned in different key aspects. Firstly, the production process gives small but critical differences in the actual dimensions of the transistors and in the interconnection threads, changes that can thinly move the electrical characteristics, leading to disparity in the travel times of the signal. Secondly, operating conditions such as the temperature and power voltage can affect the mobility of electrons within the semiconductor material, further contributing to the variability of the delay. Thirdly, the intrinsic stochastic nature of these lines of delay guarantees that identical devices will also produce different answers to the same challenge, granting the PUF Arbiters their fundamental characteristics of uniqueness.

In terms of resistance to reverse engineering, the PUF Arbiters are designed in such a way that the internal mechanisms cannot be easily discerned by the exit produced. The complexity and unpredictability of the physical layout and intrinsic randomness in the delays of the path make it extremely difficult for the opponents to reconstruct the PUF or deduce its specific output characteristics. This aspect is essential for applications that require the maintenance of secret key information safely within a chip, as it limits exposure to contradictory attacks that generally try to extract the cryptographic keys through invasive techniques.

The implications of these advantages become particularly relevant in applications such as safe communications, in which the integrity and authenticity of the information transmitted on the networks are crucial. The ability to generate unique cryptographic keys that are firmly connected to the hardware platform allows safe distribution and authentication protocols that significantly reduce the risk of man-in-the-middle attacks. [33]

Ring Oscillator PUF

Ring oscillator PUF represent a specific implementation that exploit the dynamic behavior of ring oscillator circuits to generate unique answers. A ring oscillator is an arrangement of a odd number of inverters connected in a loop forming a closed circuit. The operational mechanics of a PUF ring oscillator depends on variations in the propagation delay of each inverter, which can be influenced by factors such as temperature, tension and inconsistencies inherent in manufacturing. By using two multiplexer is possible to select two oscillators an then compare their frequencies. In base of the result of the comparison the output will be zero or one. The challenge is the input of the two multiplexers, so in base of the challenge two specific oscillators will be selected. [33]

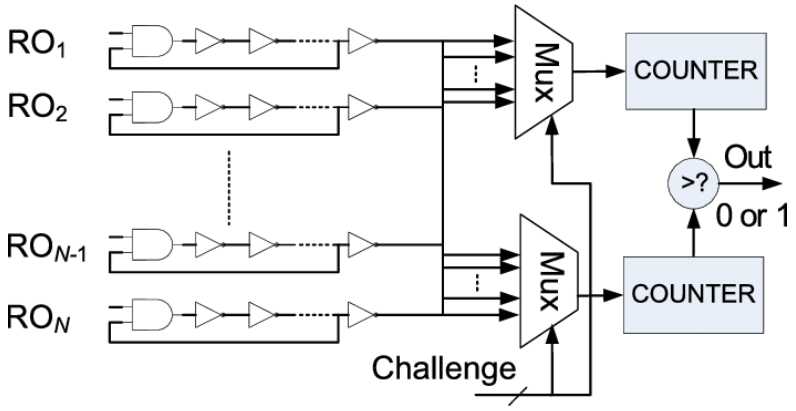

Figure 2.7: Ring Oscillator Design - figure taken from [35, p. 1]

When measuring the frequency of ring oscillator is possible to derive a unique output that can be used as an cryptographic key or for device authentication. The output of a ring oscillator PUF depends also on the specific circuit configuration, which is dictated by the stochastic nature of the silicon manufacturing process. This variability ensures that each instance of a ring oscillator PUF is distinct.
In addition, the deployment of ring oscillators can mitigate the risks associated with the attacks of the side channel. An opponent tries to collect information from the physical implementation of the device, including the analysis of the times or the consumption of energy, but the unpredictable intrinsic behavior of ring oscillators complicates this type of analysis, since the characteristics of the output remain significantly obscured. [33]

SRAM-based PUF

SRAM PUF uses the unique behavior of SRAM cells during the energy phase. When the power is provided to a SRAM, the cells enter a state of power and stabilize at a particular binary value, "0" or "1". The state of single tension for each cell constitutes a distinct bits chain, which serves as a signature in the device's silicon. This bits chain remains consistent for a given device but is unpredictable and unique on different devices, influenced by minimum variations in the manufacturing process, temperature and other environmental conditions. This inherent randomness leads to the generation of an exclusive distinct response from each device, actually serving as a fingerprint. The selected cell is chosen from the challenge. [33]

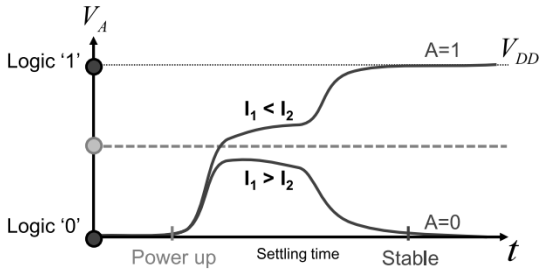

Figure 2.8: SRAM cell power-up - photo taken from Roel Maes' book [33, p. 35]

In the figure it's possible to observe that during the power-up phase the SRAM cell is in an insecure state (gray line) and based on the physical characteristics of the cell, it will stabilize either at 1 or 0.

A remarkable operating mechanism of SRAM PUFs is the process known as bits massage, which plays a key role in increasing the reliability of PUF outputs. As SRAM cell characteristics may display instability due to environmental fluctuations or aging effects, the initial response may vary in subsequent energy cycles. To address this variability, a series of bits masking algorithms can be implemented to improve output robustness. Bits massage involves the application of error correction or coding techniques that can selectively mask unreliable bits based on predefined rules or historical data collected from previous measurements. This corrective approach allows the stabilization of the PUF output, thus ensuring that, even with small fluctuations, a consistent and reproducible unique response can still be generated. The variability of power up state of SRAM cells serves as mechanisms device identification and key generation. Additionally, SRAM-based PUFs benefit of the high density of SRAM cells that allow for a large number of bits for robust key generation. [33]

New Solutions

Optical Physical Unclonable Functions (oPUFs) represent one of the first and most studied implementations of PUF technology. Unlike electronic PUFs, which rely on the microscopic variations found in semiconductor devices, oPUFs exploit the complex and inherently random interaction between light and the internal structure of a disordered optical medium. When a coherent beam of light, such as a laser, illuminates a transparent material characterized by a random distribution of diffusive centers, a unique speckle pattern is generated which constitutes the system's response to the specific challenge. Since these scattering centers are determined by uncontrollable variations introduced during the manufacturing process, each optical

medium produces a unique, unrepeatable response that is practically impossible to clone. The safety of oPUFs derives from the exponential complexity of the scattering process: even minimal differences at the nanoscale generate completely different interference patterns, making any attempt at replication computationally and physically unachievable. Furthermore, the high dimensionality of optical challenge–response pairs provides a very large source of entropy, allowing for the extraction of robust cryptographic keys. This makes oPUFs not only intrinsically unclonable, but also highly resistant to modeling attacks, since the relationship between light input and output speckle patterns is too complex to be approximated with classical algorithms. [36] [37]

The development of Quantum PUFs (qPUFs) takes this concept further by utilizing quantum mechanics, where challenges arise from quantum states rather than classical signals, in which challenges are no longer classical signals, but quantum states, and the responses derive from the interaction between these states and the unrepeatable physical properties of the device. QPUFs draw their strength from the fundamental principles of quantum mechanics. Firstly, no-cloning theorem ensure that no perfect copying of unknown quantum states can be performed, ensuring a higher level of security than traditional PUFs. Moreover, the destructive nature of quantum measurement makes it impossible for an adversary to observe the state, limiting the possibility of invasive attacks. Finally, the inherent probabilistic randomness of quantum measurements contributes to making each response unpredictable and non-modelable, further strengthening the resistance of qPUFs to cloning or simulation attempts. Quantum PUF takes advantage of the quantum tunneling phenomenon, that can be observed in transistor with dimension less then 100 nm. This method explores the unpredictable and non replicable nature of quantum mechanics, allowing the generation of exclusive cryptographic keys that are inherently linked to the individual device. In modern transistors, a layer of insulating oxide allows electrons to tunnel, generating a small electric current. Intrinsic variations in manufacturing processes make each layer unique. These differences result in leakage currents specific to each transistor that create a unique fingerprint for each chip. [38] [39]

2.2.4 PUF Application

Key Generation

Secret key generation is a cornerstone in all cryptographic systems. A key must have three main properties: randomness that ensure unpredictability of the key; uniqueness, to reduce the likelihood of collisions with other keys; and stability, so that it can be reproduced when necessary. Keys are generated through random number generators (PRNG or TRNG), usually based on physical phenomena that

are difficult to predict, such as thermal noise or temporal variations in processors. However, in many embedded and IoT devices the availability of reliable entropy sources may be limited. Once the key is generated it must be stored securely: unprotected storage in non-volatile memory poses the risk of physical extraction and cloning. Physical Unclonable Functions (PUFs) address these problems, allowing to regenerate a key "on request", avoiding having to save it permanently.

In PUF-based key generation first the provisioning phase is executed, during which a set of PUF responses is acquired and auxiliary data (or helper data) are generated. This data does not directly reveal the key, but contains enough information to regenerate it later. A classic approach uses fuzzy extractors and secure sketches, which allow stable keys to be derived from noisy sources, correcting the differences that emerge between different acquisitions. Here, reliability is crucial: key regeneration must be identical even in the presence of temperature, voltage or chip aging variations. [33] [40]

There are many schemes that have been proposed for PUF-based key generation:

- Helper data algorithms: store information derived from responses (syndrome) that allows you to correct errors. However, a critical issue is the possibility of leakage: helper data, while theoretically public, can reduce the opponent's search space and facilitate key extraction. To mitigate this problem, schemes such as Index-Based Syndrome coding (IBS) have been introduced (Yu & Devadas, 2010), which minimizes entropy loss and reduces hardware complexity. [41]
- PUFKY (Maes et al., 2012): This approach is used to generate 128-bit keys with high reliability through PUF over FPGA. This scheme combines error correction codes (BCH), helper data and fuzzy extractors, demonstrating that a PUF generator can be competitive with classical systems. [42]
- SRAM PUF key generators: exploit the initial random state of SRAM cells on lighting as a source of entropy. More stable bit selection techniques and majority voting algorithms increase reproducibility. The approach is particularly attractive for low-power IoT devices (Gao et al., 2019). [43]
- Pattern Matching Key Generators (PMKG) (Paral & Devadas, 2011): scheme that overturns the traditional paradigm. Instead of keeping responses secret and challenges public, it makes short response patterns public and keeps challenge indexes secret. The key is reconstructed via pattern recognition, eliminating the need for complex decoders. This reduces hardware costs and increases resistance to modeling attacks. [41]

Identification and Authentication

Device identification consists of associating an entity (person or device) with a specific identity within a set. While authentication consists of verifying the identity already declared through proof. Within the reign of identification based on PUF, two main distinction for the categorization of identity can be outlined: inherent identity and assigned identity. The inherent identities derive from the unique physical characteristics of the PUF, typically manifested as electrical, optical or thermal discrepancies for individual devices. These identities are not scheduled or assigned; Rather, they are a natural consequence of the complex interactions within the microfabrication process. During the production process, environmental factors and material inconsistencies contribute to the formation of distinct models that can be extracted and used for identification. The consequent integrity of inherent identities provides a solid solution to the challenge of imitation and unauthorized access, which is fundamental in an era marked by sophisticated IT threats. On the contrary, the assigned identities are summarized through discounted configuration or assignment documents once the PUF has been established. Generally, this involves the association of a specific identifier, such as a unique standard number or a cryptographic key, with the inherent characteristics of a device produced by the PUF. While the assigned identities allow customizable integration into existing identification paintings, introduce potential vulnerability, such as the risks associated with the management of keys and the possibility of reassigning identity or theft. The Identification process works is divided in two phases. The first phase is different based on the type of entity.

- For inherent identities, the first phase consists of collecting all the inherent identities of the entities that needs to be identified. This phase is called enrollment phase.
- For assigned identities, the first phase consist of giving all the unique identifier of the entities that needs to be identified. This phase is called provisioning phase.

The second phase is similar to both types. In this phase, called identification phase, the entity presenting its identity when required.
When using PUF (Physical Unclonable Function) as the identity source for a device, a challenge immediately araises: a PUF's response is never perfectly stable. Even though physical variations of silicon are unique and unclonable, the readings of a PUF may differ slightly due to electrical noise, temperature, supply voltage, or circuit aging. This means that although two consecutive readings from the same device are very similar, they are not identical bit by bit. This is where the concept of fuzzy identification comes into play.

Fuzzy identification is a technique that allows the "flawed" responses of a PUF to be used as a reliable fingerprint for identification. The basic idea is to treat identity not as a rigid, immutable sequence of bits, but as a vector that can tolerate small errors or discrepancies. In practice, instead of requiring a perfect match, you accept a match "close enough", within a pre-established threshold. The process works in the following way: the system acquires one or more PUF responses and associates them with the device identity. These responses are then processed with the help of Error Correcting Codes (ECC) algorithms or similarity metrics (e.g., the Hamming distance). When a device needs to be identified, it provides its PUF response again. Even if the new response is not 100% the same as the recorded one, the system checks whether it is close enough: if the number of discordant bits is within the acceptable threshold, the identification is considered valid.

At the heart of the authentication of PUF-based entities is the response-response mechanism, which serves as a fundamental process to check the authenticity of a device. This mechanism consists in presenting a random challenge to a PUF, which subsequently produces a corresponding response based on its unique physical characteristics. The generation and evaluation of challenges and responses create a dynamic interaction which validates not only the device but also prevents the feasibility of rereading attacks. Each challenge is generally unique and randomized, ensuring that even if an opponent captures the pairs of responses to the previous answer, the PUF will generate different responses for future challenges.The challenge mechanism is supported by two critical phases: registration and verification. During the registration phase, the PUF is exposed to a series of challenges from which it generates a set of corresponding responses. These pairs are stored in a secure manner, generally in a trust platform or a secure chip, where they can then be used for authentication purposes. The key aspect of registration is the creation of a relationship of trust, in which the responses create a secure identity linked to the physical characteristics of the PUF. After registration, the verification phase mobilizes the response-response mechanism for authentication entities. In this phase, a challenge is presented to the PUF of the entity tempting authentication, and the resulting response is compared to the stored responses derived during registration. A successful correlation confirms the authenticity of the entity, while the differences indicate attempts to falsify potential or intrusion. This type of protocol is low-cost and vary simple but there are several disadvantages:

- The challenge-response pair cannot be reused to avoid reply attack, meaning that there is the need to store a large number of CRP.
- When the CRP are disclosed the entity can no longer authenticate, and the enrollment phase must be redone.
- no mutual authentication.

Mutual authentication, in which both parties in a communication verify mutual identities before establishing a safe connection, is fundamental. The meaning of mutual authentication cannot be overrated; It prevents unauthorized access and the contrast of man-in-the-middle attacks, thus guaranteeing the integrity and confidentiality of the data transmitted. In an environment in which attacks on communication channels are increasingly widespread, the effectiveness of mutual authentication schemes is fundamental in safeguarding sensitive information against contradictory threats. The described PUF-based mutual authentication protocol is a modified version of an earlier protocol, in which the order of authentication checks is reversed: each entity authenticates only to a legitimate verifier. Each entity has a unique identifier (ID) and its own PUF instance, from which only one response is needed (no more challenge-response pairs are used). Before commissioning, the verifier (Ver) carries out an envelopment step, recording in a database the response of the PUF associated with each ID. For security reasons, the entity's envelopment interface is then blocked or destroyed, so as to prevent further direct acquisition of the PUF responses. The protocol also uses a secure sketch: the entity computes one "sketch" via a binary multiplication with a parity check matrix (derived from a block cipher), while the verifier applies the retrieval procedure (with an error correction algorithm), which requires more computational capacity. Both entities and the verifier have a cryptographically secure hash function and a secure random bit generator to produce nonces (random values used only once), which ensure message freshness and protection against replay attacks. [33]

Protection of Intellectual Property of Hardware

The protection of intellectual property (IP) in hardware has become one of the main challenges in the design of digital systems. With the spread of global supply chains, the possibility of cloning, reverse engineering and counterfeiting of integrated circuits and FPGA designs has grown significantly. In this scenario, Physical Unclonable Functions (PUFs) could be an effective tool for strengthening security and protecting intellectual property. Guajardo et al. (2007) proposed the use of intrinsic PUFs on FPGAs as a defense mechanism against cloning. In fact, FPGAs are particularly exposed to the risk of copying the design: the bitstream that defines the configuration of the programmable logic can be extracted and transferred to other devices, allowing the functionality of a circuit to be illegally replicated without authorization from the designer. The idea is to use the unique and unrepeatable response of the PUF as a cryptographic key to activate or decipher parts of the design loaded into the FPGA. During the provisioning phase, the bitstream is obfuscated or encrypted so only the PUF of the legitimate device is able to generate the correct key to make it executable. If the bitstream is copied and transferred to another FPGA it will not work because the new device will produce a different

response and therefore will not be able to reconstruct the key. [44]

A significant step in the development of intellectual property protection techniques was taken by Zheng and Potkonjak (2014), who introduced architectures based on reconfigurable digital PUFs. The central idea of this approach is to tie the proper functioning of a logic block or design to the unique behavior of the PUF integrated into the chip that hosts it. In this way, even if a hardware project is copied or cloned and loaded onto another device, it will not work properly without the authentic PUF. The novelty of these architectures consists in the ability of the PUF to dynamically modify its behavior. Unlike traditional PUFs, which produce relatively static responses to challenges, reconfigurable digital PUFs allow internal configurations to be varied, making output less predictable and more difficult to model. This feature increases resistance against attacks, particularly those that attempt to clone or simulate PUF responses through machine learning or reverse engineering techniques. This architecture is not limited to protecting against static hardware cloning, but also effective against runtime attacks, such as malicious code injection or firmware tampering. Since enabling design is continuously dependent on PUF output, an attacker cannot simply bypass the initial mechanism: the link between genuine hardware and IP remains active throughout the entire execution. [45] [46]

Other solution has explored the use of PUFs as watermarking mechanisms. By incorporating PUF responses as inherent digital signatures into the hardware synthesis process, it is possible to prove authorship of an IP design even in the event of a legal dispute. Recent studies (Sengupta et al., 2025) have even proposed the use of digital biomarkers combined with PUFs to provide robust and difficult-to-remove watermarks. [47]

PUF could be applied in device binding, where the key derived from the PUF is used to encrypt the code or pattern associated with the IP. In this way, the IP remains bound to the original chip: even if copied, it cannot be executed elsewhere without legitimate hardware (Pan et al., 2022). [48]

Secure boot

The introduction of Physical Unclonable Functions (PUFs) helps to increase the security of boot. Instead of depending on static keys stored in memory, the PUF dynamically generates the secret key every time the boot is performed. This key does not exist in any permanent storage, but is produced directly by the chip's unique physical characteristics, making it impossible for an attacker to clone or extract it. During the provisioning phase, only auxiliary data (helper data) is stored,

which does not directly reveal the key but allows it to be regenerated reliably, even in the presence of environmental variations. Upon boot, the PUF is stimulated to produce its response and, thanks to correction mechanisms such as fuzzy extractors, the original key is reconstructed. With this key, it is possible to decrypt the firmware or verify its digital signature. If the verification result is positive, the start continues; otherwise, the device stops the process, preventing unauthorized code execution. The PUF-based secure boot is therefore an important evolution compared to traditional models. It combines the advantages of encryption with the uniqueness of silicon, eliminating the problem of key storage and strengthening protection against cloning, tampering, and physical attacks. [49] [50] [51]

Vulnerabilities

The large-scale adoption of Physically Unclonable Functions (PUFs) is conditioned by a number of design and implementation challenges that limit their full technological maturity. One of the main difficulties concerns the stability of the responses: since PUFs are based on physical and parametric variations intrinsic to the manufacturing process, they are sensitive to environmental factors such as temperature, supply voltage and aging of the devices. This variability can generate errors in key regeneration and requires the adoption of correction mechanisms (fuzzy extractors, helper data), resulting in an increase in complexity and hardware area.

A further challenge is related to technological scalability. With the progressive miniaturization of production processes, physical variations tend to reduce, making it more complex to obtain sufficiently unique and unrelated answers. In parallel, some PUF implementations require significant additional circuitry, with negative impacts on energy consumption and production costs.

From a security point of view, a crucial problem is the number of challenge-response pairs (CRPs) that can be managed efficiently. PUFs based on high CRP space offer greater resistance to modeling attacks, but lead to storage and communication difficulties in real systems. Conversely, reducing the number of CRPs exposes the PUF to predictability risks.

Two categories of attacks are particularly relevant and have had a strong impact on the development of countermeasures: modeling attacks and side-channel attacks. Modeling attacks typology exploits the observation of a set of challenge-response pairs (CRPs) to train a mathematical or statistical model capable of predicting with high probability the responses of the PUF to new challenges. Machine learning techniques such as Support Vector Machines (SVM), logistic regression, deep neural

networks, and evolutionary algorithms have proven effective against several implementations, most notably Arbiter PUF and Ring Oscillator PUF. In addition to CRP analysis, an adversary can resort to indirect methods by exploiting auxiliary signals produced by the circuit during operation. Side-channel analysis techniques include observing power consumption, electromagnetic emissions, or circuit response times. Such information can reveal correlations between the internal state of the PUF and the responses generated, facilitating the reconstruction of patterns useful for prediction or even physical cloning.

Replay attacks have also been reported, which exploit the lack of secure protocols in managing CRPs: an adversary can store legitimate responses and reuse them in subsequent authentication processes. Finally, the dependence of PUFs on environmental conditions and operating parameters introduces the risk of instability: an attacker can manipulate voltage or temperature to induce systematic errors and compromise system reliability. [52] [53] [54]

Due to the existence of the challenges and vulnerabilities analyzed, the research on Physically Unclonable Functions focused on the elaboration of different mitigation strategies, aimed at increasing reliability and resistance to attacks. To mitigate the predictability of machine learning algorithms, architectural variants of classical PUFs have been proposed. These include PUF XORs and PUF Feed-Forwards, which introduce nonlinearity and make modeling the function more complex. Since modeling attacks require a large number of challenge-response pairs, an effective strategy is to reduce the amount of CRPs accessible to the adversary. This can be achieved by constraining the number of queries allowed or by adopting protocols that derive internal cryptographic keys without making the raw answers public. To reduce information leakage through side channels, masking techniques and randomization of execution times are adopted, in addition to the insertion of artificial noise in consumption signals and electromagnetic emissions. [55] [52] [56]

2.3 Integration between TPM and PUF

2.3.1 Introduction

The objective of this section is to analyze how the integration between TPM and PUF can constitute a complementary approach, capable of combining the functional robustness of TPM with the physical uniqueness guaranteed by PUFs. The integration between TPM and PUF aims to combine the strengths of both technologies, bridging their respective weaknesses. One of the most promising applications is to use PUFs for the generation of the TPM primary key in a way that it is no longer necessary to store the key in a permanent way inside the module,

reducing the risk of extraction by an attacker. Furthermore, PUFs can serve as a physical attestation mechanism of the TPM itself. Since the PUF response is unique and unclonable, it becomes extremely difficult to make a fake or emulated form that is accepted as genuine. This strengthens trust not only in the software certified by the TPM, but also in the physical device that hosts it. Finally, the use of PUFs as a root of trust allows mitigating key-extraction attacks, where an adversary manages to obtain keys stored within the TPM via side-channel or invasive analysis. Since the key derived from a PUF does not exist until the time of its reconstruction and is never stably stored, the attack surface is drastically reduced.

2.3.2 Emerging Architectures

In recent years, academic and industrial research has explored different architectures that combine the standard functionalities of TPM with the intrinsic properties of PUFs, with the aim of building more robust solutions against cloning, key extraction and advanced physical attacks.

PUFchain 4.0

A first approach is the one presented in PUFchain 4.0 (Bathalapalli, Mohanty, Kougianos, et al., 2023). In this model, the key generated by the PUF is used to perform the sealing and unsealing operations of the TPM. In practice, data in the TPM can only be retrieved if the same PUF regenerates the same response, making impossible using the module on a falsified device. Integration is particularly suited to IoT and blockchain scenarios, where it is crucial to link the logical identity of the TPM to the physical identity of the hardware that hosts it. [57]

iTPM

An evolution of the previous paradigm is represented by the iTPM (keyless TPM) model proposed by the same authors (Bathalapalli, Mohanty, Kougianos, Iyer, et al., 2023). In this protocol, primary TPM keys – such as the Endorsement Key (EK) and the Storage Root Key (SRK) – are no longer stored in non-volatile memory. On the contrary, they are dynamically regenerated by the PUF whenever needed. This completely eliminates the risk of key extraction from NVRAM and reduces the attack surface against physical compromise of the module. [58]

TPM2.0-ready

From an industrial point of view, the company PUFsecurity presented a platform "TPM 2.0-ready", the PUF takes on the role of Physical Root of Trust (PhRoT),

while the TPM manages PCR, attestations and cryptographic protocols. The central idea is to replace traditional non-volatile memories (NVMs), used to store keys and certificates, with intrinsic PUFs that derive secrets directly from physical variations in silicon. The PUF response is stabilized through fuzzy extractor algorithms and used to securely power standard TPM functions. The approach aims to reduce complexity, cost and consumption, while maintaining compatibility with TCG standards. [59]

Dual Architecture

An interesting approach, discussed by Sharma, Joshi and Mohanty (2023), is the one of dual architecture, in which TPM and PUF operate as two parallel and complementary entities. In the traditional model, TPM represents the root of logical trust: it stores or manages cryptographic keys, certifies the status of the software via Platform Configuration Registers (PCRs) and provides standardized authentication and encryption services. However, this approach focuses primarily on the functional and logical plane of the system, and does not directly address the possibility that the hardware itself could be counterfeited or replaced. This is where PUF comes in, acting as a kind of physical certificate of the device. Since the response generated by the PUF is closely linked to microscopic variations in silicon, each chip possesses a unique and non-replicable identity. Integrating a PUF alongside the TPM allows the introduction of an independent physical authentication mechanism, which makes it possible to verify not only the logical correctness of the operations of the TPM, but also the authenticity of the hardware hosting it. This combination proves particularly useful in critical contexts such as smart grids, where the compromise of a single device can have lead to dangerous consequences across the entire infrastructure. [60]

2.3.3 Practical applications of TPM and PUF

Internet of Things

A critical point of IoTs is the need to guarantee the authenticity of devices and the protection of cryptographic keys, without being able to count on expensive dedicated hardware solutions. The TPM offers standardized authentication, attestation and secure key management capabilities. The PUF provides a unique, unclonable physical identity, avoiding the need to store static keys in unprotected memories. This approach ensures protection of attacks based on cloning IoT devices and greater lightness of security modules, thanks to the elimination of secure NVMs. [61] [62]

Cloud Computing

In this context, TPMs are already widely used as the root of trust for remote attestation and encryption key protection. However, there remain issues related to possible physical compromise of servers or cloning of security modules. PUFs allow unique keys to be derived for every physical server, making it impossible to emulate a TPM in another hardware context. This type of approach ensures security in provisioning operations: VMs can only be started on authentic machines. [61] [63]

2.3.4 Advantages

- Better strength in cloning; PUFs provide a unique, non-replicable physical identity for each chip. By integrating this mechanism into a TPM, it makes it extremely difficult to create counterfeit module that can be accepted as genuine. This strengthens attestation protocols and prevents attacks based on hardware replacement.
- No need of permanent storage; Models such as iTPM (Bathalapalli et al., 2023) allow the primary keys of the TPM to be dynamically regenerated starting from a PUF, without the need to store them in NVRAM. This reduces the risk of key extraction through physical or side-channel attacks.
- Two level security; Dual architecture (Sharma, Joshi & Mohanty, 2023) ensures dual control: TPM attests to the logical state of the system, while PUF certifies hardware authenticity.
- Compatibility with existing standards; Industrial solutions such as PUFsecurity PUFcc [PUFsecurity, 2022] show that it is possible to integrate PUF while maintaining full compatibility with the TPM 2.0 standard.

2.3.5 Challenges

- PUF responses may vary due to environmental factors such as temperature, supply voltage, or aging of the device. Error correction algorithms (fuzzy extractors) are needed, but these involve additional complexity and possible costs in terms of latency and hardware area (Li et al., 2016).
- While TPM is well defined by the Trusted Computing Group, the use of PUFs is not yet regulated by established standards. This creates difficulties in interoperability and large-scale industrial deployment.
- The integration of PUFs within chips requires specialized hardware design skills and, in some cases, more complex manufacturing processes. This can be an obstacle for companies that produce low-cost devices.

- Despite resistance to cloning, PUFs are not threat-free. Machine learning-based modeling attacks can try to predict PUF responses from large challenge–response pair (CRP) datasets.

Chapter 3

Gem5

3.1 Introduction

"The gem5 simulator is a modular platform for computer-system architecture research, encompassing system-level architecture as well as processor microarchitecture."[01-simulation-background 64, p.8] In the context of hardware security, the creation of devices dedicated exclusively to simulation is extremely expensive; for this reason, software solutions are preferred, which allow a compromise between accuracy and flexibility. Moreover, the increasing complexity of modern computing systems, composed of a lot of interconnected components, makes difficult to conduct accurate analysis without an integrated view of the whole system: gem5 attempts to fill that gap.

gem5 is an open-source simulator widely used in both academia and industry. It was founded about 15 years ago at the University of Michigan with the m5 project, conceived as a framework for event-driven simulation (events, objects, statistics, and configurations), but also as a collection of models of hardware components such as CPUs, caches, buses, and I/O devices. In parallel, the GEMS (General Execution-driven Multiprocessor Simulator) project, specialized in the simulation of multiprocessor systems, was born at the University of Wisconsin. In 2011, the two projects were unified, resulting in the current gem5. Nowadays, gem5 represents a central standard for computer architecture research: it has been cited in over 2900 scientific publications and is employed by major industrial laboratories such as AMD Research, Google, Samsung, ARM Research, as well as numerous university research groups. The community that revolves around gem5 has hundreds of active contributors and thousands of users, with the aim of providing a flexible, extensible tool capable of covering the needs of both academic research and industrial development. A central role is played by the community,

which has hundreds of developers and thousands of users. The gem5 community implements patches, extensions and always-up-to-date templates, but also provides support through forums and shared documentation, ensuring continuous simulator improvement and adjustment to emerging research and industry needs. [64] [65]
A key aspect of gem5 is its modularity, which enables the configuration of targeted experiments including the processor microarchitecture level, such as pipeline and branch predictor design, to the full system level, including memory controllers, devices, interconnections, and operating systems. This flexibility allows simulations to be adaptable to different scenarios, including the study of new caching techniques, the analysis of memory coherence protocols, and the evaluation of applications in multicore and manycore environments. The primary objective of gem5 is to provide a integrated and realistic platform for simulating computing systems by trying to balance accuracy, flexibility, and efficiency. [64]

3.2 Architecture of gem5

3.2.1 Discrete event simulation model

Gem5 is based on a discrete event simulation model. In this paradigm, simulation time advances as a function of events occurring within the system, rather than continuously. Each event represents an elementary action (memory access, completion of an instruction, arrival of a packet on a bus) and is programmed to occur at a certain simulated instant of time.

Gem5's simulation engine manages a global queue of events, ordered temporally, meaning that are ordered based on the time in which their execution must be perform. Execution proceeds by extracting the event into the head of the queue from time to time, updating the system state, and queuing any new events generated by the operation. This approach allows you to obtain faithful and flexible simulations, as it allows you to accurately represent both the competition between components and the asynchronous interactions between devices. [64]

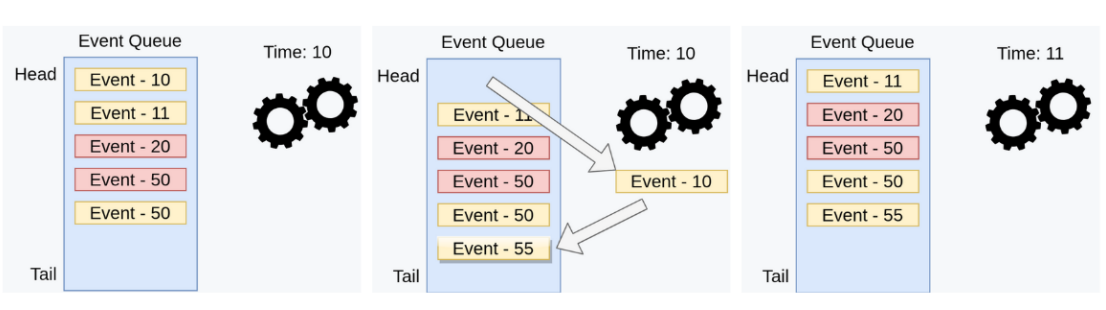

Figure 3.1: Discrete event simulation model-photo taken from [01-simulation-background 64, p.33]

Figure 3.1 illustrates how the Discrete Event Simulation model adopted by gem5 works. At the current time $t = 10$, the event at the head of the queue (Event - 10) is extracted and processed. During processing, the event may generate other future events (in the example Event - 55), which are immediately inserted into the queue in an ordered position relative to the timestamp. Once execution is complete, the simulator updates the simulated time to the next available event (Event - 11), and the process is repeated.

3.2.2 SimObject

At the heart of the gem5 architecture is the concept of SimObject. Every simulated component — be it a CPU, cache, bus, or I/O device — is modeled as SimObject. SimObjects play a dual role:

- common abstraction, as they define the basic interface that all simulator modules have in common;
- event containers, as each SimObject can generate, manage and respond to events inserted into the global queue of the simulator.

Each SimObject can be configured through specific parameters (cache size, CPU frequency, replacement policies, etc.) and interacts with other components via ports and links. This makes gem5 highly modular: new components can be added by simply defining new SimObjects, reusing much of the common infrastructure. From an implementation point of view, SimObjects constitute the basis for building complex configurations: by combining different objects it is possible to model entire computing systems, from the single core with private memory to multiprocessor platforms complete with devices and peripherals. [64]

3.2.3 CPU Models

gem5 provides different CPU models, which allow you to simulate microarchitectures with different levels of detail. They differ mainly in the compromise between accuracy and simulation speed. The main CPU models supported are [65]:

- AtomicSimpleCPU: is the simplest and fastest model. It does not simulate the time of operations in detail, but uses an "atomic" memory model: each access is considered instantaneous and returns the data immediately, applying only an estimated latency. This approach makes it ideal for warm-up tasks, debugging, or getting very crude estimates. However, it is not suitable for studies that require an accurate representation of timing or microarchitectural effects.

- TimingSimpleCPU: is an in-order core, it executes instructions sequentially, but unlike the atomic model it simulates memory access times and pipeline latencies more realistically. It is slower than the AtomicSimpleCPU, but offers more useful information in terms of time performance without arriving at the complexity of an out-of-order model.

- MinorCPU: represents a more detailed and microarchitectural in-order model. It implements a four-stage pipeline (fetch, decode, execute, writeback), allowing to analyze phenomena such as hazards, stalls and forwarding mechanisms. It is particularly useful for those who want to study in depth the behavior of simple pipelines, such as those typical of in-order ARM processors. It is slower than TimingSimpleCPU, but provides much greater fidelity than actual behavior.

- DerivO3CPU: is the most complex and detailed model of gem5. It simulates a full out-of-order processor, including advanced mechanisms such as register renaming, dynamic scheduling, speculative execution, branch prediction, and Reorder Buffer (ROB). This model is the most expensive in terms of simulation time, but at the same time, the most accurate. It is used to study modern high-performance architectures.

3.2.4 Memory Hierarchy

The memory hierarchy is another key element to perform simulations in gem5. It includes models for multi-level caches (L1, L2, L3), memory controllers and coherence protocols, fundamental for the simulation of multiprocessor systems. Gem5 offers great flexibility in defining cache size, latencies, and replacement policies. Furthermore, it is possible to configure different memory coherence protocols (MSI, MESI, MOESI), allowing to study the behavior of parallel applications and the

traffic generated by synchronization mechanisms. The simulator also enables to connect custom DRAM models and memory controllers, facilitating the study of new techniques for managing main memory. [65] [64]

3.2.5 I/O Devices

In addition to processors and memory, gem5 includes a wide range of I/O devices and modules, allowing simulation to be extended to complete systems. The main elements include:

- buses and interconnections, such as crossbars;
- I/O devices including disk controllers, network interfaces, UARTs;
- support for basic peripherals needed to run operating systems in full-system mode.

This capability makes gem5 more than just a CPU simulator: it can represent entire computing systems, enabling experiments that also include kernel management, drivers and hardware-software interactions. [65][64]

3.3 Type of simulation

Gem5 offers different simulation modes that meet different experimental needs. The simulation mode is chosen based on the level of detail required, the type of analysis to be conducted and the trade-off between accuracy and simulation performance. In particular, gem5 supports two main approaches:

- System Emulation (SE) enables running individual applications without the involvement of a full operating system;
- Full System (FS), which simulate the entire hardware-software stack, including operating system, drivers and peripherals.

Both modes are based on the same discrete event simulation engine and use the same components (CPU, memory, devices). However, they differ in the level of abstraction adopted and the experimental scenario that is intended to be analyzed.

3.3.1 System Emulation

System Emulation (SE) mode allows you to simulate the execution of individual applications without including a complete operating system. In this scenario, gem5 provides the simulated program with a simplified environment, in which system calls

are intercepted and managed directly by the simulator and not by the operating system.
This mode has several advantages:

- Speed: The absence of the operating system significantly reduces complexity, allowing faster simulations.

- Simplicity of configuration: There is no need to load kernels or disk images, but just provide the simulator with the program executable.

- Focus: This approach is particularly advantageous for studying CPU or memory behavior in relation to a single workload, as it avoids interference from other system processes.

However, SE mode presents several limitations. It does not enable analysis of interactions between software and the operating system, lacks support for complex drivers or devices, and is unsuitable for simulations that require a realistic system context. [64][65]

3.3.2 Full System

Full System (FS) mode allows to simulate a complete calculation system, including processors, memory, I/O devices and operating system. In this case, gem5 runs a real kernel image (for example, Linux) and allows the boot and management of applications as in a real system.
The main advantages of this mode are:

- Fidelity: allows to reproduce the behavior of a real system in detail, including kernels, drivers and hardware-software interfaces.

- Flexibility: Makes possible to analyze complex applications, multiprocessor systems, memory coherence protocols, and interactions between multiple devices.

- Realistic Experimentation: is the ideal choice for assessing the impact of architectural choices on complete systems and real software environments.

The downside is the high computational cost. FS mode takes longer to configure and simulate, because it's necessary to load disk images, kernels, and configure devices. Furthermore, the complexity of the simulation results in significantly longer execution times than in the SE mode.[64][65]

3.4 Extendability of gem5

A primary strength of gem5 is its high extensibility. Due to its open-source nature and modular design, based on SimObject and the discrete event model, gem5 allows new features, models, or protocols to be easily integrated, without rewriting the entire infrastructure. This design makes gem5 an effective tool to support research in diverse domains.

3.4.1 Open-source nature and the role of the community

The gem5 community plays a central role in the extension process. There are hundreds of active contributors that regularly propose new patches, templates, and improvements, which are integrated after an open review process.
The use of collaborative platforms (such as GitHub and dedicated mailing lists) allows:

- the rapid sharing of new features;
- peer review, which ensures code quality and consistency;
- the public availability of extensions developed in both academic and industrial fields.

This collaborative mechanism led to the inclusion of advanced CPU models, memories, coherence protocols, and interconnections that were unlikely to be developed by a single research group. [64][65]

3.4.2 Extension mechanisms

Each new component (e.g. an accelerator, a memory protocol, a new bus) is integrated as a SimObject; it uses common APIs and integrates with the global event queue. Gem5 uses scripts in Python for configuring simulations, enabling easy addition of parameters and creation of complex scenarios without requiring recompilation of the simulator, which requires a lot of time. In addition, gem5 provides an extensible infrastructure to collect detailed statistics (such as instruction counters, latencies, throughput, etc.) that can be expanded with new researcher-defined metrics. Finally, gem5 can be integrated with other tools, such as simulators for DRAM memory, networks on chips, or energy analysis tools like McPAT, allowing to further expand the search domain. [64][65]

Some examples of extensions that are commonly developed with gem5 are:

- new CPU models, such as specialized microarchitectures or RISC-V prototypes;

- custom cache coherence protocols, useful for studying multiprocessor systems;
- emergent memory models (such as PCM, ReRAM, HBM);
- dedicated hardware accelerators (such as for artificial intelligence, encryption, or graphics);
- new interconnections and on-chip networks (NoCs).

3.5 Workfolw of simulation in gem5

The use of gem5 is not limited to understanding its internal architecture. It also requires a very precise simulation workflow. This process includes selecting the operating mode, configuring components, loading workloads, and finally, collecting and analyzing statistics. This section will describe the basic steps to set up a simulation correctly and will present some typical examples, such as running standard benchmarks like the SPEC suite. The first step is to establish the most suitable simulation mode. The choice is generally between System Emulation (SE) and Full System (FS). SE is used for fast application or microbenchmark execution. FS is needed when analyzing more complex scenarios, including operating systems, drivers, and multiprocessor configurations. Once you have selected the mode, you move on to configuring the simulation environment. This is primarily done via scripts in Python. These scripts allow you to flexibly define the target architecture features. At this stage, specify the number and type of CPU to be used and the structure of the memory hierarchy with cache and controller. In FS mode, you can also include devices and peripherals. This step represents the heart of the workflow, as it allows you to model the hardware configuration you intend to analyze. The next step is loading the workload, which varies depending on the chosen mode. In System Emulation, provide the simulator with the executable of the program to be executed. In Full System mode, prepare an operating system kernel, for example Linux, and a disk image containing the applications to be started. After the simulation begins, gem5 generates statistics that are saved in the stats.txt file. Key metrics include instructions executed, IPC, cache miss rate, average memory latency, and overall system throughput. Analyzing these data is the final step in the workflow and supports conclusions about the performance and behavior of the simulated configuration. [64][65]

3.5.1 Examples

The use of gem5 lends itself to a wide range of scenarios, ranging from simple microbenchmarks up to complex, standardized benchmarks. A first category of experiments is represented by microbenchmarks, small programs specifically created

to solicit a specific component of the system. For example, synthetic loads that generate regular or irregular memory access patterns can be designed to analyze cache behavior, or branch sequences can be designed to evaluate the effectiveness of jump prediction mechanisms. These tests have the advantage of being quick to perform and easily interpretable, thus representing a helpful tool in the early stages of developing and can be used to validate simulation configurations.
For more systematic and precise analyses, standardized benchmarks such as the SPEC CPU suite are used. This collection in the scientific community enables the objective measurement of processor and memory hierarchy performance through workloads that represent real or nearly-real-world applications. Running SPEC in gem5 provides reliable metrics, such as average IPC, memory access latencies, and cache miss rates, offering a solid foundation for comparing different architectural configurations. [64][65]

3.6 Adding a new SimObject in gem5

The extensibility of gem5 is also realized through the possibility of introducing new SimObjects, that is, the fundamental entities with which all the components of the simulator are modeled, ranging from CPUs to caches and I/O devices. Each new element intended to be added to the system, whether a functional unit, an accelerator, or a memory-mapped peripheral, is implemented as a SimObject that interacts with others via events and communication ports. The extension process follows a well-defined path. First, it is necessary to develop the structure in C++, defining a class that inherits from the basic classes of gem5, specifically from SimObject or its specializations, such as MemObject or BasicPioDevice. Within the classroom, the internal state of the component and the methods for handling read, write, or process operations are declared, along with the logic for generating and handling events. In this phase that the basic behaviors of the new object are established, i.e. the reactions to certain stimuli and the production of new events in the global queue of the simulator. A crucial aspect of integration concerns communication interfaces. SimObjects in gem5 interact with each other via ports, which represent connection channels with other modules. After making the part in C++, the corresponding representation in Python needs to be defined, so that the new SimObject can be invoked within the configuration scripts. At this stage, configurable parameters, such as addresses, dimensions, or latencies, are exposed, allowing the user to modify them without intervening in the source code. This mechanism, characteristic of gem5, provides high flexibility and allows the same object to be reused in different scenarios. Once the definition in C++ and Python is complete, it's essential to update the gem5 build system, based on SCons, to include the new files. The recompilation allows the new component to

be definitively integrated into the simulator, making it available like any other native SimObject. The last step concerns validation. To check that it is working properly, the new object is placed in a test configuration and linked to existing modules. In the case of a memory-mapped device, for example, it is possible to connect it to the bus and test its behavior using programs that read and write to the corresponding addresses. Validation is not limited to functional control, but must also ensure temporal consistency with the discrete event model that regulates the entire simulator. [64][65]

3.7 TPM Simulations

3.7.1 Motivations

A crucial aspect in modern architectural research is the development of accurate and modular simulation environments. Among the most popular simulators, gem5 is distinguished for its ability to model complex systems at different levels of abstraction, offering the researcher an in-depth control over both the behavior of the microarchitecture and the interaction between hardware and software components. Despite its flexibility, gem5 does not include models for hardware security devices, such as trusted module. This absence limits the ability to realistically analyze how security features affect a system's overall performance or how they interact with the processor and peripherals during software execution. The introduction of a component dedicated to security allows the potential of the simulator to be extended, enabling the testing of scenarios that are difficult to observe on real hardware. The integration of a security module in gem5, like the one developed in the present work, responds to three main research needs. First, it aims to fill a structural void in the architectural simulator landscape by introducing a device that can handle trusted operations and protect sensitive data. This allows the impact of such operations on the simulated system to be studied in a systematic way, measuring its computational cost and influence on overall latency. Secondly, the simulation of a security component allows to analyze the trade-off between security and performance, evaluating how protection functions — such as the generation and management of cryptographic keys or the attestation of software components — affect the execution time and the efficiency of the system. In real scenarios, such operations introduce non-negligible overhead, which can only be studied in a controlled way via a reproducible simulation environment. Third, adding a trusted module inside gem5 provides an open and customizable experimental environment for hardware security research. Unlike proprietary simulators or closed implementations, an open-source model allows to freely extend and modify security functions, introduce new authentication logic, or explore emerging mechanisms such as Physical Unclonable Functions (PUFs). This approach paves the way for

the creation of virtual prototypes and the analysis of new trusted architectures without the need to have expensive or unavailable physical hardware. Finally, the availability of a safety model integrated into gem5 favors the reproducibility and comparability of experimental results. In the academic field, the possibility of replicating experiments in a completely open-source environment represents a fundamental requirement for validating results and building a shared knowledge base. The adoption of open simulators such as gem5 therefore allows reliable tools to be made available to the community for the joint study of performance and security in advanced architectures.

3.7.2 Correlated Works

In recent years, architectural simulation has evolved from a simple performance analysis tool to a platform capable of also supporting research in the field of hardware security. In this context, various works have attempted to model or emulate the behavior of the Trusted Platform Module (TPM), also with different approaches and purposes. However, most of these attempts favored emulation at the operating system or firmware level, rather than real module integration within an architectural simulator such as gem5.

One of the first relevant contributions is represented by TPM-SIM, a framework designed to analyze the performance of TPM in combination with the CPU, through the execution of micro-benchmarks that reproduce the most common commands of the module, such as key generation and digital signature. The authors show how the simulation of TPM operations allows to evaluate the load and overhead induced on the system, highlighting how parallelization or batching of requests can improve throughput in the presence of competing applications (Hu et al., 2010). [66]

A conceptual evolution of this approach is proposed by the TPMSim (Extensible TPM Simulator) project, which introduces more accurate time modeling, allowing the exploration of what-if scenarios in the trust domain. The simulator implements a modular architecture capable of precisely emulating latencies and command execution times, offering a flexible platform for experimental analyses. The authors show how accurate time simulation can reproduce the behaviors of real devices with average error margins of less than one percent (Tate et al., 2009). [67]

Since 2013, Pirker and Winter have explored a different approach, based on the semi-automatic generation of TPM 2.0 simulators directly from the TCG specifications. Their proposal links the simulation process with hardware synthesis on FPGAs, suggesting a methodology for moving from abstract model to concrete

prototyping. Such methodology reinforces the idea that an accurate and formally derived TPM model can serve as an intermediate tool between software simulation and physical realisation. [68]

libtpms and swtpm

In the emulation domain, the libtpms and swtpm projects, developed and maintained mainly by Stefan Berger (IBM Research), are now the reference base for Trusted Platform Module virtualization in software environments. Both are compatible with the specificications issued by TCG for TPM 1.2 and TPM 2.0 and constitute the TPM backend used by QEMU, KVM, Proxmox, and in general by many Linux-based hypervisors. libtpms is a library implemented in C that performs functional primitives of TPM: it internally manages key tables, monotonic counters, non-volatile memory (NVRAM), Platform Configuration Registers (PCRs), and the command interpreter. Each TPM command is processed by the library, which returns a binary response structured according to the standard format.
The library does not interface directly with hardware or the operating system, but exposes an API that can be called by a higher-level emulator, such as swtpm. In this way, libtpms represents the "logical engine" of the TPM, while swtpm provides the communication infrastructure with the virtualized system. swtpm (Software TPM Emulator) is a user-space application that encapsulates libtpms and offers the external system a virtual TPM device accessible via several channels:

- UNIX or TCP sockets (used by QEMU to connect to the emulator);
- device character (`/dev/vtpm0`);
- or a character device in userspace (CUSE) interface that allows you to map the TPM as a kernel device.

Since swtpm operates as an external process to the simulator or hypervisor, communication between virtual CPU and TPM occurs through an asynchronous, typically socket-based, messaging channel. Each command involves serializing data, switching to the kernel, performing context switching, and deserializing on the swtpm side. As a result, the observed latency is not the device's actual latency but is affected by inter-process communication overhead. Moreover, the TPM response times are not synchronized with the simulator's virtual clock. In QEMU, for example, the virtual CPU can advance through simulated time independently of the swtpm process, leading to a discrepancy between simulated time and real processing time. This makes it difficult to conduct accurate timing analysis to studying timing attacks or accurately measuring the security overhead introduced by the module. A further limitation concerns the lack of internal visibility: swtpm and libtpms operate as black boxes. It is not possible to directly access the internal state of the TPM

(PCR, counters, NVRAM) nor to dynamically modify its behaviors to simulate faults, fault injection or software attacks. This limits their usefulness in research areas oriented towards architectural or experimental simulation. Despite these limitations, libtpms and swtpm are essential tools for testing complete software stacks (drivers, OS, applications) that require a TPM. They are also widely adopted by open-source hypervisors and cloud-native projects, and provide an established foundation for integrating trusted functionality into virtualized systems. [69] [70]

3.7.3 Simulation Architectures

In the context of architectural simulation, the integration of security modules such as the Trusted Platform Module can be defined following different approaches, depending on the desired level of fidelity and the degree of interaction required with the simulated processor. The most used architectures are: external co-simulation, integrated internal model, and hybrid approach.

In the first scenario, the TPM module is implemented as an independent process that communicates with the main simulator through a standard interface channel, such as socket or pipe. This approach is typically used in virtualization systems such as QEMU, which rely on the swtpm emulator to provide a virtual TPM compatible with the TCG 2.0 specification. Although this architecture ensures excellent functional compatibility and allows the execution of real software that interacts with the TPM, it has limitations from an architectural point of view. In particular, the TPM module operates outside the time domain of the simulator, and inter-process communication introduces delays that are difficult to control. This makes the study of temporal or performance phenomena complex and limits internal visibility into cryptographic operations. [71]

The second approach consists in integrated internal model, in which the TPM is implemented as a native component of the simulator. In environments such as gem5, such integration can be accomplished by defining the TPM as SimObject, that is, an entity with behavior, state and communication interfaces internal to the simulator. This solution allows the device to be modeled directly into the processor time domain and connected to the system bus via an MMIO interface. In this way, each operation sent by the core is managed synchronously, allowing accurate temporal simulation and a realistic representation of the interactions between CPU and trusted module. The internal implementation also offers high flexibility: it is possible to access the TPM's internal state, introduce controlled faults and observe the real latency, or add new features, such as authentication modules or PUFs.

The third approach combines elements of the previous two. In this case, the

main logic of the TPM is maintained in an external component, but communication with the simulator is synchronized via timed exchange mechanisms or dedicated connectors. This solution is used in some hardware/software co-simulation environments, where a functional emulator is connected to a system simulator via deterministic data exchange protocols (e.g., TLM or SystemC). While this model allows existing implementations to be reused, maintaining time synchronization and managing communication makes the design complex and less flexible.

In the specific case of gem5, the choice of an internal architecture is the most effective trade-off between accuracy and control. It allows to extend the simulator with fully customized safety modules, maintaining synchronization consistent with the simulation cycle and guaranteeing the reproducibility of the results. This approach constitutes the conceptual basis of the model developed in this work, which will be described in the next chapter. [72]

3.7.4 TPM Simulation Challenges

Simulating a security module such as the Trusted Platform Module in an architectural environment presents a number of technical and conceptual challenges that arise from the very complexity of the TPM 2.0 specification and the trusted nature of that component. An accurate model must in fact combine functional fidelity, i.e. the correct implementation of the commands and data structures defined by the Trusted Computing Group (TCG), with temporal coherence, i.e. the ability to realistically represent the sequence and duration of the operations performed by the module.
A first complex aspect is represented by the modeling of internal memory and trusted resources. The TPM has volatile and non-volatile memory areas, which are used to store keys, monotonic counters, and configuration data. Reproducing such structures in simulation implies the need to implement persistence and data protection mechanisms consistent with TCG specifications. Moreover, randomness management is an obstacle: in a deterministic environment such as gem5, generating random numbers must be simulated without compromising experiment reproducibility while maintaining statistically realistic behavior.

Another challenge relates to managing the functional complexity of TPM 2.0. The official specification includes hundreds of distinct commands, each with specific parameters, permissions, and access policies. Fully implementing this set would be burdensome.

From a security point of view, the simulation of adversarial behavior constitutes a further element of complexity. Reproducing scenarios such as timing attacks, fault

injection or PCR register manipulations requires a flexible and observable model, capable of exposing the internal state of the TPM while preserving the consistency of the simulation environment. [73]

Chapter 4

Implementation

This chapter describes in detail the practical implementation of the Trusted Platform Module (TPM) model integrated in the gem5 simulation environment. After presenting the development tools used, will be illustrated how the simulated TPM device was made within gem5, highlighting the extensions and modifications to the simulator. The main steps of the implementation will then be described: the design of the hardware communication interface (mapped memory) between RISC-V processor and TPM, the internal management of TPM commands according to TPM 2.0 specifications defined by TCG, and the modifications performed to support PUFs.

4.1 Introduction

The TPM module was designed as a simulated safety device, integrated directly into the system as a memory-mapped peripheral, accessible from the RISC-V CPU via read and write operations on dedicated registers. The architecture follows a modular model that mirrors, in a simplified form, the functional structure of a real TPM 2.0, including the main logical sections: CRB (Command Response Buffer), FIFO, PCR, NV Memory, and the PUF extension. All these components are implemented in the C++ source code within the `src/dev/` directory of gem5, where the device logic and the management of its interactions with the simulated processor resides.

The simulated system configuration was realized in the `configs/` directory of gem5, through dedicated Python scripts that define the RISC-V system architecture and link the TPM module to the main components. Furthermore, it was possible to implement simulations in bare-metal mode, directly testing the communication between the CPU and the TPM device within the gem5 environment.

4.2 Development Environment and Build Process

The implementation of the TPM was developed and tested in a gem5 environment, which was configured for the 64-bit RISC-V architecture. Gem5 is a modular architecture simulator that allows to add custom hardware devices using extensions in C++ and Python. In particular, TPM has been implemented as a memory-mapped I/O device using the basic classes of gem5, in particular BasicPioDevice. The code is written in C++ and integrated into the gem5 source code; Python scripts provided by gem5 (which define system parameters and map device addresses) were used to configure and dock the TPM device to the simulated system. SCons is the build system used by gem5 to manage the modular compilation of the simulator. Each component of the project is described by a file `SConscript`, which specifies dependencies, the compilation parameters and libraries to be linked. In the case of TPM, it was therefore necessary to extend the `SConscript` file to include the `libcrypto` libraries of OpenSSL. the OpenSSL library (version 3.0) was used to implement the cryptographic primitives required by the TPM, such as SHA-256 hash computation, cryptographic random number generation, and HMAC. The integration of OpenSSL into gem5 required modifying the build system, so the SConscript file, to link libcrypto libraries.

Listing 4.1: Extension of the file SConscript to use OpenSSL

```
# Link with OpenSSL
if env['PLATFORM'] != 'darwin':  # Linux only
    env.ParseConfig('pkg-config --cflags --libs openssl')
```

This code allows SCons to automatically include the compilation and link options needed to use OpenSSL, avoiding having to manually configure paths or libraries. The host operating system for compiling and running simulations is Ubuntu 20.04 LTS 64-bit, with GNU C++ toolchain for gem5 and RISC-V GCC toolchain to compile any test bare-metal programs run in the simulator.

4.3 TPM Model Architecture in gem5

4.3.1 Integration as a BasicPioDevice device

On the implementation side in C++ the TPM module was realized by defining a new SimObject in gem5. Creating a SimObject allows to have a C++ object inside the simulator that interacts with the rest through standard interfaces (doors, events, etc). The TPMDevice class is derived from BasicPioDevice and overwrites the key methods read() and write() to intercept CPU accesses to registers of the simulated TPM. At initialization, the TPMDevice constructor sets the device occupied memory size (pioSize) and base address (pioAddr) by taking them from

the configuration parameters (provided by the Python script of gem5). In this way, gem5 knows that every address range access [pioAddr, pioAddr+pioSize) must be handled by TPM device.

4.3.2 Memory-mapped communication interface

The TPM module communicates with the simulated CPU through a Memory-Mapped I/O interface (MMIO). In gem5, this means that to the TPM device is assigned a range of addresses in the physical address space of the simulation; when the CPU accesses those addresses, the calls are intercepted and managed by device logic instead of standard memory.

The address range reserved for TPM records is `0x10001000 - 0x10001FFF`. That's 4 kB of space (4,096 B), analogous to the size of memory pages, to map registers. This space is subdivided further to implement both the FIFO interface classic TPM 1.2/2.0 (based on the TIS – TPM Interface Specification standard) is the new interface CRB (Command Response Buffer) introduced with TPM 2.0 to improve performances.

In particular, in the `read(PacketPtr pkt)` method (and symmetrically in `write(PacketPtr pkt)`), the offset relative to the base address `pioAddr` is calculated and the location determined as `loc = offset/LocalitySpacing`; next, the function checks whether the offset corresponds to a register belonging to the FIFO or CRB interface, invoking respectively, `handleFifoRead/Write` or `handleCrbRead/Write`.

Listing 4.2: implementation of the function read()

```
1  Tick TPMDevice::read(PacketPtr pkt)
2     {
3         Addr addr = pkt->getAddr();
4         Addr offset = addr - pioAddr;
5         unsigned loc = offset / LocalitySpacing;
6
7         if (loc >= MaxLocalities)
8         {
9             panic("Invalid locality read: %u", loc);
10        }
11
12        Addr locality_offset = offset % LocalitySpacing;
13
14        if (isFifoOffset(locality_offset))
15        {
16            return handleFifoRead(pkt, loc);
17        }
```

```
18
19          if (isCrbOffset(locality_offset))
20          {
21              return handleCrbRead(pkt, loc);
22          }
23
24          panic("Unhandled TPM read: locality %u offset %#x size %u",
25                loc, locality_offset, pkt->getSize());
26      }
```

The auxiliary functions `isFifoOffset()` and `isCrbOffset()` recognize the relevant registers of the two interfaces: for FIFO, for example `TPM_ACCESS_REG_OFFSET`, `TPM_STS_REG_OFFSET` and the data window between `FIFO_DATA_BASE_OFFSET` and `FIFO_DATA_END_OFFSET`; for CRB, the registers as `CRB_CTRL`, `CRB_CMD_ADDR_` and `CRB_RSP_ADDR_`. This logic allows maintaining compatibility with drivers that still use FIFO, while supporting the most modern and high-performance CRB interface. In case of unrecognized out-of-range or offset accesses, the device generates an exception via `panic()`, ensuring deterministic behavior and facilitating debugging during simulation.

Locality Management

In the Trusted Platform Module, the concept of locality represents a logical isolation mechanism that allows multiple software components to interact with the TPM in a controlled and mutually exclusive way. In particular, localities (from 0 to 4, for a maximum of five) allow distinct entities — such as BIOS, operating system, hypervisor, or privileged applications — to access the TPM without interfering with each other. Each location has its own set of registers and internal status, and can be activated or released via dedicated registers.

In the simulated device, the main data structure implementing this logic is `TPMRegisters`, defined as a collection of registers for each interface (FIFO and CRB), replicated for all localities:

Listing 4.3: Management of Locality

```
std::array<TPMRegisters, MaxLocalities> tpmRegs;
static const int MaxLocalities = 5;
static constexpr Addr LocalitySpacing = 0x10000;
```

Where `TPMRegisters` is implemented as follows.

Listing 4.4: TPMRegisters

```
    struct TPMRegisters
    {
      // FIFO interface
```

```
4         uint8_t access = 0;
5         uint8_t active = 0;
6         uint8_t sts = 0;
7         uint8_t intf_capability = 0;
8         uint8_t interface_id = 0;
9         std::vector<uint8_t> fifo;
10        uint32_t fifoExpectedSize = 0;
11        size_t fifoBytesReceived = 0;
12        bool fifoReceiving = false;
13        // CRB interface
14        uint32_t crb_ctrl_req = 0;
15        uint32_t crb_ctrl_sts = 0;
16        uint32_t crb_ctrl_cancel = 0;
17        uint32_t crb_ctrl_start = 0;
18        uint32_t crb_cmd_size = 0;
19        uint32_t crb_rsp_size = 0;
20        uint64_t crb_cmd_addr = 0;
21        uint64_t crb_rsp_addr = 0;
22        std::vector<uint8_t> crb_cmd_buf;
23        std::vector<uint8_t> crb_rsp_buf;
24        CrbState crb_state = CrbState::Init;
25        // Locality registers (CRB-specific)
26        uint8_t loc_ctrl = 0;
27        uint8_t loc_sts = 0;
28    };
```

In the code 4.3 there is the `LocalitySpacing` field which defines the distance, in bytes, between the address space reserved for two consecutive localities. In this way, each locality is mapped into a distinct range of the MMIO address space. The actual offset associated with locality n is calculated as `offset = loc * LocalitySpacing`. Indeed, if the base address of the TPM is `0x10001000`, the locality 0 will occupy the addresses `0x10001000-0x10010FFF`, the locality 1 the next interval `0x10011000-0x10020FFF`, and so on, up to a maximum of five blocks of 64 kB.

Each element of the tpmRegs array contains the logical registers for its locality, including registers specific to control and status of locality in CRB mode, defined as aliases on the same offset `0x008C`:

Listing 4.5: CRB lolcality control

```
1     static constexpr Addr LOC_CTRL_OFFSET = 0x008C; // write, to
    activate a locality (0x1 for request, 0x0 for release)
2     static constexpr Addr LOC_STS_OFFSET = 0x008C;  // read, to check
     the status of the locality (0x1 if the locality is active, 0x0 if
     not)
```

The `LOC_CTRL` register allows the host software to request or release a locality by

setting a specific value: writing `0x1` to `LOC_CTRL_OFFSET` triggers the locality, while writing `0x0` drops it. Instead, the `LOC_STS` register, read from the same address, allows checking whether the locality is currently active (returning `0x1`) or inactive (`0x0`). Such address aliasing faithfully reproduces the behavior of the real TPM, where the same physical register is read and write accessible for different functions.

Operational management of localities occurs during read and write operations on the FIFO or CRB. In the case of FIFO, the `handleFifoRead()` function calculates the address offset within the current locality and selects the corresponding register structure:

Listing 4.6: Calculation of the locality

```
1     Addr offset = pkt->getAddr() % LocalitySpacing;
2    auto &regs = tpmRegs[loc];
```

During simulation, only the active locality (indicated by the global `active_locality` field) is enabled to execute TPM commands; any access from other inactive localities is ignored or generates an exception (panic("Invalid locality")). This mechanism guarantees that each communication session between host and TPM is mutually exclusive, faithfully reproducing the localities arbitrage logic required by the TIS standard.

FIFO Interface

In FIFO mode, the TPM simulates the classical register-based interface: Access, Status, and Data FIFO, which is historically used by TPMs on LPC, SPI, or I^2C. FIFO mode management is implemented mainly in the two functions `handleFifoRead()` and `handleFifoWrite()`, which represent the two fundamental communication paths between the simulated CPU and the TPM device. The first is invoked when the host performs a *read* operation on one of the MMIO addresses belonging to the FIFO range, while the second handles *write* operations. Basically, `handleFifoWrite()` receives and interprets data from the host — as a request to activate a locality, bytes of a TPM command, or the `CommandReady` flag setting — while `handleFifoRead()` returns data that the TPM makes available, such as device status or response bytes.

Each locality has a dedicated address window, and within it, the FIFO mode registers are mapped at fixed offsets with respect to the base address. In particular:

Listing 4.7: FIFO Registers

```
1 static constexpr Addr TPM_ACCESS_REG_OFFSET          = 0x0000;
2 static constexpr Addr TPM_INTF_CAPABILITY_OFFSET     = 0x0014;
3 static constexpr Addr TPM_STS_REG_OFFSET             = 0x0018;
```

```
4 static constexpr Addr FIFO_DATA_BASE_OFFSET        = 0x0024;
5 static constexpr Addr FIFO_DATA_END_OFFSET         = 0x0027;
6 static constexpr Addr TPM_INTERFACE_ID_OFFSET      = 0x0030;
```

These offsets allow quickly distinguishing in which register the incoming packet should be handled. The communication, through the FIFO, between the TPM and the software happens in the following way.

Activation of locality. First of all, software (for example, a TPM driver) must "require" the use of a locality by writing the value `0x20` to the `Access(0)` registry (bit `TPM_ACCESS_REQUEST_USE`). When bit `0x20` is written, a check is performed to activate the selected locality and deactivate any other.

Listing 4.8: Request of Locality activation

```
1             // Host is requesting to activate this locality
2             for (auto &r : tpmRegs)
3                 r.active = false;
4             regs.active = true;
```

When the locality is active, the TPM performs a read of the Access register and returns `0x20` (the `TPM_ACCESS_ACTIVE_LOCALITY` bit) to confirm that the host can use the required locality.

Writing the command to FIFO. The software prepares a TPM command compliant with the TPM 2.0 standard (2 bytes of tags, 4 bytes of size, 4 bytes of command code, any handles and parameters). The TPM sees it as a blob of N bytes; the host writes it byte-per-byte (or in blocks) in the FIFO Data window. With each write, `handleFifoWrite()` appends the byte in the `regs.fifo` buffer and updates the `fifoBytesReceived` and `fifoExpectedSize` counters. When at least 6 bytes are received, the device can extract the overall command size from the `[2..5]` fields of the TPM header and use it to determine how many bytes it expects in total. Until all the expected bytes have arrived, the TPM keeps the `EXPECT` bit active (`0x02`) in the `STS` register, to signal that other input is expected:

Listing 4.9: Write FIFO

```
1     else if (offset >= FIFO_DATA_BASE_OFFSET && offset <=
  FIFO_DATA_END_OFFSET) {
2         // Ignore writes while a response is still pending
3         if ((regs.sts & 0x01) && !regs.fifoReceiving) {
4             pkt->makeResponse();
5             return pioDelay;
6         }
7
8         if (!regs.fifoReceiving) {
9             regs.fifo.clear();
```

```
10              regs.fifoReceiving = true;
11              regs.fifoExpectedSize = 0;
12              regs.fifoBytesReceived = 0;
13          }
14
15          if (regs.fifoExpectedSize && regs.fifoBytesReceived >= regs.
    fifoExpectedSize) {
16              pkt->makeResponse();
17              return pioDelay;
18          }
19
20          regs.fifo.push_back(val);
21          regs.fifoBytesReceived++;
22
23          if (regs.fifoBytesReceived >= 6 && regs.fifoExpectedSize ==
    0) {
24              regs.fifoExpectedSize =
25                  (static_cast<uint32_t>(regs.fifo[2]) << 24) |
26                  (static_cast<uint32_t>(regs.fifo[3]) << 16) |
27                  (static_cast<uint32_t>(regs.fifo[4]) << 8)  |
28                   static_cast<uint32_t>(regs.fifo[5]);
29              if (regs.fifoExpectedSize < regs.fifoBytesReceived) {
30                  regs.fifoExpectedSize = static_cast<uint32_t>(regs.
    fifoBytesReceived);
31              }
32          }
33
34          if (regs.fifoExpectedSize == 0 || regs.fifoBytesReceived <
    regs.fifoExpectedSize) {
35              regs.sts |= 0x02; // EXPECT more data
36          } else {
37              regs.sts &= ~0x02;
38          }
39      }
```

`val` is the byte written by the host; as soon as it arrives, it ends up in `regs.fifo` for that location. The `STS` log can be read by the host to monitor status.

Sending the command to the TPM. When the host has finished writing the command, it must signal that it is ready for processing by setting the `CommandReady` bit (`0x40`) in the `STS` register (write to the offset of `STS` with that bit active).

Listing 4.10: Command Ready

```
1       if (val & 0x40) {
2           if (!regs.fifoReceiving ||
3               regs.fifoExpectedSize == 0 ||
4               regs.fifoBytesReceived < regs.fifoExpectedSize) {
5               regs.sts |= 0x02;
```

```
6                   pkt->makeResponse();
7                   return pioDelay;
8               }
9
10              std::vector<uint8_t> command = regs.fifo;
11              regs.fifo.clear();
12              regs.fifoReceiving = false;
13              regs.fifoExpectedSize = 0;
14              regs.fifoBytesReceived = 0;
15              regs.sts &= ~0x02;
16              regs.sts &= ~0x40;
17              regs.sts &= ~0x01;
18
19              // Process the command
20              std::vector<uint8_t> response = processCommand(command);
21
22              // Load response into FIFO
23              sendFifoResponse(loc, response);
24          }
```

The code accomplishes the following operations:
(i) As soon as it sees the `CommandReady` bit, the TPM knows that the host has finished the input phase and that the received command is complete. It then copies the accumulated bytes into the `regs.fifo` buffer to the local vector `command`, and empties the input FIFO —that data is considered "consumed" and ready for processing.
(ii) Clears the internal state, resetting the flags `CommandReady`, `DataAvail` and `EXPECT` within the register `STS`, so as to signal that there are no more input bytes to receive and that the TPM is being processed.
(iii) Invokes the function `processCommand(command)`, which represents the logical engine of the TPM: this function interprets the command according to the TPM 2.0 format (header, parameters, handles, etc.), performs the requested operation (for example extension of a PCR) and produces a response vector `response` containing the bytes of the output.
(iv) Finally, it calls `sendFifoResponse(loc, response)`, which takes care of loading the response bytes into the output FIFO buffer and updating the status flags consistently: if there is data available, the `DataAvail (0x01)` bit is set, while `EXPECT (0x02)` is reset to zero, indicating that the TPM has completed processing and is ready for the reading phase by the host.

Reading the answer; the host reads the response byte by byte from `FIFO_DATA`. Each read returns the first available byte and removes it from the buffer. When all bytes have been read, the TPM resets `DataAvail` and reports `EXPECT=1`, signaling that it is ready for a new command.

CRB Interface

The Command Response Buffer (CRB) mode implements the interface introduced with TPM 2.0 to improve the communication performances between hosts and TPM compared to the classical FIFO. In this mode, the host and TPM exchange commands and responses through two shared memory areas, named `Command Buffer` and `Response Buffer`, respectively. The device no longer receives bytes through registers; instead, it only receives control signals indicating where to read and write data, thereby realizing a more efficient mapped memory interface.
Within the TPM model, communication with the simulated CPU occurs via two main functions: `handleCrbWrite()` and `handleCrbRead()`. The first is invoked whenever the host (or simulated CPU) performs a *write* operation on an address belonging to the TPM device space; the second is instead invoked at a *read*. In other words, `handleCrbWrite()` handles all commands sent from the host side — such as configuring control registers, starting a command, or asking to delete— by updating the module's internal state and the state machine `CrbState`. The function `handleCrbRead()`, on the contrary, serves to return the contents of the TPM registers to the host, allowing the status of the device to be queried or the parameters currently configured to be read. The communication, through the CRB, between the TPM and the software happens in the following way.

Activation of locality; Locality management is performed via the `LOC_CTRL` register, which shares the same memory address as `LOC_STS`. Depending on the type of access, a write to this address is interpreted as a request to activate or release the locality (`LOC_CTRL`), while a read returns the current state of the same (`LOC_STS`). By writing the value `0x1` to `LOC_CTRL`, the host requests activation of the corresponding location. Our implementation in `handleCrbWrite()` updates the internal state of the device by setting `regs.active` and `regs.loc_sts` and disabling all other locations:

Listing 4.11: Request Locality

```
1     if (offset == LOC_CTRL_OFFSET)
2     {
3         regs.loc_ctrl = val32;
4         regs.active = val32 & 0x1;
5         regs.loc_sts = regs.active ? 0x1 : 0x0;
6         if (regs.active)
7         {
8             for (unsigned i = 0; i < MaxLocalities; ++i)
9             {
10                if (i != loc)
11                {
12                    tpmRegs[i].active = 0;
13                    tpmRegs[i].loc_sts = 0;
```

```
14                  }
15              }
16          }
17          pkt->makeResponse();
18          return pioDelay;
19      }
```

Reading the same offset, handled in `handleCrbRead()`, returns `0x1` if the location is active, or `0x0` otherwise.

Start of the Execution. Execution of a CRB command is initiated by writing the value `0x1` to the register `CRB_CTRL_START` (offset 0x004C). When the TPM is in the `Ready` state, the host indicates that the command is ready in the previously configured memory. The function `handleCrbWrite()` captures this write, updates the internal state and reads the command from the simulated memory via the function `dmaRead()`:

Listing 4.12: Execution Start

```
1       case CRB_CTRL_START_OFFSET:
2           regs.crb_ctrl_start = val32;
3           if ((val32 & 0x1) && regs.crb_state == CrbState::Ready)
4           {
5               regs.crb_state = CrbState::Execution;
6               regs.crb_cmd_buf = dmaRead(regs.crb_cmd_addr, regs.
    crb_cmd_size);
7               execCrbCommand(regs);
8               if (!regs.crb_rsp_buf.empty() && regs.crb_rsp_addr != 0)
9               {
10                  dmaWrite(regs.crb_rsp_addr, regs.crb_rsp_buf);
11              }
12          }
13          break;
```

At this stage, the TPM reads from the Command Buffer the bytes of the full command, processes them through `execCrbCommand()`, and writes the response to the memory pointed to by `CRB_RSP_ADDR`.
The function `execCrbCommand()` encapsulates the command execution logic, including exception handling. If successful, the result is passed to the function `finishCommand()` with return code `TPM_RC::SUCCESS`; if not, an error response is constructed:

Listing 4.13: execCrbCommand()

```
1   void TPMDevice::execCrbCommand(TPMRegisters &regs)
2   {
3       try
4       {
```

```
5           auto rsp = processCommand(regs.crb_cmd_buf);
6           regs.crb_rsp_buf = std::move(rsp);
7           finishCommand(regs, static_cast<uint32_t>(TPM_RC::SUCCESS));
8       }
9       catch (const std::exception &e)
10      {
11          DPRINTF(TPMDevice, "CRB command failed: %s", e.what());
12          regs.crb_rsp_buf.clear();
13          finishCommand(regs, static_cast<uint32_t>(TPM_RC::FAILURE));
14      }
15  }
```

The function `finishCommand()` constructs the final response, setting the TPM status flags. If the response is too short or the return code does not indicate success, a minimum 10-byte packet compliant with the TPM 2.0 standard is generated, with the fields `tag`, `size`, and `responseCode`. The completion stage also updates the control logs:

Listing 4.14: Construction of the response packet

```
1           regs.crb_rsp_buf.resize(10);
2           regs.crb_rsp_buf[0] = (tag >> 8) & 0xFF;
3           regs.crb_rsp_buf[1] = tag & 0xFF;
4           regs.crb_rsp_buf[2] = 0x00;
5           regs.crb_rsp_buf[3] = 0x00;
6           regs.crb_rsp_buf[4] = 0x00;
7           regs.crb_rsp_buf[5] = 0x0A;
8           regs.crb_rsp_buf[6] = (responseCode >> 24) & 0xFF;
9           regs.crb_rsp_buf[7] = (responseCode >> 16) & 0xFF;
10          regs.crb_rsp_buf[8] = (responseCode >> 8) & 0xFF;
11          regs.crb_rsp_buf[9] = responseCode & 0xFF;
```

During execution, an ongoing command can be canceled by writing `0x1` to the `CRB_CTRL_CANCEL` register. In this case, if the TPM is in the `Execution` state, the `handleCrbWrite()` function calls `finishCommand()` with error code `TPM_RC::CANCELED`, and, if a response address has been configured, writes an error packet compliant with the TPM specification to memory. The `handleCrbRead()` function allows the host to read the TPM status at any time. The main registers (`CRB_CTRL_STS`, `CRB_CTRL_REQ`, `CRB_CMD_SIZE`, `CRB_RSP_SIZE`, `LOC_STS`) return the current values stored in `regs`. If the location is not active, the value `0xFF` is returned, simulating unauthorized access.

4.3.3 Basic commands implemented in the TPM module

To validate the operation of the simulated TPM module and ensure compatibility with the fundamental operations envisioned by the TPM 2.0 standard, a set of basic commands was implemented, selected to cover the main phases of the device's life cycle. In particular, the module supports the commands `TPM2_Startup`, `TPM2_Shutdown`, `TPM2_Create`, and `_TPM_Init`, each of which has been modeled to faithfully reproduce the logical behavior described in the TCG specification, while maintaining an abstraction layer compatible with the gem5 simulation environment.

_TPM_Init — Module initialization The command `_TPM_Init` represents the initialization stage of the module, and is the first logical operation performed after the device is created. In the model, this function is called directly inside the constructor of the `TPMDevice`, so that each new instance of the TPM in the gem5 simulator starts from a coherent and completely reset state. Its purpose is to restore the internal state of the TPM by removing any residual information from previous executions. All status flags are cleared (`initialized`, `failureMode`, `orderlyShutdownFlag`) and the shutdown type before `TPM_SU::CLEAR` is set.

Listing 4.15: _TPM_Init() function

```
void TPMDevice::_TPM_Init()
{
    DPRINTF(TPMDevice, "_TPM_Init: resetting volatile state\n");

    state.initialized = false;
    state.failureMode = false;
    state.orderlyShutdownFlag = false;
    state.lastShutdownType = TPM_SU::CLEAR;

    discardOrderlySnapshot();
    purgeStClearNvIndexes();
    clearTransientState();
    populatePufsFromConfig();
    resetPcrBanks(true);
    resetCrbState();
}
```

At this stage, the device registers are not yet configured, but only the internal status structures. The `_TPM_Init` command must always precede a subsequent call to `TPM2_Startup`, as required from the standard.

TPM2_Startup() — Logical start of TPM The command `TPM2_Startup()` is the one that performs the actual initialization of the TPM, bringing it into a consistent operational state. It can be invoked with two modes: `TPM_SU::CLEAR`,

which performs a “cold” start by clearing all volatile information, or `TPM_SU::STATE`, which attempts to restore the sorted state (orderly state) previously saved via `TPM2_Shutdown`.

Listing 4.16: TPM2_Startup() implementation

```
TPMDevice::TPM_RC TPMDevice::TPM2_Startup(TPM_SU startupType)
{
    DPRINTF(TPMDevice, "TPM2_Startup: startupType = %d\n",
            static_cast<int>(startupType));

    // 1) Must be first logical command after _TPM_Init
    if (state.initialized)
        return TPM_RC::INITIALIZE; // Startup already done

    resetCrbState();

    if (startupType == TPM_SU::STATE) {
        if (state.lastShutdownType != TPM_SU::STATE || !state.
    orderlyShutdownFlag)
            return TPM_RC::VALUE;

        if (!restoreOrderlyState()) {
            discardOrderlySnapshot();
            state.orderlyShutdownFlag = false;
            return TPM_RC::FAILURE;
        }

        discardOrderlySnapshot();
    } else if (startupType == TPM_SU::CLEAR) {
        discardOrderlySnapshot();
        clearTransientState();
        purgeStClearNvIndexes();
        populatePufsFromConfig();
        resetPcrBanks(true);
        state.failureMode = false;
    } else {
        return TPM_RC::VALUE;
    }

    state.initialized = true;
    state.orderlyShutdownFlag = false;

    if (startupType == TPM_SU::CLEAR)
        measureSystemState();

    DPRINTF(TPMDevice, "TPM2_Startup: completed successfully\n");
    return TPM_RC::SUCCESS;
}
```

When a boot of type `CLEAR` is performed, the TPM completely erases the volatile state: the PCR banks are reset, the volatile NV memory is cleaned up and the PUF instances are reloaded from configuration. If, on the other hand, the type is `STATE`, the restoration of the last saved orderly state is attempted, provided that the previous shutdown was successfully carried out by a command of type `STATE`. In both cases, when finished, the `state.initialized` flag is set to `true`, indicating that the TPM is now operational. In case of startup `CLEAR`, the system measurement (`measureSystemState()`) is also performed, simulating the initial attestation phase of the real TPM.

TPM2_Shutdown() — Stop ordered or clear status The command `TPM2_Shutdown()` handles the logical TPM shutdown stage and internal state storage. As with `Startup`, there are two types of shutdown: `TPM_SU::STATE`, which saves the contents of PCR banks, sessions, and NV indexes (orderly state) to memory, and `TPM_SU::CLEAR`, which instead performs a "clear" shutdown by wiping volatile data.

Listing 4.17: TPM2_Shutdown() implementation

```
TPMDevice::TPM_RC TPMDevice::TPM2_Shutdown(TPM_SU shutdownType)
{
    if (!state.initialized)
        return TPM_RC::INITIALIZE; // must call Startup first

    if (shutdownType != TPM_SU::CLEAR && shutdownType != TPM_SU::
    STATE)
        return TPM_RC::VALUE;

    resetCrbState();
    TPM_RC result = TPM_RC::SUCCESS;

    if (shutdownType == TPM_SU::STATE) {
        if (!persistOrderlyState()) {
            result = TPM_RC::FAILURE;
            state.orderlyShutdownFlag = false;
        } else {
            state.orderlyShutdownFlag = true;
        }
    } else { // TPM_SU_CLEAR
        discardOrderlySnapshot();
        clearTransientState();
        state.orderlyShutdownFlag = false;
    }

    state.lastShutdownType = shutdownType;
    state.initialized = false;
    return result;
```

```
28 }
```

In the case of `STATE`, the function `persistOrderlyState()` serializes the internal state structures (`pcrBanks`, `sessionTable`, `chosenPufs`, `nvIndices`) in memory, creating a snapshot that can be restored the next time you boot. If a shutdown of type `CLEAR` is requested, all temporary data is deleted and the PUF structures are cleaned. In both cases, the TPM status is returned to uninitialized, ready for a new boot cycle.

TPM_Create — Creating PUF objects The command `TPM_Create()` has been implemented to allow dynamic configuration of Physical Unclonable Functions (PUFs). This command receives as input a data buffer with the number and type of PUF to activate, and updates the `chosenPufs` structure accordingly.
The function decodes the command according to the standard TPM format (tag, size, command code, payload) and uses the next field to identify the types of PUFs to create. Each type is validated and added to the set `chosenPufs`, which represents the simulated physical generators of PUFs.

4.3.4 Authorization and Session Management

A key feature of TPM 2.0 is its sophisticated authorization mechanisms for commands affecting sensitive objects. TPM 1.2 used fixed entities with dedicated passwords. TPM 2.0 introduces authorization sessions that offer HMAC authorization, policy, and parameter encryption in a flexible manner. Our model implements a subset of these features: authorization with a simple password, the command directly contains the cleartext value of the shared password; HMAC authorization is based on creating a temporary session identified by a nonce and, optionally, a salt. This type of authorization uses HMAC SHA-256 to cover the secret `AuthValue` of the target object. Each entity protected in the TPM, such as a key, NV index, or PUF, has its own value `AuthValue`, which represents the "secret key" used to validate permissions and calculate authorization HMACs.

Authorization type identification

According to the TCG TPM 2.0 Library Specification, Part 3, the authorization type used in a command is determined by the `tag` field in the command packet header. This field lets the TPM know if it should expect an authorization area (`TPMS_AUTH_COMMAND`) and which mechanism to apply: `TPM_ST_NO_SESSIONS (0x8001)` — indicates that the command contains no authorization session. In this case, the TPM interprets authorization as "password-only" or "null" (no active session); `TPM_ST_SESSIONS (0x8002)` — indicates that the command includes one or more `TPMS_AUTH_COMMAND` structures in the parameter area. The TPM then

reviews the handle of each session to determine its type: if `sessionContext.type == TPM_SE::HMAC`, the session is of type HMAC; if instead it is `POLICY` or `TRIAL`, it is treated according to the other mechanisms provided by the standard.
The two main cases are handled in our model: `NO_SESSIONS` (password authorization) and `SESSIONS` with session `HMAC`, enough to cover the most common authorization operations.

Authorization session structure

Each session information is stored in the `SessionContext` structure, which maintains the status of the active HMAC session.

Listing 4.18: Authorization Session

```
struct SessionContext {
    uint32_t handle; // session handle
    TPM_SE type; // session type HMAC or POLICY
    uint32_t nonceSize; // size of the nonce
    std::vector<uint8_t> nonceCaller; // nonce for the session
    std::vector<uint8_t> nonceTPM; // TPM nonce
    HashAlg hashParam; // hash algorithm used to encrypt session
    parameters
    HashAlg authHashAlg; // hash algorithm used for authorization
    std::vector<uint8_t> salt; // salt for the session
    std::vector<uint8_t> sessionKey; // derived key

    std::vector<uint8_t> cachedAuthValue; // cached authorization
    value for unbound sessions
    bool isBounded;

};
```

Each active session is stored using a global table `sessionTable`, which maps the session handle to its context.

Creating a HMAC session - TPM2_StartAuthSession

The command `TPM2_StartAuthSession` allows the software to start a new authorization session. This is used to negotiate security parameters between hosts and TPM, as shown below:

- `tpmKey` — Handle of a TPM key used to encrypt any *salt* of the session. If set to `TPM_RH_NULL`, it means that no key is used (*unsalted* session).
- `bind` (authHandle) — Handle of the object to which the session is linked. If not null, the session is "bound": the `AuthValue` of the object is included in the session key derivation, making the session valid only for that object.

- `nonceCaller` — Random value sent by caller (host).
- `encryptedSalt` — Optional salt value encrypted with the key specified in `tpmKey`. If present, the TPM decrypts it with the function `decryptSaltWithEVP()`.
- `sessionType` — Session type: HMAC (0x00), Policy (0x01), Trial, etc.
- `authHash` — Hash algorithm used for session digests and HMACs (SHA-256, SHA-1, SHA-384)

During command processing, the TPM:

1. Checks the received parameters and checks the validity of the handles (NV, Hierarchy, PUF, etc.).
2. Reads `nonceCaller` and, if present, decrypts the `salt` through RSA key. In case of an error (missing key or decryption failed), the command is rejected.
3. Generates a random value `nonceTPM` of the same length as `nonceCaller`.
4. Determines whether the session is bound or salted:
   - Bound: `authHandle` $\neq$ `TPM_RH_NULL`, then l'`AuthValue` of the associated object is retrieved.
   - Salted: the field `salt` is non-empty.
5. Derives session key with function `KDFa()`:

   $K = \text{KDFa}(\text{authValue} \,\|\, \text{salt},\ \text{“TPM2\_HMAC”},\ \text{nonceCaller} \,\|\, \text{nonceTPM},\ L)$

   where L is the length of the algorithm uses for calculate the HMAC (for example, SHA-256 the length is 256 bits).

Listing 4.19: KDFa()

```
    ctx.sessionKey = KDFa(
    getEVPFromHashAlg(ctx.authHashAlg),
    authValue, ctx.salt,
    "TPM2_HMAC",
    ctx.nonceCaller, ctx.nonceTPM,
    getHashSize(ctx.authHashAlg) * 8
);
```

6. Allocates a unique session handle and registers the context in the table `sessionTable`. The handle is allocated progressively starting from `0x03000000`, as in the real TPM:

Listing 4.20: Allocate Session

```
uint32_t allocateSessionHandle() {
    static uint32_t nextHandle = 0x03000000;
    return nextHandle++;
}
```

7. Constructs the response for the host with the handle and the new `nonceTPM`.

At this point, the host can use the returned handle to authenticate subsequent commands.

Bounded and Unbounded Sessions

In this model, the distinction between *bounded* and *unbounded* sessions mirrors the behavior defined by the TCG in TPM 2.0 standard. A session is considered *bounded* when an `authHandle` other than `TPM_RH_NULL` is specified: in that case, the session is logically tied to an object of the TPM, and its session key (`sessionKey`) is derived by including the `AuthValue` of that object. This means that the authorization obtained through that session is valid only for that specific object: if the object changes authorization value, or if the session is used for another handle, the HMAC calculation will no longer coincide and the authorization will fail. In other words, binding ensures that the session cannot be reused for different resources, reinforcing the isolation of credentials in the TPM. Sessions *unbounded*, in contrast, are initialized with `authHandle = TPM_RH_NULL`. In this case, the session key does not depend on any `AuthValue` inside the TPM, but only on the `salt` (if present) and the nonces exchanged between host and TPM. This makes them more flexible, but also less tied to a specific context: typically, these sessions are used for temporary authorization or to establish encrypted data exchange channels, without tying the session to a particular object. In the code, such a distinction is represented by the flag:

```
ctx.isBounded = (authHandle != TPM_RH_NULL);
```

Where the `authValue` parameter is empty for unbounded sessions, while for bounded sessions it contains the secret associated with the bound object (for example the password of an NV index or the authorization value of a PUF). This allows that the session key is effectively different for each combination of handle and nonce, ensuring that two nominally identical sessions but tied to different objects produce distinct HMACs.
The diagram below summarizes how bounded and unbounded sessions are handled.

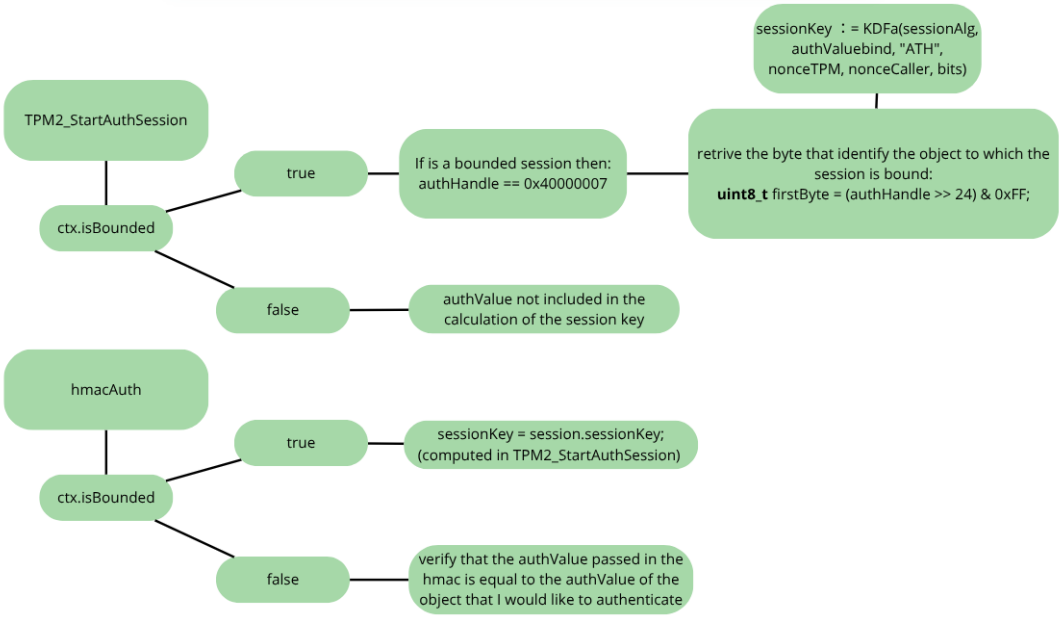

Figure 4.1: Overview of Bounded and Unbounded Session Handling

The first part of the scheme describes how the session is created in the case of bounded or unbounded sessions. While the second part describe where the session key is taken based on the type of session.

Using the HMAC session

When the host sends a protected command, it includes a `TPMS_AUTH_COMMAND` structure containing:

- the session handle (4 bytes),
- the updated `nonceCaller`,
- one byte of attributes,
- and finally two bytes of size followed by the HMAC or password value.

In the case of a `Password Session`, the final field contains the cleartext password; in the case of an HMAC session, it contains the HMAC calculated according to the formula:

$$\text{HMAC}(\text{sessionKey} \,\|\, \text{authValue},\ \text{cpHash} \,\|\, \text{nonceCaller} \,\|\, \text{nonceTPM} \,\|\, \text{sessionAttr})$$

where `cpHash` is the hash of the command parameters and command code, excluding the authorization area.

Authorization verification

The function `hmacAuth()` implements all the validation logic of the authorization areas present in the received commands.
Depending on the type of session, the behavior is different:

- Password session: checks that the nonce is null and directly compares the received password with the expected `AuthValue`.
- HMAC session:
  - Extracts `nonceCaller`, attributes and HMAC value.
  - Calculate the `cpHash` of the command.
  - Retrieve `sessionKey` and `AuthValue` from context.
  - Calculate the expected HMAC with `computeTPMAuthHMAC()` and compare it with the sent one.
  - If valid, it updates the session by generating a new `nonceTPM`.

In case of an error, `TPM_RC_AUTH_FAIL` or `TPM_RC_SESSION_HANDLE` is returned. This mechanism is used by all functions that require authorization, such as `verifyNvAuthorization()` or `generateChallengeResponse()` for PUFs.
Finally, after each valid authorization, the TPM generates a new random `nonceTPM`:

```
1 RAND_bytes(sessionPtr->nonceTPM.data(), sessionPtr->nonceTPM.size());
```

This behavior, compliant with TCG specifications, ensures that each authenticated command uses a unique nonce, preventing the reuse of old HMACs and therefore replay attacks. Furthermore, the new nonce is stored in the `SessionContext` structure, making it possible to safely continue multi-command sessions.

Decrypt salt and load RSA keys

To support "salted" sessions, the module includes the function `decryptSaltWithEVP()`, which uses OpenSSL APIs to decipher an RSA-OAEP-encrypted `salt`. The RSA keys are preloaded into the simulated TPM via `loadKeyFromHex()`, which constructs a `EVP_PKEY` object starting from hexadecimal parameters (modulus, exponent, private exponent). This part allows to realistically simulate the creation of cryptographic sessions based on asymmetric keys.

Closing sessions — TPM2_FlushContext()

Sessions can be closed explicitly via the `TPM2_FlushContext(handle)` command, which removes the corresponding entry from the `sessionTable`. During `Shutdown`, all sessions are automatically deleted, since — as in real TPM — they are volatile entities that do not survive a reboot.

Security Considerations

This authorization model makes the simulated TPM capable of requesting the correct credentials (passwords or HMAC) to access protected resources. If an NV index or a PUF has a defined `AuthValue`, any command that does not provide proper authorization is rejected.

4.3.5 Platform Configuration Registers (PCR)

PCRs are registers dedicated to the secure storage of cryptographic measurements that reflect the state of the platform over the course of the boot process and during execution. Every time a system component is loaded (for example BIOS, bootloader, kernel or critical modules), its hash is calculated and this value is "extended" in the corresponding PCR, creating a chain of trust. This chain helps ensure that the current state of the system is the result of a sequence of verifiable, unaltered events. The fundamental property of PCRs is retroactive non-modifiability: once a value is extended in the register, it is not possible to trace the previous value or modify it arbitrarily, but only to add new measurements cumulatively.
According to the TCG (Part 2: Structures) specification, a compliant TPM module must implement at least 24 PCRs, numbered 0 through 23, each of which may be associated with one or more hashing algorithms (for example, SHA-1, SHA-256, SHA-384, SHA-512).

General operation of the PCR mechanism

The logical operation of a PCR can be described through three basic operations:
Initialization; at the time of TPM start or initialization, each PCR is set to a known value, generally a sequence of zeros of the adopted hash length.
Extension; whenever a new event or component needs to be measured, its hash is calculated as H(date). The new PCR value is the result of the concatenated hash of the previous value and the new digest:

$$\mathrm{PCR}_i^{\mathrm{new}} = H\Big(\mathrm{PCR}_i^{\mathrm{old}} \parallel H(\mathrm{data})\Big)$$

This operation is called an extend and ensures that each change in the state of the system deterministically changes the PCR value, preserving the integrity of the measurement chain.

Verification and attestation; the contents of the PCRs can be read and signed by the TPM to generate a quote, which is a cryptographically verifiable proof of the status of the platform. External entities can verify these quotas by comparing the PCR values with the expected ones, ensuring that the system has not been compromised.

PCR Data Structure

In the code, the PCR data structure is defined as follows:

Listing 4.21: PCR Data Structure

```
1    struct PCR
```

```
2     {
3       std::vector<uint8_t> value;
4       bool resettable = false; // for TPM2_PCR_Reset (generally false
       for 0-15 PCRs)
5       std::vector<uint8_t> authValue; // authorization value
      associated with the PCR
6
7       PCR() : value() {} // default constructor leaves value empty
8       PCR(HashAlg alg, bool isResettable = false)
9           : value(getHashSize(alg), 0), resettable(isResettable) {}
      // initialize with hashSize zeroed bytes
10    };
```

Each PCR includes:

- value: the current value of the register, represented as a vector of bytes. This value is initialized with a sequence of zeros the size of the hashing algorithm used (for example, 32 bytes for SHA-256);
- resettable: a Boolean flag indicating whether the PCR can be zeroed (only some registers, typically 16 onwards, are resettable);
- authValue: An associated authorization value, used to control access to or modification of the PCR.

The total number of PCRs is set at 24, accoding to TCG specification.

Software implementation operation

The software implementation of PCRs within the `TPMDevice` class faithfully replicates the logic of TPM, providing the following main operations:

Reading; the `pcrRead()` function provides access to the contents of a log, returning the current digest. It preliminarily checks the validity of the required index and the presence of the "bank" corresponding to the hashing algorithm used (for example SHA-256).

Extension; the `pcrExtend()` function implements hash concatenation and recalculation logic. The new PCR value is calculated by concatenating the current register value with the event digest and applying the selected hash function:

Listing 4.22: pcrExtend()

```
1 toHash.insert(toHash.end(), pcr.value.begin(), pcr.value.end());
2 toHash.insert(toHash.end(), digest.begin(), digest.end());
3 pcr.value = shaHash(alg, toHash);
```

This implementation guarantees the forward integrity property, since each subsequent value cryptographically depends on all previous values.

Reset. Some PCRs can be returned to zero, marked as resettable. The function `pcrReset()` checks for that condition and, if successful, zeroes the registry contents for all active algorithms. Non-reset PCRs (typically 0 to 15) retain their value until the TPM is restarted.

Events and measurements; the `pcrEvent()` and `measureSystemState()` functions allow simulating the measurement of the system state. In particular, `measuresystemState()` collects structural system information (name, available memory, number of threads, active workload) and calculates an SHA-256 digest of that data. The result is then extended into PCR0, which takes the role of the main register for measuring platform integrity. This approach allows the chain of trust to be emulated at the software level, allowing any changes to the state of the platform to be verified by comparing the current value of the PCR with the expected one.

Listing 4.23: measuresystemState()

```
appendString(sys->name());
appendU64(sys->memSize());
appendU32(static_cast<uint32_t>(sys->cacheLineSize()));
appendU32(static_cast<uint32_t>(sys->threads.size()));
appendU32(static_cast<uint32_t>(sys->threads.numActive()));
if (sys->workload)
    appendString(sys->workload->name());
```

Authorization; the `verifyPcrAuthorization()` method implements an access control on PCRs, verifying that the provided handle actually belongs to the PCR category and that the authorization (authValue) is valid.

Function of cryptographic hashes and libraries

The function `shaHash()` represents the base cryptographic primitive used to calculate integrity measures associated with Platform Configuration Registers (PCR). It performs digest calculation from an arbitrary data buffer, using one of the hashing algorithms required by the TPM 2.0 standard. Its behavior complies with what is defined in the *TCG TPM 2.0 Library Specification – Part 2: Structures*, which specifies the use of hash functions such as SHA-1, SHA-256 and SHA-384 for digest generation used in extension and attestation operations.
The function has the following implementation:

Listing 4.24: shaHash()

```
std::vector<uint8_t> TPMDevice::shaHash(HashAlg alg,
                                        const std::vector<uint8_t> &
    data)
{
    const EVP_MD *md = getEVPFromHashAlg(alg);
    if (!md)
        throw std::runtime_error("TPM_RC_HASH: Unsupported hash
    algorithm");

    EVP_MD_CTX *ctx = EVP_MD_CTX_new();
    if (!ctx)
        throw std::runtime_error("Failed to allocate digest context")
    ;

    std::vector<uint8_t> result(EVP_MD_size(md));
    unsigned int len = 0;

    EVP_DigestInit_ex(ctx, md, nullptr);
    if (!data.empty())
        EVP_DigestUpdate(ctx, data.data(), data.size());
    EVP_DigestFinal_ex(ctx, result.data(), &len);
    EVP_MD_CTX_free(ctx);

    result.resize(len);
    return result;
}
```

The function `shaHash()` is used in multiple places in the implementation:

- to calculate the digest of data to be extended in PCRs;
- to generate the measurement values of the system during the phase *boot measurement*;
- for extension operations in NV type registers `EXTEND`, in combination with `hashConcat()`.

The developed implementation accurately reproduces the behavior of the PCRs defined by the TCG, ensuring cryptographic consistency and correct updating of the platform status. Each extension generates a new value dependent on all previous events, preventing any retroactive manipulation. Moreover, integrating the `measureSystemState()` function allows maintaining a synthetic yet reliable representation of the system state that's useful for remote attestation or integrity verification operations.

4.3.6 NV Memory

General context and operation

In the Trusted Platform Module (TPM), non-volatile memory (NV) is a persistent area intended for storing NV indices and related data even in the absence of power. According to the TCG TPM 2.0 specification, each NV index is identified by a unique handle and described by a public block (policy, attributes, size, name algorithm) and a private state (permissions, values and locks).
Conceptually:

- a *NV Index* is a persistent object characterized by a type (`ORDINARY`, `COUNTER`, `BITS`, `EXTEND`, `PIN_FAIL`, `PIN_PASS`), by a series of attributes (`TPMA_NV`) governing its behavior, and by a set of authorization policies and values;
- fundamental operations include defining and removing NV space, reading and writing with offset and bounds checking, managing locks (*read/write/-global*), typed operations (`increment`, `setBits`, `extend`), and changing login credentials.

NV attributes (`TPMA_NV`) enable security and operational consistency constraints, including:

- `TPMA_NV_WRITTEN`: indicates whether the index has been initialized or written at least once;
- `TPMA_NV_WRITELOCKED` and `TPMA_NV_READLOCKED`: prevent index writing or reading, respectively;
- `TPMA_NV_WRITEDEFINE`, `TPMA_NV_WRITE_STCLEAR`, `TPMA_NV_READ_STCLEAR`: specify lock persistence (permanent or until TPM is reset);
- `TPMA_NV_GLOBALLOCK`: enable global locking on all NV indexes;
- `TPMA_NV_ORDERLY`: differs the persistent update by moving it to a volatile representation.

The NV area then provides secure persistence of platform security-related data and policies, maintaining strict control over access and integrity of operations.

NV Data Structure

Non-volatile memory management is modeled in the project through three main structures: `TPMS_NV_PUBLIC`, `TPMS_NV_PUBLIC2` and `NVIndex`. These represent, respectively, the public portion of an NV index, its extended variant with handle-type information, and the complete internal representation maintained by the TPM

device.

The structure **TPMS_NV_PUBLIC** defines the public area of an NV index, i.e. the set of static parameters that describe how the register is configured. It is modeled according to the **TCG TPM 2.0, Part 2 – Structures** specification and includes the following fields:

- `nvIndex`: unique NV index identifier, encoded as a 32-bit *handle*. NV handles belong to the category `TPM_HT_NV_INDEX` and permanently identify the memory location reserved for an object.
- `nameAlg`: specifies the hashing algorithm (for example SHA-256, SHA-384, etc.) used to calculate the Name of the index. This is used as a digest that uniquely represents NV Index public definition. This field allows you to ensure the integrity of the public descriptor and verify it cryptographically during attestation operations.
- `attributes`: contains configuration bits defined by the field `TPMA_NV`. Each bit represents an index property or restriction, such as read and write locks, index persistence after TPM reset, or the use of the `ORDERLY` flag for partial maintenance in volatile memory.
- `authPolicy`: represents the **authorization policy** associated with the index, stored as a byte vector. It is the digest of a policy structure generated by the TPM, and is compared with the authorization provided during operations to verify that access conditions are respected.
- `dataSize`: indicates the maximum size, in bytes, of the data that the index can contain. This value is used to validate write operations and to allocate the necessary NV memory.
- `type`: specifies the **index type**, via enumeration
  - `ORDINARY`: generic read/write space;
  - `COUNTER`: numeric register incremental with `TPM2_NV_Increment()`;
  - `BITS`: set of bits that can be manipulated via OR logical operations;
  - `EXTEND`: PCR-like extension register, based on concatenated hashes;
  - `PIN_FAIL` and `PIN_PASS`: indexes dedicated to managing PIN-based access policies.

The **TPMS_NV_PUBLIC** structure describes all **immutable and verifiable** parameters of an NV Index, providing the information necessary to calculate its

Name and control its security policies.
The `TPMS_NV_PUBLIC2` structure represents a extension of `TPMS_NV_PUBLIC`, introduced for more modular handling. Includes two fields:

- `handleType`: identifies the type of handle associated with the object (for example `TPM_HT_NV_INDEX` for NV indexes). This field allows different objects to be distinguished within the TPM handle space.
- `nvPublic`: structure `TPMS_NV_PUBLIC` proper, which contains index details.

The **NVIndex** structure represents the internal and dynamic part of an NV Index, that is, the actual state of the object within the TPM. It contains both information derived from the public part, as well as additional fields necessary for managing content and permissions.
The main fields are:

- `handle`: index internal identifier (full 32-bit handle). The most significant byte encodes the handle type according to TCG rules, while the remaining 24 bits identify the specific index.
- `nameAlg`: hashing algorithm used to calculate the index name, as defined in the public section.
- `data`: byte vector containing the **persistent data** stored in NV. In special cases (`COUNTER`, `BITS`, `EXTEND`), it represents the numeric counter, bit value, or cumulative digest, respectively.
- `size`: field size `data`, consistent with `dataSize` in the public part. It is used to validate offsets and write limits.
- `attributes`: bit field (`TPMA_NV`) that keeps the updated state of the index, including dynamic flags such as `WRITTEN` or `WRITELOCKED`.
- `authValue`: secret value associated with the index, used for password authorization operations or HMAC. It is distinct from `authPolicy`, which instead defines a logical constraint on access conditions.
- `authPolicy`: copy of the associated public policy, necessary to verify the coherence between the public and internal areas.
- `type`: logical index type, as defined in `TPM_NT`.
- `volatileCounter`: temporary counter used to manage indexes of type `COUNTER` with attribute `ORDERLY`. It is maintained in RAM and consolidated on NV only when a threshold is exceeded or when the TPM status is saved.

- `volatileData`: temporary buffer in RAM used for indexes with attribute `ORDERLY`. It allows you to reduce physical writes to non-volatile memory, preserving their duration.
- `writeLocked`: Boolean flag indicating whether the index is currently locked in write.
- `written`: flag that signals whether the index has been written at least once (equivalent to `TPMA_NV_WRITTEN`).

The distinction between `TPMS_NV_PUBLIC` and `NVIndex` reflects the conceptual division between **static and verifiable description** (public area) and **operational status** (private area). This separation allows only information necessary for integrity verification to be made public, keeping sensitive data and the internal dynamic state of the TPM confidential.

Operations implemented

The functions `writeNV()` and `readNV()` allow writing and reading data on the NV indices, verifying the attributes and memory limits in each operation and the index permissions.
The **write** performs the following checks:

1. checking for index presence and authorization by `verifyNvAuthorization()`;
2. checking absence of flag `TPMA_NV_WRITELOCKED`;
3. consistency between offset, size and attributes (eg. `TPMA_NV_WRITEALL`).

Listing 4.25: Verifying limits in NV writing

```
1 if (offset + input.size() > index.size)
2     throw std::runtime_error("TPM_RC_NV_RANGE: Write exceeds NV index
      size");
3
4 std::copy(input.begin(), input.end(), index.data.begin() + offset);
5
6 index.attributes |= NVAttr::WRITTEN;
```

The **read** similarly checks block attributes (`TPMA_NV_READLOCKED`), the offset and size limits, and the actual initializing the data (`TPMA_NV_WRITTEN`), raising exceptions in case of violations of policies or constraints.

The **undefineNVSpace()** function allows you to delete an existing NV index, updating available memory and checking security policies. The operation is blocked

for indexes that have the attribute `TPMA_NV_POLICY_DELETE`, and it requires platform permission if the corresponding bit is set. For cases where there is an explicit cancellation policy, the `undefineNVSpaceSpecial()` function allows conditional removal to the validation of the policy digest and the authorized command code (`TPM_CC_NV_UndefineSpaceSpecial`).

Typed indices

NV indices can take on different logical types, defined by the field `TPM_NT`, which determine its operating behavior:

- **Counter** (`incrementNV()`): Handles persistent counters and `ORDERLY`. The value is stored in the first four bytes of `data` and updated in RAM if the `ORDERLY` attribute is active:

$$\text{counter}_{\text{new}} = \text{counter}_{\text{old}} + 1$$

- **Bitfield** (`setBitsNV()`): allows logical operation OR between the current value and a 64-bit mask:

$$\text{bits}_{\text{new}} = \text{bits}_{\text{old}} \vee \text{mask}$$

- **Extend** (`extendNV()`): implements an extension analogous to that of the PCRs, concatenating the current value and the supplied buffer, therefore calculating:

$$\text{new} = H(\text{current} \,\|\, \text{buffer})$$

  The result is stored in `data` (for persistent indexes) or in `volatileData` (for indexes `ORDERLY`).

For indexes with attribute `TPMA_NV_ORDERLY`, the implementation maintains a temporary copy of the value in volatile memory (`volatileData` or `volatileCounter`), which is synchronized with NV memory only in system stability conditions or when a predefined threshold is exceeded. This behavior replicates the *orderly* update policy described in the TPM 2.0 Part 3 specifications.

NV memory persistence between Simulations

To ensure the persistence of non-volatile memory (NV) of the TPM between different simulations runs, a new feature was introduced in the model: the *serialization of NV state to file*. In particular, the `TPMDevice::savePersistentNvState()` method takes care of exporting the contents of the NV memory to a binary file, saving all the metadata necessary to correctly restore the state in subsequent

executions. Saving includes a "magic number" and a version number, useful for checking format consistency, as well as information on the number of NV indexes present and the amount of memory used. For each index, the main fields of the `NVIndex` structure are then written to file, such as the identifier (`handle`), the associated hashing algorithm (`nameAlg`), attributes, data size, type, vectors `data`, `authPolicy`, and `authValue`, as well as flags that indicate whether the index has been written or blocked in writing.

The `persistNvState()` method acts as a *wrapper* for `savePersistentNvState()`, automatically invoking it at critical points in the TPM lifecycle (e.g. at *shutdown* or *reset*) and handling any errors via log messages. In this way, the state of NV memory is preserved even between successive simulations, making it possible to model in `gem5` a TPM with realistic behavior, in which permanent data - such as keys, monotonic counters or authorization policies - survive reboots of the simulated system.

Instead, the complementary function, `loadPersistentNvState()`, is invoked during device initialization for **reconstruct NV status** starting from the saved file. It reads the fields in order, verifying the *magic number* and the version, and recreates the map of the NV indices in memory, allocating the structures and populating the respective fields with deserialized data. In case of inconsistencies or missing files, loading is ignored and NV memory is initialized to a clean state, thus ensuring the robustness of the persistence mechanism. This extension allows to more faithfully simulate the behavior of a real TPM, which maintains sensitive information and internal state in non-volatile memory between reboots.

Lock and Policy

The blocking functions, `writeLockNV()`, `readLockNV()` and `globalWriteLockNV()`, apply write and read restrictions on NV indices according to `TPMA_NV` attribute bits. The function `changeNVAuth()` allows updating the value of authorization (`authValue`), after checking the associated policy (`TPM_CC_NV_ChangeAuth`) and the imposed maximum digest limit from the hashing algorithm used. authorization is handled by the `verifyNvAuthorization()` function, supporting both password-based logins and HMAC sessions, checking the match between the calculated digest and the fields `authValue` of the index.
Although the authorization mechanism using **policy** was not fully implemented in this version, the data structure and management functions have been designed to accept one in the future **extension compatible with policy-based authorization**. Specifically, the fields `authPolicy` and `policyDigest` are already foreseen in the data structures (`NVIndex` and `TPMS_NV_PUBLIC`), allowing future integration

of a system of authorization policy.

Hash function and calculation of integrity measures

The function `hashConcat()` constitutes one of the fundamental primitives for the calculation of integrity measures and for the management of NV indices of type `EXTEND`. It implements the concatenation and hashing operation defined in the specification *TCG TPM 2.0 Library Specification – Part 2: Structures*, required for obtain a new value cryptographically bound to the previous data.
The function calculates:

$$H(a \,\|\, b)$$

where H represents the selected hash function, while a and b they are two byte vectors (e.g. the current value and the extension buffer). The use of binary concatenation ensures that the resulting digest is uniquely determined by the order of the data, preventing any possibility of inversion or independent manipulation of the parts. From an implementation point of view, the function uses cryptographic APIs `EVP` from OpenSSL, as shown in the fragment below:

Listing 4.26: hashConcat()

```
std::vector<uint8_t> TPMDevice::hashConcat(uint32_t alg,
                                           const std::vector<uint8_t>
    &a,
                                           const std::vector<uint8_t>
    &b)
{
    const EVP_MD *md = nullptr;
    if (alg == TPM_ALG_SHA256)
        md = EVP_sha256();
    else if (alg == TPM_ALG_SHA384)
        md = EVP_sha384();
    else if (alg == TPM_ALG_SHA1)
        md = EVP_sha1();
    else
        throw std::runtime_error("Unsupported hash algorithm");

    EVP_MD_CTX *ctx = EVP_MD_CTX_new();
    std::vector<uint8_t> result(EVP_MD_size(md));

    EVP_DigestInit_ex(ctx, md, nullptr);
    EVP_DigestUpdate(ctx, a.data(), a.size());
    EVP_DigestUpdate(ctx, b.data(), b.size());
    EVP_DigestFinal_ex(ctx, result.data(), nullptr);
    EVP_MD_CTX_free(ctx);

    return result;
}
```

Using the `EVP` APIs allows you to abstract from the algorithm type and to easily extend support to additional hash functions without changing the architecture of the code. Currently, the algorithms `SHA-1`, `SHA-256`, and `SHA-384` are supported, in line with the most common implementations of TPM 2.0. This function is used by indexes of type `EXTEND` to replicate the behavior of **Platform Configuration Registers (PCR)**, calculating a cumulative digest that maintains the **forward integrity** of the value. Each new extension generates a result that uniquely depends on all the older extensions, depending on the model:

$$\mathrm{NV}_{\mathrm{new}} = H(\mathrm{NV}_{\mathrm{old}} \,\|\, \mathrm{data})$$

so that no retroactive modification of the content can produce a value consistent with the legitimate state of the registry.

4.3.7 TPM commands: mechanism of dispatch

The component `TPMDevice::processCommand()` represents the entry point of the command interface of the simulated TPM. All commands coming from the top level (software or CPU) are forwarded to this function, which deals with:

1. validate binary packet structure;
2. identify the command type by the field `cmdCode`;
3. perform routing (dispatch) to the appropriate management function;
4. build and return the formatted response according to the TPM 2.0 standard.

Each packet sent to the TPM follows the standard format defined in the specifications TCG *TPM 2.0 Part 3 – Commands*:

`tag (2B)` | `size (4B)` | `commandCode (4B)` | `parameters`

The dispatcher initially checks that the package complies with this structure, checking that the actual buffer length matches the value `size` declared in the header field. Such checks prevent memory corruption and attack attempts based on malformed packages or with inconsistent lengths.
The routing to the correct function is handled by a `switch` on the field `cmdCode`. Each code identifies a distinct TPM operation — be it standard (eg. `NV_Read`) or an extended one (eg. `PUF_ChallengeResponse`). The architecture of the dispatcher is deliberately modular: command codes are declared as readable constants,

and adding a new command only requires the insertion of a new `case` with the corresponding management function.
In the context of the thesis, two custom commands were implemented for PUF management:

- `0x0A0A0A0A`: executes the flow of challenge–response by calling the `generateChallengeRespo` function;

- `0x0A0A0A0B`: Dynamically selects active PUFs within the TPM lifecycle via `TPM_Create()`.

This approach allows the TPM behavior to be extended without altering compatibility with standard commands defined by the TCG.
An example of how the dispatch process has been performed is the follows:

Listing 4.27: hashConcat()

```
1     std::vector<uint8_t> response{};
2     switch (cmdCode) {
3     case 0x0A0A0A0A:{ // puf - challenge-response
4         response = generateChallengeResponse(command, tag); // tag ==
      0x8001 o 0x8002
5         break;
6     }
7
8     case 0x20000000:{ // flush session
9         response = TPM2_FlushContext(command);
10        break;
11    }
12
13    case 0x0A0A0A0B: { // TPM_Create (custom): choose active PUF(s)
14        // Execute and return a minimal success header
15        TPM_Create(command);
16        uint16_t cmd_tag =
17            (static_cast<uint16_t>(command[0]) << 8) |
18             static_cast<uint16_t>(command[1]);
19        response = makeRespHeader(cmd_tag, 0x0000);
20        break;
21    }
```

The function `makeRespHeader()` constructs the standard response header, including the return code (`TPM_RC`) that indicates the operation's outcome. Any exceptions raised during command execution are intercepted and relaunched as errors consistent with TPM semantics, preserving interface robustness and traceability.

4.4 Integration of PUFs

The objective of this section is to illustrate in detail the integration of the *Physical Unclonable Functions* (PUFs) in the simulated TPM module, highlighting the solutions adopted for the management of several heterogeneous types (`SRAM`, `RING_OSCILLATOR`, `ARBITER`) and for their interoperability with the security infrastructure of the TPM.
The integration was designed according to two fundamental architectural principles:

1. **Separation between configuration and instance**: each PUF is defined by a configuration object (`PufConfig`), which specifies its static parameters, and by an instance object (`PufWrapper`), which manages the operational status and authorization logic.

2. **Static polymorphism and typed security**: the different PUF families are managed uniformly via `std::variant`, allowing dynamic selection of the concrete type while maintaining compile-time type security.

The resulting infrastructure allows heterogeneous PUFs to be defined and used while maintaining semantics consistent with the TPM 2.0 model, in which each resource is associated with a unique identifier, an authorization area and a set of access attributes.

4.4.1 PUFs Data Structure

To integrate heterogeneous PUFs into the simulated TPM, the implementation clearly distinguishes between: (i) **declarative configurations** (`PufConfig`) and (ii) **runtime instances** (`PufWrapper`) ready to use. At the system level, facilities are organized into maps by type, with a set dedicated to operational selection.

Listing 4.28: PUF types

```
1  enum class PufType : uint32_t
2  {
3    GENERIC = 0x06000004
4    // other types can be added here
5  };
```

The 32-bit value allows stable identification even in the binary protocol (explicit endianness). The `0x06` prefix in the high byte facilitates quick checks of "namespaces" of handles. The `isValidPufType()` function avoids the use of unsupported types.
The structure `PufConfig` represents the logical and static description of a PUF, that is, the set of parameters that define its expected behavior before the instance is created. It is therefore a "configuration", non-operational object that contains all

the information needed to initialize a PUF properly. The information defined here are used by the function `createPufFromConfig()` to construct the corresponding `PufWrapper` instance.

Listing 4.29: PUF configuration structure

```
struct PufConfig {
  PufType type;
  std::vector<uint8_t> authValue; //secret/key for PUF access
  uint8_t authSize; //actual length of authValue
  uint16_t challengeSize; // in bytes (CRP input)
  uint16_t responseSize; // in bytes (CRP output expected)
  bool userWithAuth; //requires user auth (pw/HMAC)
  bool adminWithPolicy; //requires administrative policy
  bool hasSeed = false; //if present, use static seed
  std::string seedLabel; //seed label (for logging/debugging)
  PufSeedT explicitSeed = 0; // static seed value
};
```

- `type`: select the PUF to use.
- `authValue`, `authSize`: authorization material (password/HMAC key) and its length; used in `generateChallengeResponse()` to authorize access to the PUF.
- `challengeSize`, `responseSize`: sizing of the CRP (validation constraint in input/output); keeps buffers consistent and prevents overflow/underflow.
- `userWithAuth`: enable "user" access control (TPM tag `TPM_ST_SESSIONS` required for authenticated access).
- `adminWithPolicy`: reserved for advanced policies (not yet operational here), but already provided for future command constraints/conditions.
- `hasSeed`, `explicitSeed`, `seedLabel`: enable a declarative *static seed* for determinism and test reproducibility; the label facilitates traceability and debugging.

The structure `PufWrapper` instead represents the **materialization runtime** of a PUF within the TPM module. While the configuration defines static parameters, the wrapper maintains the PUF *operational status*, such as:

- the concrete implementation reference (`Generic`);
- the active authorization values;
- the seed initialization state (static or generated);

- the login information (user or policy).

It is the logical equivalent of an "active object" within the TPM, used directly by challenge–response functions.

Listing 4.30: PUF instance wrapper

```
using ConcretePuf = std::variant<SramPuf, RingOscillatorPuf,
    ArbiterPuf>;

struct PufWrapper {
  PufType type; //concrete type of instance
  ConcretePuf pufInstance; //concrete implementation (variant)

  uint8_t authValue[32]; // copy of secret (bounded)
  uint8_t authSize; //actual length of secret

  bool userwithAuth = true; //requires auth user
  bool adminWithPolicy = false; //policy admin preparation

  bool hasStaticSeed = false; //use of static seed
  PufSeedT staticSeed = 0; // static seed value

  bool hasGeneratedSeed = false; //if generated at runtime
  PufSeedT generatedSeed = 0; // generated seed value
};
```

- `type`: replicate configuration type for consistency and runtime controls.
- `pufInstance`: encapsulates the concrete object (`Generic`) into a `std::variant`.
- `authValue[32]`, `authSize`: local copy and size of authorization material; fixed maximum length array to avoid reallocations and simplified bounds checks.
- `userwithAuth`, `adminWithPolicy`: enable access control paths; if either is `true` authenticated access requires `0x8002` tags (TPM sessions).
- `hasStaticSeed`, `staticSeed`: if present, the PUF uses a predefined (deterministic) seed; useful for repeatable tests and comparisons against *ground truth*.
- `hasGeneratedSeed`, `generatedSeed`: in the absence of static seed, the first access generates a random seed (persisted in the wrapper) to make the behavior of the instance stable along the TPM lifecycle.

The `PufWrapper` then provides the direct interface between the logic of the TPM and the concrete implementations of PUFs, maintaining a polymorphic model via `std::variant` and `std::visit`.

Listing 4.31: Maps and PUF management sets

```
1  std::unordered_map<PufType, PufConfig> _pufConfigs; // copy of the
     params
2     std::unordered_map<PufType, PufWrapper> pufs; //each PUF is
     identified by the
3 type std::unordered_set<PufType> chosenPufs; //this set is used to
     take trace to the puf that we would like to use from the startup
     of the TPM until the shutdown of the TPM.
```

- `_pufConfigs`: "catalogue" of declarative configurations; each entry defines how a PUF of that type will be constructed and protected.
- `pufs`: map of instances actually created and ready to use; key `PufType` for O(1) access at CRP request stage.
- `chosenPufs`: set of PUF types "enabled" at runtime via the `TPM_Create` command; avoids accidental use of PUFs not intended in the current scenario.

The process of initializing PUFs occurs in two distinct stages:

1. the stage of **declaration**, where configurations (`PufConfig`) are registered in `_pufConfigs`;
2. the step of **materialization**, in which `populatePufsFromConfig()` creates the concrete instances (`PufWrapper`) using the function `createPufFromConfig()`.

The `createPufFromConfig()` acts as a factory, selecting the appropriate constructor based on the type of PUF indicated in the configuration. The sizes of `challenge` and texttt response are converted to bits.
Populating via `populatePufsFromConfig()` transfers all security parameters (authorization value, digest size, access policies, seeding strategy) to the runtime structure and allocates instances in memory, maintaining explicit control over the lifecycle. During this stage, initialization of the `authValue` and `authSize` fields establishes object access limits, while the `userWithAuth` and `adminWithPolicy` flags define the level of protection required for access. The function is designed to be idempotent: any instances already created are not reconstructed, ensuring consistency of the `pufs` map content and reducing risks of state inconsistency.

4.4.2 Challenge-Response flow

The heart of the implementation lies in `generateChallengeResponse()`, which implements the entire generation flow of the *Challenge–Response Pair (CRP)*. The structure of the data packet sent to the `generateChallengeResponse()` function is shown below.

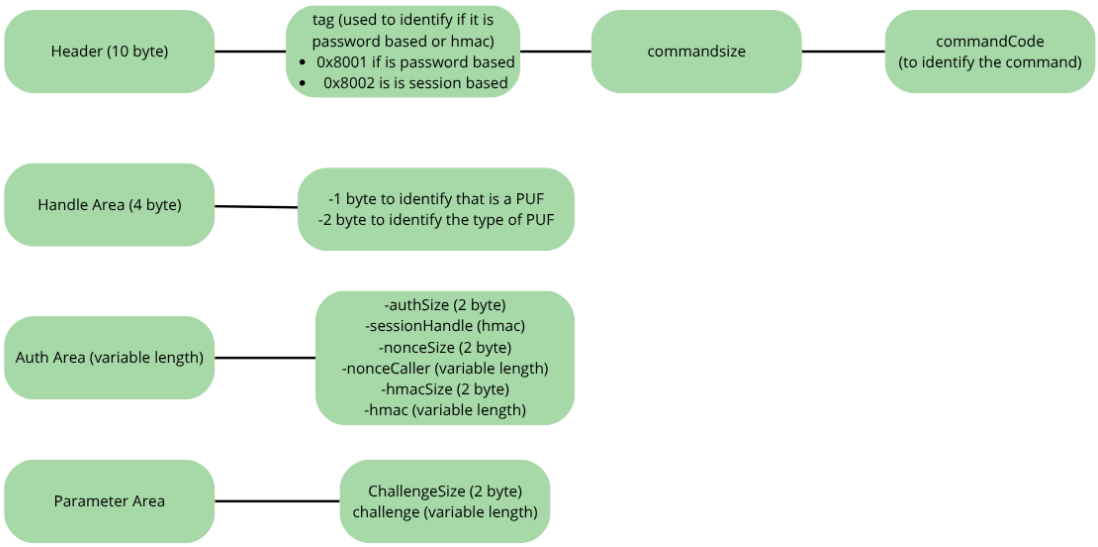

Figure 4.2: Structure of the data packet sent to the generateChallengeResponse function

Parsing and preliminary validation

The function parses the received packet according to the binary format of TPM commands (`tag|size|cmd|payload`) and extracts the PUF identifier (handle). Handle validity is checked in three steps:

1. prefix check `0x06` (PUF namespace identifier);
2. type validation via `isValidPufType()`;
3. checks that the PUF is present in the `chosenPufs` set, i.e. among those activated in the current life cycle.

This step ensures that each access is consistent with the current state of the TPM and that uninitialized or deactivated types cannot be addressed.

Authorization

Authorization control is then performed, compliant with the TPM 2.0 model. Each PUF can be:

- **free**, if no authorization mechanism is required;
- **authorization with password** (tag `0x8001`);
- **authorization via HMAC/policy session** (tag `0x8002`).

For PUFs with flags `userWithAuth` or `adminWithPolicy`, the use of the tag `TPM_ST_SESSIONS (0x8002)` is mandatory; any requests with legacy tags `0x8001` are rejected. In the case of authorization via HMAC, the function delegates validation to `hmacAuth()`, which calculates the digest on the session area and returns the new nonce to be included in the response.
The following snippet highlights security enforcement:

Listing 4.32: Tag control and TPM authorization

```
if ((puf.userwithAuth || puf.adminWithPolicy) && authSelector != 0x8002)
    throw std::runtime_error("TPM_ST_SESSIONS required for PUF access");
```

Even though the mechanism of *policy-based authorization* is not yet fully operational, the data structures (`authPolicy`, `adminWithPolicy`) and control paths are already in place, allowing in the future the introduction of complex policies based on digest and command constraints, in line with the TCG specifications.

Validation of the challenge

After the authorization step, the function `generateChallengeResponse()` proceeds with the validation of the *challenge* field, which represents the data of entrance to the PUF. The challenge is a byte vector provided by the caller and must respect the dimensions expected from the configuration associated with the PUF type. Incorrect validation of this phase causes errors.

Listing 4.33: Parsing and challenge validation

```
uint16_t challengeSize = read_u16_be(offset);
auto it = _pufConfigs.find(static_cast<PufType>(pufType));
if (en == _pufConfigs.end())
    throw std::runtime_error("PUF type not found in _pufConfigs");

if (challenge Size != it->second.challengeSize)
    throw std::runtime_error("Wrong challenge size");

//offset now points to the actual start of the challenge
if (offset >= command.size())
    throw std::runtime_error("packet missing challenge");

//extracting the challenge from the package
std::vector<uint8_t> challenge(command.begin() + offset, command.end());

if (challenge.size() != it->second.challengeSize)
    throw std::runtime_error("wrong challenge size");
```

Three main checks are performed at this stage:

1. correspondence between the PUF type and the configuration registered in `_pufConfigs`;
2. consistency of the challenge size with that declared in `PufConfig`;
3. validity of the offset within the packet to avoid out-of-bounds readings.

Seed management and controlled determinism

Once the challenge has been validated, the function determines the seed (`seed`) to be used for processing. In the real model, the seed represents the origin of the physical uniqueness of the PUF; in the simulated context, it is modeled as a pseudorandom integer value that guarantees the same property of non-replicability and stability over time.
Each PUF can operate in two modes:

- **Static seed:** the seed is declared in the configuration (`hasSeed = true`) and is used for all subsequent responses. This mode is useful in testing and validation, as it allows to obtain deterministic and reproducible results.
- **Runtime seed:** if there is no explicit seed, it generates a random one on the first request via the function `generateRandomSeed()`. The generated value is then stored inside the `PufWrapper`, so that all subsequent responses of the same PUF remain consistent throughout the session.

Listing 4.34: Managment of Seed

```
PufSeedT chosenSeed = 0;
if (puf.hasStaticSeed) {
    chosenSeed = puf.staticSeed;
} else {
    if (!puf.hasGeneratedSeed) {
        chosenSeed = generateRandomSeed();
        puf.generatedSeed = chosenSeed;
        puf.hasGeneratedSeed = true;
    } else {
        chosenSeed = puf.generatedSeed;
    }
}
```

This logic allows **reproducibility and realism** to be combined: in the absence of static seed, the simulation generates a pseudorandom behavior analogous to the one equal to a physical system, but ensuring consistency between multiple invocations of the same PUF. After seed selection, the value is injected into the

concrete instance via the functions `serialize()` and `deserialize()`, to change its internal state in a transparent but controlled manner. The combination of validated challenge and controlled seed ensures that the CRP calculation is both **safe**, **reproducible** and **traceable** in the TPM lifecycle.

Challenge-Response Elaboration

Once the *challenge* has been validated and the *seed* to be used has been determined, the module proceeds with the actual processing of the *Challenge–Response Pair* (CRP). In the implemented model, the concrete PUF (`Generic`) exposes a function `crp()` that receives as input the vector `challenge` and returns a vector `response` containing the response generated. The call is made polymorphically via the standard function `std::visit`, which applies a *lambda* to the instance contained in the variant `pufInstance`, ensuring that the correct method is invoked on the concrete type.

```
1     appPayload = std::visit(
2             [&](auto& impl) -> std::vector<uint8_t> {
3                 using ImplT = std::decay_t<decltype(impl)>;
4                 if constexpr (std::is_same_v<ImplT, std::monostate>)
    {
5                     throw std::runtime_error("PUF instance not
    initialized");
6                 } else {
7                     return impl.crp(challenge);
8                 }
9             },
10            puf.pufInstance
11        );
```

Using `std::visit` allows to maintain a dynamic **dispatch** and safe to compile-time, avoiding the use of virtual pointers and ensuring efficiency and typified safety. In this way, the infrastructure can handle multiple PUF implementations without sacrificing performance or breaking the abstraction of the TPM model.

Response generation

The final response is encapsulated in a packet conforming to the TPM 2.0 response format: `tag | size | rc | payload`, including the new nonce generated for HMAC sessions.

4.5 Integration of the TPM module into the gem5 simulator

After implementing the internal logic of the TPM module, it's necessary to integrate the module within the simulation environment gem5. That phase included two main components:

1. the device definition in Python, through the class `TPMDevice` (file `TPMDevice.py`), describing parameters of configuration and the module interface;
2. the inclusion of the device in the simulated RISC-V platform, by the script `tpm.py`, which builds the whole system (CPU, memory, bus and peripherals).

4.5.1 Device definition: `TPMDevice.py`

The file `TPMDevice.py` introduces a new Python class that extends `BasicPioDevice`, the basic gem5 abstraction for devices memory-mapped (MMIO). The class associates the C++ module (`dev/TPM_Device/tpm_device.hh`) to its Python representation, specifying its fundamental parameters:

- `pio_addr`: base address in I/O bus (0x30000000);
- `pio_size`: address space reserved for the device;
- `pio_delay`: simulated access latency (default 10ns);
- `PufConfig`: the structure of the parameters associated to the PUFs;
- `tpm_keys`: structure of tthe default keys loaded in TPM;

- $\texttt{nv}_s torage_p ath : structure of the file's path to save persistent memory.$

Listing 4.35: TPMDevice.py

```
1  from m5.params import *
2  from m5.objects import BasicPioDevice
3
4  class TPMDevice(BasicPioDevice):
5      type = 'TPMDevice'
6      cxx_header = "dev/TPM_Device/tpm_device.hh"
7      cxx_class = "gem5::TPMDevice"
8
9      pio_addr = Param.Addr(0x30000000, "Base address")
10     pio_size = Param.Addr(0x50000, "Size of address range")
11     pio_delay = Param.Latency('10ns', "Delay")
12     nv_storage_path = Param.String("", "Optional path to persistent
   NV storage file")
```

```
13
14      PufConfig = VectorParam.String([
15          "0x06000004@default_generic:AABBCC:64:32:1:0"  # SRAM PUF
    seed label optional
16      ], "Fixed PUF configurations")
17
18      tpm_keys = VectorParam.String([], "List of predefined keys:
    handle:modulus:exponent:privExponent")
```

Using `VectorParam.String` parameters allows passing directly from the Python layer to the C++ logic the PUF configurations, encoding them in compact form.

4.5.2 Integration into simulated RISC-V system: `tpm.py`

The script `tpm.py` builds the entire system architecture simulated in gem5, integrating the TPM module as a peripheral *memory-mapped* connected to main bus. The implemented architecture reproduces a minimal but complete RISC-V system, built to support TPM module operation in *full-system* mode.
The platform consists of:

- a CPU of type `TimingSimpleCPU`, capable of modeling execution times and memory accesses in detail;
- a DRAM memory subsystem `DDR3_1600_8x8`, connected using the main memory bus;
- a dual communication bus (`SystemXBar` and `IOXBar`) to separate memory traffic and I/O traffic;
- a TPM device connected via MMIO to the address `0x30000000`;
- a RISC-V interrupt system (`CLINT` and `PLIC`) and a hardware reference timer (`RiscvRTC`).

Listing 4.36: CPU definition and memory subsystem

```
system.cpu = TimingSimpleCPU()
system.cpu.isa = RiscvISA()
system.cpu.createInterruptController()

system.mem_ranges = [AddrRange(0x80000000, size='512MB')]
system.mem_ctrl = MemCtrl()
system.mem_ctrl.dram = DDR3_1600_8x8()
system.mem_ctrl.dram.range = system.mem_ranges[0]
system.mem_ctrl.port = system.membus.mem_side_ports
```

The choice of CPU `TimingSimpleCPU` was dictated by the balance between simplicity and temporal realism: the model allows for simulating delays in accessing the bus and memory, providing an accurate basis for evaluating the behavior of the TPM and its PIO operations.
The 512 MB DDR3 DRAM represents a typical configuration for embedded systems or experimental RISC-V platforms, sufficient for performing bare-metal tests and for managing TPM communication buffers.

Listing 4.37: Integration of TPM into the gem5 system

```
system.tpm = TPMDevice(
    pio_addr=0x30000000,
    pio_size=0x50000,
    pio_delay='10ns',
    PufConfig=[
        "0x06000001:AABBCC:64:32:1:0",
        "0x06000003@0x9ABCDEF0:AABBCC:128:16:1:0"
    ],
    tpm_keys=[
        "0x81000001:D2C1337B...A1"
    ]
)
system.tpm.pio = system.membus.mem_side_ports
```

The device is then connected to the main memory bus via the `pio` interface, making it accessible to the CPU as a peripheral MMIO. Thanks to this approach, TPM operations can be performed by simulated software (e.g. a bare-metal kernel) writing and reading in range-mapped addresses `0x30000000-0x3004FFFF`.

To enable firmware execution and communication with the simulated TPM, the device was integrated within the RISC-V platform `HiFive`, already available in the `gem5` framework. This platform provides a complete environment, including an interrupt handler, a real-time clock (RTC), a serial console (UART) and a PCI host for routing peripherals off-chip.
The aim has been to place the TPM in an architectural context realistic, where the CPU can communicate with the module via the I/O bus and generate events or prints via the UART.

Listing 4.38: Integration of TPM in HiFive platform

```
# RISC-V Platform (HiFive)
system.platform = HiFive()
system.platform.rtc = RiscvRTC(frequency=Frequency('100MHz'))
system.platform.clint.int_pin = system.platform.rtc.int_pin

# Terminale UART
system.platform.terminal.outfile = 'stdoutput'
```

This section of the configuration code in gem5 is crucial because it allows to customize the parameters of the simulated TPM without the need to recompile the entire environment or modify the simulator source code. The TPM is connected to the HiFive platform via the `membus`, which conveys memory-mapped I/O operations. While the UART remains responsible for external communication (textual output and debugging). Synchronization with the timer and interrupts is handled by the subsystem CLINT/PLIC, which guarantees a correct execution sequence even in presence of simulated latencies (eg. `pio_delay = 10ns`). During the simulation, the bare-metal firmware (`tpm_test.elf`) communicates directly with the TPM and produces messages through the UART, redirected on the file `stdoutput`. This makes it possible to monitor the TPM initialization steps in real time, the responses to commands and the execution of PUF operations.

The script loads an ELF executable (`tpm_test.elf`) compiled for RISC-V which interacts with the simulated TPM, initializing it and sending test commands such as `Startup`, `NV_Write`, or `PUF_ChallengeResponse`. Uploading is done via the `system.workload.bootloader` field, typical of bare-metal gem5 models.

Listing 4.39: Configuring workload and starting simulation

```
system.workload = RiscvBareMetal()
system.workload.bootloader = "tpm_test.elf"
system.cpu.createThreads()

root = Root(full_system=True, system=system)
m5.instantiate()

print("Beginning simulation!")
exit_event = m5.simulate()
print(f"Exiting @tick {m5.curTick()} because {exit_event.getCause()}"
    )
```

In this way, the complete simulation reproduces a hardware environment RISC-V with a TPM compliant with the TCG specification.

Chapter 5

Results

To assess the correctness and reliability of the implemented TPM module, a series of experimental tests was conducted within the environment of simulation `gem5`. The main objective of this evidence was to verify both the **functional behavior** of TPM commands (`_TPM_Init`, `TPM2_Startup`, `TPM2_Shutdown`), NV memory operations, PCR extensions, PUF management, and the correct **architectural integration** of the device within of a complete RISC-V system.

The test environment was configured as a *bare-metal* system, in which the CPU directly executes firmware written in C language, without the presence of an operating system. This choice allows precise control of the interface with the TPM, allowing command encoding to be analyzed in detail, the structure of the responses and the temporal effects introduced by the model of PIO communication (Programmed I/O).

All tests were performed on a 64-bit RISC-V architecture simulated in `gem5`, using a dedicated firmware (`tpm_test.elf`) that implements a minimal set of TPM commands and a simple UART-based debug interface. Analysis of the results confirmed the correct implementation of the TPM logic, demonstrating the possibility of integrating PUF-based cryptographic primitives in a reliable hardware component simulated in a software environment. Subsequent sections detail the experimental environment, the structure of the test programs, and the main results obtained.

5.1 Validation of the PUF Interaction

To verify the correct functioning of the TPM, PUF (Physically Unclonable Function) and its integration within the TPM module, several experimental tests were conducted, each designed to evaluate a different scenario of authentication and

management of cryptographic parameters. The main objective was to observe the coherence and variability of the PUF response as the initial conditions — namely the seed and the challenge — were varied.

Four main test cases were run, differentiated by authentication mode and handling of seed and salt parameters:

- **Test 1 – HMAC without salt, with default seed**: In this scenario, a standard HMAC authentication was used, without the addition of a salt value. The initial seed, is provided through the configuration file `tpm.py`, remained constant to allow verification of consistency of the PUF response under equal initial conditions.

- **Test 2 – Password-based, with default seed**: In this test, authentication was performed using a static password. Again, the seed was kept constant to verify that the behavior of the module remained deterministic in the absence of random variations.

- **Test 3 – HMAC with salt, with default seed**: A known salt value was introduced, in order to verify the correct propagation of the parameter within the authentication. Two different challenges were sent to the PUF to verify that two different challenges give two different responses.

- **Test 4 – HMAC with salt and randomly generated seed**: In the last experiment, a seed was dynamically generated at the start of the simulation. The results showed a consistent variation of responses, confirming the correct implementation of the internal entropy mechanism.

During execution: the **challenge** sent to the PUF consists of a vector of 64 bytes at incremental values (from 0x00 to 0x3F); the **response** from the PUF is calculated internally on the device, authenticated using HMAC SHA-256 for test1, test3 and test4 and using the `RS_PW command`, which includes the login password directly in the packet for test2; the entire communication is via Programmed I/O (PIO) interface, which allows direct packet transfer between CPU and simulated TPM. The firmware, written in C language, explicitly handles command encoding, binary packet construction, and synchronization with the TPM status register (STS).

5.1.1 Execution Flow

Each test was performed in a bare-metal environment, with a sequence of standard TPM commands:

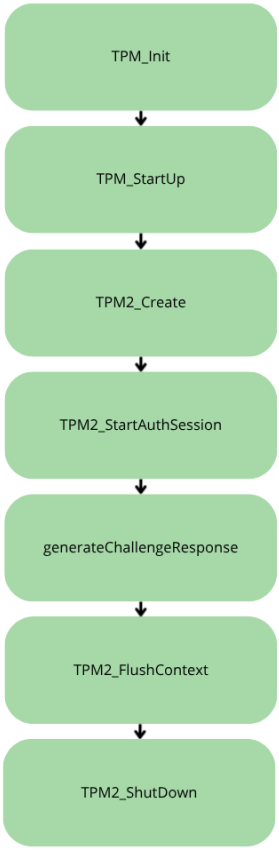

Figure 5.1: Testing workflow

TPM_StartUp / TPM_Init

- **Host:** Builds the command using *build_cmd_startup_clear*() (TAG+SIZE+CM). Sends the command to FIFO followed by $0x40$ (commandReady).
- **Device (TPM):** Waits for the Host to write $0x40$. Executes *processCommand*() $\rightarrow$| TPM2_StartUp() and returns the response code $0x20$.

TPM2_Create

- **Host:** Constructs the command as header + payload($N + N \times 4$), where N is the number of chosen PUFs. For each PUF, 4 bytes contain its identifier. Sends the command bytes to FIFO followed by $0x40$.
- **Device (TPM):** Executes *processCommand*() $\rightarrow$ TPM_Create() and returns $0x20$.

TPM2_StartAuthSession (If HMAC Authentication is Used)

- **Host:** Constructs the command using *build_cmd_start_auth_session*(), specifying tpmKeyHandle, authHandle, nonce, salt, HMAC session($0x00$), and hashAlg($0x000B$ for SHA256).
- **Device (TPM):** Executes *processCommand*() → TPM_StartAuthSession(), constructs the session, and returns the response: $0x20$ + handleOfTheSession + nonceLen + nonce.

generateChallengeResponse

- **Host:** Constructs the command using *build_cmd_challenge_hmac*(), including the PUF handle ($0x06000001$u), auth_sram, sessHandle (if applicable), nonces, and the **challenge** vector. Sends the command bytes to FIFO followed by $0x40$.
- **Device (TPM):** Executes *processCommand*() → generateChallengeResponse(). Verifies authentication, calls the PUF to generate the response, and returns: $0x20$ + nonceLen + nonce + response to the challenge.

TPM2_FlushContext

- **Host:** Constructs the command with the session handle to be flushed. Sends the command bytes to FIFO followed by $0x40$.
- **Device (TPM):** Executes *processCommand*() → TPM2_FlushContext(), flushes the session, and returns $0x20$.

TPM2_Shutdown

- **Host:** Constructs the shutdown command (command +$0x0000$).
- **Device (TPM):** Executes *processCommand*() → TPM2_Shutdown() and returns $0x20$.

In case of test 3, TPM2_StartAuthSession function was not called because a password-based authentication method was used.

During execution, two types of logs were obtained:

- Bare-metal output: containing the low-level messages and hex packets sent and received by the TPM;
- Debug log (`tpm.log`): reporting the command tags, session handles, and internal states of the TPM during processing.

Both logs confirm the correct generation of the challenge, implementation of the TPM, and authentication protocols.

5.1.2 HMAC without salt, with default seed

The first test was conducted to verify the correct functioning of the authentication phase via HMAC session and the direct interaction with the PUF configured as a TPM peripheral.
In this scenario, no salt value was used, while the seed was provided deterministically within the module, to ensure repeatability of the behavior. The HMAC session was defined as bounded, as it was associated with a specific PUF identifier (authHandle = 0x06000004), corresponding to the GENERIC PUF type.

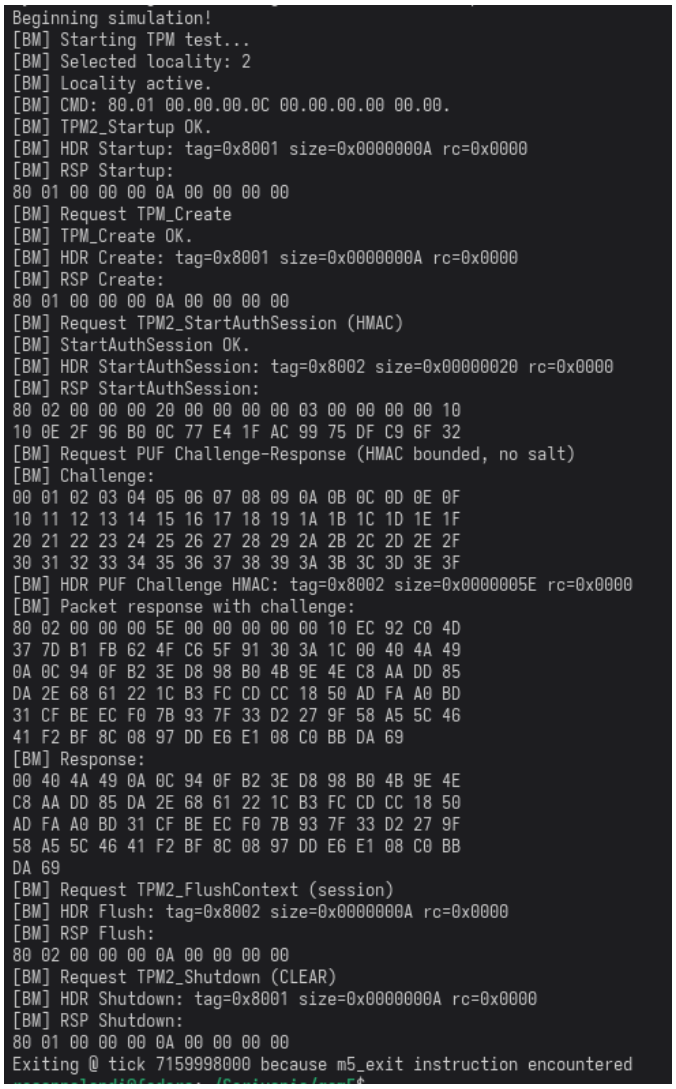

Figure 5.2: Bare-metal output - test 1

The figure 5.2 gives the log of bare-metal execution. The initial message confirms

the activation of location 2 and the start of the test.
The sequence of commands highlights the following main stages:

1. **Initializing the TPM** – the `TPM2_Startup(CLEAR)` command is successfully executed (`rc = 0x0000`), activating the module's operating context.

2. **Creating the PUF** – the `TPM_Create` command instantiates a PUF of type `0x06000004`, registering it as an authenticable entity.

3. **HMAC session start** – the `TPM2_StartAuthSession` command initializes an authentication session with a 16-byte nonceCaller and null salt. The TPM returns a valid session handle (`0x03000000`), which will be used in subsequent commands.

4. **Execute challenge–response protocol** – the firmware sends the challenge 64 bytes in length and receives a response of the same length as shown in the [BM] Response field. Note that the output buffer contains 66 bytes, since the first two bytes encode the length of the response (64 bytes in this case), followed by the actual PUF response.

5. **State termination and persistence** – the `TPM2_FlushContext` and `TPM2_Shutdown(CLEAR)` commands run regularly, ensuring session closure and TPM NV state storage.

At all stages, the return code (`rc = 0x0000`) confirms the absence of functional errors and the correct interpretation of the packets by the simulated device.

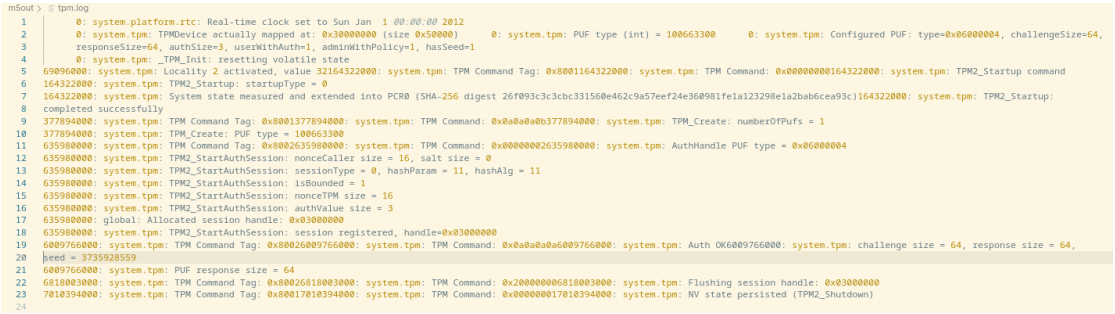

Figure 5.3: Debug - test1

The internal log (`tpm.log`) confirms the sequence and parameters of the executed commands:

- Configured PUF: type = 0x06000004, challengeSize = 64, responseSize = 64, authSize = 3;

- HMAC session: nonceCaller = 16 bytes, salt = 0, hashAlg = SHA-256;
- Session handle: 0x03000000;
- PUF response: PUF response size = 64, Auth OK;
- State persistence: NV state persisted (`TPM2_Shutdown`).

The absence of errors and the consistency of the parameters demonstrate the correct execution of the TPM command chain and the full integration of the PUF logic in the simulated model.
During the initialization phase, the `TPM2_Startup (CLEAR)` command also activates the system measurement procedure, as reported in the log:

```
system.tpm:  System state measured and extended into PCR0 (SHA-256
digest 26f093c3c3cbc331560e462c9a57eef24e360981fe1a123298e1a2bab6cea93c)
```

This step confirms that the model retains the behavior predicted by the TPM 2.0 standard, performing SHA-256 hash calculation of the initial system content and subsequent value extension in the Platform Configuration Register (PCR0). Such an operation represents the first step in the secure boot measurement chain, ensuring the integrity of the execution environment.
From the gem5 log it's possible to observe the tick values associated with the main test events:

- `TPM2_Startup`: completed at approximately 1.64×10^8 ticks;
- `PUF Challenge-Response`: Performed between $6.00 \times 10^9$and 6.81×10^9 ticks, with an overall latency of approximately 8.00×10^8 ticks.

Comparison with standard TPM model timelines shows that PUF integration does not introduce significant time overheads. The additional latency is negligible compared to the total command processing time, confirming that the extension of the model does not compromise efficiency or synchronization between CPU and peripheral.

The test confirms the correct implementation of the HMAC protocol and the functional consistency between the TPM module and the integrated PUF. The 64 byte PUF response is stable between successive executions, certifying the repeatability of the behavior with the same seed and challenge. Furthermore, the initial extended system measure in PCR0 demonstrates the preservation of the secure boot functionalities of the standard TPM, while simulation time analysis highlights the absence of measurable overheads. Overall, the experiment confirms that PUF integration occurs transparently and efficiently, preserving the safety and performance properties of the original model.

5.1.3 Password-based, with default seed

The second experiment aimed to check how the PUF challenge–response command works using a password-based authentication mode (`RS_PW`), as an alternative to the HMAC session from the previous test. In this scenario, communication with the TPM does not involve the creation of an HMAC-type session nor the use of nonces or salts, but is based on the direct sending of a secret key (password) associated with the selected PUF.

```
Beginning simulation!
[BM2] Starting TPM test (PWD) for password-based case...
[BM2] Selected locality: 2
[BM2] Locality active.
[BM2] HDR Startup: tag=0x8001 size=0x0000000A rc=0x0000
[BM2] RSP Startup:
80 01 00 00 00 0A 00 00 00 00
[BM2] TPM2_Startup OK.
[BM2] Request TPM_Create
[BM2] HDR Create: tag=0x8001 size=0x0000000A rc=0x0000
[BM2] RSP Create:
80 01 00 00 00 0A 00 00 00 00
[BM2] TPM_Create OK.
[BM2] Request PUF Challenge-Response (PWD)
[BM2] Challenge:
00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F
10 11 12 13 14 15 16 17 18 19 1A 1B 1C 1D 1E 1F
20 21 22 23 24 25 26 27 28 29 2A 2B 2C 2D 2E 2F
30 31 32 33 34 35 36 37 38 39 3A 3B 3C 3D 3E 3F
[BM2] HDR PUF Challenge: tag=0x8002 size=0x0000004E rc=0x0000
[BM2] Packet response with challenge:
80 02 00 00 00 4E 00 00 00 00 00 00 00 40 4A 49
0A 0C 94 0F B2 3E D8 98 B0 4B 9E 4E C8 AA DD 85
DA 2E 68 61 22 1C B3 FC CD CC 18 50 AD FA A0 BD
31 CF BE EC F0 7B 93 7F 33 D2 27 9F 58 A5 5C 46
41 F2 BF 8C 08 97 DD E6 E1 08 C0 BB DA 69
[BM2] Response:
00 40 4A 49 0A 0C 94 0F B2 3E D8 98 B0 4B 9E 4E
C8 AA DD 85 DA 2E 68 61 22 1C B3 FC CD CC 18 50
AD FA A0 BD 31 CF BE EC F0 7B 93 7F 33 D2 27 9F
58 A5 5C 46 41 F2 BF 8C 08 97 DD E6 E1 08 C0 BB
DA 69
[BM2] PUF Challenge OK.
[BM2] Request TPM2_Shutdown (CLEAR)
[BM2] HDR Shutdown: tag=0x8001 size=0x0000000A rc=0x0000
[BM2] RSP Shutdown:
80 01 00 00 00 0A 00 00 00 00
[BM2] Shutdown OK.
Exiting @ tick 1847007000 because m5_exit instruction encountered
```

Figure 5.4: Bare-metal output - test 2

The execution bare metal shows the correct sequence of commands and absence of errors at all stages:

1. **Initializing the TPM** – the `TPM2_Startup(CLEAR)` command completes with `rc = 0x0000`.

2. **Creating the PUF** – `TPM_Create` registers a PUF of type `0x06000004`, as in the previous test.

3. **Execute Challenge–Response (Password-based)** – the PUF Challenge–Response (PWD) command receives a valid 64-byte response, indicated by the [BM2] Response log.

4. **Correct termination** – `TPM2_Shutdown(CLEAR)` persists non-volatile state (NV state persisted).

All commands return `rc = 0x0000`, confirming correct packet decoding and consistent PUF handling even with direct authentication.

Figure 5.5: Debug - test 2

Test 2 highlights the versatility of the PUF model integrated into the TPM, capable of operating in both authenticated mode (HMAC) and direct mode (password). With the same challenge and seed, the PUF response is identical to that obtained in test 1, confirming the internal consistency of the generator in absence of variations. Furthermore, execution occurs with latency comparable to the HMAC case, without introducing measurable overheads.

5.1.4 HMAC with salt, with default seed

In this test, the session is:

- **unbounded**, as it is not bound to a specific object (`bind = RH_NULL`);

- **salted**, as it uses an RSA-OAEP salt value encrypted with the TPM public key (`tpmKeyHandle = 0x81000001`).

After the session was created, two consecutive Challenge–Response PUF commands were executed, each authenticated using HMAC. Finally, the session was closed with `TPM2_FlushContext`, followed by `TPM2_Shutdown(CLEAR)`.

```
Beginning simulation!
[BM3] Starting TPM test (unbounded+salt, PUF SRAM+ARBITER)...
[BM3] Selected locality: 2
[BM3] Locality active.
[BM3] HDR Startup: tag=0x8001 size=0x0000000A rc=0x0000
[BM3] RSP Startup:
80 01 00 00 00 0A 00 00 00 00
[BM3] Request TPM_Create (GENERIC)
[BM3] HDR Create: tag=0x8001 size=0x0000000A rc=0x0000
[BM3] RSP Create:
80 01 00 00 00 0A 00 00 00 00
[BM3] Request TPM2_StartAuthSession (unbounded, salted)
[BM3] HDR StartAuthSession: tag=0x8002 size=0x00000020 rc=0x0000
[BM3] RSP StartAuthSession:
80 02 00 00 00 20 00 00 00 00 03 00 00 00 00 10
51 21 F3 2A 65 92 B1 41 9A 7F F3 0E CC 81 9D A1
[BM3] Request PUF Challenge-Response (HMAC, GENERIC #1)
[BM3] HDR PUF GENERIC #1: tag=0x8002 size=0x0000005E rc=0x0000
[BM3] RSP PUF GENERIC #1:
80 02 00 00 00 5E 00 00 00 00 00 10 38 F4 C5 6A
22 2D 27 7C 19 E9 A5 BC 67 B6 F9 8B 00 40 4A 49
0A 0C 94 0F B2 3E D8 98 B0 4B 9E 4E C8 AA DD 85
DA 2E 68 61 22 1C B3 FC CD CC 18 50 AD FA A0 BD
31 CF BE EC F0 7B 93 7F 33 D2 27 9F 58 A5 5C 46
41 F2 BF 8C 08 97 DD E6 E1 08 C0 BB DA 69
[BM3] Request PUF Challenge-Response (HMAC, GENERIC #2)
[BM3] HDR PUF GENERIC #2: tag=0x8002 size=0x0000005E rc=0x0000
[BM3] RSP PUF GENERIC #2:
80 02 00 00 00 5E 00 00 00 00 00 10 7C 7F C3 53
0E EF B7 EF E6 DA 1A 6C 4E C1 7E 50 00 40 98 77
8B 21 0C A5 04 61 1F D8 5F 71 52 B3 43 D0 A9 A2
EE 56 C9 B3 45 32 FD E6 3A FE B8 83 5C 7B 57 18
82 2B 65 09 09 B7 75 94 A8 FE 5D 58 0D 9D 02 8B
D1 3B B5 2B 09 D4 62 E3 30 E6 5F 85 42 29
[BM3] Request TPM2_FlushContext (session)
[BM3] HDR Flush: tag=0x8002 size=0x0000000A rc=0x0000
[BM3] RSP Flush:
80 02 00 00 00 0A 00 00 00 00
[BM3] Request TPM2_Shutdown (CLEAR)
[BM3] HDR Shutdown: tag=0x8001 size=0x0000000A rc=0x0000
[BM3] RSP Shutdown:
80 01 00 00 00 0A 00 00 00 00
Exiting @ tick 12182347000 because m5_exit instruction encountered
```

Figure 5.6: Bare-metal output - test 3

In the following figure, it's possible to observe that, differently from test 1, the size of salt is different from zero.

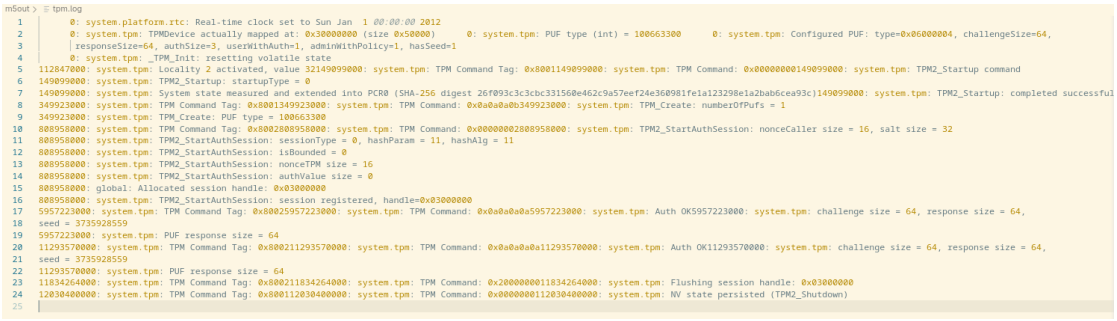

Figure 5.7: Debug - test 3

Test 3 demonstrates that the HMAC mechanism with external salt is fully compatible with PUF management. The use of the KDFa function (SHA-256) to derive the session key from nonceCaller, nonceTPM, and salt ensures a high level of security and prevents accidental reuse of keys between different sessions.
The TPM was able to:

- create and manage salted session,
- correctly calculates authentication HMACs,
- generate consistent and repeatable PUF responses.

The two consecutive requests (GENERIC #1 and #2) returned different responses because the PUF received two different challenges.

5.1.5 HMAC with salt and randomly generated seed

The fourth test replicates the configuration of Test 3 (HMAC session unbounded + salted), but with one substantial difference: the enabling of random seed within the TPM. In this case, the PUF component and the KDFa function are initialized with a dynamically generated seed value at startup.

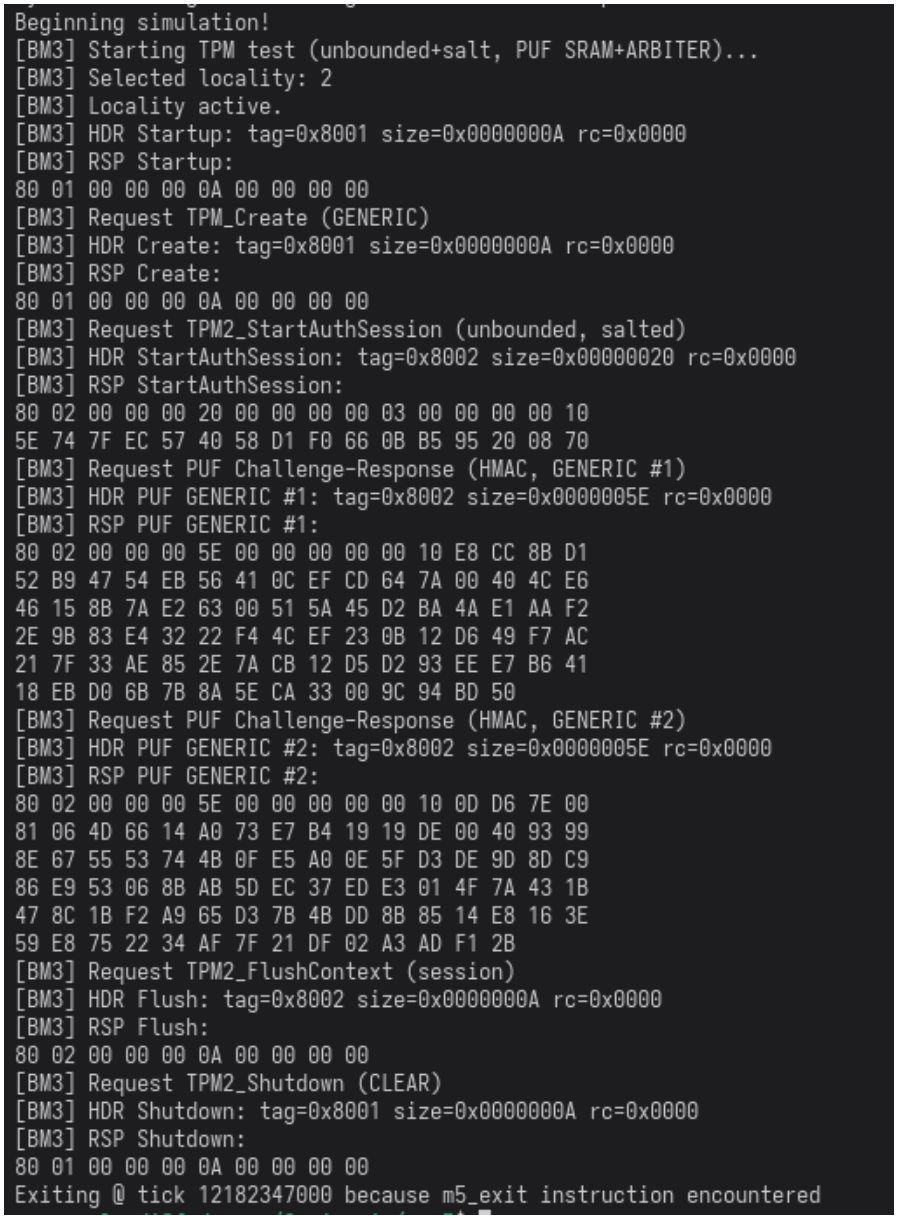

Figure 5.8: Bare-metal output - test 4

The only functional difference lies in the `hasSeed=0` configuration, which enables pseudo–random generation of the seed internal to the PUF component, visible in the TPM initialization log.

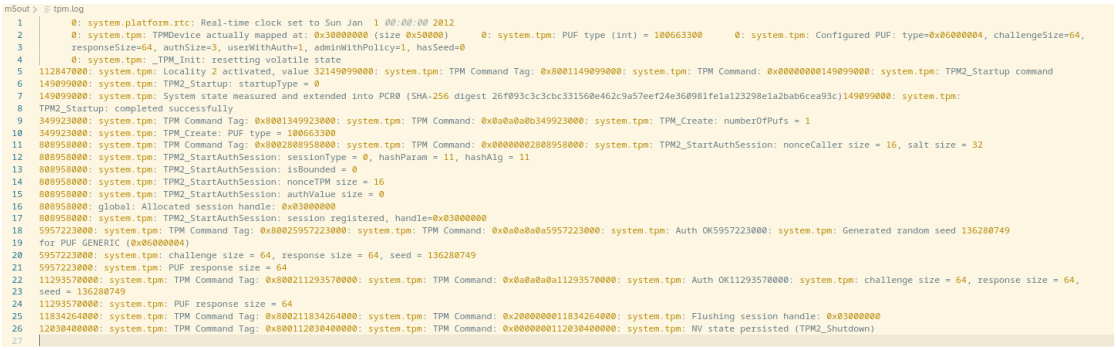

Figure 5.9: Debug - test 4

Introduction of a random seed has a direct impact on the variability of the PUF response, but does not alter the behavior of the rest of the system. Authentication pipeline, HMAC session management and communication through PIO maintain the same latency and structure.

This experiment demonstrates that internal seed can be used as a source of hardware entropy to increase the diversity of PUF responses, without introducing significant instability or overhead into the simulation.

5.1.6 Final Evaluation

To evaluate the fundamental properties of the PUF — **coherence** (stability of the response with the same input) and **variability** (difference between responses to different inputs) —, the challenge and response values for each test were collected. The results will be summarized in a comparative table, showing for each configuration:

Table 5.1: Analysis of the variability of the PUF response

Test	Challenge (hexadecimal)	Seed PUF	Response (64B)
Test 1	00010203...3E3F	3735928559	4A 49 0A 0C ... C0 BB DA 69
Test 2	00010203...3E3F	3735928559	4A 49 0A 0C ... C0 BB DA 69
Test 3	00010203...3E3F	3735928559	4A 49 0A 0C ... C0 BB DA 69
	40414243...7E7F	3735928559	98 77 8B 21 ... 5F 85 42 29
Test 4	00010203...3E3F	136280749	4C E6 46 15 ... 9C 94 BD 50
	40414243...7E7F	136280749	93 99 8E 67 ... A3 AD F1 2B

Such analysis allows to check:

- that with the same seed and challenge the response remains stable (consistency);

- that at different seeds or challenges the response varies significantly (variability), confirming the correct behavior of the PUF as a source of hardware entropy.

5.2 Validation of Non-Volatile Memory Retention Across Simulations

To check that the simulated TPM is handling non-volatile memory (NV) correctly, a bare-metal test (`test4_tpm_nv.c`) has been developed that defines, writes and reads an NV index. The program also runs a persistent counter, updated at each startup, to demonstrate data survival between distinct simulation sessions.

The following code defines an NV space (`NV_DefineSpace`), increments a counter, writes updated data, and finally reads it again for verification. At the end, a command from `TPM2_Shutdown` is executed to force persistent memory to be written to disk:

```
[BM4] NV index not present - provisioning baseline data
[BM4] NV_Write updated counter to 00000001
[BM4] NV memory demo complete
[BM4] Re-run this workload with the same nv_storage_path
       to watch the counter persist.
```

During the first run, the TPM finds no pre-existing NV index:

- the line appears in the bare-metal log `NV index not present - provisioning baseline data`;

- the file `tpm.log` shows the index creation with the command `TPM2_NV_DefineSpace`;

- a start counter of 1 is written.

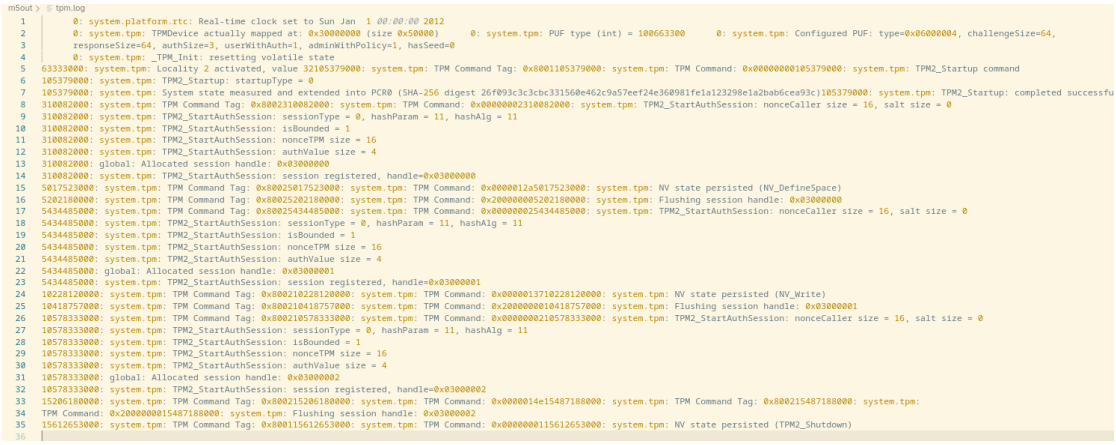

Figure 5.10: Debug - NV Memory

```
Beginning simulation!
[BM4] test4_tpm NV demo start
[BM4] Locality 2
[BM4] Startup HDR: tag=0x8001 size=0x0000000A rc=0x0000
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_DefineSpace CMD:
80 02 00 00 00 63 00 00 01 2A 40 00 00 01 00 00
00 39 03 00 00 00 00 10 B0 B1 B2 B3 B4 B5 B6 B7
B8 B9 BA BB BC BD BE BF 00 00 20 F7 4D BF F8 6C
67 1E E0 99 12 7A 4F 22 C9 13 8C 75 B6 33 1A B9
FA 54 AD 2F B1 5D F9 AA A3 4B 72 00 04 DE AD BE
EF 00 10 01 50 00 20 00 0B 00 06 00 06 00 00 00
20 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000000E rc=0x0000
[BM4] NV_DefineSpace done (fresh index)
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] NV index not present - provisioning baseline data
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_Write CMD:
80 02 00 00 00 73 00 00 01 37 01 50 00 20 01 50
00 20 00 00 00 39 03 00 00 01 00 10 D0 D1 D2 D3
D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF 00 00 20 DD
D2 BD EE 67 69 59 E9 55 C0 8C 32 EB 9F E9 09 4B
7D 6C 00 08 72 AB A7 38 8E 0D 4B 66 AD 22 D6 00
20 00 00 00 01 4E 56 2D 44 45 4D 4F 20 70 65 72
73 69 73 74 65 6E 74 20 70 61 79 6C 6F 61 64 20
31 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] NV_Write updated counter to 00000001
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_Read CMD:
80 02 00 00 00 53 00 00 01 4E 01 50 00 20 01 50
00 20 00 00 00 39 03 00 00 02 00 10 F0 F1 F2 F3
F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 00 00 20 9A
24 52 8D BA DF 01 16 C1 FA E8 3F 71 4B 99 1E D1
A8 5F 45 7F 6E 14 5C D9 85 BB 01 4D 02 86 E1 00
20 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000002C rc=0x0000
[BM4] NV_Read data:
00 00 00 01 4E 56 2D 44 45 4D 4F 20 70 65 72 73
69 73 74 65 6E 74 20 70 61 79 6C 6F 61 64 20 31
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] Shutdown HDR: tag=0x8001 size=0x0000000A rc=0x0000
[BM4] NV memory demo complete
[BM4] Re-run this workload with the same nv_storage_path to watch the counter persist.
Exiting @ tick 15853996000 because m5_exit instruction encountered
```

Figure 5.11: Bare-metal output - NV Memory

In a second simulation, launched with the same parameter `nv_storage_path`, the TPM detects the presence of the previously defined index:

- prints `NV index already exists, expecting persisted data`;
- the output clearly shows the line `Retrieved persisted counter = 0x00000001`;
- the counter stored in the NV memory is read and incremented from 1 to 2;

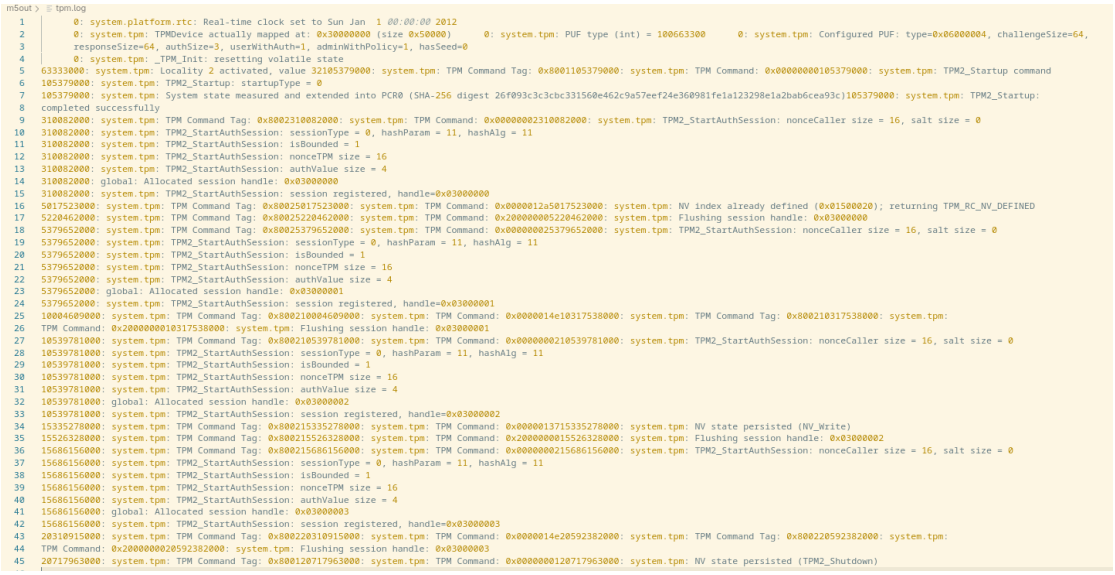

Figure 5.12: Debug - NV Memory 2

```
Beginning simulation!
[BM4] test4_tpm NV demo start
[BM4] Locality 2
[BM4] Startup HDR: tag=0x8001 size=0x0000000A rc=0x0000
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_DefineSpace CMD:
80 02 00 00 00 63 00 00 01 2A 40 00 00 01 00 00
00 39 03 00 00 00 00 10 B0 B1 B2 B3 B4 B5 B6 B7
B8 B9 BA BB BC BD BE BF 00 00 20 9A 41 43 46 6D
D3 F5 F8 18 2A 36 92 CE EB 11 5F F2 39 0A 75 7B
BE 0D DE 56 74 3F 7B C7 F4 29 BD 00 04 DE AD BE
EF 00 10 01 50 00 20 00 0B 00 06 00 06 00 00 00
20 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000000A rc=0x0140
[BM4] NV index already exists, expecting persisted data
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_Read CMD:
80 02 00 00 00 53 00 00 01 4E 01 50 00 20 01 50
00 20 00 00 00 39 03 00 00 01 00 10 91 92 93 94
95 96 97 98 99 9A 9B 9C 9D 9E 9F A0 00 00 20 65
67 89 6C A3 B7 3F 53 7F 8C 77 3F D4 43 C6 F9 43
93 CA 42 C5 B9 9C C3 0B E2 D5 4F CD 52 EB 64 00
20 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000002C rc=0x0000
[BM4] Persisted NV payload:
00 00 00 01 4E 56 2D 44 45 4D 4F 20 70 65 72 73
69 73 74 65 6E 74 20 70 61 79 6C 6F 61 64 20 31
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] Retrieved persisted counter = 0x00000001
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_Write CMD:
80 02 00 00 00 73 00 00 01 37 01 50 00 20 01 50
00 20 00 00 00 39 03 00 00 02 00 10 D0 D1 D2 D3
D4 D5 D6 D7 D8 D9 DA DB DC DD DE DF 00 00 20 CB
FD 4F 47 42 D5 2B 93 DA 69 8F DE 70 0D E4 79 62
2F F7 9D BD 43 C3 C7 99 D3 34 7D B5 16 2D FA 00
20 00 00 00 02 4E 56 2D 44 45 4D 4F 20 70 65 72
73 69 73 74 65 6E 74 20 70 61 79 6C 6F 61 64 20
32 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] NV_Write updated counter to 00000002
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] StartAuthSession HDR: tag=0x8002 size=0x00000020 rc=0x0000
[BM4] NV_Read CMD:
80 02 00 00 00 53 00 00 01 4E 01 50 00 20 01 50
00 20 00 00 00 39 03 00 00 03 00 10 F0 F1 F2 F3
F4 F5 F6 F7 F8 F9 FA FB FC FD FE FF 00 00 20 09
A8 85 5E B1 B0 21 C4 A3 AC DF BC 81 4F 37 36 F2
79 2F 28 23 D5 6A 80 B5 32 A8 2B 5C 19 55 EC 00
20 00 00
[BM4] Command HDR: tag=0x8002 size=0x0000002C rc=0x0000
[BM4] NV_Read data:
00 00 00 02 4E 56 2D 44 45 4D 4F 20 70 65 72 73
69 73 74 65 6E 74 20 70 61 79 6C 6F 61 64 20 32
[BM4] Flush HDR: tag=0x8002 size=0x0000000A rc=0x0000
[BM4] Shutdown HDR: tag=0x8001 size=0x0000000A rc=0x0000
[BM4] NV memory demo complete
[BM4] Re-run this workload with the same nv_storage_path to watch the counter persist.
Exiting @ tick 20959346000 because m5_exit instruction encountered
```


Figure 5.13: Bare-metal output - NV Memory 2

The results show that:

- NV memory correctly maintains content between runs, as required by the TPM specification;
- the command `TPM2_Shutdown` forces data persistence on the status file;
- on reboot, the module is able to reread previously written values.

5.2.1 Final Evaluation

Results obtained confirm the correct functional implementation of the non-volatile memory (NV) management protocol according to the specifications of TPM 2.0. The executed command sequence — `NV_DefineSpace` → `NV_Write` → `NV_Read` → `TPM2_Shutdown` — it shows that the simulated module is able to define an NV index, write a persistent payload, read it correctly in subsequent runs, and finally save the state securely after the simulation is finished.

From a performance perspective, log measurements show that write and read operations on NV introduce negligible time overhead compared to the overall flow of TPM commands. Persistent state file management does not significantly alter simulation cycle execution times, highlighting efficient integration of non-volatile memory functionality into the gem5 model.

Chapter 6

Conclusions

Despite the existence of increasingly sophisticated software security solutions — such as advanced cryptographic libraries or fully virtualized Trusted Execution environments — such measures often prove insufficient when employed in systems subject to physical attacks. The growing popularity of RISC-V architectures, especially in IoT contexts and embedded systems, makes the adoption of hardware-based security mechanisms indispensable. In this scenario, the Trusted Computing paradigm takes on a central role. This thesis aims to fill this gap by strengthening hardware security in RISC-V architectures through the integration of a Trusted Platform Module (TPM) and a Physical Unclonable Function (PUF) within the gem5 simulation environment. The main goal is to ensure a high level of protection from the early stages of system startup —a requirement not achievable through software security alone — while exploiting the unique potential of PUFs, capable of providing cryptographic guarantees based on the unique physical properties of the circuit.

The work was developed in two complementary phases:

- Implementation of the TPM 2.0 Model: A software model of TPM 2.0 has been designed and built, faithful to the specifications of the TCG (Trusted Computing Group). This model includes support for key authentication mechanisms (Password, HMAC), communication interfaces (FIFO, CRB), Platform Configuration Registers (PCR), and non-volatile memory for key and counter management.
- PUF Module Integration: The TPM model has been extended with the integration of a simulated PUF module. This allows the TPM to forward challenges and receive unique responses derived from the intrinsic physical characteristics of the circuit. This integration is essential for on-the-fly generation of hardware-related keys, eliminating the need to store static secrets

and consequently the risk of extracting a key saved in memory.

The entire TPM system was integrated into the RISC-V architecture within the gem5 emulator, exploiting its open-source nature to ensure flexibility and experimental reproducibility.

The simulation results fully confirmed the validity of the proposed approach:

- **Functional Validity of TPM**: The TPM framework developed in gem5 is able to faithfully reproduce the behavior of a real TPM. Tests have shown that standard calls to the TPM receive the expected responses, and security features, such as integrity measurement, operate according to specifications. It was possible to launch a simulated RISC-V platform and observe the correct updating of the PCR logs during the secure boot sequence.
- **Dynamic Key Generation via PUF**: PUF integration demonstrated its primary benefit: the simulated TPM was able to generate unique keys for each PUF run or fingerprint, without reusing or storing a fixed key. For example, the change in response value was verified as the challenge sent to the PUF changed (or as the simulated intrinsic characteristics of the PUF itself changed).
- **Minimum Impact on Performance**: Although a systematic and rigorous performance analysis in terms of latency and throughput was not conducted, observation of simulation logs and execution times allowed us to note that the integration of the TPM module and the addition of PUF logic do not introduce significant overhead into the system. In particular, critical operations such as system boot (boot) or integrity measurement occurred with latency times that, while observable, were found to be reasonable and not such as to compromise the usability of the system in an embedded or IoT context. This suggests that the approach is not only security-friendly, but also efficient.

This work has generated benefits that go beyond a single functional demonstration. The most significant advantage is the drastic reduction of the attack surface related to static secrets. Since there are no permanent keys stored in the TPM's non-volatile memory (NV-RAM), an attacker cannot obtain sensitive information through physical memory extraction. Any critical secret is recreated at startup using the PUF and vanishes at shutdown (power-off). This increases the level of protection against low-level attacks (such as hardware sniffing or reverse engineering of the chip). The inherent uniqueness introduced by the PUF means that even compromising a single device does not provide useful information to attack another. Each device has unpredictable and unrelated PUF responses. This is crucial in distributed network contexts such as IoT or data centers, where compromising a

node often preludes attacking similar nodes.

The simulation-based approach in gem5 constitutes a crucial methodological benefit.

- **Total Observability**: Unlike a sealed real chip, in gem5 you can print and verify all TPM internals (registers, authorization states, hash values), facilitating system debugging and validation.
- **Economical and Flexible Experimentation**: The developed framework is open source and can be used by the research community as a test bed to evaluate new security ideas. It is now possible to simulate complex scenarios, such as client-server networks or multi-device ecosystems (each node with its own TPM), at no hardware cost and without the need to build initial physical prototypes.

Developing and integrating a complex system into a simulation environment presented significant challenges. The implementation of TPM 2.0 required a thorough understanding and faithful translation of the TCG specifications, which are extremely detailed and extensive. Maintaining the consistency and logical security of the software model has been a challenging task. Integrating the TPM model into gem5, and in particular its correct interface with the RISC-V architecture (e.g., for managing memory registers and interrupts), required significant learning and debugging effort. The gem5 ecosystem, while powerful, has a steep learning curve and requires careful configuration for hardware/software customizations.

The limitations found suggest directions for improvement and new research opportunities:

- **Extension of TPM Functional Coverage**: The current implementation, while widely usable, does not cover all the functionality under TPM 2.0 (for example, advanced authorization policies or some specific types of keys). The model can be extended to achieve full equivalence with a commercial hardware TPM.
- **Formal and Performance Analysis**: A systematic performance evaluation was not conducted. It will be essential to measure the overhead introduced by the use of PUF (in terms of latency in TPM operations or boot time) using gem5's profiling capabilities.
- **Advanced Security Testing**: The framework can be used as a basis for studying the resilience of the TPM integrated with a PUF against more sophisticated attacks. Logical Side-Channel Attacks analyze memory access patterns or cache usage generated by the system to infer sensitive information. Fault

Injection Simulation inserts hooks into the simulator to emulate disturbances or errors (such as power glitches) when generating PUF responses to check their robustness. Modeling ML Attacks that repeatedly query the TPM model to attempt to build a machine learning-based predictive model, evaluating the intrinsic robustness of the simulated PUF against automated learning.

In conclusion, the work achieved the goal of creating an open source and innovative simulation environment for hardware security on RISC-V. This framework not only fills a gap in RISC-V trusted computing tools but tangibly demonstrates the feasibility and utility of combining TPM and PUF to raise the level of security, fully in tune with RISC-V's open source philosophy. It is hoped that this work can serve as a solid basis for further research and implementations.

Bibliography

[1] Andrea Pasquinucci. «Sicurezza, Hardware e Confidential Computing – Parte 1». In: *ICT Security Magazine* (2021). Accessed: 2025-11-12. URL: https://www.ictsecuritymagazine.com/articoli/sicurezza-hardware-e-confidential-computing-parte-1/ (cit. on p. 5).

[2] Wikipedia contributors. *Trusted Computing Base*. 2025-11-12. URL: https://en.wikipedia.org/wiki/Trusted_computing_base (cit. on p. 5).

[3] Wikipedia contributors. *Trusted Computing Group*. Accessed: 2025-11-12. URL: https://en.wikipedia.org/wiki/Trusted_Computing_Group (cit. on p. 6).

[4] Trusted Computing Group. *Trusted Platform Module (TPM) Summary*. Accessed: 2025-09-08. 2025. URL: https://trustedcomputinggroup.org/resource/trusted-platform-module-tpm-summary/ (cit. on p. 6).

[5] Trusted Computing Group (TCG). *What is a Trusted Platform Module (TPM)?* Accessed: 2025-09-08. URL: https://trustedcomputinggroup.org/about/what-is-a-trusted-platform-module-tpm/ (visited on 11/12/2025) (cit. on p. 6).

[6] Trusted Computing Group. *TCG Roots of Trust Specification v0.20 (Public Review)*. Tech. rep. Accessed: 2025-09-09. Trusted Computing Group, 2018. URL: https://trustedcomputinggroup.org/wp-content/uploads/TCG_Roots_of_Trust_Specification_v0p20_PUBLIC_REVIEW.pdf (cit. on p. 7).

[7] Trusted Computing Group (TCG). *Trusted Platform Module Library Specification, Family '2.0', Level 00, Part 0: Architecture*. Tech. rep. Accessed: 2025-09-08. Trusted Computing Group, 2014. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-0-Version-184_pub.pdf (cit. on pp. 7, 15).

[8] Synopsys. *What is Root of Trust?* Accessed: 2025-09-08. 2023. URL: https://www.synopsys.com/glossary/what-is-root-of-trust.html (cit. on p. 7).

[9] Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 1: Architecture*. Accessed: 2025-09-09. 2024. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-1-Version-184_pub.pdf (cit. on pp. 8, 15).

[10] Cyberraiden. *Trusted Platform Module (TPM) and its uses in Windows Operating System*. Blog Post. 2025. URL: https://cyberraiden.wordpress.com/2025/03/28/trusted-platform-module-tpm-and-its-uses-in-windows-operating-system/ (visited on 11/12/2025) (cit. on pp. 8, 12, 17).

[11] Trusted Computing Group (TCG). *TSS Overview: Common Architectures*. White Paper/Overview. Version v1.0, Revision 10. Trusted Computing Group, 2021. URL: https://trustedcomputinggroup.org/wp-content/uploads/TSS_Overview_Common_v1_r10_pub09232021.pdf (cit. on p. 9).

[12] tpm2-software contributors. *tpm2-tss: TPM2 Software Stack*. Codice sorgente e documentazione del TPM Software Stack. GitHub. URL: https://github.com/tpm2-software/tpm2-tss (visited on 11/12/2025) (cit. on p. 9).

[13] Trusted Computing Group (TCG). *Trusted Platform Module 2.0 Brief Overview*. Overview Document. Document Revision 02. Trusted Computing Group, 2019. URL: https://trustedcomputinggroup.org/wp-content/uploads/2019_TCG_TPM2_BriefOverview_DR02web.pdf (cit. on p. 10).

[14] Eric Chiang. *The Trusted Platform Module Key Hierarchy*. Accessed: 2025-09-10. 2025. URL: https://ericchiang.github.io/post/tpm-keys/ (cit. on pp. 11, 12).

[15] Sundeep Bajikar. *Trusted Platform Module (TPM) Based Security on Notebook PCs - White Paper*. White paper. Accessed: 2025-09-08. 2002. URL: https://www.intel.com/content/www/us/en/architecture-and-technology/trusted-platform-module-tpm-based-security-paper.html (cit. on pp. 11, 15).

[16] tpm2-software contributors. *Remote Attestation*. Accessed: 2025-09-10. 2019. URL: https://tpm2-software.github.io/tpm2-tss/getting-started/2019/12/18/Remote-Attestation.html (cit. on p. 13).

[17] CYSEC. *FluidOS Project*. 2025-09-11. URL: https://www.cysec.com/fluid-os-project/ (cit. on p. 14).

[18] Margie Ruffin, Chenkai Wang, Gheorghe Almasi, Abdulhamid Adebayo, Hubertus Franke, and Gang Wang. «Towards Continuous Integrity Attestation and Its Challenges in Practice: A Case Study of Keylime». In: *IBM Research* (2023). Accessed: 2025-09-13. URL: https://github.com/mruffin/Dynamic-Policy-Generator (cit. on p. 14).

[19] Trusted Computing Group (TCG). *TPM Keys for Platform Identity*. Tech. rep. Accessed: 2025-09-08. Trusted Computing Group. URL: https://trustedcomputinggroup.org/wp-content/uploads/TPM_Keys_for_Platform_Identity_v1_0_r3_Final.pdf (cit. on p. 15).

[20] Microsoft. *Panoramica di BitLocker*. Accessed: 2025-09-08. URL: https://learn.microsoft.com/it-it/windows/security/operating-system-security/data-protection/bitlocker/ (cit. on p. 17).

[21] Microsoft. *Modalità di utilizzo del TPM (Trusted Platform Module) da parte di Windows*. Accessed: 2025-09-08. URL: https://learn.microsoft.com/it-it/windows/security/hardware-security/tpm/how-windows-uses-the-tpm (cit. on p. 17).

[22] Anestis Papakotoulas, Theodoros Mylonas, Kakia Panagidi, and Stathes Hadjiefthymiades. «Optimizing IoT Security via TPM Integration: An Energy Efficiency Case Study for Node Authentication». In: *ITU Journal on Future and Evolving Technologies* (2024). Accessed: 2025-09-11. URL: https://www.itu.int/pub/S-JNL-VOL5.ISSUE1-2024-A06 (cit. on p. 19).

[23] Trusted Computing Group (TCG). *Advisory on TCG Critical Vulnerability TCGVRT0007*. Tech. rep. Accessed: 2025-09-11. Trusted Computing Group (cit. on p. 19).

[24] Jeremy Boone and NCC Group. *TPM Genie: A Framework for Hardware-Level TPM Security Research*. Tech. rep. Accessed: 2025-09-13. NCC Group, 2020 (cit. on p. 19).

[25] Elaine Barker and John Kelsey. *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*. Tech. rep. SP 800-90A Rev. 1. Accessed: 2025-09-13. National Institute of Standards and Technology (NIST), 2015. DOI: 10.6028/NIST.SP.800-90Ar1. URL: https://csrc.nist.gov/pubs/sp/800/90/a/r1/final (cit. on p. 19).

[26] Robert Markel, Dennis Scharwachter, Jan Schilling, Christopher Kühr, and Thorsten Holz. «faulTPM: Exposing AMD fTPMs' Deepest Secrets». In: *31st USENIX Security Symposium (USENIX Security 22)*. Accessed: 2025-09-13. USENIX Association, 2022. URL: https://www.usenix.org/conference/usenixsecurity22/presentation/markel (cit. on p. 20).

[27] Michael Bernhard, Stephan Fischer, Bernhard Krenn, Peter Mautner, Christian Moos, Eric Pöppelmann, and Simon Staudacher. «TPM-FAIL: TPM meets Timing and Lattice Attacks». In: *29th USENIX Security Symposium (USENIX Security 20)*. Accessed: 2025-09-13. 2020. URL: https://www.usenix.org/conference/usenixsecurity20/presentation/bernhard (cit. on p. 20).

[28] [Inserisci Autore/i qui]. «[Inserisci Titolo qui]». In: *arXiv preprint arXiv:2503.12248* (2025). Accessed: 2025-09-13. arXiv: 2503.12248 [cs.CR]. URL: https://arxiv.org/html/2503.12248v1 (cit. on p. 20).

[29] WithSecure Labs. *Sniff, There Leaks My BitLocker Key*. Accessed: 2025-09-08. URL: https://labs.withsecure.com/publications/sniff-there-leaks-my-bitlocker-key (cit. on p. 20).

[30] [Roua Boulifa]. «[Countermeasures Against Fault Injection Attacks in Processors: A Review]». In: *Information* (2024). Accessed: 2025-09-08. URL: https://www.mdpi.com/2078-2489/16/4/293 (cit. on p. 20).

[31] [Shams Shaikh]. «[Rethinking HSM and TPM Security in the Cloud: Real-World Attacks and Next-Gen Defenses]». In: *arXiv preprint arXiv:2507.17655* (2025). Accessed: 2025-09-08. URL: https://arxiv.org/html/2507.17655v1 (cit. on p. 21).

[32] Christina Brzuska, Marc Fischlin, Heike Schröder, and Stefan Katzenbeisser. *Physically Uncloneable Functions in the Universal Composition Framework*. Tech. rep. Accessed: 2025-09-12. techreport, 2014 (cit. on p. 22).

[33] Roel Maes. *Physically Unclonable Functions: Constructions, Properties and Applications*. Springer, 2020 (cit. on pp. 22–27, 29, 32).

[34] [Inserisci Autore/i dell'articolo qui]. *Conventional Arbiter PUF Design (Figure 2)*. 2025-11-13. URL: https://www.researchgate.net/figure/Conventional-Arbiter-PUF-Design_fig2_323950967 (cit. on p. 24).

[35] [Inserisci Autore/i dell'articolo qui]. *A conventional ring oscillator PUF (RO-PUF) (Figure 1)*. 2025-11-13. URL: https://www.researchgate.net/figure/A-conventional-ring-oscillator-PUF-RO-PUF_fig1_314595204 (cit. on p. 26).

[36] Muneer A. Al-Saraj, Xiaoxu Luan, Wei Liu, and Tian Jiang. «Optimal performance of simple low-cost optical physical unclonable functions resilient to machine learning attacks». In: *Optics Express* (2021) (cit. on p. 28).

[37] Giuseppe Emanuele Lio, Sara Nocentini, Lorenzo Pattelli, Eleonora Cara, Diederik Sybolt Wiersma, Ulrich Rührmair, and Francesco Riboli. «Quantifying the Sensitivity and Unclonability of Optical Physical Unclonable Functions». In: *Advanced Materials Technologies* (2024) (cit. on p. 28).

[38] Maurizio Di Paolo Emilio. *Quantum Tunneling e PUF*. Accessed: 2025-09-12. 2024. URL: https://tinnovamag.com/quantum-tunneling-e-puf/ (cit. on p. 28).

[39] [Inserisci Autore/i qui]. «[Inserisci Titolo dell'Articolo qui]». In: *Quantum* 5 (2021), p. 475. URL: https://quantum-journal.org/papers/q-2021-06-15-475/pdf/ (cit. on p. 28).

[40] Yevgeniy Dodis, Rafail Ostrovsky, Leonid Reyzin, and Adam Smith. «Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data». In: *Advances in Cryptology – EUROCRYPT 2004*. Lecture Notes in Computer Science. Springer, 2004. URL: https://link.springer.com/chapter/10.1007/978-3-540-24676-3_31 (cit. on p. 29).

[41] John O'Donnell, Domenico Forte, and Albin Chandy. «Reliable and Efficient PUF-Based Key Generation Using Pattern Matching». In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017. URL: https://people.csail.mit.edu/devadas/pubs/host2011.pdf (cit. on p. 29).

[42] David Merli, Giusi Satori, Alessandro Teseo, Paolo Cocco, and Massimo Poncino. «PUFKY: A Fully Functional PUF-based Cryptographic Key Generator». In: *2017 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2012. DOI: 10.23919/DATE.2017.7927056. URL: https://ieeexplore.ieee.org/document/7927056 (cit. on p. 29).

[43] Yansong Gao, Yang Su, Wei Yang, Shiping Chen, Surya Nepal, and Damith C. Ranasinghe. «Building Secure SRAM PUF Key Generators on Resource Constrained Devices». In: *2017 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*. IEEE, 2017. URL: https://ieeexplore.ieee.org/document/8730781 (cit. on p. 29).

[44] Jorge Guajardo, Sandeep S. Kumar, Geert-Jan Schrijen, and Pim Tuyls. «FPGA Intrinsic PUFs and Their Use for IP Protection». In: *Information and System Security Group*. Eindhoven, The Netherlands: Philips Research Laboratories, 2007. URL: http://link.springer.com/10.1007/978-3-540-74735-2_5 (cit. on p. 33).

[45] Jason Xin Zheng and Miodrag Potkonjak. «A Digital PUF-based IP Protection Architecture for Network Embedded Systems». In: *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. Los Angeles, CA, USA: ACM/IEEE, 2014. URL: https://dl.acm.org/doi/10.1145/2658260.2661776 (cit. on p. 33).

[46] Jason Xin Zheng. «PUF-based Hardware and Software Active IP Protection». Ph.D. Dissertation. Los Angeles, California: University of California, Los Angeles, 2014. URL: https://escholarship.org/uc/item/3n22352m (cit. on p. 33).

[47] Anirban Sengupta, Nabendu Bhui, and Vishal Chourasia. «Hardware IP protection by exploiting IP vendor's proteogenomic BioMarker as digital watermark during behavioral synthesis». In: *Scientific Reports* (2025). URL: https://www.nature.com/articles/s41598-025-96495-5 (cit. on p. 33).

[48] Qianqian Pan, Mianxiong Dong, Kaoru Ota, and Jun Wu. «Device-Bind Key-Storageless Hardware AI Model IP Protection: Joint PUF and Permute-Diffusion Encryption-Enabled Approach». In: *IEEE Conference*. IEEE. 2022. URL: http://arxiv.org/abs/2212.11133 (cit. on p. 33).

[49] Don Owen Jr., Derek Heeger, Calvin Chan, Wenjie Che, Fareena Saqib, Matt Areno, and Jim Plusquellic. «An Autonomous, Self-Authenticating, and Self-Contained Secure Boot Process for Field-Programmable Gate Arrays». In: *Cryptography* (2018). URL: https://www.mdpi.com/2410-387X/2/3/15?utm_source=chatgpt.com (cit. on p. 34).

[50] Florian Unterstein. «High Precision Electromagnetic Analysis of Leakage Resilient Cryptographic Constructions». Ph.D. Dissertation. Munich, Germany: Technical University of Munich, 2021. URL: https://mediatum.ub.tum.de/doc/1602042/1602042.pdf?utm_source=chatgpt.com (cit. on p. 34).

[51] Florian Unterstein, Nisha Jacob, Neil Hanley, Chongyan Gu, and Johann Heyszl. «SCA secure and updatable crypto engines for FPGA SoC bitstream decryption: extended version». In: *Journal of Cryptographic Engineering* (2021). URL: https://link.springer.com/article/10.1007/s13389-020-00247-2?utm_source=chatgpt.com (cit. on p. 34).

[52] Joseph Y. Halpern. «Chapter 4: The Art of Causal Modeling». In: *Actual Causality*. MIT Press, 2016 (cit. on p. 35).

[53] Roel Maes and Ingrid Verbauwhede. «From Statistics to Circuits: Foundations for Future Physical Unclonable Functions». In: *Towards Hardware-Intrinsic Security: Foundations and Practice*. Ed. by Ahmad-Reza Sadeghi and Dominik Naccache. Springer, 2010. URL: https://doi.org/10.1007/978-3-642-14452-3_3 (cit. on p. 35).

[54] Ulrich Rührmair, Frank Sehnke, Jan Sölter, Gideon Dror, Srinivas Devadas, and Jürgen Schmidhuber. «Modeling Attacks on Physical Unclonable Functions». In: *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010. URL: https://doi.org/10.1145/1866307.1866335 (cit. on p. 35).

[55] Mahmoud Khalafalla, Mahmoud A. Elmohr, and Catherine H. Gebotys. «Going Deep: Using Deep Learning Techniques with Simplified Mathematical Models Against XOR BR and TBR PUFs (Attacks and Countermeasures)». In: *arXiv preprint arXiv:2009.04063* (2020). URL: https://arxiv.org/abs/2009.04063 (cit. on p. 35).

[56] Meenatchi Jagasivamani, Peter Gadfort, Michel Sika, Michael Bajura, and Michael Fritze. «Split-Fabrication Obfuscation: Metrics and Techniques». In: *Proceedings of the IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*. IEEE, 2014. URL: https://ieeexplore.ieee.org/document/6855560 (cit. on p. 35).

[57] Venkata K. V. V. Bathalapalli et al. «PUFchain 4.0: Integrating PUF-Based TPM in Distributed Ledger for Security-by-Design of IoT». In: *Conference on Distributed Ledger Technologies*. Accessed: 2025-09-13. 2023. DOI: 10.1145/3583781.3590206. URL: https://www.researchgate.net/publication/371305995_PUFchain_40_Integrating_PUF-based_TPM_in_Distributed_Ledger_for_Security-by-Design_of_IoT (cit. on p. 36).

[58] Venkata K. V. V. Bathalapalli et al. «iTPM: Exploring PUF-Based Keyless TPM for Security-by-Design of Smart Electronics». In: *International Symposium on VLSI (ISVLSI)*. Accessed: 2025-09-13. 2023. DOI: 10.1109/ISVLSI59464.2023.10238586. URL: https://www.researchgate.net/publication/373721477_iTPM_Exploring_PUF-based_Keyless_TPM_for_Security-by-Design_of_Smart_Electronics (cit. on p. 36).

[59] PUFsecurity. *TPM 2.0-Ready: Top Security with PUFcc*. Accessed: 2025-09-13. 2022. URL: https://www.pufsecurity.com/document/tpm-2-0-ready-top-security-with-pufcc/ (cit. on p. 37).

[60] Giriraj Sharma, Amit M. Joshi, and Saraju P. Mohanty. «Fortified-Grid: Fortifying Smart Grids through the Integration of the Trusted Platform Module in Internet of Things Devices». In: *Information* (2023). Accessed: 2025-09-13. DOI: 10.3390/info14090491. URL: https://www.mdpi.com/2078-2489/14/9/491 (cit. on p. 37).

[61] Cédrick De Pauw, Jan Mühlberg, and Jean-Michel Dricot. «An Improved PUF-Based Privacy-Preserving IoT Protocol for Cloud Storage». In: *Proceedings of the 10th International Conference on Information Systems Security and Privacy*. Accessed: 2025-09-13. SCITEPRESS - Science and Technology Publications, 2024. DOI: 10.5220/0012326000003648. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0012326000003648 (cit. on pp. 37, 38).

[62] Krzysztof Gołofit. «Security Primitives for Memoryless IoT Devices Based on Physical Unclonable Functions and True Random Number Generators». In: *Scientific Reports* (2024). Accessed: 2025-09-13. URL: https://www.nature.com/articles/s41598-024-75373-6 (cit. on p. 37).

[63] Joshua Tito Amael, Oskar Natan, and Jazi Eko Istiyanto. «High-Security Hardware Module with PUF and Hybrid Cryptography for Data Security». In: *arXiv preprint* (2024). Accessed: 2025-09-13. URL: https://arxiv.org/html/2409.09928v1 (cit. on p. 38).

[64] gem5 Project. *gem5 Bootcamp 2024: Introduction and Simulation Background*. Accessed: 2025-09-09. 2024. URL: https://gem5bootcamp.github.io/2024/#01-Introduction/01-simulation-background (cit. on pp. 40–42, 44–49).

[65] gem5 Project. *gem5: gem5 Documentation*. Accessed: 2025-09-15. 2025. URL: https://www.gem5.org/documentation/ (cit. on pp. 41, 43–49).

[66] Kai Hu, Xia Zhu, and Min Zhao. «TPM-SIM: A Framework for Performance Evaluation of Trusted Platform Modules». In: *International Conference on Multimedia Technology*. Accessed: 2025-10-09. IEEE. 2010. URL: https://www.researchgate.net/publication/221058807_TPM-SIM_a_framework_for_performance_evaluation_of_trusted_platform_modules (cit. on p. 50).

[67] Stephen Tate and Dylan Doss. «Timing-Accurate TPM Simulation for What-If Explorations in Trusted Computing». In: *IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Accessed: 2025-10-09. IEEE. 2009. URL: https://www.researchgate.net/publication/224178772_Timing-accurate_TPM_simulation_for_what-if_explorations_in_Trusted_Computing (cit. on p. 50).

[68] Markus Pirker and Johannes Winter. «Automatic Generation of TPM 2.0 Simulators from TCG Specifications». In: *Information Security Theory and Practice: Security, Privacy and Trust in Computing Systems and Ambient Intelligent Ecosystems*. Accessed: 2025-10-09. Springer, 2013. DOI: 10.1007/978-3-642-38908-5_8. URL: https://link.springer.com/chapter/10.1007/978-3-642-38908-5_8 (cit. on p. 51).

[69] Stefan Berger et al. *libtpms: Software Implementation of a TPM 1.2/2.0*. Software project. Accessed: 2025-10-09. 2024. URL: https://github.com/stefanberger/libtpms (cit. on p. 52).

[70] Stefan Berger et al. *swtpm: TPM Emulator for QEMU/KVM*. Software project. Accessed: 2025-10-09. 2024. URL: https://github.com/stefanberger/swtpm (cit. on p. 52).

[71] QEMU Project. *TPM Device Emulation Specification*. Accessed: 2025-10-09. 2024. URL: https://qemu-project.gitlab.io/qemu/specs/tpm.html (cit. on p. 52).

[72] Hugues Cassé, Xavier Delaunay, Stéphane Jean, Cédric Jalier, Antoine Hammad, and Hamid Aboushady. «Parallel Simulation of Loosely Timed SystemC/TLM Programs: Challenges Raised by an Industrial Case Study». In: *Electronics* (2016). URL: https://www.mdpi.com/2079-9292/5/2/22?utm_source=chatgpt.com (cit. on p. 53).

[73] Cheng Zhao, Lei Wang, Qian Li, and Yifan Zhang. «Reproducing Spectre Attack with gem5: How To Do It Right». In: *ResearchGate* (2021). Accessed: 2025-10-09. URL: https://www.researchgate.net/publication/350810796_Reproducing_Spectre_Attack_with_gem5_How_To_Do_It_Right (cit. on p. 54).

[74] Jyotiprakash Mishra, Sanjay K. Sahay, and Aman Pathak. «A Hardware Security Review of RISC-V». In: *ICT Systems and Sustainability*. Accessed: 2025-11-04. 2026. URL: https://doi.org/10.1007/978-3-032-06665-7_38.

[75] Philip Roman, Vouthanack Sovann, Kenneth Triplin, David Um, Michael Violante, and Amy Wees. *Trusted Platform Module*. Accessed: 2025-09-08. 2012. URL: https://researchedsolution.wordpress.com/2013/09/14/trusted-platform-module/.

[76] Shohreh Hosseinzadeh, Bernardo Sequeiros, Pedro R. M. Inácio, and Ville Leppänen. «Recent Trends in Applying TPM to Cloud Computing». In: *Security and Privacy* (2020). Accessed: 2025-09-11. URL: https://onlinelibrary.wiley.com/doi/10.1002/spy2.93.

[77] Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 2: Structures*. Accessed: 2025-09-09. 2024. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-2-Structures-Version-184_pub.pdf.

[78] Trusted Computing Group. *Trusted Platform Module 2.0 Library Part 3: Commands*. Accessed: 2025-09-09. 2024. URL: https://trustedcomputinggroup.org/wp-content/uploads/Trusted-Platform-Module-2.0-Library-Part-3-Commands-Version-184_pub.pdf.