Advanced Computer Architecture

—

Part III: Hardware Security
Trusted Execution Environments

Paolo Ienne
<paolo.ienne@epfl.ch>

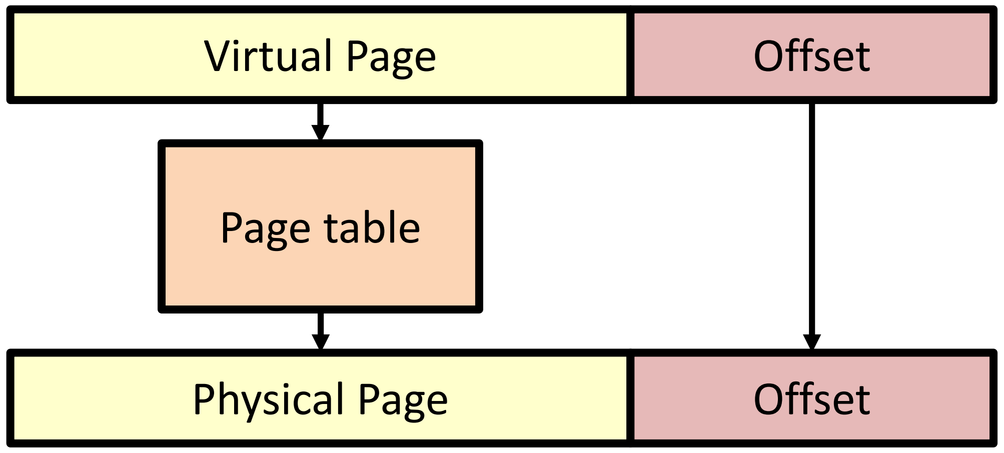
Outline

1

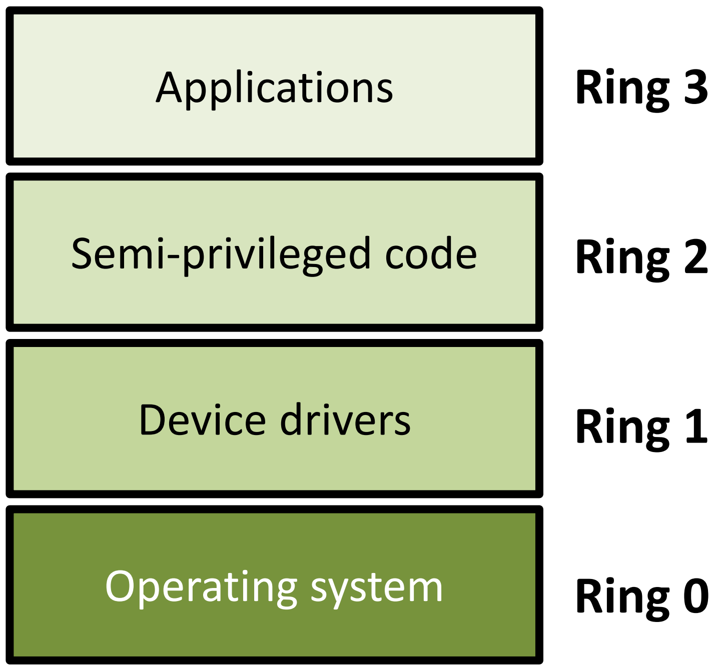
The Classic Approach to Confidentiality and Integrity

Isolation = Confidentiality and Integrity

Virtual Memory



Processor Privilege Levels



Virtual Memory

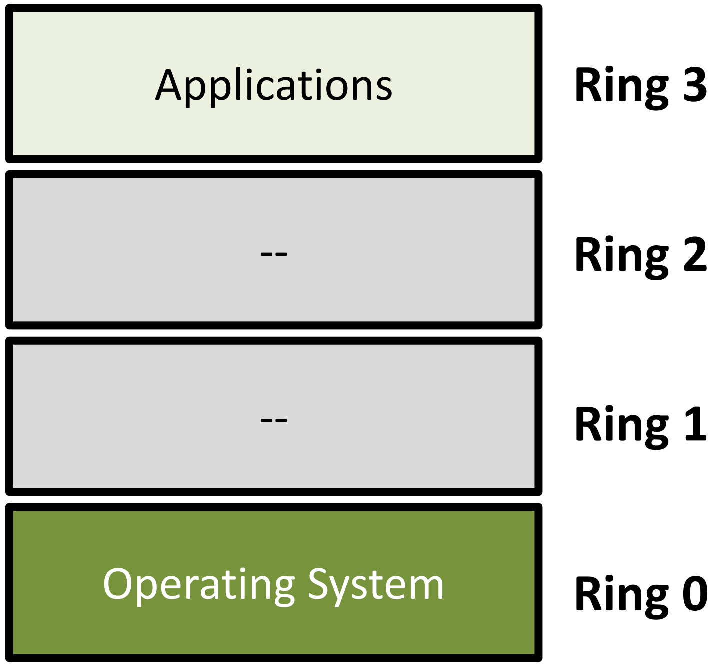
- The key to isolation across processes is the creation of a memory **indirection**
  - Processes "speak" in terms of **virtual memory addresses** (conventional addresses defined independently per process) and they need ultimately to be converted into **physical addresses** (the ones used on the electrical buses to the memory chips)
  - A **central trusted entity** (e.g., the OS) is charged of the allocation of these virtual memory addresses and of the translation
  - Isolation is achieved by the **trusted entity allowing only translations compliant** with the desired isolation property

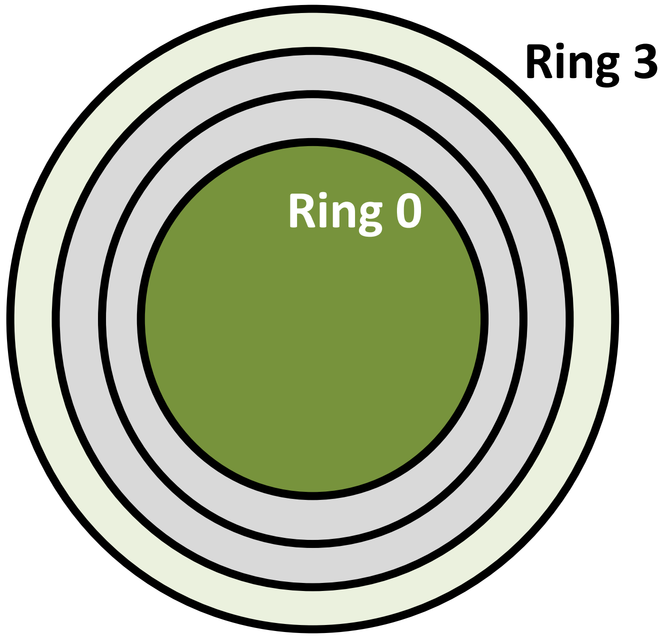
Processor Privilege Levels

- The key to the ability of limiting the possible translations depends on the existence of **processor privilege levels**
  - Some instructions can be **executed only in some privilege levels**
  - Instructions lowering the privilege level do not need to be restricted to a particular level: there is no harm in deciding that one can do less
  - Critical is the **mechanism to raise the privilege level**, of course
    - Link raising the privilege level to a predefined change in control flow (i.e., some form of jump): if the privilege level raises, only some specific code can be executed
    - Usually in the form of a **software exception instruction**: raise the privilege and then raise the exception to execute the exception handler
    - If the virtual memory mechanism has been used well to protect the exception handler code, there is confidence that when the privilege level is high, only the OS can be executing

Classic Privilege Levels

- Traditionally, **multiple privilege levels** (or rings) with varying capabilities tuned to some particular purposes
- Lower levels (or inner rings) **add to the capabilities** of levels above (or rings outside)
- In practice, most processors evolved to have only two privilege levels: **user mode** and **kernel mode** (names vary)





Virtual Machines (VMs)

- At the turn of the millennium there started to be (renewed) interest in hosting virtual machines (complete OS and applications) inside another OS and, in particular, inside a dedicated monitor (**hypervisor**)
- In particular, **full virtualization**: run the very same OS and applications in the virtual machine that one would run on the bare hardware
- Many reasons:
  - **Consolidation** of multiple small machines in a powerful one (lower cost and energy)
  - **Flexible deployment** (no need to buy a machine upfront)
  - **Lower dependence** from the **hardware** details (easy to move across servers)
  - **Better isolation** (not processes of the same OS but different OSes)

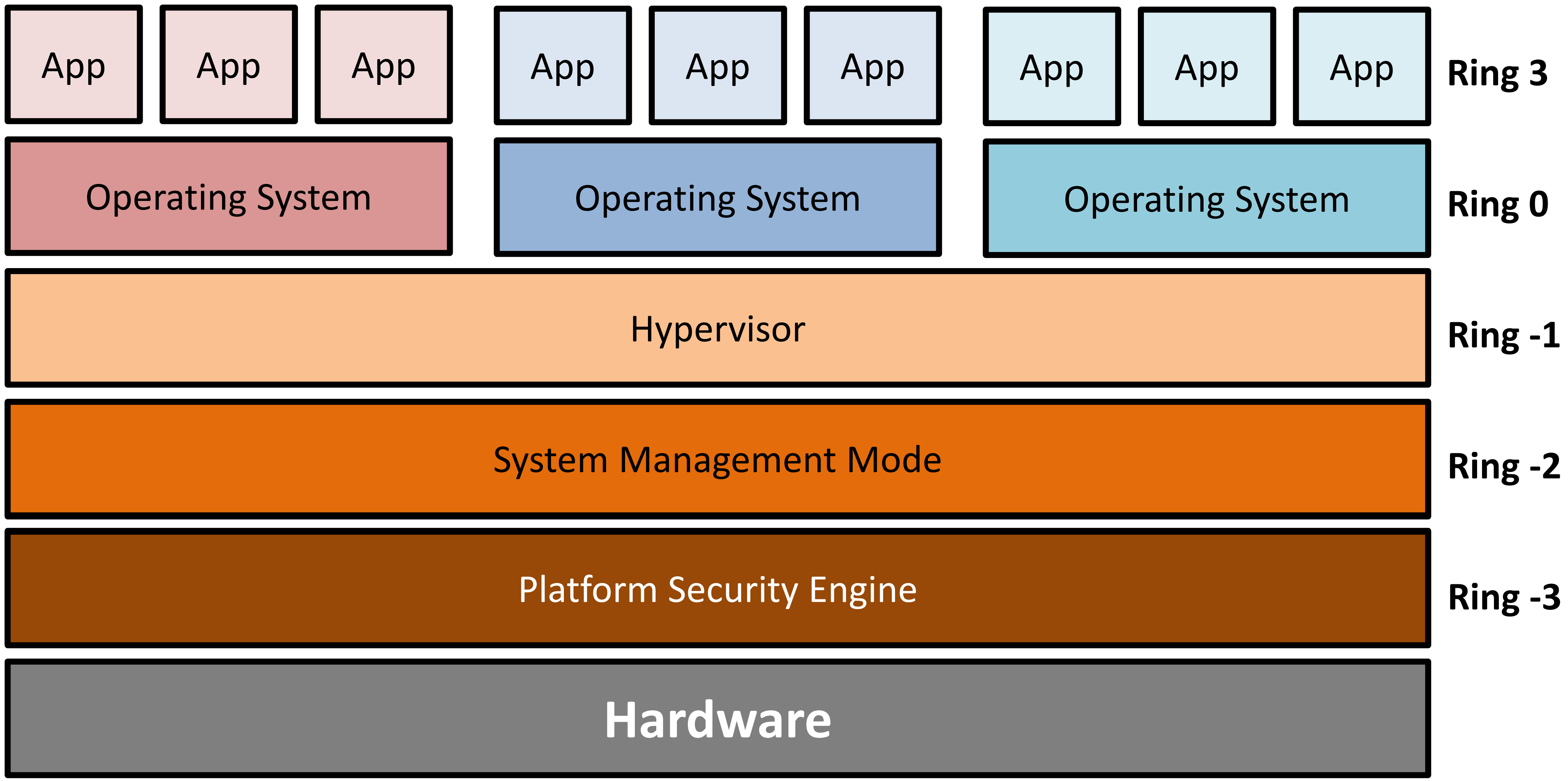
Software-Based Virtualization

- Mostly, the ingredients for process virtualization enable also full virtualization:
  - Memory is accessed via **TLBs**, violations results in **exceptions** being raised, etc.
- Achieving full virtualization on a CPU not meant for it is challenging:
  - If guest OSes need to be isolated, they cannot run all in kernel mode
  - But if **guest OSes run in user mode**, how can they do their job?!
- The classic approach is called **trap-and-emulate**:
  - Guest OS will create exceptions when trying to do its normal job (loading a TLB)
  - Hypervisor will check the pertinence and, if appropriate, emulate
  - Many key data structures will be replicated (shadow page tables)
- But some instructions simply behave differently in user and kernel mode!
  - **Dynamic Binary Translation** (remember?!...) to rewrite the functionality with user mode instructions
- VMware achieved full software virtualization in 1999 (its author is not too far away…)

Hardware-Assisted Virtualization

- Around 2005-06, both AMD and Intel introduced **ISA extensions and hardware support** for full virtualization and progressively extended it
  - **More privilege levels** (Ring -1, Hypervisor)
  - **Another level of address translation** (nested paging) supported by the hardware page walker
  - Interrupt virtualization
  - IOMMU virtualization
  - …

More High-Privilege Levels



More High-Privilege Levels

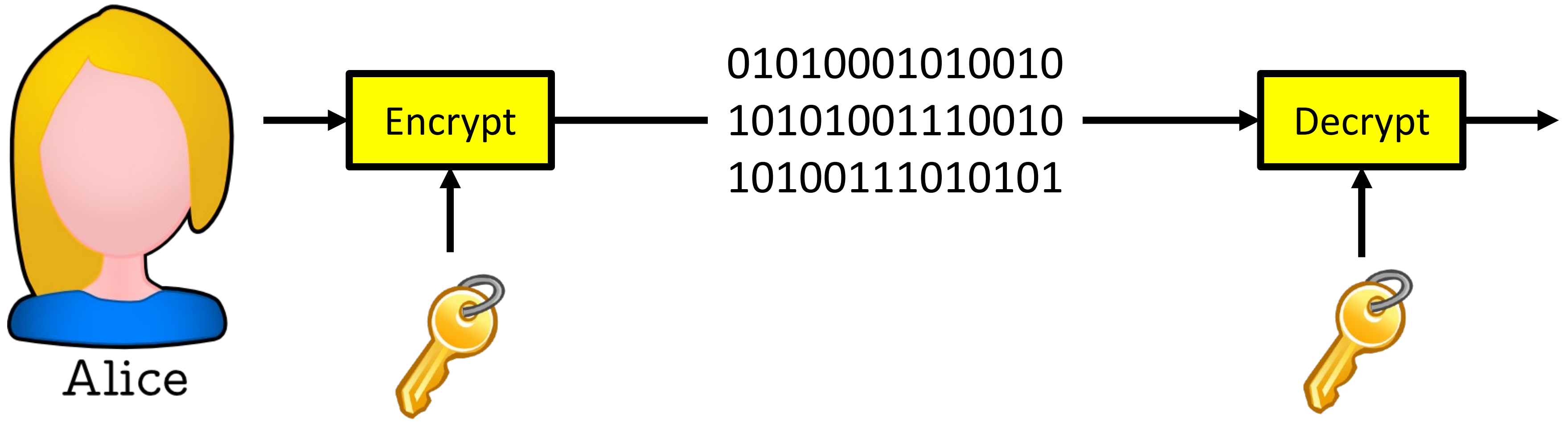
- **System Management Mode** (Ring -2)
  - First introduced by Intel and now in all x86 processors
  - Guarantee some management functionality in firmware even if the OS or the hypervisor are compromised; accessible by dedicated interrupts
  - Mostly used for power and thermal management or handling hardware errors
- **Platform Security Engine** (Ring -3)
  - Intel's Management Engine (ME) or AMD's Platform Secure Processor
  - **Physical isolation** through a piece of hardware independent from the processor
    - Power up and down the processor, network connected, reserved main memory, etc.
- Not just more levels but **dedicated hardware** and **physical isolation**
  - FSM or small processor independent of the main cores
    - Intel: ARC (from ARC International, now Synopsys), Quark
    - AMD: ARM
- Largely implement **security by obscurity**

2

The Universal Ingredients of (Hardware) Security Recipes

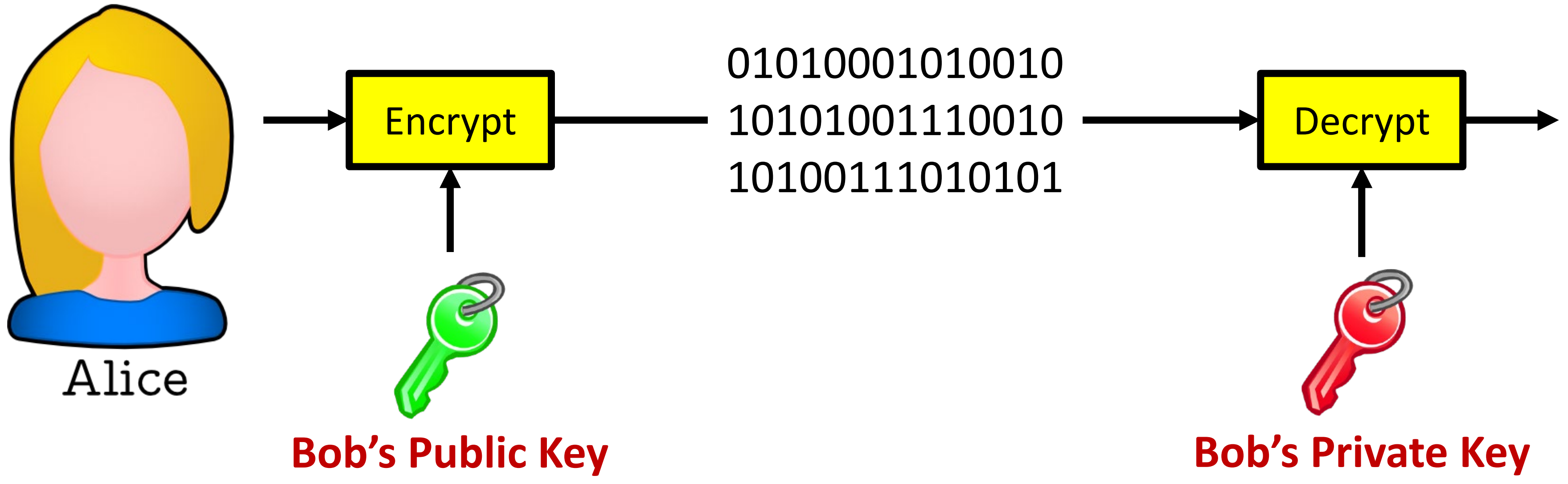
Symmetric-Key Cryptography

- Also called private-key cryptography



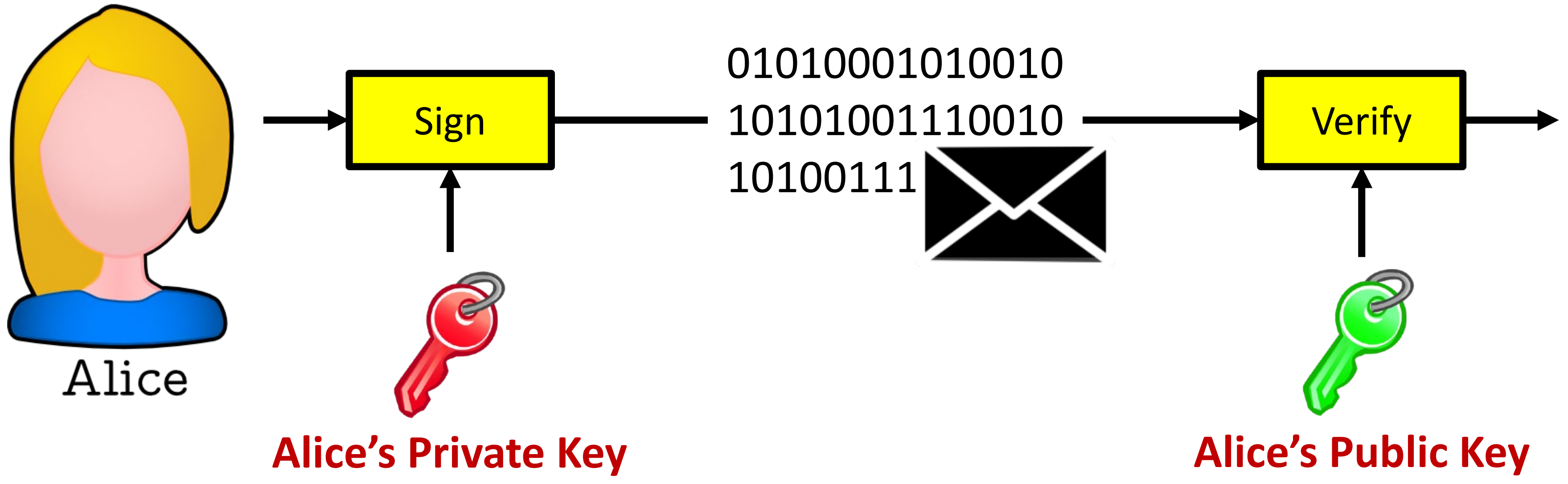
- A single **secret key** (*symmetric* key), shared by Alice and Bob
- Typically used for **confidentiality**: without the key, one cannot read the message
- Typical examples: RC4, DES, 3DES, AES,…

Public-Key Cryptography: Encryption



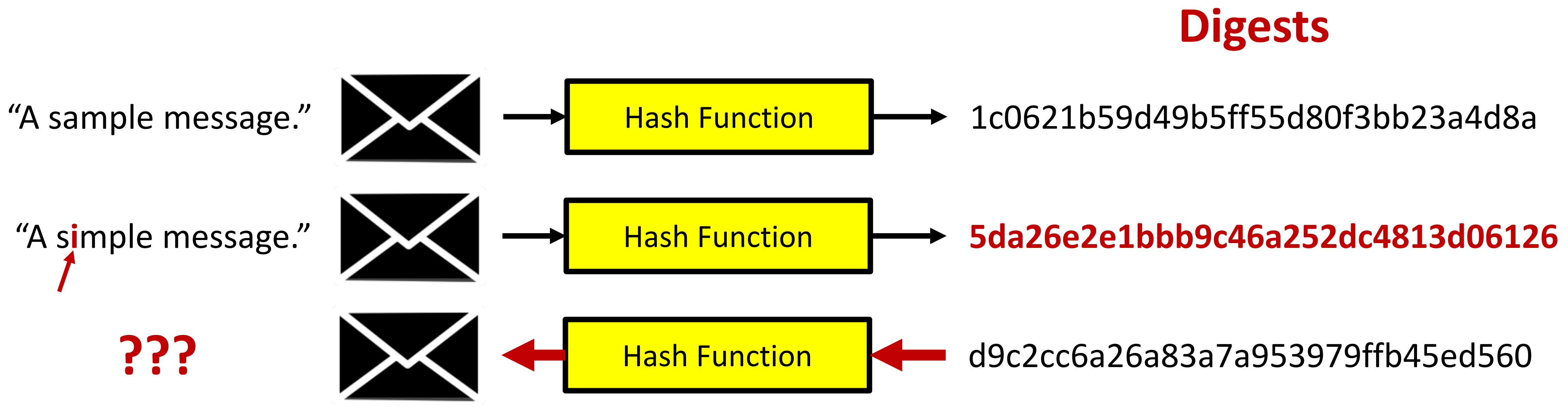
- **Two keys** per user, typically generated from a large random number, **one public and one private (secret)**
- Can be used for **confidentiality** as shown above: everyone can encrypt a message but only Bob has the key to decode it
- **Much slower** than symmetric encryption

Public-Key Cryptography: Digital Signature



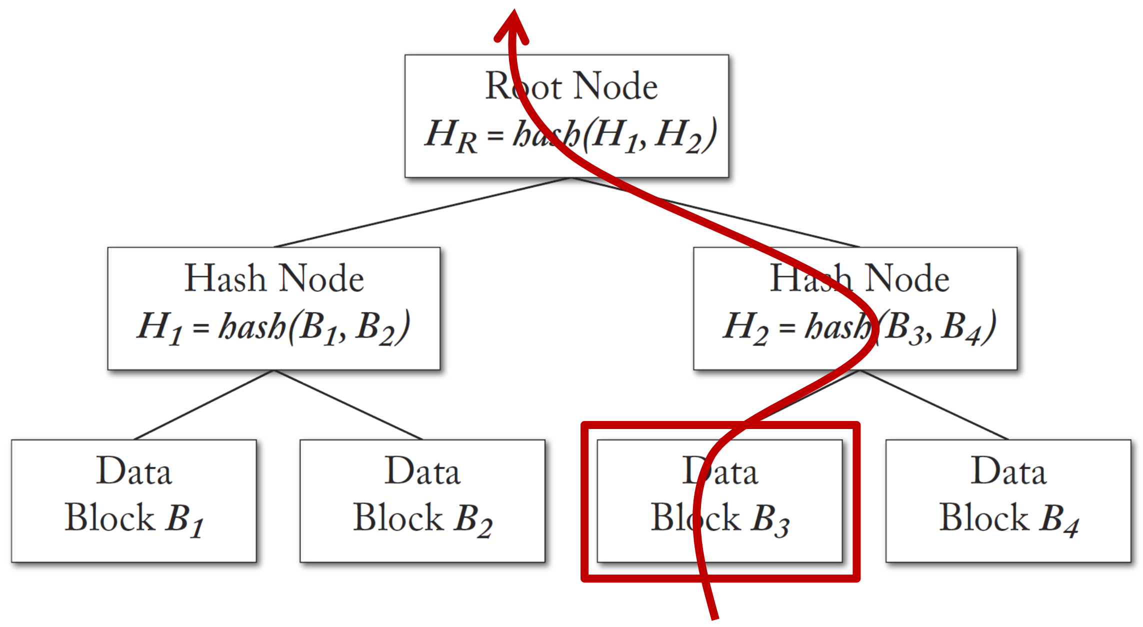
- Exchanging the order of the keys makes it possible to verify **authenticity**: everybody can tell that only Alice could have sent the message (but only **provided one can trust Alice's public key to be genuine!**)
- Typical examples of public-key algorithms: RSA, ECC,…

One-Way Hash Functions


Digests
“A sample message.”
Hash Function
1c0621b59d49b5ff55d80f3bb23a4d8a
“A simple message.”
Hash Function
5da26e2e1bbb9c46a252dc4813d06126
???
Hash Function
d9c2cc6a26a83a7a953979ffb45ed560

- Typically used for **integrity**: it should be impossible to create a new message or modify one such that it results in the same hash (also called *digest* or *fingerprint*) as the original
- Typical examples: MD5, SHA-2, SHA-3,…

Hash Trees (or Merkle Trees)



Source: Szefer, 2019

- Recursive application of one-way hashes on a dataset split in blocks (file, memory,...)
- Useful to keep hashes up to date in case of local changes: one needs only to **recompute the hash of the block where the change took place and of the parents**

Random Number Generators

- Main distinction:
  - **Pseudo-Random Number Generators** are algorithms to produce from a few initial bits (a *seed*) a deterministic long string of random-looking numbers
  - **True Random Number Generators** are typically hardware components which exploit physical phenomena (electrical or thermal noise, temperature variations, etc.) to generate truly random numbers
- **TRNGs are slow**, thus often TRNGs generate seeds and PRNG generate strings of random numbers for practical use
- TRNG can be **sensitive to tampering** or may provide *backdoors*

Physical Unclonable Functions (PUFs)

- Circuits exploiting intrinsic random physical features to produce a fingerprint **uniquely identifying each chip**
- Infeasible for the manufacturer to produce a chip with a specific identifier—as opposed to have the manufacturer *write* into each device a specific identifier, such as a serial number (which is also costly)
- Used today in relatively specific contexts and several vulnerabilities have been discovered for existing PUFs

Freshness and Nonces

- In **replay attacks**, an adversary intercepts a piece of data and **resends it at a later time**
- The authenticity and integrity of the message is guaranteed by the fact that the message was a genuine one once first sent—what it misses is **freshness**
- The typical solution is to introduce **nonces**, that is numbers used only once during the lifetime of the system: if a message contains a previously used nonce, it is not fresh
- Nonces can be produced by **monotonic counters**, for instance

Homomorphic Encryption

- Form of encryption which allows **computing over encrypted data** without access to the secret key
- Ultimate solution to secure remote computation: user ships encrypted data, they get **processed by an untrusted party who does never see data in clear**, and user receives back encrypted results
- Extremely intellectually appealing idea, but, in practice, today there is **no general solution** except for limited families of computation and with **impractical performance overheads**

3

Trusted Computing Base and Trusted Execution Environments

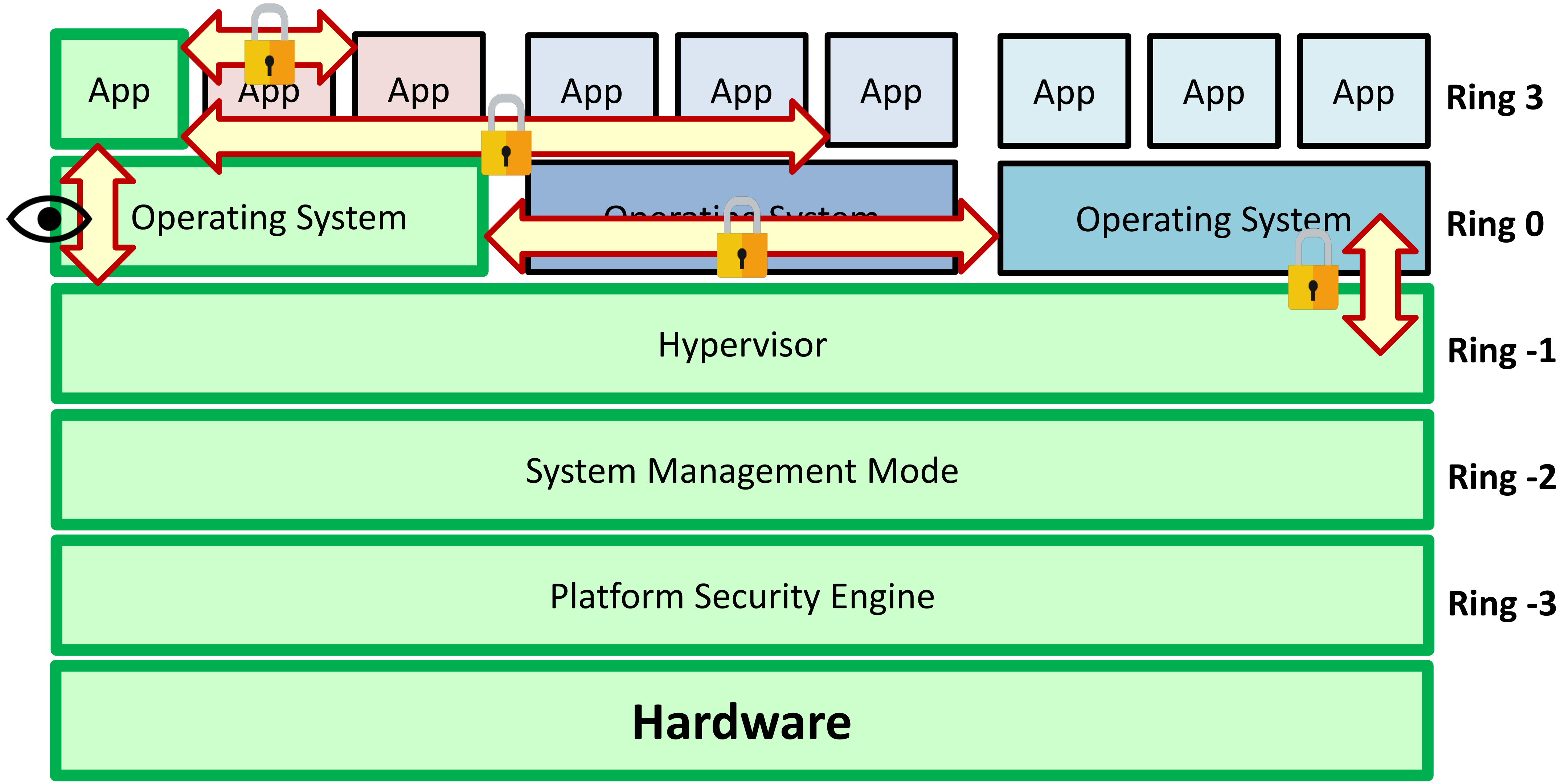
Trusted Computing Base (TCB)

- The set of **trusted hardware and software components** which can be object of an attack
- Important: what is **trusted** is not necessarily **trustworthy**!
- The purpose is to separate clearly
  1. what is **supposed to be trustworthy** and simply may not be such because of **bugs or conceptual oversights**

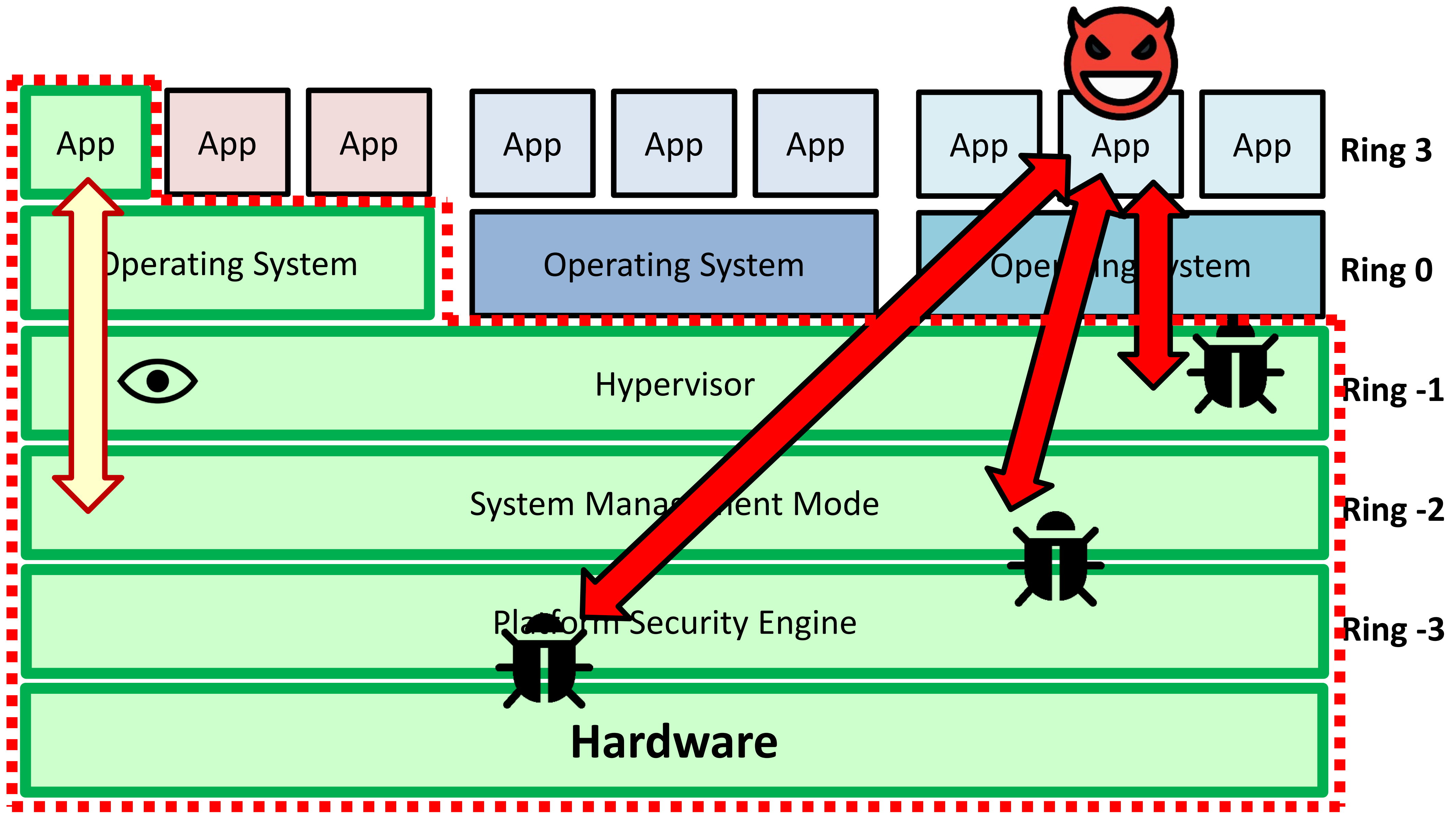
  from

  2. what is **clearly untrustworthy** and against which the system has been designed with explicitly defenses

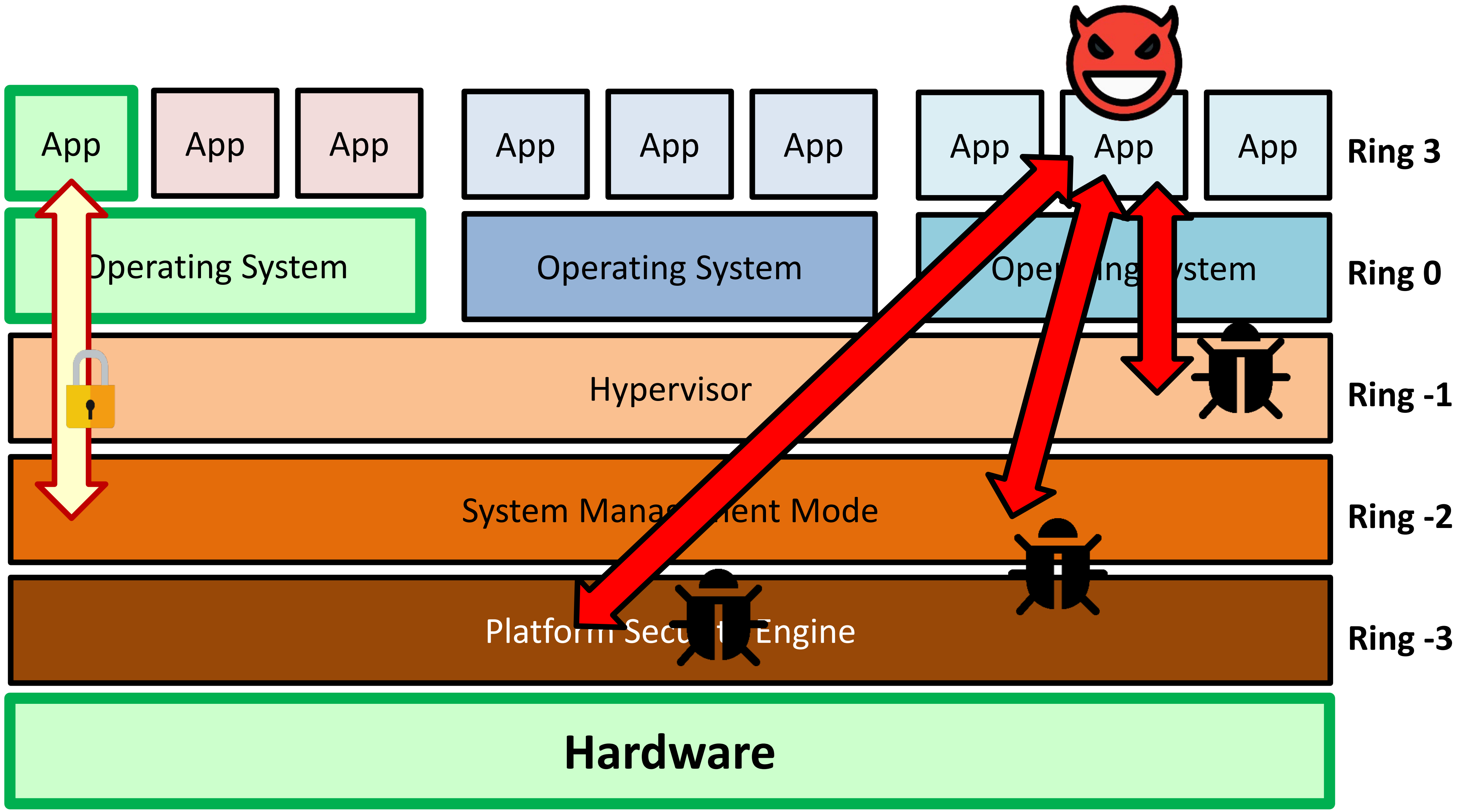
The Classic TCB



Surface of Attack

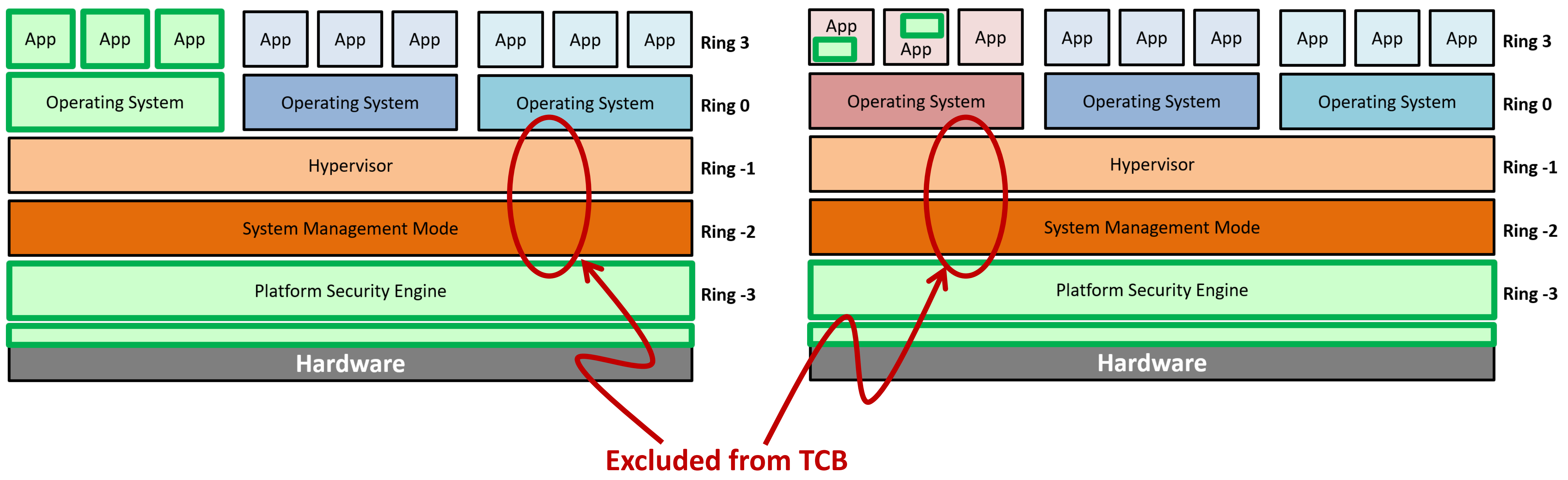


Make TCB Small!



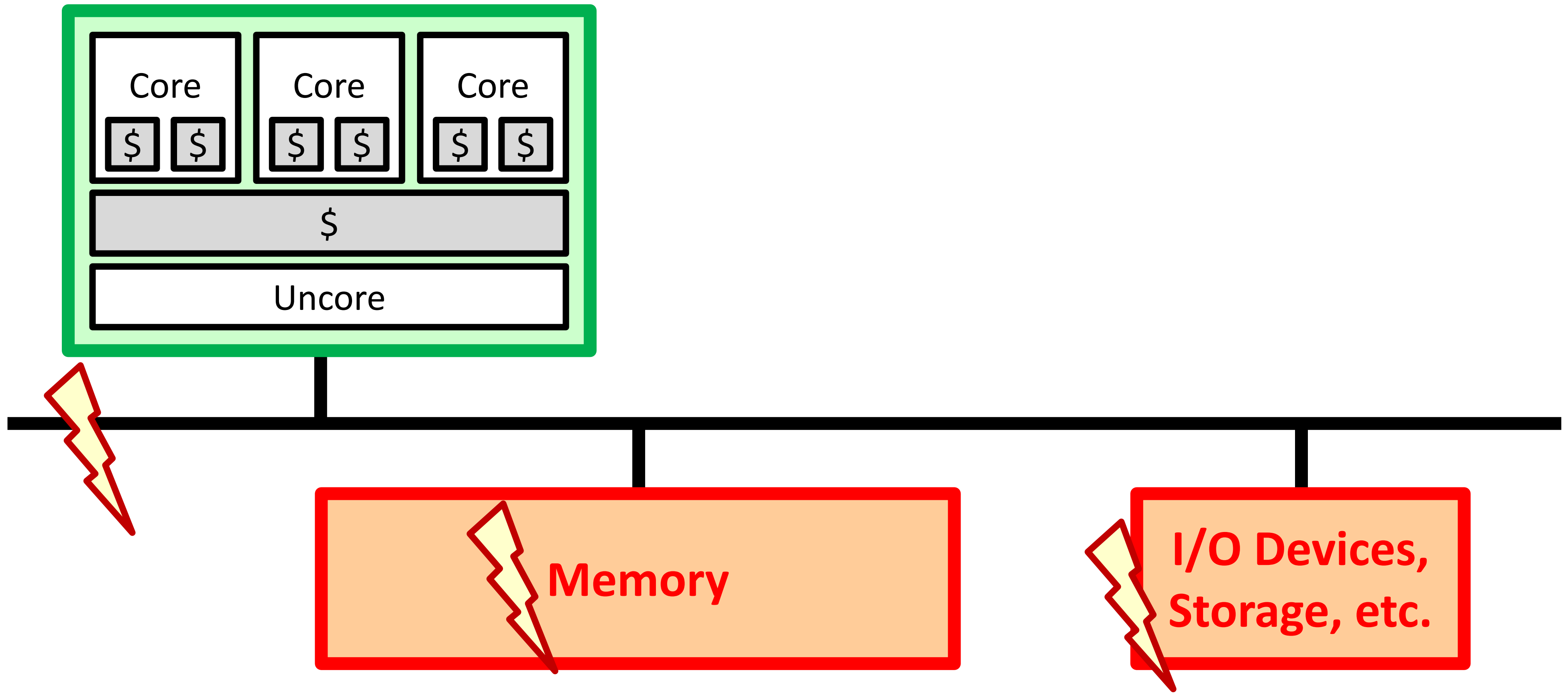
Evolving TCB Needs

- TCB evolving also due to **new business models**
- Cloud users trust their own apps, their own guest OSes, and the processor manufacturer, but **not the cloud operator**



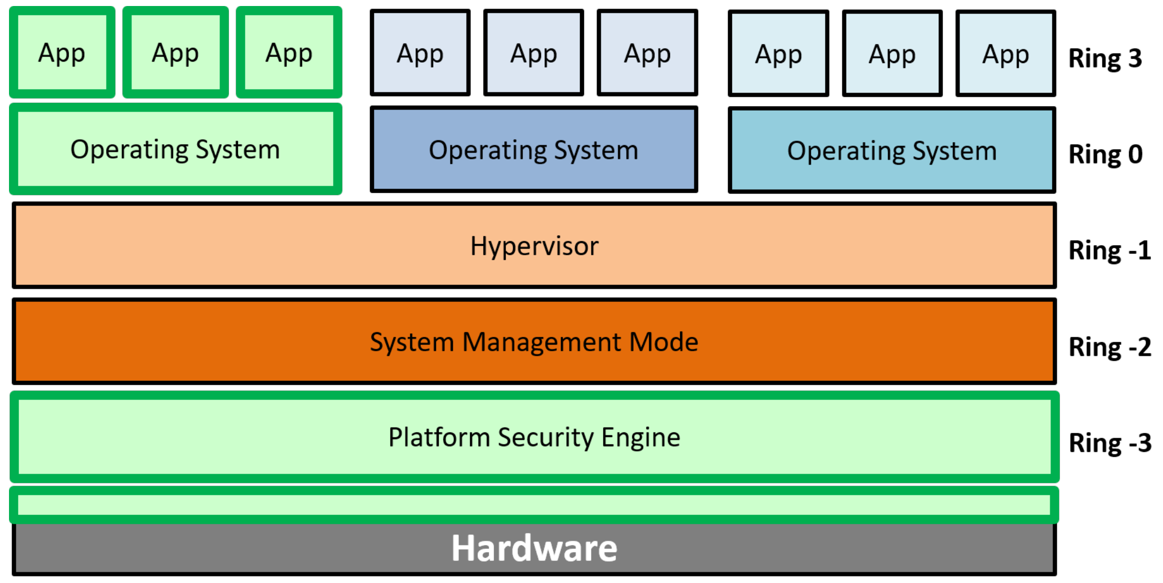
Trusted Processor Chip

- The fact that the cloud operator is not considered trusted means also that **not the whole computer hardware is trusted**

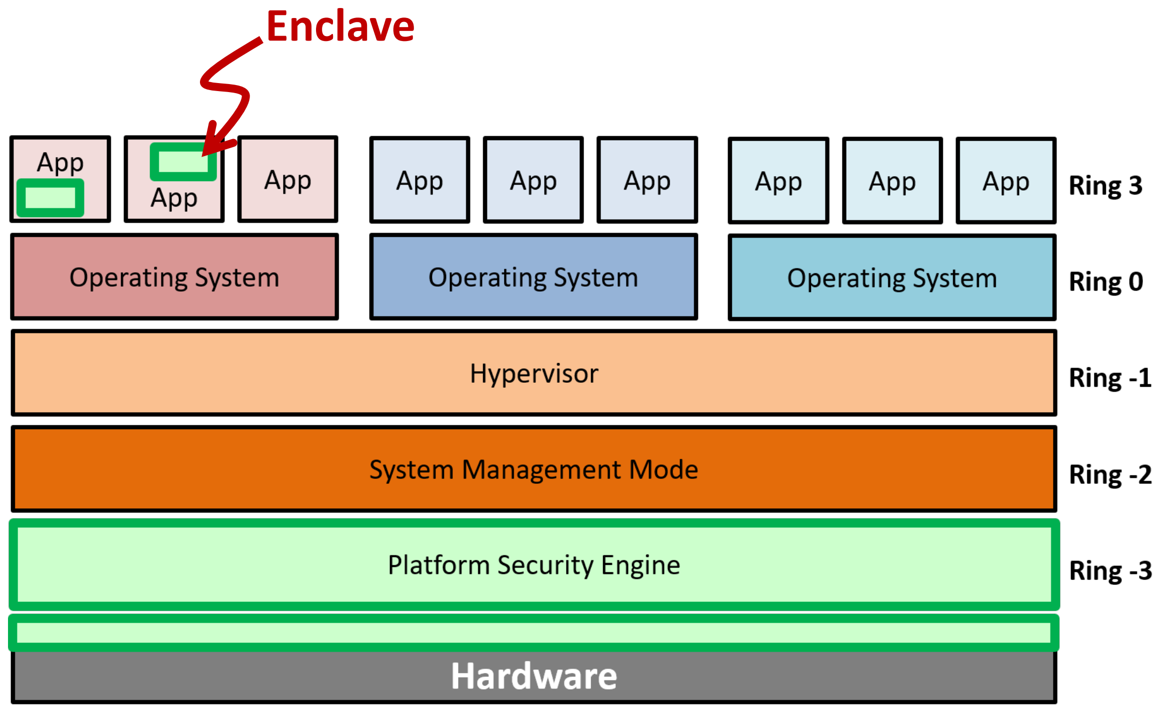


Trusted Execution Environments

- Create environment where **only protected software resides and executes**, supported by a **minimal TCB**



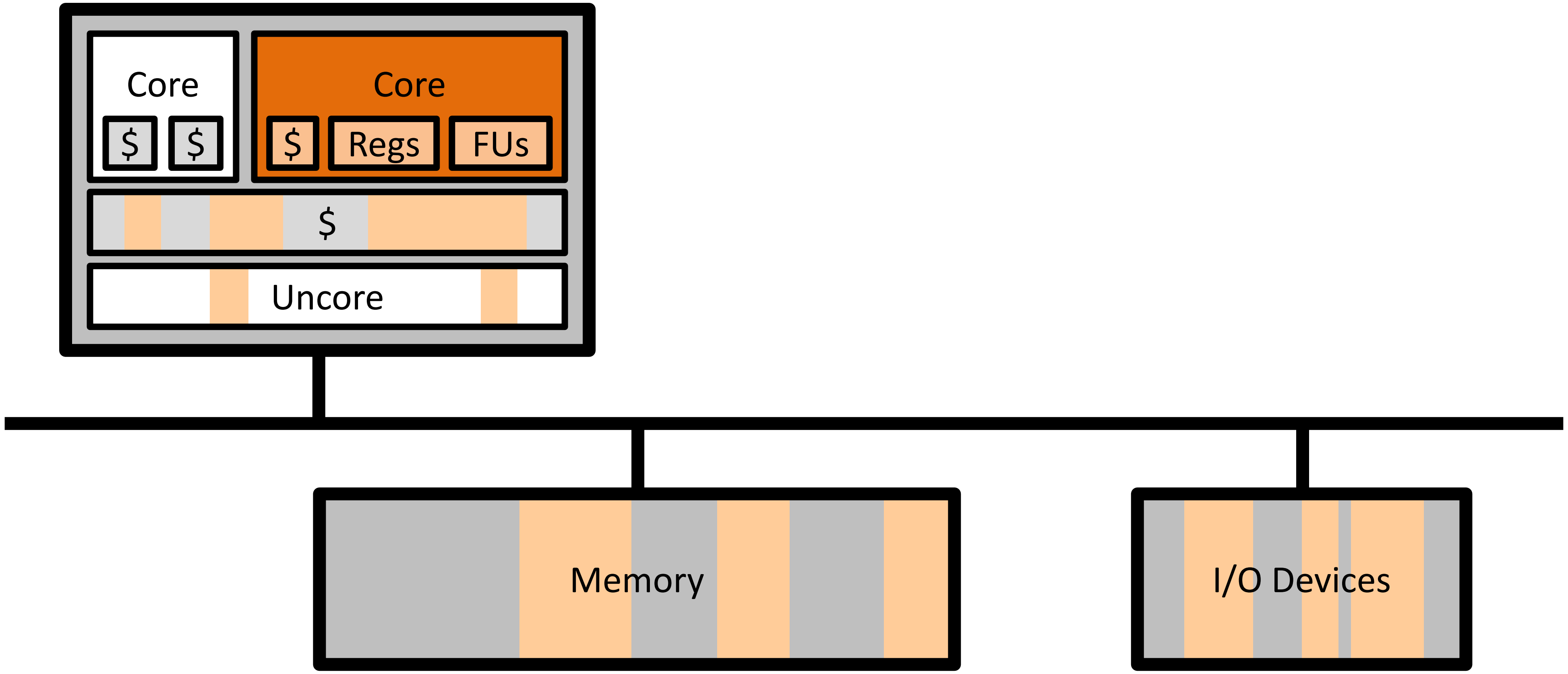
AMD Secure Encrypted Virtualization (SEV)



Intel Software Guard Extensions (SGX)

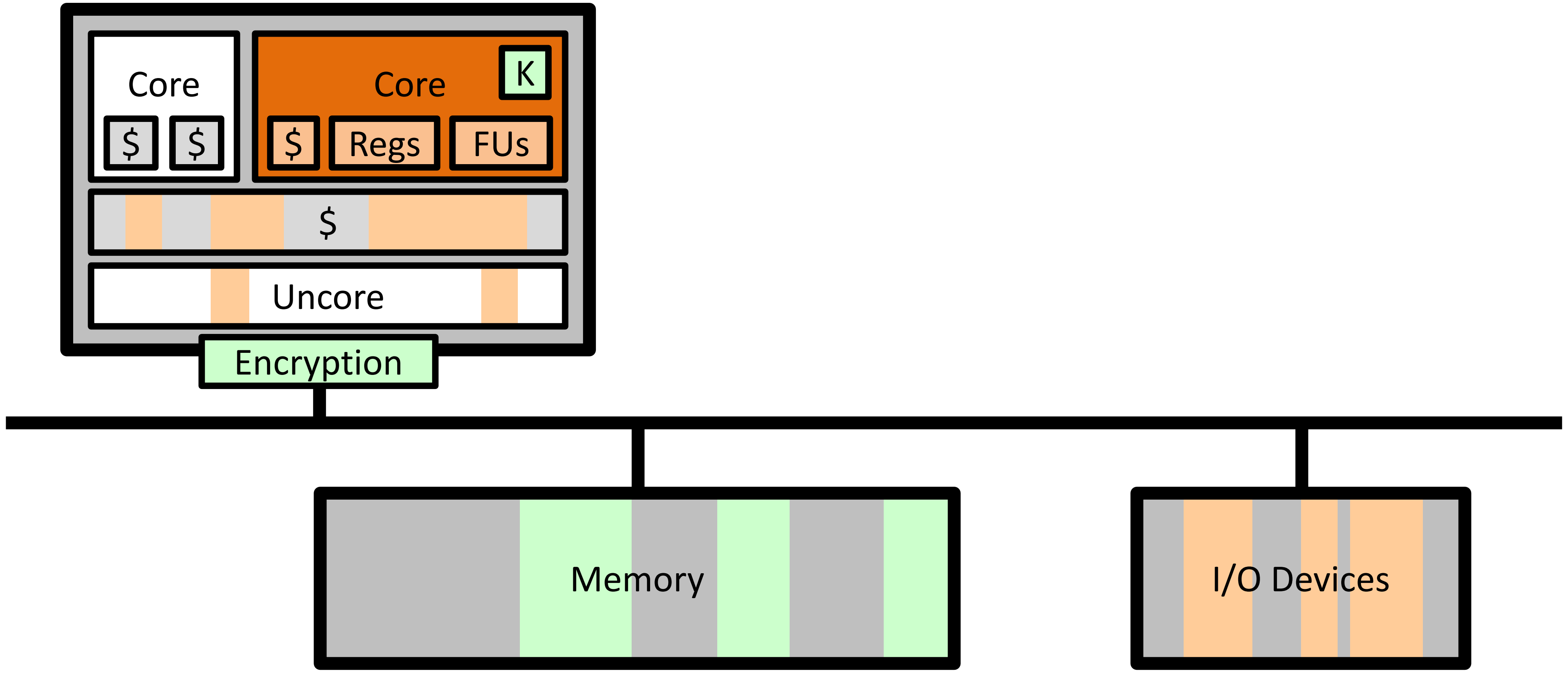
Trusted Execution Environments

- The main challenge is to protect the software state of the TEE given the fact that its state is unavoidably **dispersed all over the system** and specifically outside of the TCB and inside untrusted software and hardware components



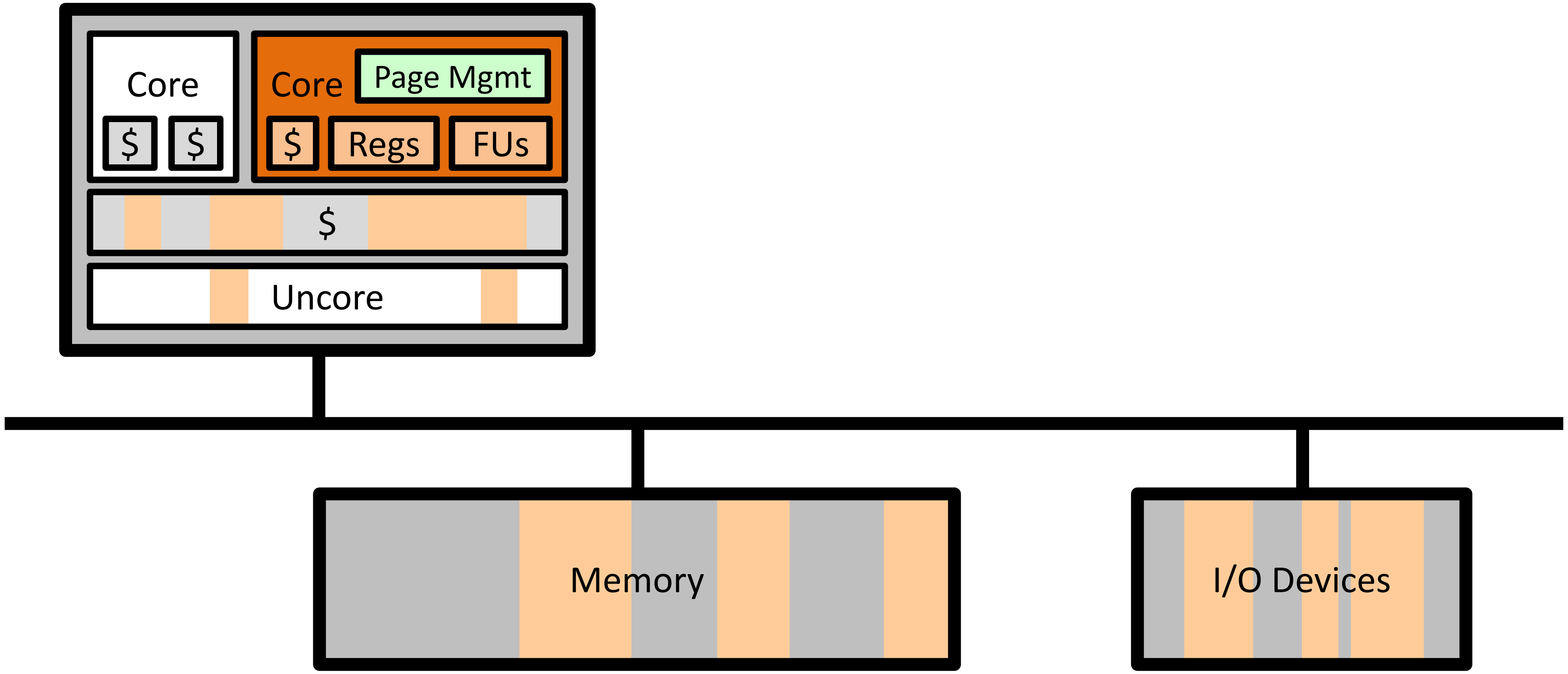
Confidentiality through Encryption

- **Symmetric encryption** ensures confidentiality outside of the processor
- This usually implies **hardware encryption/decryption modules** at the edge and locally stored keys



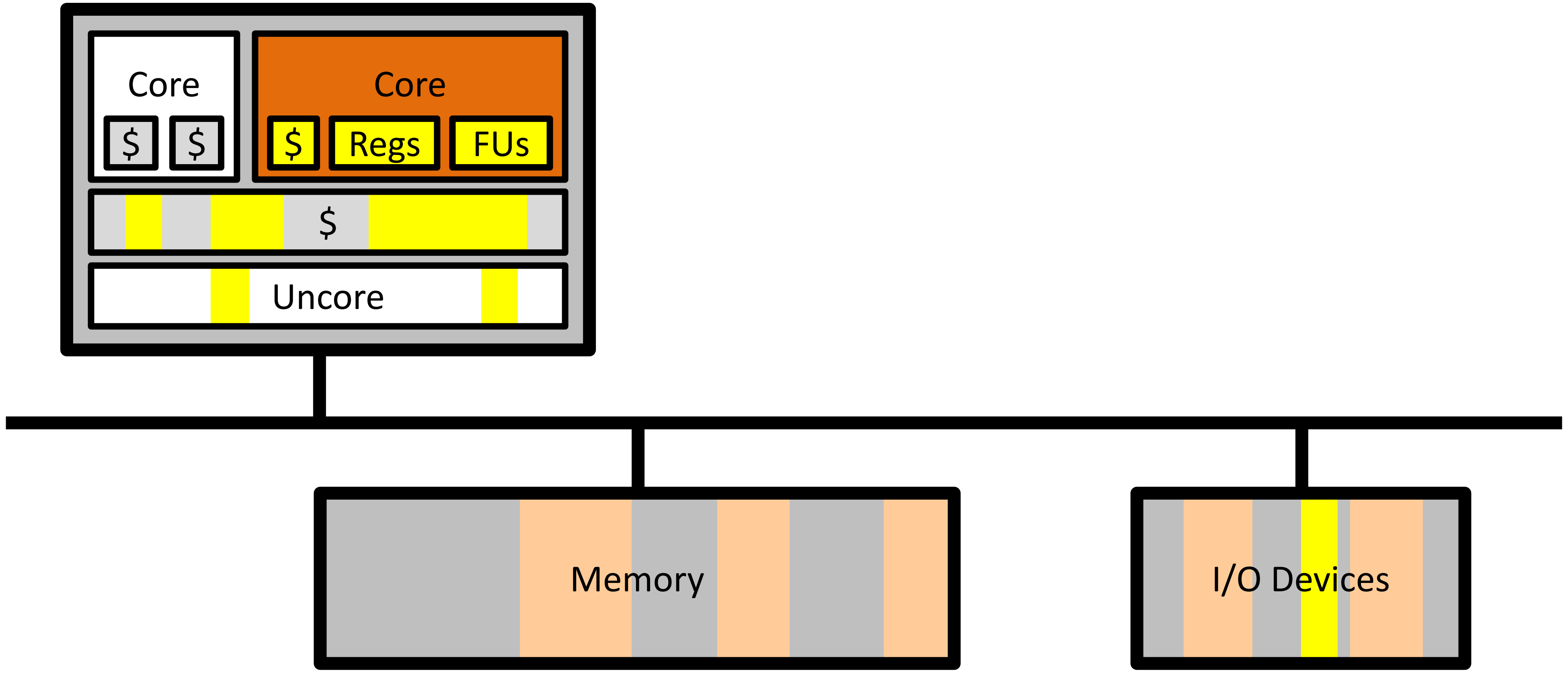
Confidentiality through Isolation

- Isolation can happen through usual means (page tables, etc.) but **memory management cannot be under the control of untrusted entities**
- TEEs and their TCB hardware should be in charge of their own page management



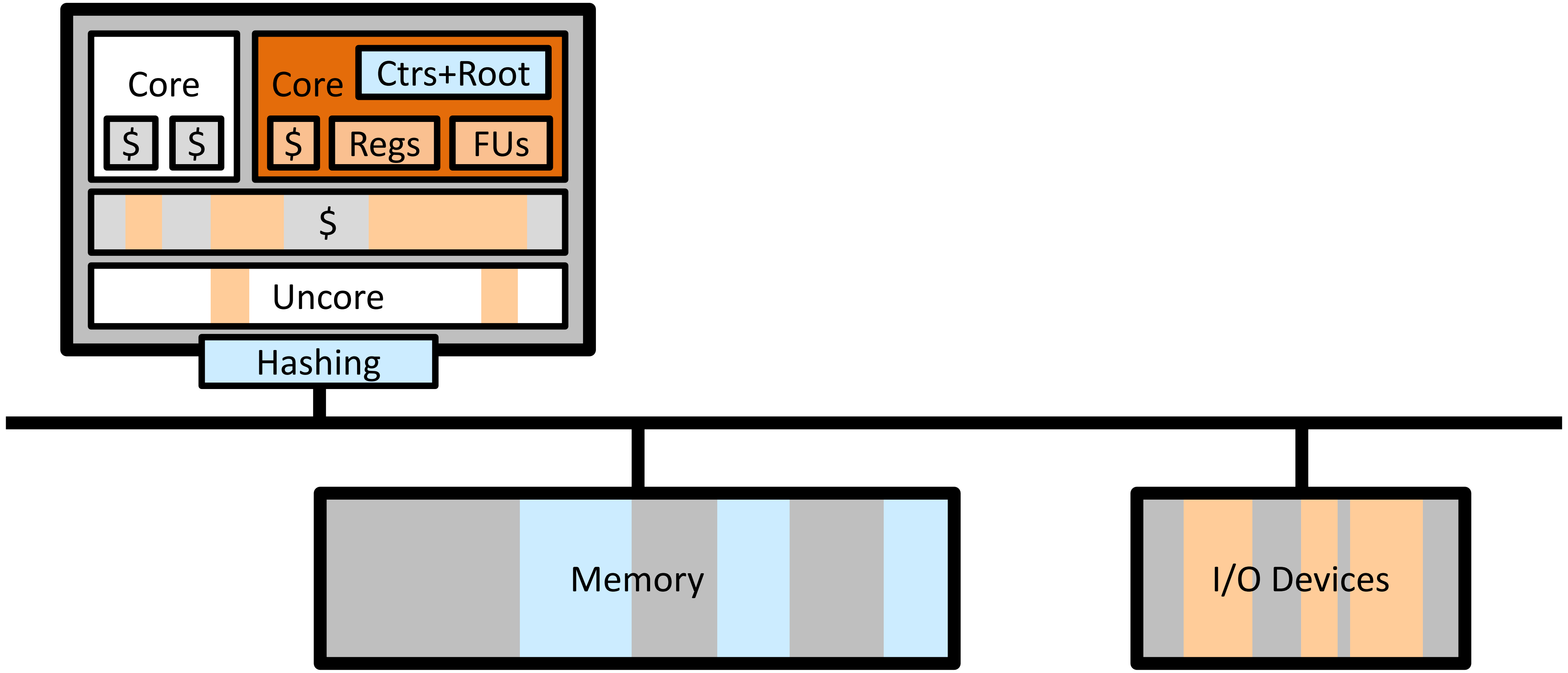
Confidentiality through State Flushing

- **Architectural and microarchitectural state** across all parts of the system need to be **flushed** before untrusted entities control the system (classic target of side-channel attacks)
- The challenge is to **identify all places** where there is confidential state



Integrity through Cryptographic Hashing

- **One-way hashing** ensures integrity of everything stored outside of the processor
- Again, this usually implies **hardware modules** at the edge and **locally stored nonces and root hashes**



4

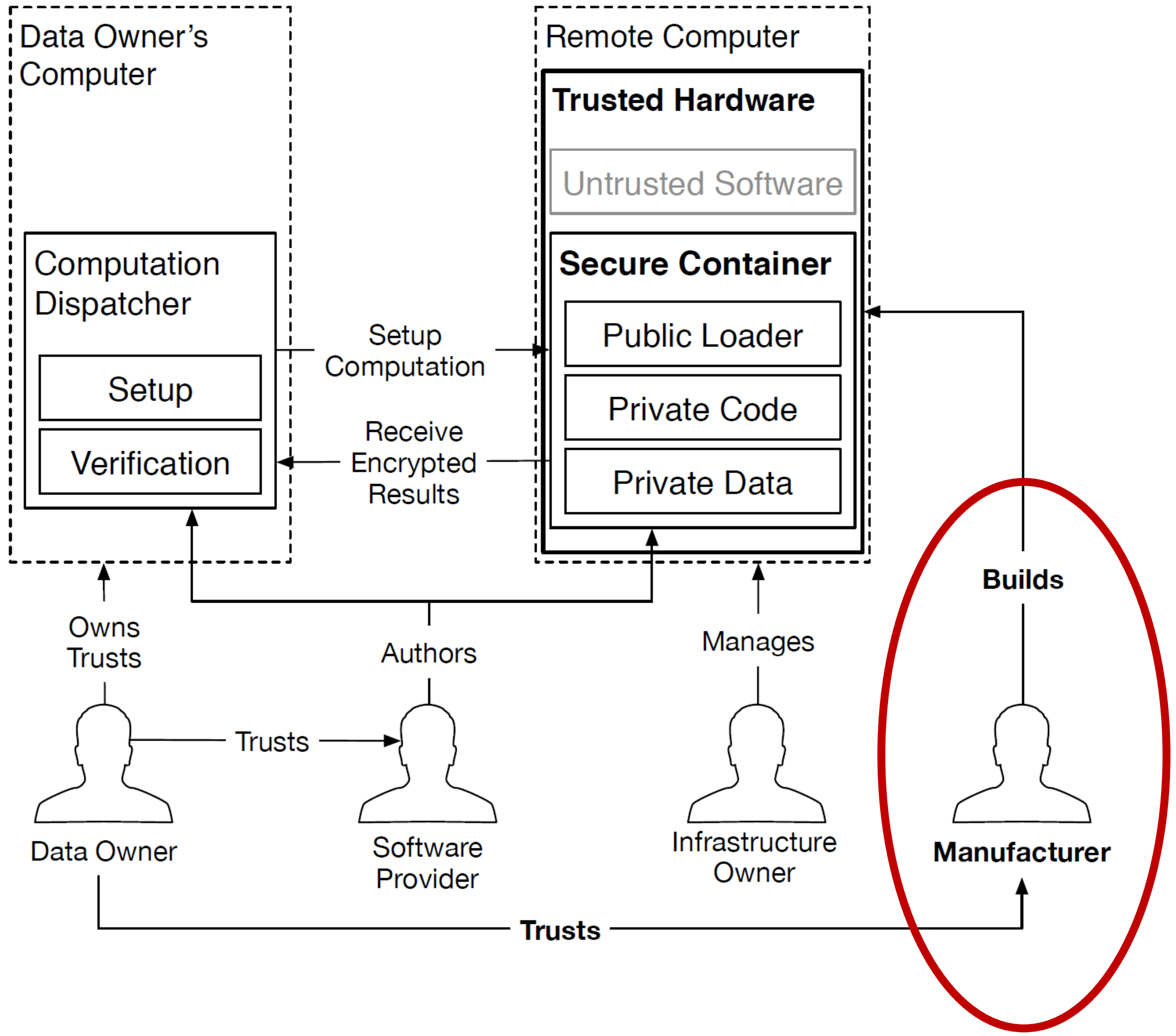
Intel Software Guard Extensions (SGX)

Intel SGX

- Problem: Execute **critical software on a remote computer owned and maintained by an untrusted party**, with some integrity and confidentiality guarantees
- Needs two fundamental properties
  - **Isolation**
    - Each secured environment is protected from all other software running on the machine (including OS, hypervisor, etc. and other secured environments)
  - **Attestation**
    - Provide a proof that the software running inside the protected environment is genuine and untampered
- TCB reduces to the CPU chip (hardware) and the critical code (software)

Intel SGX

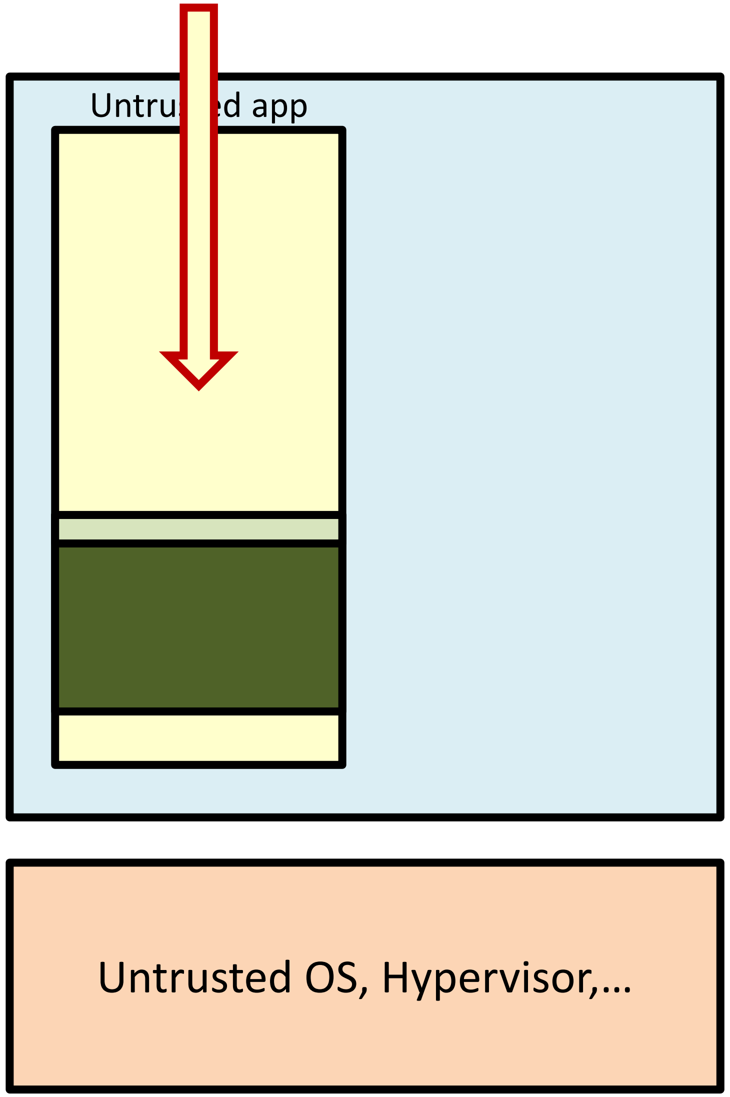
- Data owner **trusts the hardware** running in a remote computer operated by an **untrusted infrastructure owner**
- The trusted hardware establishes a **secure container (enclave)** and supplies the user with a **proof** that they are accessing a specific piece of software running into the enclave
- The data owner **uploads encrypted** data that the software in the enclave decrypts and processes
- The enclave software **encrypts the results** and sends them back to the data owner
- The system software of the **infrastructure owner** is in charge of managing resources and devices as in ordinary systems, but has **no access to the code and data of the enclave**



Source: Costan and Devadas, Cryptology ePrint, 2016

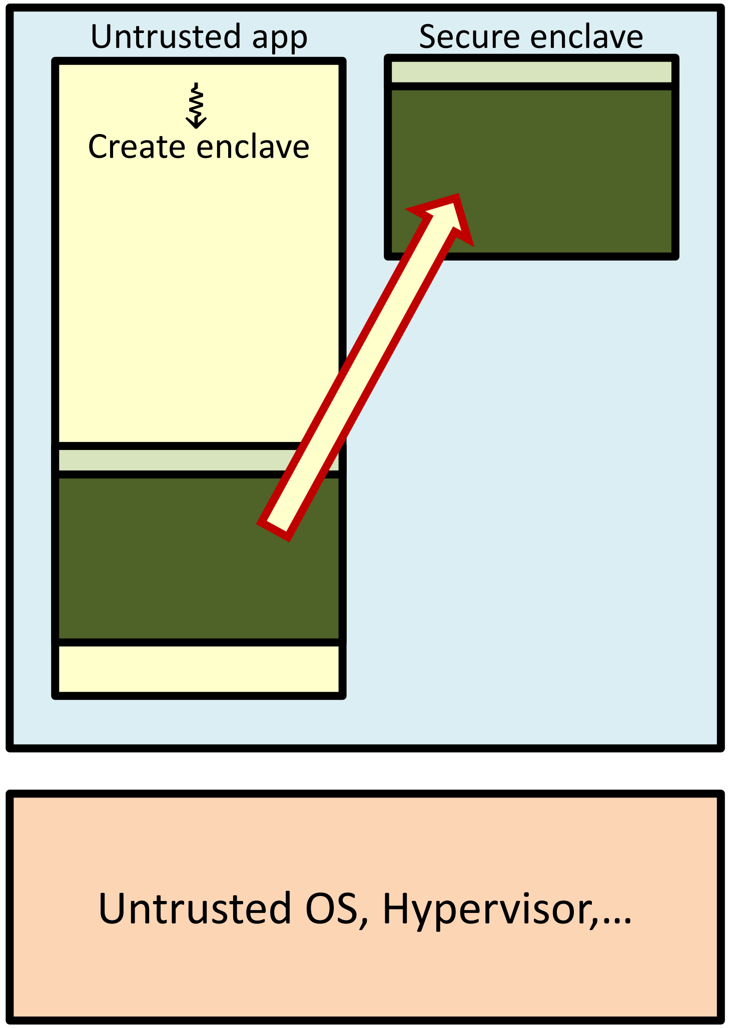
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**



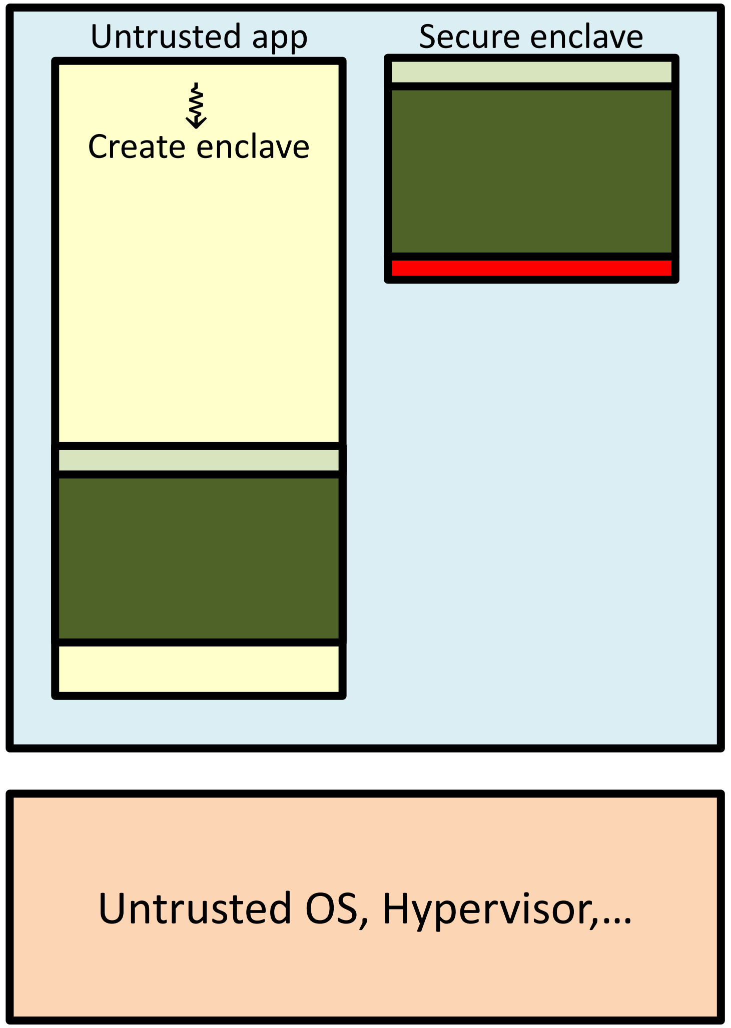
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**
2. Untrusted code in the application, through an untrusted OS, asks SGX to **setup the enclave and copy there code and data** from unprotected memory; initial payload is unprotected



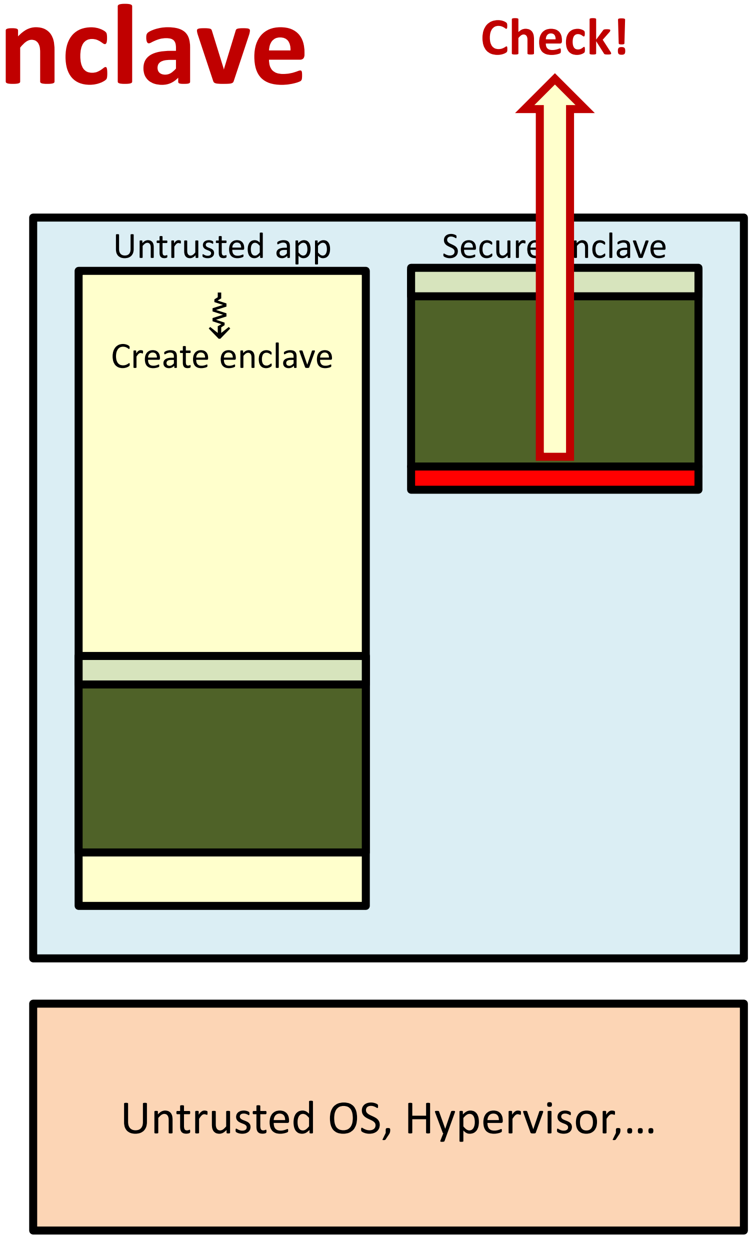
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**
2. Untrusted code in the application, through an untrusted OS, asks SGX to **setup the enclave and copy there code and data** from unprotected memory; initial payload is unprotected
3. Once done, the enclave is marked as initialized and the **content is cryptographically hashed** into a final measurement hash



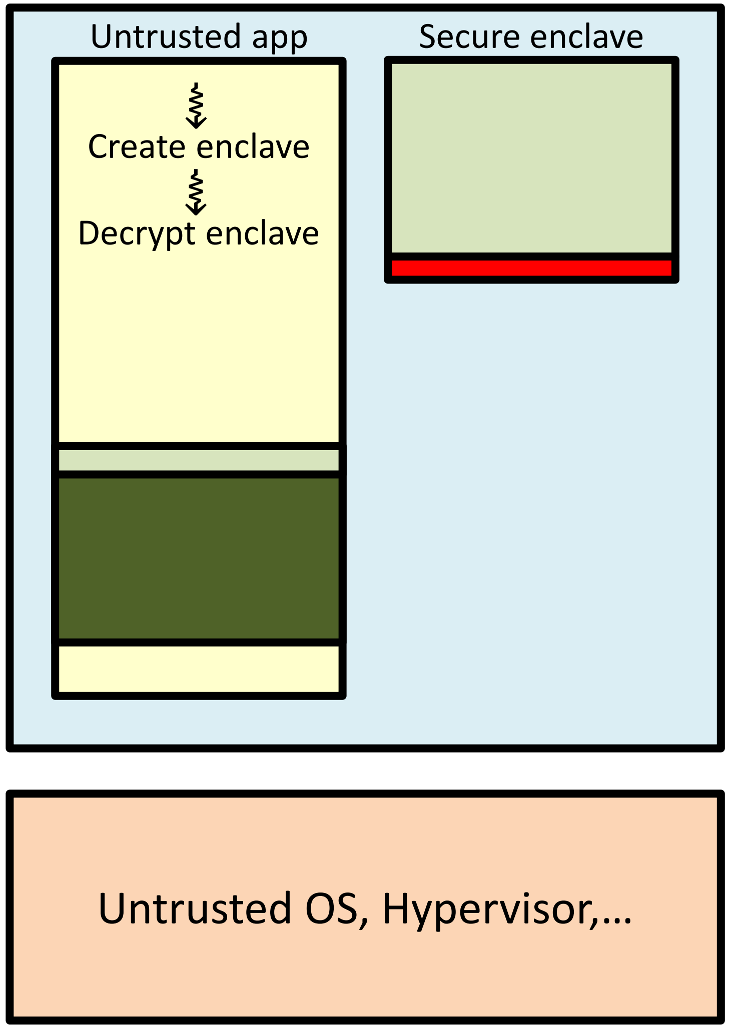
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**
2. Untrusted code in the application, through an untrusted OS, asks SGX to **setup the enclave and copy there code and data** from unprotected memory; initial payload is unprotected
3. Once done, the enclave is marked as initialized and the **content is cryptographically hashed** into a final measurement hash
4. The remote user can undergo a **software attestation process** to obtain a proof, through the measurement hash, that the enclave is setup properly



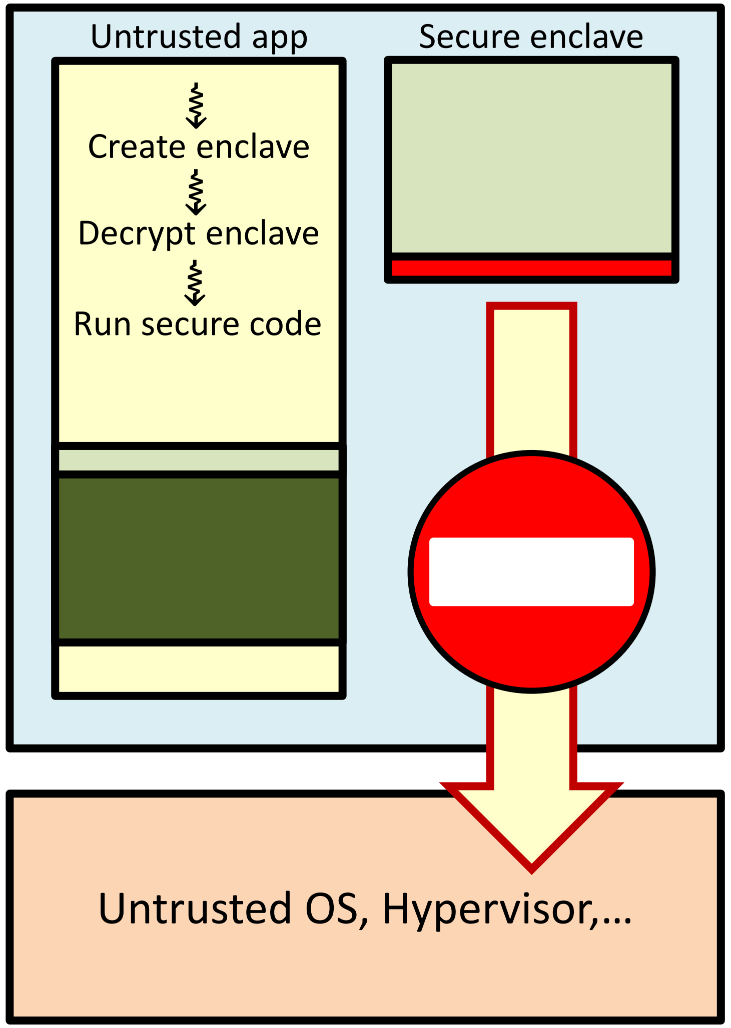
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**
2. Untrusted code in the application, through an untrusted OS, asks SGX to **setup the enclave and copy there code and data** from unprotected memory; initial payload is unprotected
3. Once done, the enclave is marked as initialized and the **content is cryptographically hashed** into a final measurement hash
4. The remote user can undergo a **software attestation process** to obtain a proof, through the measurement hash, that the enclave is setup properly
5. Trusted code in the enclave can now **decrypt the payload**, now protected by being inside the enclave



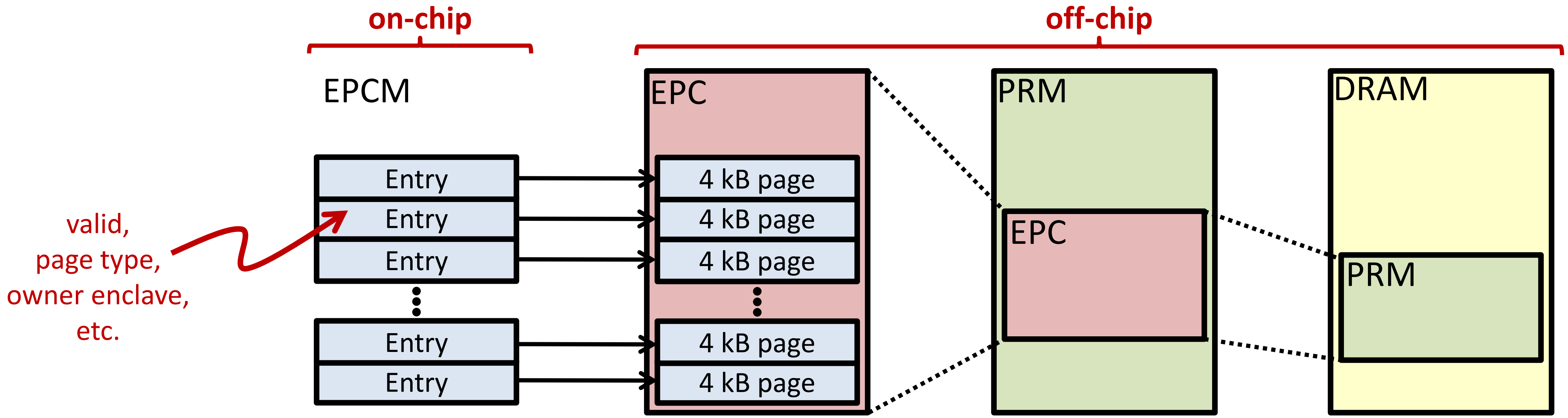
Functionality of an Enclave

1. The **remote user** launches their **untrusted app**
2. Untrusted code in the application, through an untrusted OS, asks SGX to **setup the enclave and copy there code and data** from unprotected memory; initial payload is unprotected
3. Once done, the enclave is marked as initialized and the **content is cryptographically hashed** into a final measurement hash
4. The remote user can undergo a **software attestation process** to obtain a proof, through the measurement hash, that the enclave is setup properly
5. Trusted code in the enclave can now **decrypt the payload**, now protected by being inside the enclave
6. **Trusted code in the enclave can be invoked** through mechanism similar to those used to switch to kernel mode
7. **Exceptions** while executing enclave code are **handled by SGX first** (see later) to protect secrets



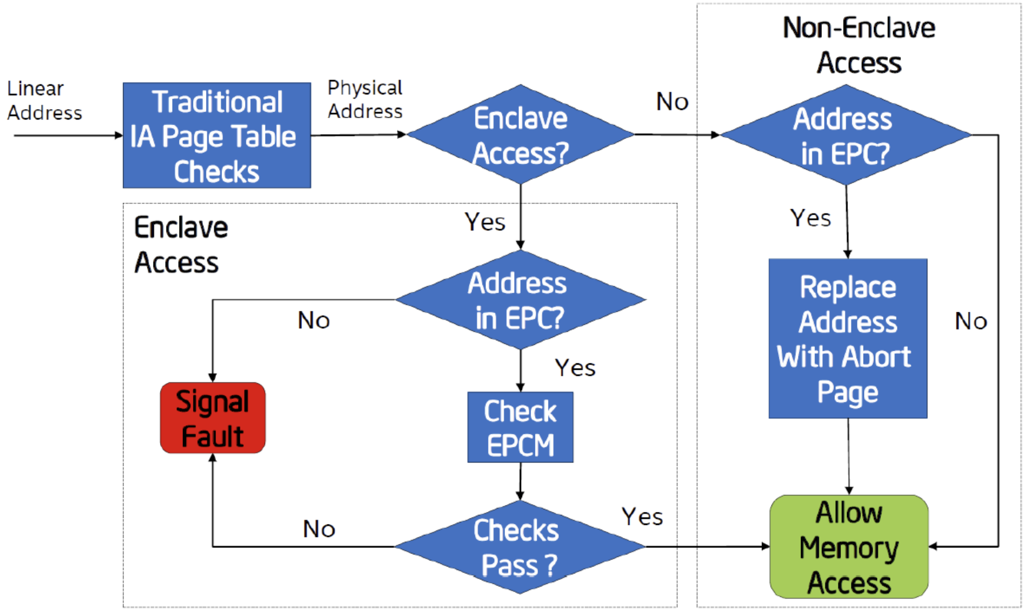
Physical Memory Organization

- PRM: Processor Reserved Memory = pages reserved by SGX for enclaves
  - Defined in the BIOS, adjacent power-of-two area of physical memory
- EPC: Enclave Page Cache = pages allocated by SGX for enclaves
  - Allocated by kernel or hypervisor, encrypted in hardware with keys generated at boot time
- EPCM: Enclave Page Cache Map = metadata of each EPC page such as valid, owner, etc.
  - Inside the processor, fixed size, limits the maximum EPC size (e.g., 128 MB)



EPC Isolation

Memory Access



- If enclave access:
  - Address in EPC? → "Address in the page table?"
  - Check EPCM → "Check page table metadata"
- Nothing really surprising: the **classic job** of the OS or hypervisor now done at the physical page level by the hardware (special instructions, etc.)
- OS code replaced by **processor firmware**
- **Smallest TCB**
- **Security by obscurity?**
  - Nobody is supposed to change the firmware
  - Nobody is supposed to see or understand the firmware

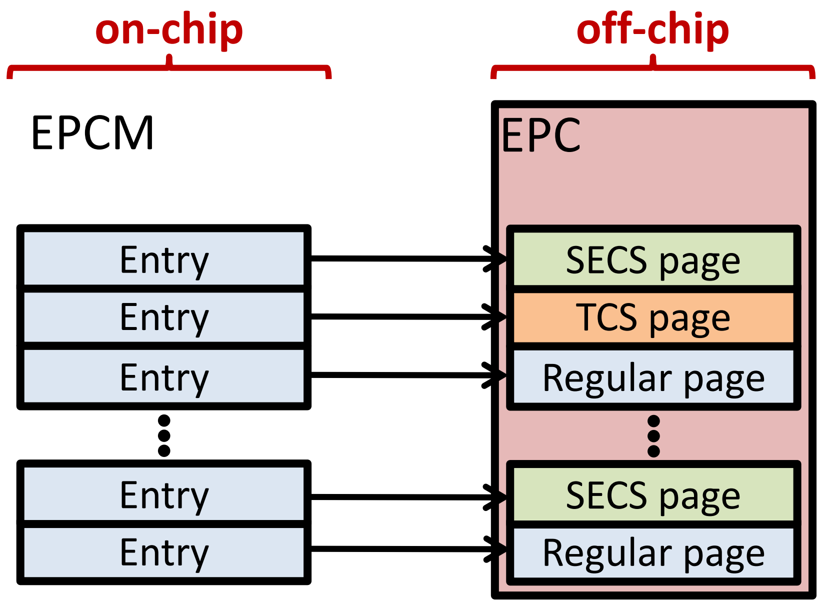
Source: van Dijk, Uconn CSE-5095, 2017

EPC Management Pages

- Most pages in EPC are code or data of the enclaves
- Some pages are reserved for SGX management
  - **SGX Enclave Control Structure (SECS)** pages: enclave attributes, hashes for attestations, etc.
  - **Thread Control Structure (TCS)** pages: allow multiple threads to execute the enclave code concurrently
- These pages are neither accessible to hypervisors, OSs, etc. nor to the enclave code itself, only to SGX

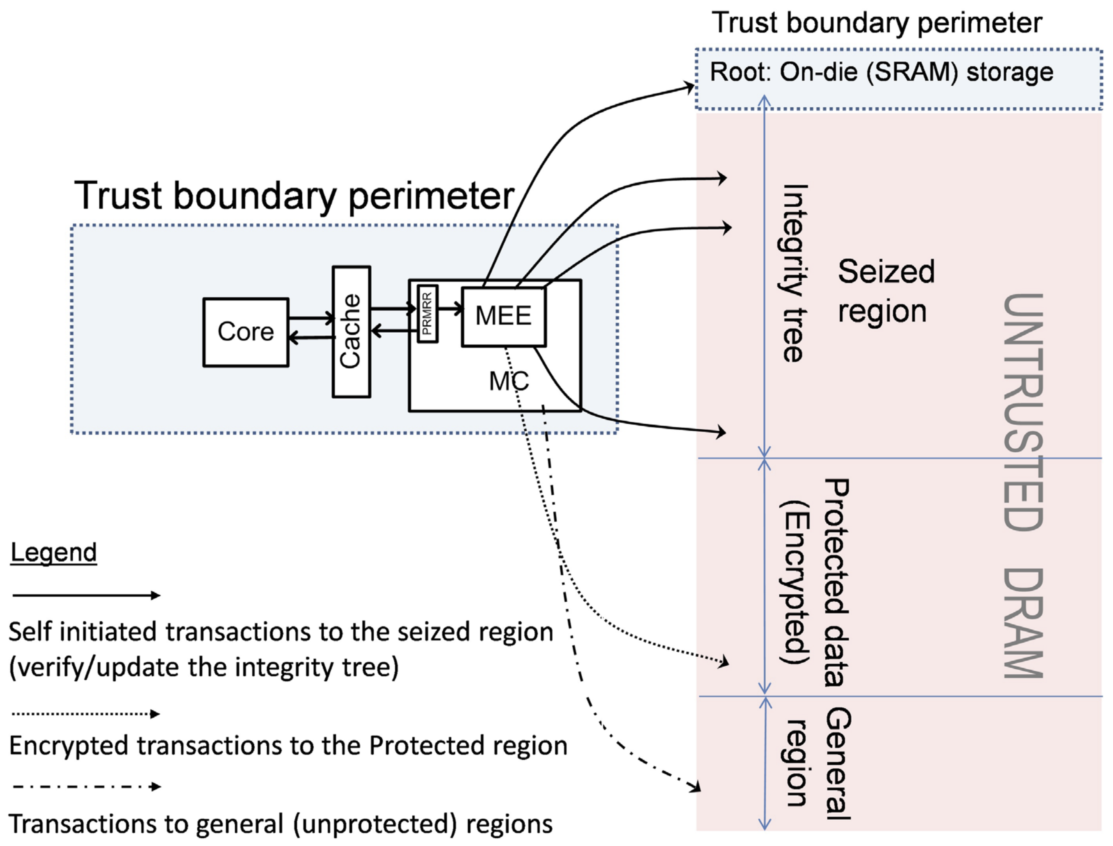
Exceptions

- On an exception during the execution of enclave code, **SGX dumps the state in EPC pages linked to the TCS** and restores the application state (thus hiding the enclave state to the application), before executing the exception handler
- Again, a bit more of the classic kernel job shifted into the processor (a pre-handler part of SGX)



Memory Encryption and Integrity

- EPC pages are **encrypted** by a hardware **Memory Encryption Engine (MEE)** so that no snooping or Coldboot attacks can succeed
- MEE works at the resolution of **cache lines (512 bits)**
- MEE encrypts every piece of data in a protected region of untrusted memory but also spontaneously **maintains an integrity tree in untrusted memory**
- The root of the integrity tree is stored in protected memory inside the hardware TBC (processor chip)



Trust boundary perimeter
Root: On-die (SRAM) storage
Trust boundary perimeter
Core
Cache
PRMRR
MEE
MC
Integrity tree
Seized region
UNTRUSTED DRAM
Protected data (Encrypted)
General region
Legend
Self initiated transactions to the seized region (verify/update the integrity tree)
Encrypted transactions to the Protected region
Transactions to general (unprotected) regions

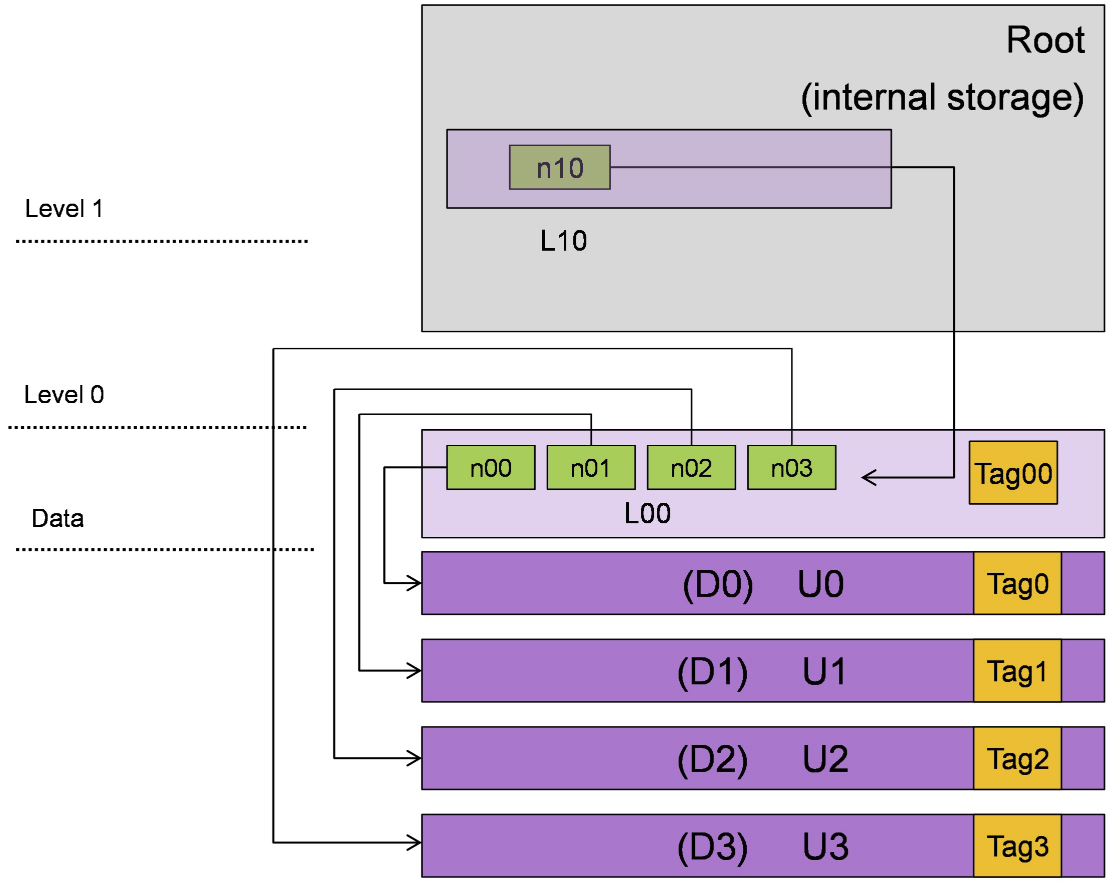
Source: Gueron, Cryptology ePrint, 2016

EPC Swapping

- But EPC is only a limited-size cache: **enclave pages** may need to be **swapped out to non-EPC memory**
  - Non-EPC memory is **unsecure** and replay attacks may happen
- On EPC swap out:
  - EPC page is **decrypted**
  - EPC page is **encrypted again** with **versioning** information to ensure freshness and **signed** to check for integrity, and saved to non-EPC memory together with **128 byte of metadata**
  - Versioning information is saved in **dedicated EPC management pages**
- On EPC swap in:
  - Complementary actions

Principle of the Memory Integrity Tree

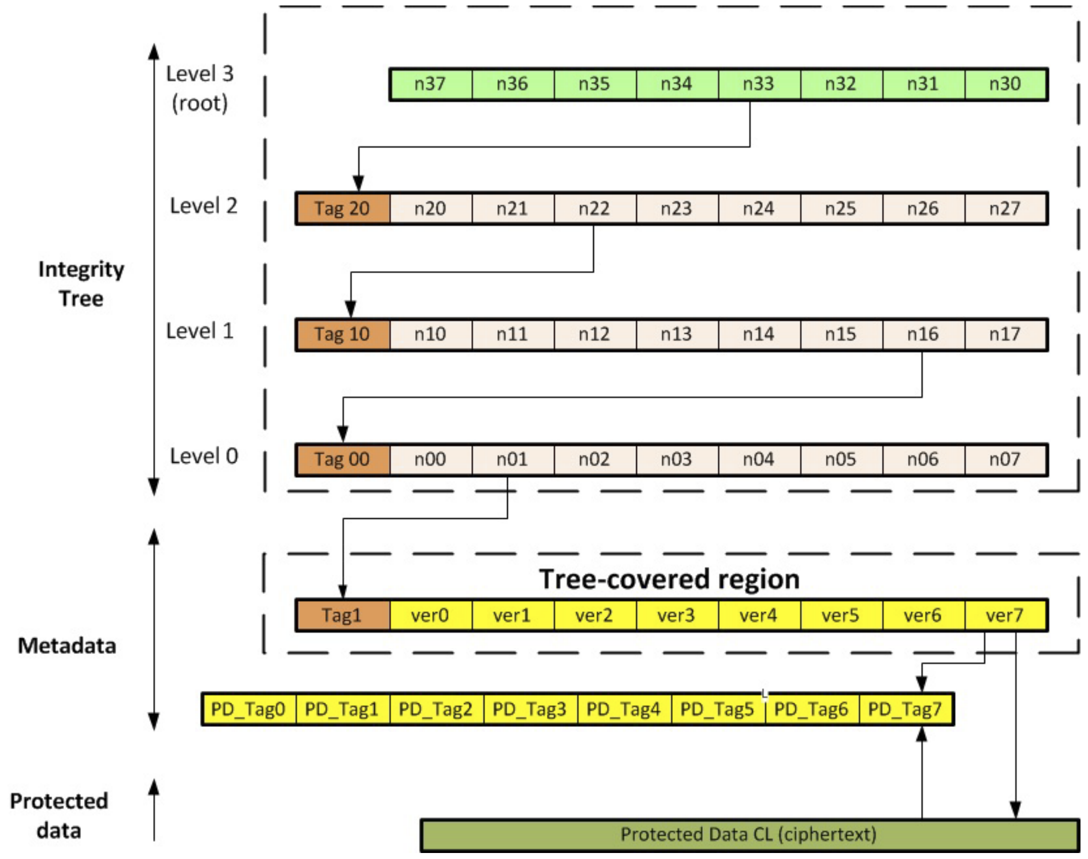
- Tag$x$ is the digest of data D0 together with nonce n0x
- Tag0x is the digest of nonces n00-n04 together with n1x
- The **tree root** nonce is secure because **stored in memory internal to the processor**
- Read and verify:
  1. All tags can be computed **independently** and in any order
  2. If **any** check failed, integrity compromised
- Write and update:
  1. **Preemptive check** to avoid replay attacks
  2. Update, increment nonces, and **recompute tags**
  3. **Write** tags



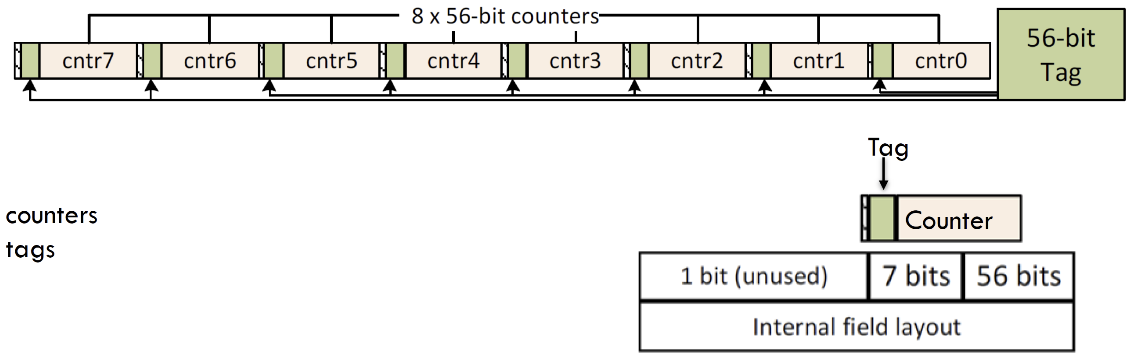
Source: Gueron, Cryptology ePrint, 2016

MEE Actual Data Structure

- Same principle but **8-ary tree** with everything organized in **512-bit cache lines** and packed appropriately
- Everything accessible with simple **hardware friendly** bit-shift operations
- A memory region of 128 MB contains
  - 96 MB of protected data (**efficiency = 3/4**)
  - 24 MB of metadata (nonces and tags = 1/8 data + 1/8 data)
  - 1.5 MB of tree's L0 (= 1/8 of metadata)
  - 192 kB of tree's L1 (= 1/8 of L0)
  - 24 kB of tree's L2 (= 1/8 of L1)
  - Waste for the alignment
  - 3kB of tree's L3 (= 1/8 of L2) → **in SRAM**
- MEE **performance** overhead around **2-14%**



Level 3 (root)
n37 n36 n35 n34 n33 n32 n31 n30
Level 2
Tag 20 n20 n21 n22 n23 n24 n25 n26 n27
Integrity Tree
Level 1
Tag 10 n10 n11 n12 n13 n14 n15 n16 n17
Level 0
Tag 00 n00 n01 n02 n03 n04 n05 n06 n07
Tree-covered region
Tag1 ver0 ver1 ver2 ver3 ver4 ver5 ver6 ver7
Metadata
PD_Tag0 PD_Tag1 PD_Tag2 PD_Tag3 PD_Tag4 PD_Tag5 PD_Tag6 PD_Tag7
Protected data
Protected Data CL (ciphertext)



56-bit counters
56-bit tags

Source: Gueron, Cryptology ePrint, 2016

Root of Trust

- Security requires many **different cryptographic keys** for multiple purposes
  - Private and public keys for authentication
  - Secret keys for confidentiality
  - Keys for integrity checks
- Some can be **random and ephemeral** (e.g., for encrypting data into DRAM) → generated at boot
- Most need to depend on a something **unique and persistent**: a root of trust key stored in the processor and accessible only to the TCB
  - **Root Provisioning Key (RPK)**, stored by Intel
  - **Root Sealing Key (RSK)**, that Intel declares to erase after manufacturing
- Classic security issues: **Public Key Infrastructure, Certificate Authority, revocation, etc.**

5

ARM TrustZone

ARM TrustZone

- A very different system from Intel SGX
- The basic business model and market is very different: **ARM does not build chips but licenses Intellectual Property**; many customers only license the architecture and build the processor themselves (e.g., Apple)
- TrustZone is a collection of hardware mechanisms which conceptually **partition a system and its resources in a secure and a nonsecure world**

ARM TrustZone

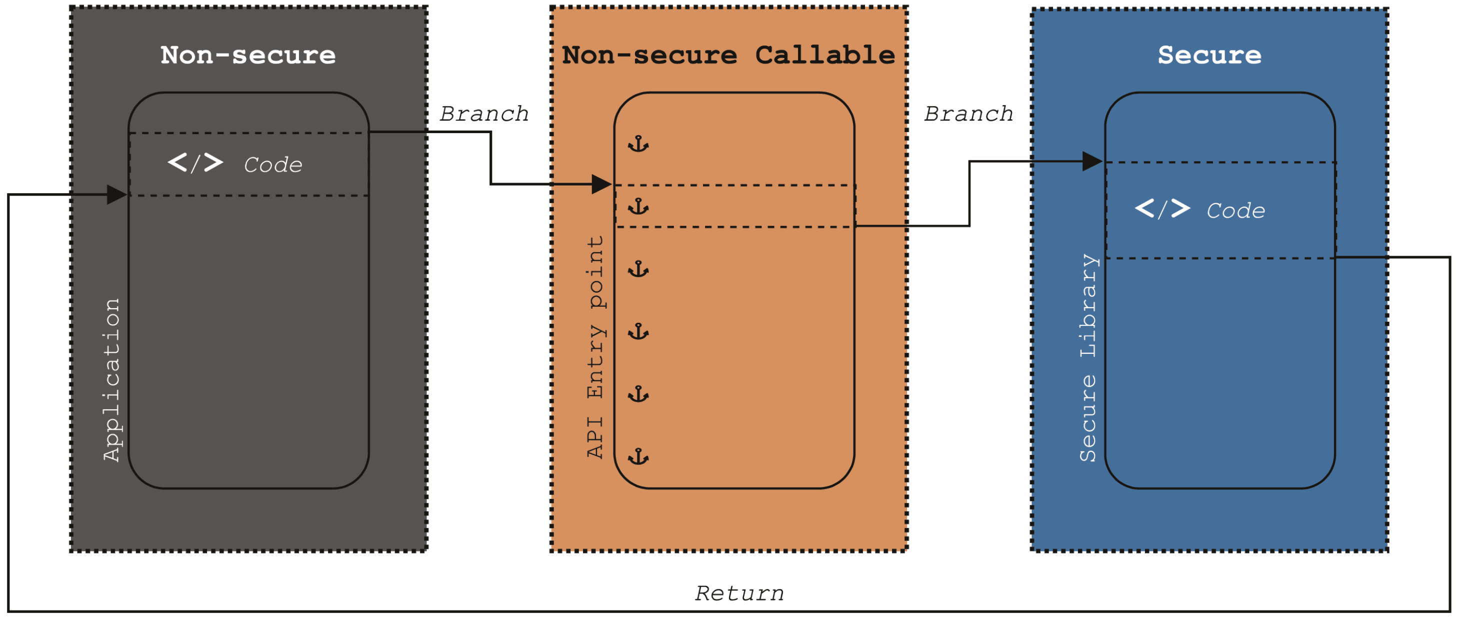
- **Mainly about isolation**
  - **Hardware**: an additional bit in the AMBA AXI bus protocol informs the system components (e.g., caches) and peripherals of accesses within the secure world
    - Hardware TCB is essentially the chip (components either handle securely accesses as appropriate or are trusted to refuse secure requests)
  - **Software**: partitioned in two parts with a special **monitor** to transition between them
    - Software TCB is the software in the trusted part
- Only one TEE per system (vs. multiple enclaves in Intel SGX)

Extended Privilege and Memory Isolation

- Essentially, introduces a **secure/nonsecure partition mode** orthogonal to the classic privilege levels (Thread/Handler)
  - Fairly classic register banking and duplications—e.g., four copies of register R13 (stack pointer) instead of only two
- Memory split in three classes
  - **Secure** and **Nonsecure**
  - **Secure but callable from nonsecure** code (a special API is responsible of the return to the nonsecure world)
- Hardware **Attribution Units** and **Protection Controllers** intercept addresses to memory and raise exceptions in case of violation
  - Much simpler than SGX enclave accesses, but conceptually similar
  - Limited number of secure regions (e.g., eight)

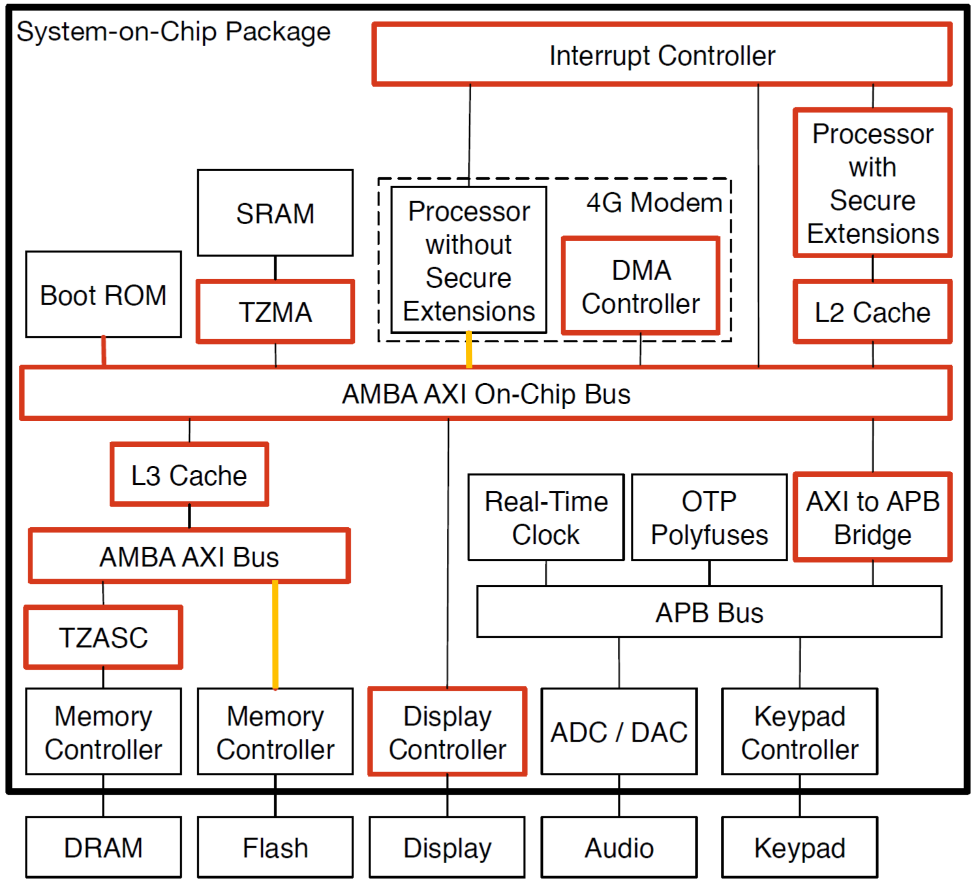
Calling Restrictions

- Branching into nonsecure callable region **checks that the first instruction at the destination** is a **Secure Gateway** (SG) instruction which sets the processor in secure mode; **special branch instructions** to return from secure to nonsecure
- The nonsecure callable region is a **bridge to call secure code**, not callable directly
- Alternatively, the more classic privileged instruction **Secure Monitor Call** (SMC) jumps into the monitor



Hardware View

- System components receive a special AXI bit as a sort of extension of the address and can thus be TrustZone-aware (in **red** on the figure)
- Some busses omit the TrustZone secure bit in the bus address (in **orange** on the the figure)
- Since there is no ARM chip but only some architectural definitions, the security **properties depend on the actual system design** on the ARM licensee and on the details of all components



Source: Costan and Devadas, Cryptology ePrint, 2016

Software View

- The **secure world** contains several necessary components
  - A trusted boot loader (hardware reset → secure mode)
  - A small trusted OS
  - A monitor to switch back and forth from the nonsecure world
  - Security critical applications
- The secure monitor has unrestricted access to the nonsecure world
- ARM provides **reference implementations** of secure firmware inclusive of secure boot services and the secure monitor
- **No security by obscurity** on the software side (at least not from ARM, but probably quite a bit by the system designers)

Attack	TrustZone	TPM	TPM+TXT	SGX	XOM	Aegis	Bastion	Ascend, Phantom	Sanctum
Malicious containers (direct probing)	N/A (secure world is trusted)	N/A (The whole computer is one container)	N/A (Does not allow concurrent containers)	Access checks on TLB misses	Identifier tag checks	Security kernel separates containers	Access checks on each memory access	OS separates containers	Access checks on TLB misses
Malicious OS (direct probing)	Access checks on TLB misses	N/A (OS measured and trusted)	Host OS preempted during late launch	Access checks on TLB misses	OS has its own identifier	Security kernel measured and isolated	Memory encryption and HMAC	X	Access checks on TLB misses
Malicious hypervisor (direct probing)	Access checks on TLB misses	N/A (Hypervisor measured and trusted)	Hypervisor preempted during late launch	Access checks on TLB misses	N/A (No hypervisor support)	N/A (No hypervisor support)	Hypervisor measured and trusted	N/A (No hypervisor support)	Access checks on TLB misses
Malicious firmware	N/A (firmware is a part of the secure world)	CPU microcode measures PEI firmware	SINIT ACM signed by Intel key and measured	SMM handler is subject to TLB access checks	N/A (Firmware is not active after booting)	N/A (Firmware is not active after booting)	Hypervisor measured after boot	N/A (Firmware is not active after booting)	Firmware is measured and trusted
Malicious containers (cache timing)	N/A (secure world is trusted)	N/A (Does not allow concurrent containers)	N/A (Does not allow concurrent containers)	X	X	X	X	X	Each enclave its gets own cache partition
Malicious OS (page fault recording)	Secure world has own page tables	N/A (OS measured and trusted)	Host OS preempted during late launch	X	N/A (Paging not supported)	X	X	X	Per-enclave page tables
Malicious OS (cache timing)	X	N/A (OS measured and trusted)	Host OS preempted during late launch	X	X	X	X	X	Non-enclave software uses a separate cache partition
DMA from malicious peripheral	On-chip bus bounces secure world accesses	X	IOMMU bounces DMA into TXT memory range	IOMMU bounces DMA into PRM	Equivalent to physical DRAM access	Equivalent to physical DRAM access	Equivalent to physical DRAM access	Equivalent to physical DRAM access	MC bounces DMA outside allowed range
Physical DRAM read	Secure world limited to on-chip SRAM	X	X	Undocumented memory encryption engine	DRAM encryption	DRAM encryption	DRAM encryption	DRAM encryption	X
Physical DRAM write	Secure world limited to on-chip SRAM	X	X	Undocumented memory encryption engine	HMAC of address and data	HMAC of address, data, timestamp	Merkle tree over DRAM	HMAC of address, data, timestamp	X
Physical DRAM rollback write	Secure world limited to on-chip SRAM	X	X	Undocumented memory encryption engine	X	Merkle tree over HMAC timestamps	Merkle tree over DRAM	Merkle tree over HMAC timestamps	X
Physical DRAM address reads	Secure world in on-chip SRAM	X	X	X	X	X	X	ORAM	X
Hardware TCB size	CPU chip package	Motherboard (CPU, TPM, DRAM, buses)	Motherboard (CPU, TPM, DRAM, buses)	CPU chip package	CPU chip package	CPU chip package	CPU chip package	CPU chip package	CPU chip package
Software TCB size	Secure world (firmware, OS, application)	All software on the computer	SINIT ACM + VM (OS, application)	Application module + privileged containers	Application module + hypervisor	Application module + security kernel	Application module + hypervisor	Application process + trusted OS	Application module + security monitor

Source: Costan and Devadas, Cryptology ePrint, 2016

Conclusions

- There is definitely no magic one-stop solution for all security troubles
- Rather, we see an **enormous variety** on what TEEs are and what they are expected to protect from (= huge variety of quickly evolving business needs)
- The sole clear and sound motto appears to be "**reduce the attack surface to the bare minimum**"
- Focus appears to be mostly in **glorified versions of classic isolation mechanisms and classic security protocols** (e.g., attestation schemes), but also in some new unconventional features (e.g., memory encryption and integrity in SGX)
- The apparent **complexity** of some of these solutions seems alone and by itself a form of **fragility** (disclaimer: uninformed personal opinion)
- Commercial systems show **very little or no protection** from advanced microarchitectural and physical **side-channel attacks**
- Still a new and quickly evolving aspect of computer architecture which will **need some time to reach maturity** and some form of standardization

References

General

- J. Szefer, Principles of Secure Processor Architecture Design, Synthesis Lectures on Computer Architecture, Morgan & Claypool, 2019

Intel Software Guard Extensions (SGX)

- V. Costan and S. Devadas, *Intel SGX Explained*, Cryptology ePrint Archive, Report 2016:086, 2016
- S. Gueron, *A Memory Encryption Engine Suitable for General Purpose Processors*, Cryptology ePrint Archive, Report 2016:204, 2016

ARM TrustZone

- S. Pinto and N. Santos. *Demystifying ARM TrustZone: A Comprehensive Survey*. ACM Computing Surveys, volume 51, article 130, January 2019