Trusted Execution Environment

P. Giannoccaro

October 11, 2022

Contents

1 INTRODUCTION

In modern society the number of electronics devices is increasing exponentially. Moreover, they become more and more interconnected, thus exploiting an increasing amount of information in order to improve performances. This is the concept of **Internet of Things (IoT)** [1]: billions of intelligent and heterogeneous communicating devices, each of them with a specialized function realized by an hardware accelerator, implemented in a **System-on-chip (SoC)** architecture. These heterogeneous systems are also used in critical system such as aircraft, banks, medical devices, automobiles, etc... where security is an essential requirement [2].
In this scenario, **Trusted Execution Environments (TEE)** can provide protection of both run-time states and stored assets of the system-on-chip, thus ensuring authenticity of the executed code and resistance to software and physical attacks.
Also, the increasing number of 'intelligent' devices will require a faster workflow in the system-on-chip design and hardware accelerators integration. **ESP** [3] is an open-source **RISC-V** research platform for heterogeneous SoC desgin which combines a modular tile-based architecture with a set of application-oriented flows for the design and optimization of accelerators. The ESP platform ensure highly scalable architecture and high abstraction system-level design, with a fully automated flow from software to hardware development to full-system prototyping on FPGA, thus reducing design time and costs.
The following paper provides: (i) an overview of security and threats against system-on-chip; (ii) a description of TEE main features and goals; (iii) a possible implementation of a TEE in the RISC-V ESP platform.

2 SoC SECURITY

An **attack towards a system** is defined as an attempt from a malicious user to extract protected information from the system itself as well as modifying its behavior or stopping it at all. A TEE ideally should be designed to be resistant to all kinds of attacks, both software and physical, by also dealing with the constant trade-off between security level and performances/power consumption. Therefore, ensuring security on a SoC is really challenging because of trade-offs and since attacks are becoming more and more sophisticated.
Hardware vulnerabilities [4] may be caused by unintentional design errors and/or hardware Trojans maliciously implanted during design or manufacturing (for example, incorrect design specifications, defective design implementation, incorrect translation of design in RTL synthesis). Major risks are not only critical system/user information leakage but also system irreparability. Moreover, current industry detection methods cannot achieve a good hardware vulnerabilities detection ratio, thus enhancing the importance of a good security architecture.

2.1 Software and physical attacks

There are many kinds of attacks towards system-on-chips as described in source [4]:

- *RTL Bugs*: they are common in a lot of design platforms and can lead to hardware vulnerabilities hardly detectable such as incorrect privileged escalation, address overlapping, improper write permissions to certain system registers, insecure encryption function, insecure key storage and hard-coded passwords.

- *Hardware Trojans (HT)*: it is a malicious modification of the integrated circuit design done by malicious engineers and chip manufacturers with the scope of bypassing or disabling a security fence of a system or its components. It is activated when a certain trigger occur, such as sensors, internal logic states, a particular input pattern or an internal counter value. Trojans attacks can occur both in logic circuits, with a relatively high overhead, and in embedded memory through resistive short, bridge and open circuit node insertion. Side-channel detection is an effective method to detect HT by measuring any difference in system power consumption, electromagnetic (EM) emanation and propagation delay.

- *Logic-locking attacks*: logic-locking technology aims to solve the threat of IP piracy by adding a gate driven by a secret key to hide the internal details of the IP. Only when the programmed key is applied, the conversion is reversed to achieve the original function of the IP. Attacks against this technology consist in selectively injecting input and scan the corresponding output, functionally pirating the function of the protected IP module. A secure scan chain is needed to prevent such attacks, made with flip-flop locking at the input in order to obfuscate functional output.

- *Electromagnetic Fault Injection (EMFI)*: Due to electromagnetic interference, system-on-chips could skip or mistakenly handle the execution of instructions, allowing the attacker to bypass security steps. Moreover, these kind of attacks become more and more successful under low power supply voltage designs and at higher clock frequencies.

- *Covert channels*: it is the leakage of sensitive data between processes theoretically not allowed to communicate due to security policy of the system.

- *Physical Access Attacks*: attackers with physical access to a chip may directly probe the pin signals to observe sensitive information. To prevent such attacks it is mandatory to encrypt and check the integrity of all data outside the processor, including DRAM and non-volatile memory. Also the address should not leave any trace in order to avoid the attacker to collect sensitive data such as encryption keys.

- *Buffer Overflow Attacks*: it is an anomaly where a program, while writing data to a buffer, overruns the buffer's boundary and overwrites adjacent memory locations. Exploiting the behavior of a buffer overflow is a well-known security exploit. On many systems, the memory layout of a program, or the system as a whole, is well defined. By sending in data designed to cause a buffer overflow, it is possible to write into areas known to hold executable code and replace it with malicious code, or to selectively overwrite data pertaining to the program's state, therefore causing behavior that was not intended by the original programmer. Buffers are widespread in operating system (OS) code, so it is possible to make attacks that perform privilege escalation and gain unlimited access to the computer's resources [5].

- *Side-channel Attacks*: it is an attack based on extra information that can be obtained because of a protocol/algorithm implementation through timing analysis, power consumption analysis, cache accesses monitoring, electromagnetic radiation analysis and other techniques. By simply analyzing how the system behave depending on the input it could be possible to extract sensitive data (such as encryption keys).

2.2 Attack Countermeasures

There are many ways to protect the system from the above mentioned attacks. **Hardware-assisted Security Units** can provide a level of security that software-based solutions cannot reach. In addition they reduce the complexity of the software infrastructure, thus improving performances. Hardware assisted methods include **Trusted Computing**, **Random Number Generation**, **Cryptography Engines** and **Memory Protection**.

- *Oblivious RAM (ORAM)*: it is adopted with the scope of transforming algorithms in such a way that resulting ones keep the same input-output behavior of the original but the distribution of memory access pattern is independent of the memory access pattern of the original algorithm. This is adopted since an attacker could obtain information about the execution of a program and the nature of the data that it is dealing with, just by observing the pattern in which various locations of memory are accessed during its execution, even if stored data is encrypted.

- *Memory Protection*: can be implemented through tagged memory, memory isolation and memory encryption and authentication in order to limit as much as possible accesses to data-sensitive areas of the memory.

- *Cryptographic Engines*: are used to limit the execution of malicious code through **Digital Signature Algorithm (DSA)** or to encrypt data. Performance overhead can be fairly reduced by implementing cryptographic algorithms with hardware accelerators and by extending the **Instruction Set Architecture (ISA)** with specifically designed instructions for encryption/decryption.

- *Side-channel attack prevention*: in order to prevent these kind of attacks, hardware-based technologies focus on redesigning the cache and modify processor architecture to improve cross-processor information flow tracking. Software solutions instead consist in clearing all core states such as private caches, translation backup buffers, branch prediction units relying on refresh mechanism. Moreover, side-channel information leakage is prevented by obliviously writing program data. In fact, program writing needs to avoid sensitive data access leaving traces, as well as ISA instructions. To prevent timing attacks, it is useful to extend the ISA with an instruction to integrate the refresh operation to clear core-level state. Also, *Dynamic Frequency Scaling (DFS)*, related to dynamic change of clock frequency and associated voltage of a CPU can be a cheap and effective countermeasure against simple Power Analysis attacks.

3 TRUSTED EXECUTION ENVIRONMENT IN DETAIL

A Trusted Execution Environment is an execution environment which protects both its run-time states and its stored assets, hence the need of isolation and secure storage. Unlike dedicated hardware co-processors, a TEE is able to easily manage its content by installing or updating code and data. Moreover, it must attests its trustworthiness to third-parties for a threat model which includes both software a physical attacks on the main memory and non-volatile memory. Therefore, a TEE is a tamper resistant processing environment that runs a **separation kernel** and that guarantees:

- *Authenticity of executed code*
- *Integrity of run-time states*
- *Confidentiality of its code, data and run-time states stored on a persistent memory*
- *Remote attestation of trustworthiness for third-parties*
- *Securely updatable content*
- *Resistance to all software/physical attacks on main memory and backdoor security flaws*

In other words, no untrusted code should be able to cause, enable or prevent any kind of event (such as instructions, traps, interrupts,...) in the trusted execution environment.

3.1 What is a TEE?

To give a definition of Trusted Execution Environment (TEE), it is firstly introduced the concept of Separation Kernel.

> *"The **Separation Kernel** is a foundation component of the TEE. It is the element that assures the property of isolated execution. The separation kernel [...] is a security kernel used to simulate a distributed system. Its main design purpose is to enable the coexistence of different systems requiring different levels of security on the same platform. Basically, it divides the system into several partitions, and guarantees a strong isolation between them, except for the carefully controlled interface for inter-partition communication."* [6]

The security requirements for the separation kernel are: (i) Data separation between partitions which means that a partition cannot read or modify the data of another partition; (ii) Temporal separation, shared resources should not leak information into other partitions; (iii) Control of information flow communication between partitions can occur only when allowed; (iv) Fault isolation which means that security breach in one partition should not spread into other ones. Now, it is possible the give an accurate definition of a TEE:

> *"A **Trusted Execution Environment (TEE)** is a tamper-resistant processing environment that runs on a separation kernel. It guarantees the authenticity of the executed code, the integrity of the runtime states (e.g. CPU registers, memory and sensitive I/O), and the confidentiality of its code, data and runtime states stored on a persistent memory. In addition, it shall be able to provide remote attestation that proves its trustworthiness for third-parties. The content of TEE is no static; it can be securely updated. The TEE resists against all software attacks as well as the physical attacks performed on the main memory of the system. Attacks performed by exploiting backdoor security flaws are not possible."* [6]

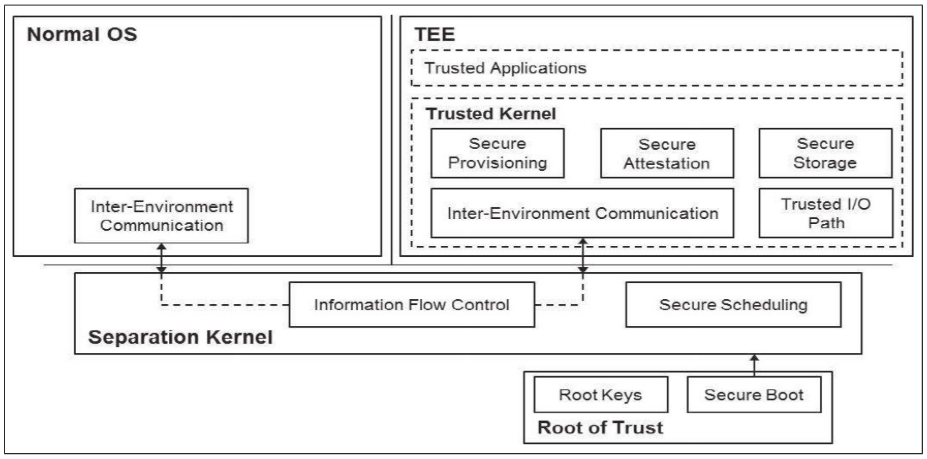


Figure 1: TEE Architecture [6]

The building blocks of a TEE are:

- **Secure Boot**: assures that only authenticated code can be executed. It has the scope of building the entire chain of trust of the system. If for any reason one stage of the secure boot procedure is not successfully completed the boot process is reset or aborted.
- **Secure Scheduling**: assures a “balanced” and “efficient” coordination between the TEE and the rest of the system. Indeed, it should assure that the tasks running in the TEE do not affect the responsiveness of the main OS. [6]

- **Inter-Environment Communication**: it is an interface that make possible the communications between TEE and the rest of the system. However it introduce new attack threats such as (i) message overload attacks; (ii) user and control data corruption attacks; (iii) memory faults caused by shared pages being removed; and (iv) unbound waits caused by the non-cooperation of the untrusted part of system.

- **Secure Storage**: it is the storage where confidentiality, integrity and authenticity of stored data are guaranteed. It can be accessed only by authorized entities. Possible implementations are the **Physical Memory Protection (PMP)** and **Tagged Memory**.

- **Trusted I/O Path**: ensure authentic communication between the TEE and external peripherals, thus protecting transferred data from malicious attacks such as screen-capture attack, key logging attack, overlaying attack, and phishing attack.

3.2 The Root of Trust (RoT)

The **Root Of Trust (RoT)** is the foundation on which all secure operations of computing system depend. It contains the keys for encryption functions and the fundamental modules to support the trusted boot and the TEE. Since it is the key element of the entire system security it must be secure by design considering that it will most likely be the target of malicious users attacks. The RoT can both have fixed functionalities or be programmable/updatable (for example with new encryption algorithms to counter evolving threats). Moreover, it should be physically isolated and have anti-tampering and side-channel resistance capabilities to prevent fault injections and side-channels attacks.
The building blocks of the Root of Trust [4] are:

- **Symmetric Cryptographic (e.g. AES) Engine**: used for message and image file encryption to guarantee data confidentiality and support secure boot.

- **Asymmetric Cryptographic (e.g. Public Key RSA) Engine**: is used for message encryption and enables the sender to combine a message with a private key to create a short digital signature on the message. Asymmetric cryptographic system use key pairs generated by a **Physical Unclonable Function (PUF)** (*see more in KMU chapter*). One key is public while the other one is private and known only to the owner. Key Public key can be used by anyone for encryption, but decryption is allowed only through private key. This block is fundamental for the Trusted Boot procedure and digital signature verification.

- **HASH Engine**: Used in many digital signature verification algorithm such as **Elliptic Curve Digital Signature Algorithm (ECDSA)** for *message hashing.* Basically, it implement an hashing algorithm that receives as input a variable length string and produces a fixed-size length hash value that can be used for message authentication, digital signatures, one-way passwords and intrusion detection.

- **Secure Memory**: features multiple interfaces and a hardened memory protection unit to store secure assets in a memory region isolated from the rest of the system. It may be both RAM and ROM, protected by *Physical Memory Protection (PMP)* to guarantee access only to entities with proper privilege areas.

- **OTP Management Module**: manages the **One Time Programmable Memory (OTP)** which is a non-volatile memory used to store keys and other secure assets such as firmware, bootloader, digital signature algorithms, encryption algorithms. The OTP can be programmed only once by an irreversible process done by a trusted factory before the chip leaves it.

- **True Random Number Generator (TRNG)**: generates random numbers or bits for multiple cryptographic algorithms and protocols such as key distribution and generation or authentication schemes. The generations of each bit can be based on an unpredictable physical process (*Non-deterministic Random Bit Generator (NRBG)* or on algorithms that generates a bit sequence starting from an initial value determined by a seed (*Deterministic Random Bit Generator (DRBG)*.

- **Trusted Boot Services**: The boot process is fundamental for correctly building the entire chain of trust of the system. It starts with authentication of the boot-loader and sequentially proceed to verify and initialize the kernel, peripherals and every trusted application in the system. If any stage does not pass the code authentication through digital signature verification done by *Code Authentication Unit (CAU)* the boot process is reset or aborted.

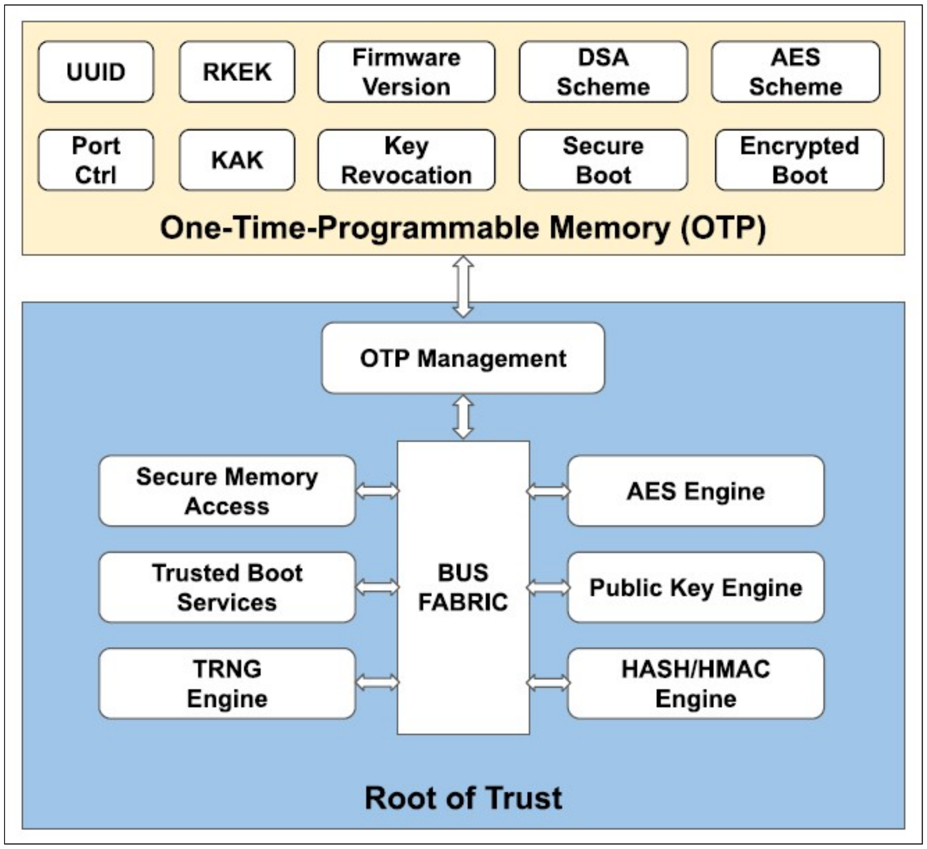


Figure 2: RoT Building Blocks [4]

3.3 Digital Signatures Algorithm (DSA)

A **Digital Signature Algorithm (DSA)** scheme is a set of algorithms used to verify digital signature of the code to be executed. It consists in three algorithms that use most of the building blocks of the RoT:

- *Key Generation Algorithm*: it provides a pair of a private and a public key, randomly selected through a *Physically Unclonable Function (PUF)*. Note that only the public key should be distributed, while the private key must be kept secret.
- *Signing Algorithm*: it receive as input a message and a private key and produces as output the message signature.
- *Signature Verifying Algorithm*: it receive as input a message and its signature and a public key. It has a binary outcome: either accept or reject message authenticity. This algorithm is implemented by the **Code Authentication Unit (CAU)**.

3.4 The Key Management Unit (KMU) and the Code Authentication Unit (CAU)

The *Key Management Unit (KMU)* and *Code Authentication Unit (CAU)* are two fundamental blocks used in the construction of the chain of trust in the system boot sequence. They uses a lot of blocks of the Root of Trust such as OTP management module, Hash engine and True Random Number Generator.

The **Key Management Unit (KMU)** generates and distributes symmetric and asymmetric keys to various security blocks such as *Code Authentication Unit (CAU)*, *Secure Debug*, *Memory Protection Unit (MPU)*. The KMU can distribute public keys stored in the OTP memory or generate them through a **Physically Unclonable Function (PUF)** module. The PUF [7] allows the KMU module to extract volatile secret keys from semiconductor random manufacturing parameters variations that only exist when the chip is powered on and depend on the uniqueness of the physical microstructure. It either generates a symmetric key to be shared with the client through a cryptographic protocol or it generates a random seed for asymmetric key generator through a *True Random Number Generator (TRNG)*.

The **Code Authentication Unit (CAU)** is a fundamental module of the RoT which has the scope of implementing a *Digital Signature Algorithm (DSA)* in order to provide code authentication relying on a set of public keys stored in the OTP memory to verify the digital signature of the stored code to be executed. It is used in the system boot sequence and in the normal OS behavior before executing trusted applications. Indeed, it also contains a *Direct Memory Access (DMA)* module to directly read code and related signatures from memories, both internal and external. Moreover, the CAU also uses the hash engine, since hash computing is a common step in many digital signature algorithms such as *Elliptic Curve Digital Signature Algorithm (ECDSA)*. The CAU execution flow is described in the following scheme:

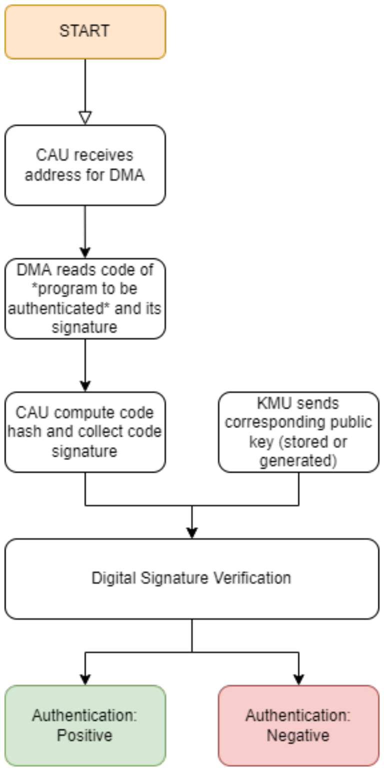

START
CAU receives address for DMA
DMA reads code of *program to be authenticated* and its signature
CAU compute code hash and collect code signature
KMU sends corresponding public key (stored or generated)
Digital Signature Verification
Authentication: Positive
Authentication: Negative

Figure 3: Code Authentication Unit (CAU) Execution Flow

3.5 Building the chain of trust: the Trusted Boot

The Code Authentication Unit (CAU) and the Key Management Unit (KMU), together with the *Boot Sequencer* [7], are the fundamental blocks in the *System Boot Sequence*, which is a crucial step for the correct initialization of a Trusted Execution Environment.

The **Boot Sequencer** is a Finite State Machine (FSM) which basically implements the entire flow of the boot sequence. It communicates with the CAU and the KMU. In each state, every code of the boot sequence steps is authenticated before being executed. If any state fails the authentication step the entire boot sequence is aborted or reset. This is a fundamental procedure since system boot is a crucial point in the device lifetime. In fact, the system boot initialize the most sensible part of the entire system, the *Hypervisor Execution Environment*

(HEE) with the highest permission (Machine Mode). Indeed, many attacks aim to break the software by exploiting system boot vulnerabilities in order to replace the OS kernel image. That is why it is crucial to verify and authenticate each code of the boot sequence.

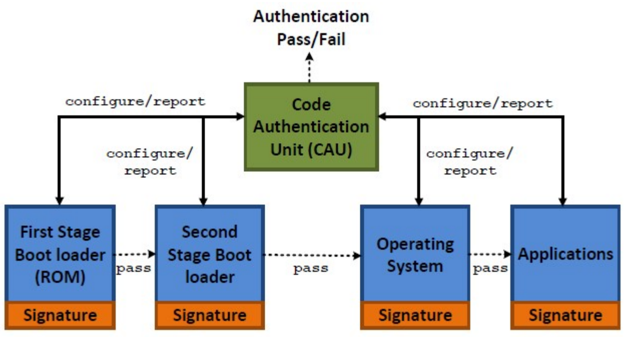


Figure 4: Steps of Boot Authentication [7]

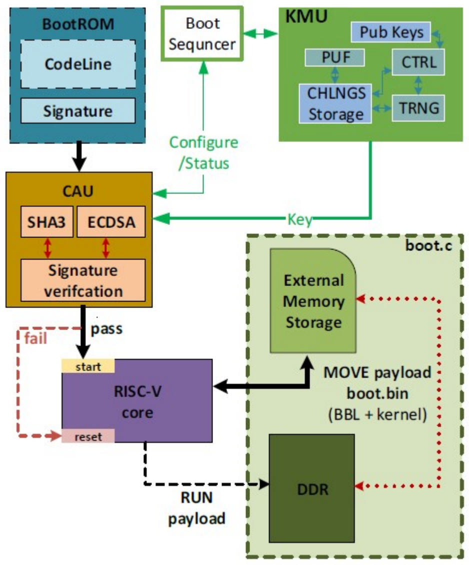


Figure 5: Boot Sequence building blocks [7]

The chain starts with an immutable first stage bootloader stored in ROM or OTP memory which cryptographically verifies the signature of the second stage bootloader. First, the boot sequencer authenticate the first stage bootloader taken from ROM/OTP memory. If authentication is successful, the first stage code, which consist in loading and authenticating the second stage, is executed.

The codes of second stage is loaded in DRAM and executed only if the authentication through CAU is again successful. The second stage code contains the so called *Berkeley Boot Loader (BBL)* which basically initialize peripherals, set up page table and virtual memory and load and authenticate the kernel. As happened before, the kernel code is loaded into DRAM and authenticated with CAU, which accesses it through DMA. If the authentication is again positive, then the kernel code is executed and the *Operating System (OS)* is up and running, as well as every setting contained in it, such as the configuration registers for *Physical Memory Protection (PMP)*, permissions areas for trusted and untrusted applications, the hypervisor infrastructure and so on. From now on, the OS is divided into a deep permission level system and areas (the separation kernel), with sensible interfaces, and supervisor system between levels which is better explained in the following paragraphs. In conclusion, any trusted application to be executed on the OS must be firstly authenticated through CAU.

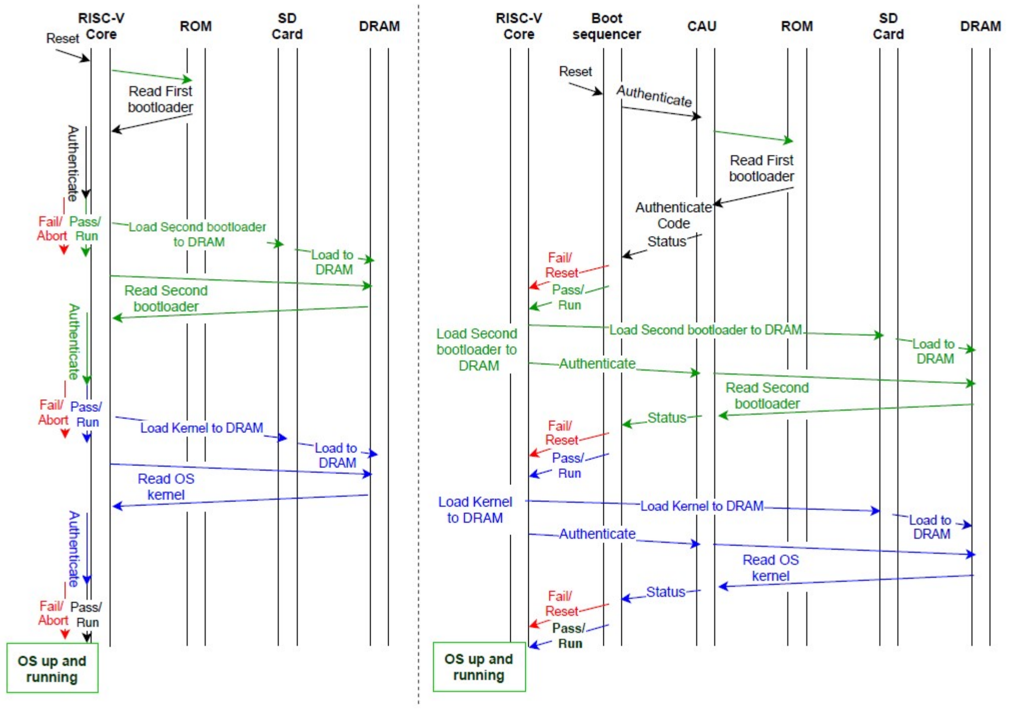


Figure 6: Boot Sequence stages [7]. On the left, software based boot; on the right hardware assisted boot sequence.

3.6 Separation Kernel Architecture

As mentioned before, the *Separation Kernel* is a crucial element of Trusted Execution Environments architecture.

> *"The **Separation Kernel** is a foundation component of the TEE. It is the element that assures the property of isolated execution. The separation kernel [...] is a security kernel used to simulate a distributed system. Its main design purpose is to enable the coexistence of different systems requiring different levels of security on the same platform. Basically, it divides the system into several partitions, and guarantees a strong isolation between them, except for the carefully controlled interface for inter-partition communication."* [6]

It must satisfy certain security requirements, such as:

- <u>Data separation</u> between partitions which means that a partition cannot read or modify the data of another partition.
- <u>Temporal separation</u>, shared resources should not leak information into other partitions.
- <u>Control of information flow</u> communication between partitions can occur only when allowed.
- <u>Fault isolation</u> which means that security breach in one partition should not spread into other ones.

A common RISC-V architecture for separation kernel uses three levels:

- <u>*Hypervisor Execution Environment (HEE)*</u>: it is the highest security level in a system (M-mode), to support multiple OS and manage the TEE settings. Code running in this mode is inherently trustworthy, because it has low-level access to the machine implementation and full permissions to access any memory region (unless specified by PMP system). Therefore, code running in this mode must be authenticated through the CAU. The HEE communicates with lower levels through the *Hypervisor Binary Interface (HBI)* and supports the trusted boot, the PMP registers and peripherals management. It also contains the **Security Monitor (SM)** which is a M-mode software that manages the lifecycle of enclaves (protected memory region) and therefore, the isolation boundary between enclaves and untrusted OS.
- <u>*Supervisor Execution Environment (SEE)*</u>: It is the environment related to a specific OS of the system and can be intended as a bios-like I/O system [4]. The SEE communicates through the *Supervisor Binary Interface (SBI)* and has the scope of handling events that stop normal execution (such as interrupts, supervisor call, trap handler) and then resume the application where it stopped. Moreover, it contains the trusted applications, that once authenticated, have higher permissions (S-mode) with respect to untrusted applications.

- *Application Execution Environment (AEE)*: it is the region in which applications (either trusted or not) run. It communicates with the supervisor through *Application Binary Interface (ABI)*.

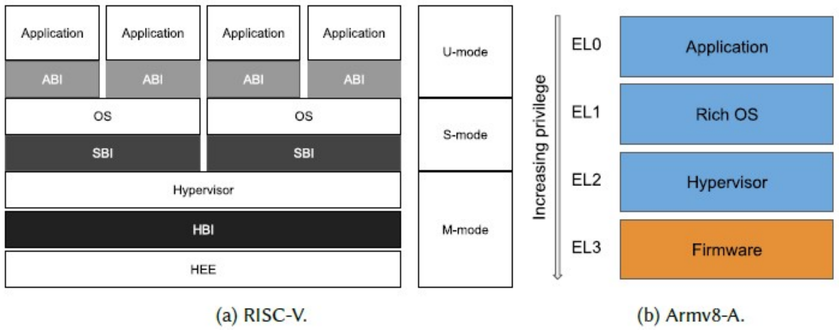


Figure 7: Privilege Level system [4]

A core normally runs application code in U-mode until some trap such as a supervisor call or a timer interrupt forces a switch to a trap handler, which usually runs in a more privileged mode. The core will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction in U-mode. Traps that increase privilege level are termed vertical traps, while traps that remain at the same privilege level are termed horizontal traps. The RISC-V privileged architecture provides flexible routing of traps to different privilege levels. Each privilege level has a core set of privileged ISA extensions with optional extensions and variants [4].

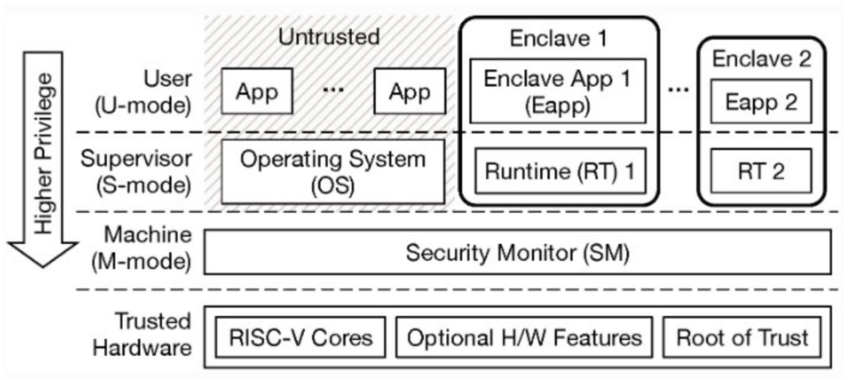


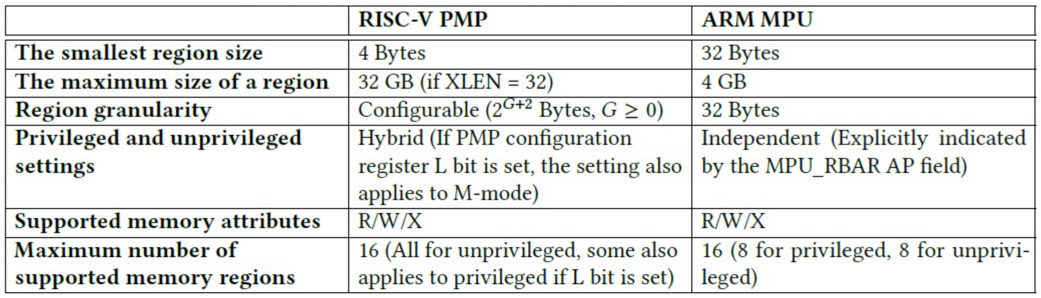
Figure 8: Keystone Privilege Level system [9]

3.7 Memory Protection Strategies: PMP and Tagged Memory

Memory protection is an essential feature of Trusted Execution Environments since it is needed for data protection and permission areas definition. The two main strategies for memory protection are: (i) *Physical Memory Protection (PMP)* and *Tagged Memory*.

The **Physical Memory Protection (PMP)** is a well known RISC-V memory protection technique which allows the firmware to limit accesses to memory regions (Read, Write or Execute) and lock it preventing accesses to Supervisor/User-running cores. This is possible thanks to *Control Status Registers (CSR)* (up to 16 per core) that configure the area limits of memory. These registers can be modified only by the Machine-Mode-running core (the highest permission level) and if needed, not even by the machine mode itself but only by reset and a new kernel firmware. This is done to prevent the machine-mode running core to accidentally change these registers, compromising the entire system security. A locked region is called **Enclave** and access are handled through a *PMP-Checker*, which is an hardware in-core module that checks every access to memory regions, in order to prevent S-mode and U-mode to read, write or execute privileged data. Violations are precisely trapped at core. Also, in an enclave is destroyed, also data contained in it must be cleared.

Table 2. Comparison of RISC-V PMP and ARM MPU Main Features.



	RISC-V PMP	ARM MPU
The smallest region size	4 Bytes	32 Bytes
The maximum size of a region	32 GB (if XLEN = 32)	4 GB
Region granularity	Configurable (2^{G+2} Bytes, $G \geq 0$)	32 Bytes
Privileged and unprivileged settings	Hybrid (If PMP configuration register L bit is set, the setting also applies to M-mode)	Independent (Explicitly indicated by the MPU_RBAR AP field)
Supported memory attributes	R/W/X	R/W/X
Maximum number of supported memory regions	16 (All for unprivileged, some also applies to privileged if L bit is set)	16 (8 for privileged, 8 for unprivileged)

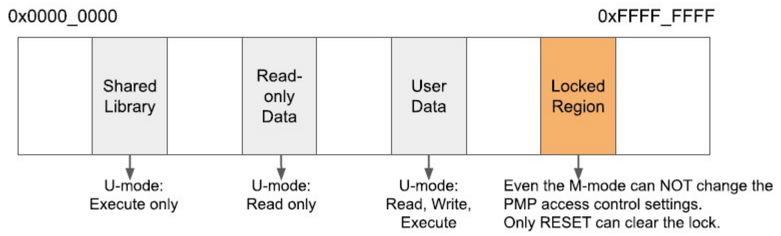

0x0000_0000
0xFFFF_FFFF
Shared Library
Read-only Data
User Data
Locked Region
U-mode: Execute only
U-mode: Read only
U-mode: Read, Write, Execute
Even the M-mode can NOT change the PMP access control settings. Only RESET can clear the lock.

Fig. 4. Demonstration of RISC-V physical memory protection.

Figure 9: RISC-V PMP and ARM Memory Protection Unit comparison [4]

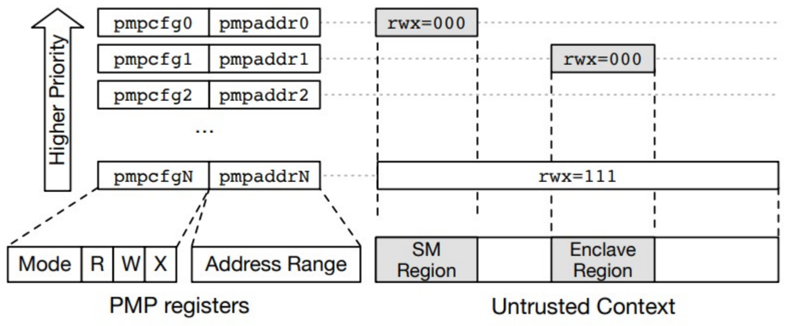
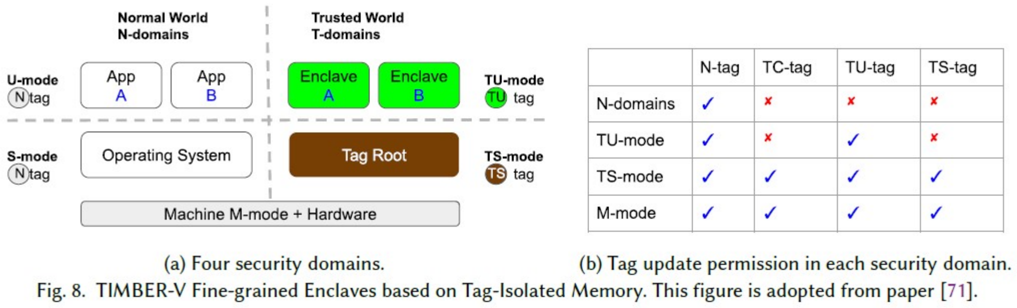


Figure 10: PMP Control Registers []

The **Tagged Memory** is a memory protection technique which allows memory pointers to have a capability tag [4]. At every memory access, the tag of the memory address will be checked to see if there is any capability violation. The implementation basically add a predefined number of bits every 64-bit word in memory. Also the caches need to be extended and coherence of tags is granted with the existing cache coherence mechanism. A possible RISC-V architecture for tagged memory is TIMBER-V [8]. With a two bits it is possible to use 4 tags: N-tag for normal world, TU-tag and TS-tag for trusted user and trusted supervisor for trusted world and TC-tag to access trusted callable entry point functions in order to switch between normal world and trusted world. Tags can only be updated within the same or lower security domain but cannot be used to elevate privileges. Combined with PMP mechanism, it can flexibly and efficiently isolate code and data to implement a Trusted Execution Environment (TEE).



	N-tag	TC-tag	TU-tag	TS-tag
N-domains	✓	✘	✘	✘
TU-mode	✓	✘	✓	✘
TS-mode	✓	✓	✓	✓
M-mode	✓	✓	✓	✓

(a) Four security domains. (b) Tag update permission in each security domain.

Fig. 8. TIMBER-V Fine-grained Enclaves based on Tag-Isolated Memory. This figure is adopted from paper [71].

Figure 11: Tagged Memory [4]

3.8 Encryption in TEE

Cryptographic algorithms are ubiquitous in security architectures and typically they are compute-intensive. There exists cryptographic standards and algorithms for a wide range of cryptographic functions such as block cipher techniques, digital signatures, hash functions, key management, memory encryption and so on. Therefore, it is fundamental to optimize the implementation of these algorithms through ISA extensions and hardware-based solutions,such as most of building blocks included in the root of trust.

The basic requirement for RISC-V Encryption integration is a correct implementation of *Advanced Encryption Standard (AES)*, which is a symmetric encryption algorithm used as standard solution from US government and require an ISA extension and an optimal implementation of widely used bit operations.

In conclusion, the development of quantum computing and the impact of Shor's algorithm has created active research to prevent these kind of attack, which is called *Post-Quantum Cryptography (PQC)*. PQC-based algorithms currently require effective arithmetic operations on hundreds to thousands of bits of data.

4 TEE IN RISC-V ESP PLATFORM

The increasing number of devices in the IoT scenario described at the beginning of the paper implies an increasing need of tools that enables fast design and fast prototyping of SoC, in order to reduce design time. Indeed, the designed SoC must also be secure and implement a Trusted Execution Environment. In the following paragraphs it is taken as tool the *RISC-V ESP Platform* and discussed how a TEE can be implemented in the tile-based architecture.

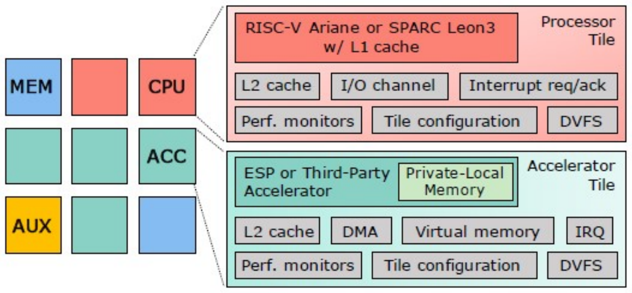


Figure 12: Tile-based SoC architecture [3]

4.1 RISC-V ESP Platform

> ”ESP is an open-source research platform for heterogeneous SoC design. The platform combines a modular tile-based architecture with a variety of application-oriented flows for the design and optimization of accelerators. The ESP architecture is highly scalable and strikes a balance between regularity and specialization. The companion methodology raises the level of abstraction to system-level design and enables an automated flow from software and hardware development to full-system prototyping on FPGA. For application developers, ESP offers domain-specific automated solutions to synthesize new accelerators for their software and to map complex workloads onto the SoC architecture. For hardware engineers, ESP offers automated solutions to integrate their accelerator designs into the complete SoC.” [3]

The ESP tile architecture is structured as a heterogeneous grid filled with a mix of tiles chosen depending on the target application. There are 4 types of tiles that can be used in the SoC design through the ESP platform: processor tile, memory tile, auxiliary I/O tile and hardware accelerator tile. They are connected by a *Multiplane Network-on-chip (NoC)*. Each tile is encapsulated into a *modular socket (aka shell)* which interface the tile to the NoC. This is a fundamental feature of the ESP design platform since it decouples the design of a tile from the design of the NoC, making easier the overall system design. In the tile-based architecture the keywords are scalability, modularity and heterogeneity. Processors and accelerators have the same importance, enhancing the system-centric view instead of a traditional processor-centric perspective.

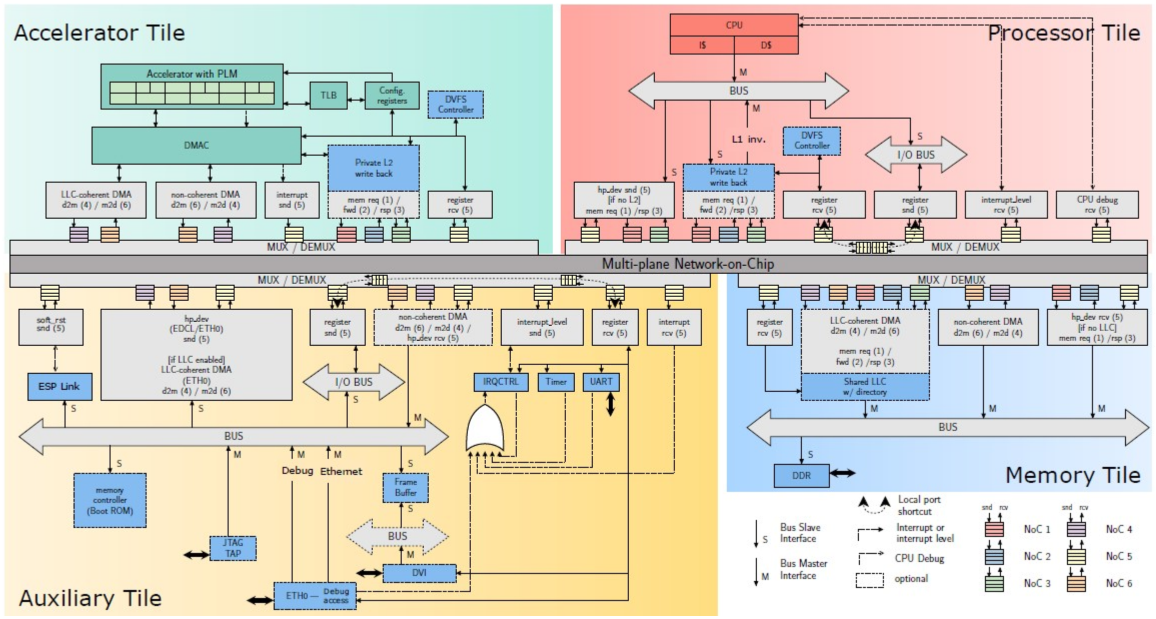


Figure 13: Tile-based SoC architecture in detail [3]

Let's analyze in detail the building blocks of the ESP tile-based architecture:

- **System Interconnect (NoC)**: processing elements act as transaction masters that access peripherals and slave devices distributed across remote tiles. All remote communication is supported by a NoC which is a *transparent communication layer*. The sockets include standard bus ports, bridges, interface adapters and proxy components that provide a complete decoupling from the network interface. Every platform service is implemented by a pair of proxy components. One proxy translates requests from bus masters, such as processors and accelerators, into transactions for one of the NoC planes. The other proxy forwards requests from the NoC planes to the target slave device, such as last-level cache (LLC) partitions, or Ethernet. For each proxy, there is a corresponding buffer queue, located between the tile port of the NoC routers and the proxy itself. The current implementation of the ESP NoC is a packet-switched 2D-mesh topology with look-ahead dimensional routing. Every hop takes a single clock cycle because arbitration and next-route computation are performed concurrently. Multiple physical planes allow protocol-deadlock prevention and provide sufficient bandwidth for the various message types. For example, a distributed directory-based protocol for cache coherence requires three separate channels, assigned to request, forward, and response messages. Concurrent DMA transactions, issued by multiple accelerators and handled by various remote memory tiles, require separate request and response planes. Instead of reusing the cache-coherence planes, the addition of two new planes increases the overall NoC bandwidth. Finally, one last plane is reserved for short messages, including interrupt, I/O configuration, monitoring and debug.

- **Processor Tile**: contain a processor core capable of running Linux and with own L1 cache. It communicates on a local bus and it is agnostic of the rest of the system. The processor tile communicates with a memory through an AXI interface. Then, socket provides proxies, bus adapters, unified L2 cache of configurable size. Any processor request to memory-mapped I/O registers are forwarded by the socket to the NoC plane. It is also present an interrupt-level proxy which implement a custom communication protocol between processor and interrupt controller and system timer in auxiliary tile.

- **Memory Tile**: each memory tile contains a a channel to external DRAM. The number of memory tiles is configured at the beginning (typically 1 to 4 depending on the target application). All necessary hardware logic to support the partitioning of the addressable memory space is automatically generated and the partitioning is completely transparent to software. Each memory tile also contains a configurably-sized partition of the LLC with the corresponding directory.

- **Hardware Accelerator Tile**: This tile contains the specialized hardware of a loosely-coupled accelerator. This type of accelerator executes a coarse-grained task independently from the processors while exchanging large datasets with the memory hierarchy. To be integrated in the ESP tile, accelerators should comply to a simple interface that includes load/store ports for latency-insensitive channels, signals to configure and start the accelerator, and an `acc_done` signal to notify the accelerator completion and generate an interrupt for the processors. ESP accelerators that are newly designed with one of the supported design flows automatically comply with this interface. For existing accelerators, ESP offers a third-party integration flow. In this case, the accelerator tile has only a subset of the proxy components because the configuration registers, DMA for memory access, and TLB for virtual memory are replaced by standard bus adapters. Furthermore, the socket enables point-to-point communication (P2P) among accelerator tiles so that they can exchange data directly instead of using necessarily shared-memory communication. Third-party accelerators can use the services to issue interrupt requests, receive configuration parameters and initiate DMA transactions. They are responsible, however, for managing shared resources, such as reserved memory regions, and for implementing their own custom hardware-software synchronization protocol.

The execution flow of an ESP accelerator consists of four phases, configure, load, compute, and store. A software application configures, checks, and starts the accelerator via memory-mapped registers. During the load and store phases, the accelerator interacts with the DMA controller, interleaving data exchanges between the system and the accelerator's private local memory (PLM) with computation. When the accelerator completes its task, an interrupt resumes the software for further processing. For better performance, the accelerator can have one or more parallel computation components that interact with the PLM. In conclusion, a latency-insensitive design is obtained by adding `valid` and `ready` signals to the channels. The valid signal indicates that the value of the data bundle is valid in the current clock cycle, while the ready signal is de-asserted to apply backpressure.

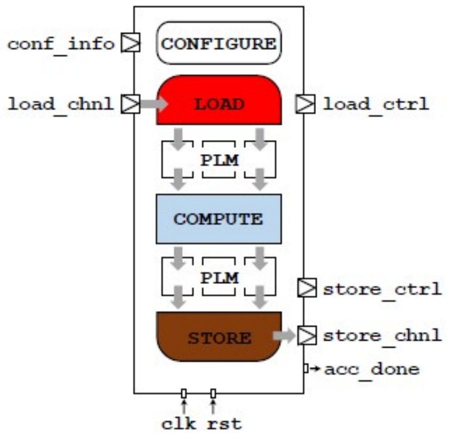


Figure 14: Hardware Accelerator Structure [3]

- **Auxiliary I/O Tile**: The auxiliary tile hosts all shared peripherals in the system except from memory: the Ethernet NIC, UART, a digital video interface, a debug link to control ESP prototypes on FPGA and a monitor module that collects various performance counters and periodically forwards them through the Ethernet interface. The socket of the auxiliary tile is the most complex because most platform services must be available to serve the devices hosted by this tile. The interrupt-level proxy, for instance, manages the communication between the processors and the interrupt controller. Ethernet, which requires coherent DMA to operate as a slave peripheral, enables users to remotely log into an ESP instance via SSH. The frame-buffer memory, dedicated to the video output, is connected to one proxy for memory-mapped I/O and one for non-coherent DMA transactions. These enable both processor cores and accelerators to write directly into the video frame buffer. The Ethernet debug interface, instead, uses the memory-mapped I/O and register access services to allow ESP users to monitor and debug the system through the ESP Link application. Symmetrically, UART, timer, interrupt controller and the bootrom are controlled by any master in the system through the counterpart proxies for memory-mapped I/O and register access.

4.2 TEE in the ESP Tile-based Architecture

In order to implement a *Trusted Execution Environment (TEE)* in the RISC-V ESP Platform all the requirements described above, such as trusted boot, separation kernel, memory protection and so on must be met.

To do so, some modifications to the traditional tile-based architecture are needed:

- It is introduced a new **Processor-Root of Trust Tile** which has the scope of implementing all the Root of Trust building blocks as well as the *Code Authentication Unit (CAU)*, *Key Management Unit (KMU)* and the *Boot Sequencer* for the System Boot Sequence implementation. This tile is a special processor tile which can run only in M-mode. Together with the Auxiliary I/O tile and a Memory Tile, define the *Hypervisor Execution Environment (HEE)*, that manages enclaves, boot sequence, peripherals, firmware updates and contain the security monitor for the management of overall system security (enclave lifecycle, isolation between execution environments, trap handling, permission levels).

- Every other processor tile can run in S-mode or U-mode or both. Together with a designer-selected number of memory tiles and hardware accelerators tiles it is possible to define the *Supervisor Execution Environment (SEE)* and the *Application Execution Environment (AEE)*. Moreover, every processor tile must contain a *PMPChecker* Module for enclave accesses and tagged memory infrastructure for permission-based division of memory regions.

- *Physical Memory Protection (PMP)* mechanism and *Tagged Memory* should be hardware-implemented for every memory tile, completely transparent to software. Both memory division mechanisms must be managed by the hypervisor processor tile and it should be possible to associate a certain permission tag to an entire memory tile.

- For the Hardware Accelerator Tile integration it is fundamental to manage the memory access permission associated since in the ESP architecture it is implemented a *Direct Memory Access (DMA)* and a *P2P Interconnect System* in each accelerator tile. The core processor that calls the hardware accelerator must also send, together with data and other configuration parameters, a configuration permission tag that is equal or lower in level with respect to the calling tile. In this way the accelerator is prevented from accessing sensible data. Also, memory privileges escalation is prevented.

- Any stored app can be trusted or untrusted. If it is untrusted, the system normally execute it through a processor in U-mode and with the lowest permission tag until an event such as interrupt occur. On the other hand, if it is trusted it can be executed by a processor either in U-mode or S-mode and with a memory permission tag TU. However, before being executed, the execution core must ask the hypervisor to authenticate the code through the CAU.

- Firmware updates can be easily implemented by simply updating the boot loader code and kernel image with the new ones, digitally signed with the private keys known to designers. The system boot sequence will automatically authenticate every code ensuring the correct and secure firmware update.

- It is fundamental to extend the *Instruction Set Architecture (ISA)* with cryptographic primitives in order to optimize performances since cryptographic engines are used in trusted computing, communication, storage, execution and many other security objectives.

5 NOTES

5.1 References

[1] ”The internet of things: a survey” - Shancang Li, Li Da Xu, Shanshan Zhao
[2] ”Securing Hardware Accelerators: A New Challenge for High-Level Synthesis” Christian Pilato , Member, IEEE, Siddharth Garg, Kaijie Wu, Ramesh Karri, Senior Member, IEEE, and Francesco Regazzoni, Member, IEEE
[3] ”Agile SoC Development with Open ESP” - Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni
[4] ”A Survey on RISC-V Security: Hardware and Architecture” - TAO LU, Marvell Semiconductor Ltd., USA
[5] https://en.wikipedia.org/wiki/Buffer_overflow
[6] ”Trusted Execution Environment: What It Is, and What It Is Not” Mohamed Sabt, Mohammed Achemlal and Abdelmadjid Bouabdallah
[7] ”Lightweight Secure-Boot Architecture for RISC-V System-on-Chip” Jawad Haj-Yahya, Ming Ming Wong, Vikramkumar Pudi, Shivam Bhasin, Anupam Chattopadhyay
[8] “TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V.” SamuelWeiser, MarioWerner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi.
[9] http://docs.keystone-enclave.org/en/latest/Getting-Started/How-Keystone-Works/Keystone-Basics.html

5.2 Abbreviations and Acronyms

ABI	Application Binary Interface
AEE	Application Execution Environment
AES	Advanced Encryption Standard
BBL	Berkeley Boot Loader
CAU	Code Authentication Unit
CPU	Central Processing Unit
CSR	Control Status Registers
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access
DRAM	Dynamic Random Access Memory
DRGB	Deterministic Random Bit Generator
DSA	Digital Signature Algorithm
ECDSA	Elliptic Curve Digital Signature Algorithm
EMFI	Electromagnetic Fault Injection
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
HBI	Hypervisor Binary Interface
HEE	Hypervisor Execution Environment
HT	Hardware Trojans
I/O	Input/Output
IoT	Internet of Things
IP	Intellectual Property
ISA	Instruction Set Architecture
KMU	Key Management Unit
LLC	Last Level Cache
NoC	Network-on-Chip
NRBG	Non-deterministic Random Bit Generator
ORAM	Oblivious Random Access Memory
OS	Operating System
OTP	One-Time Programmable
P2P	Point-to-Point
PLM	Private Local Memory
PMP	Physical Memory Protection
PQC	Post-Quantum Cryptography
PUF	Physically Unclonable Function
RISC	Reduced Instruction Set Computer
ROM	Read-Only Memory
RoT	Root of Trust
RSA	Rivest–Shamir–Adleman
SBI	Supervisor Binary Interface
SEE	Supervisor Execution Environment
SM	Security Monitor
SoC	System-on-chip
SSH	Secure SHell

TEE	Trusted Execution Environment
TLB	Translation Lookaside Buffer
TRNG	True Random Number Generator
UART	Universal Asynchronous Receiver-Transmitter