Secure Processor Architectures

Chapter 3

[1] J. Szefer, “Principles of secure processor architecture design,” *Synth. Lect. Comput. Archit.*, vol. 13, no. 3, pp. 1–173, 2018.

Real-World Attacks

- Motivates the **need** for secure processor architectures

- Provide a **glimpse** of how wrong assumptions about hardware behavior (e.g., DRAM refresh) or unintended consequences of performance optimizations (e.g., speculative execution) affect security

- Same bugs and vulnerabilities of regular processors can affect secure processors too
  - Processor has bugs too, and sometimes are very expensive! (e.g., FDIV Pentium cost Intel millions)
  - Complexity implies bugs

Real-World Attacks: Coldboot

- Used to stole information from RAM while the system is powered off
  - DRAM capacitors charge don't disappear studently when turned off: there is a slowly decay. **The assumption about DRAM fast volatility is wrong**

- If the DRAM chips are cooled (e.g., via compressed air) the decay is slower (capacitor decay is **temperature dependent**)

- Interchange the DRAM chips to another computer and dump the contend of the chip it at **rogue OS boot**
  - **Stole keys, passwords, etc…**

- Solutions
  - Explicitly erase sensitive at power off. Use battery support to perform the operation
  - Encrypt memory content

Real-World Attacks: Rowhammer

- Modify memory non-accessible from the attacker process (assuming OS/VM are ok)

- Bypass OS/VM isolation by exploiting DRAM cross/contaminations of row contents
  - A specific and **repetitive** access pattern to accessible memory to the attacker can "modify" adjacent non-accessible rows

- Identify the victim's physical memory target (in an OS can be fixed) and try to allocate process data in an adjacent row (e.g., malloc across all memory). When achieved, **attack!**
  - **The attack might consist in code injection (write sniped of code that exploits system security).**

- Solutions:
  - Use better DRAM (hard do it. From generation to generation of DRAM the promises repeats, but the problem does not go away)
  - Prevent allocate the code in the same spots of the virtual-memory: **Address Space Layout Randomization** (ASLR)
  - Encrypt memory content

Real-World Attacks: Meltdown

- Exploits **side effects of out-of-order execution** and design decisions of certain **Intel** processor families to read arbitrary memory of kernel (mapped in process address space)
  - Share kernel addressing space is advantageous for system calls
  - Share doesn't mean the user code can access (`pte.us=1`)
- Privilege level in memory access (loads) is only checked at commit stage (Intel processors)
- Execute loads to kernel addresses (stores the data in cache) and prevent the load to reach commit (e.g., raising an exception before to issue the load)
- Apply cache-side attack to the set of the cache that stores the data to infer the value
  - Piece-by-piece (~1 byte) using to timing determine the "stolen" address

Real-World Attacks: Spectre

- Breaks isolation between apps by exploiting executive execution of instructions following branches
  - Like meltdown side-channel but affect any processor with branch–prediction
  - Harder to implement and cross application-attack

- Train in the attacker app the branch prediction (shared by all process running in the CPU) to force in the victim app a miss-prediction (branch prediction uses PC+history to make the prediction) using a sensitive piece of code (gadget)

```
if (x < array1_size)
    y = array2[array1[x]*256)
```

- Force the gadget to reach out-of-bounds access in array2 (will be cancelled later)
  - The remains of such access is in the cache and can be inferred via side-channel attack

- Solutions:
  - Disable branch predictor
  - Don't share branch predictor content between processes
  - Loads inside branch's acts as memory barriers

Real-World Attacks: Other Bugs and Vulnerabilites

- Processor has bugs, documented by manufacturers. Some of them can be security-critical (30 on 300 in [4])
  - Examples System Management Mode, Message Signaling Interrupt, etc..
  - GPU vulnerabilities can be used also to break isolation and steal information

- Attacks can focus on non-compute components: Thermal Sensors, DVFS, can be abused. Change the timing of certain operations and introduce faults (to leak information)

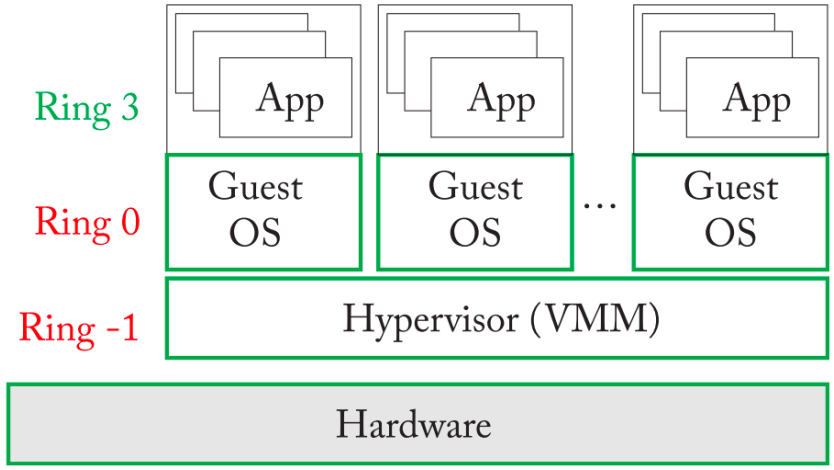
- Many exploits design goals (performance, area and energy) are susceptible of being exploited
  - v.gr. Performance: caches via timing side-channel
  - v.gr. Area: DRAM via Rowhammer
  - v.gr. Power: DVFS can generate faults

General-Purpose Architectures

- Secure processors (SP) are built on top of GP Processors (GPP) and expand them with security features
  - Its part of the Trusted Compute Base (**TCB**) Supports Trusted Execution Environment (**TEE**)

- GPP uses a ring-based protection mechanism to isolate App/OS/VMM
  - Certain ISA features available in each ring
  - When needed, controlled mechani `syscalls`, `vmexists`)

- **Compromised OS or VMM can attack Apps**. SP tries to address that (untrusted OS/VM)

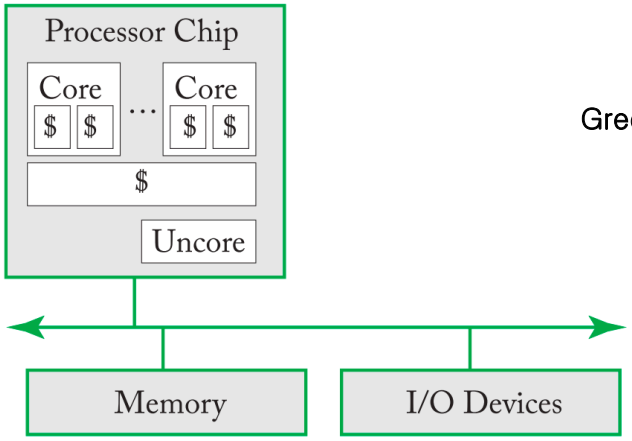


Green == Considered trusted

Typical components (traditional view)

- Software Components
  - Ring 1: Apps
  - Ring 0: OS
  - Ring -1: VMM

- Hardware Components
  - CPU, Mem, complex I/O systems

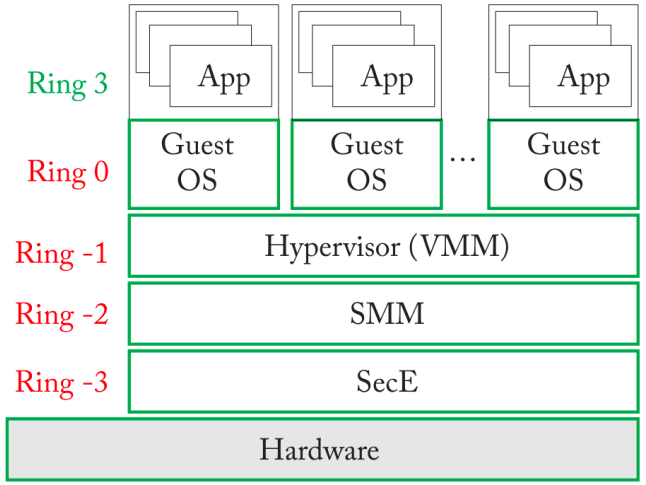


Green == traditionally considered trusted

Secure Processor Architectures (real view)

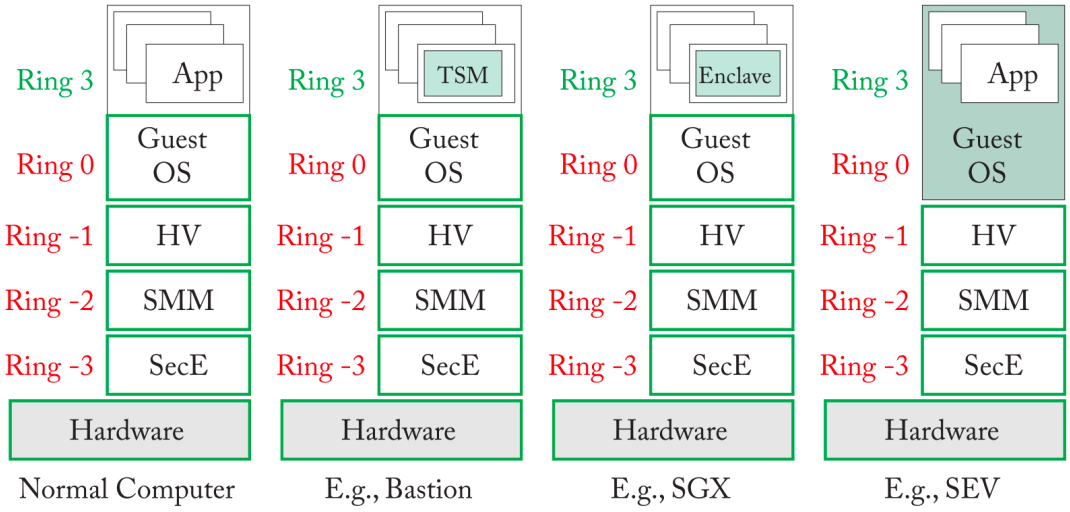
- **System Management Mode** (SMM) (ring -2)
  - Code part of the firmware run by GPP
  - Accessible via System Management Interface (SMI), asserting a pin in processor chip package or I/O over specific port
  - Management functionalities (e.g., IPMI) even if OS/VMM compromised
  - Uses security through obscurity

- **Platform Security Engine** (SecE) (ring -3)
  - Intel Management Engine, AMD Platform Secure Processor (it's an ARM with TrustZone)
  - **Small processor** isolated from the rest system (Intel in North-bridge), (AMD integrated)
  - Can be online even with power-down
  - Control system execution and emulate some hw features such as AMD SEV
  - Uses security through obscurity



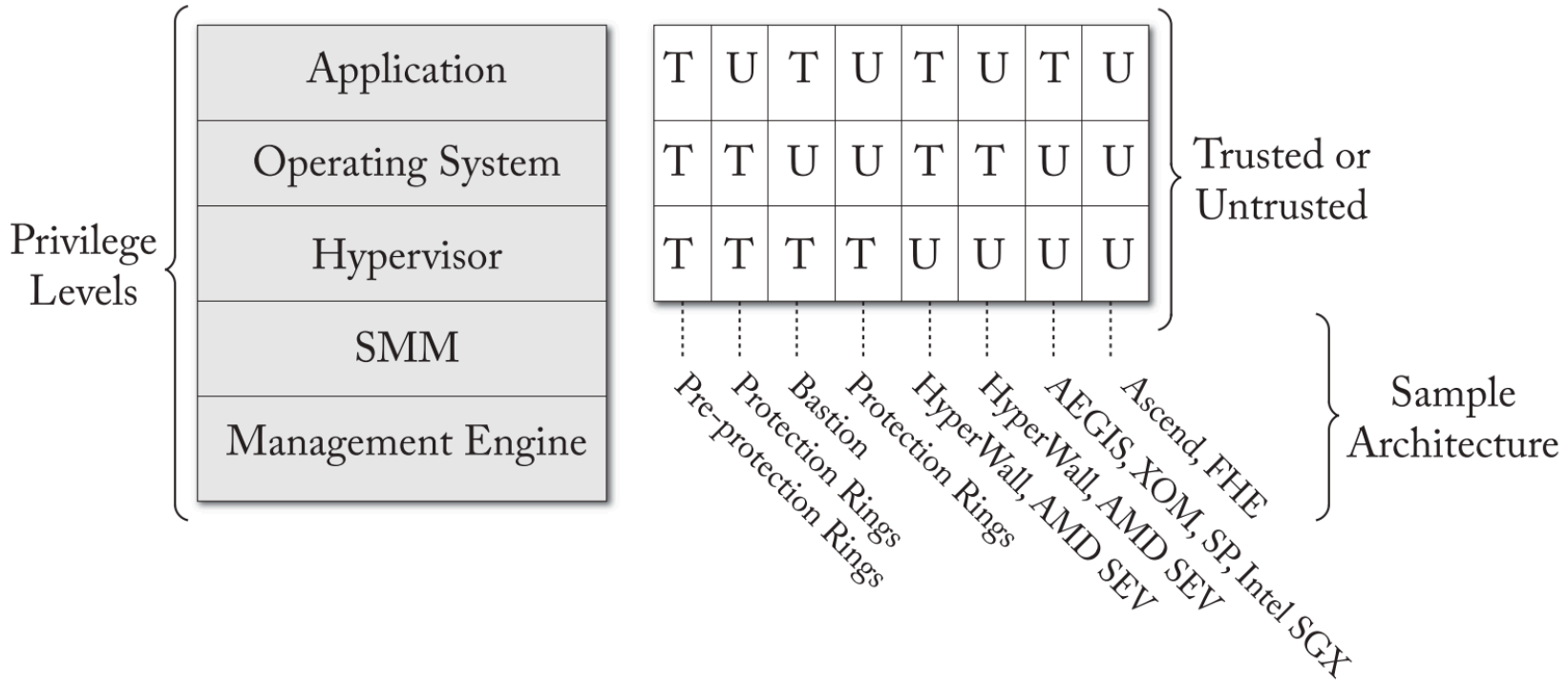
Isolation Barriers

- SMM y SecE extends vertical privilege levels
- Horizontal privilege levels separation can be added also (e.g., ARM trustZone)
- Breaking vertical hierarchy of protection levels
- In secure mode (regardless of the level) software is more privileged than software in normal mode



Architectures for Different Software Threats

- Diversifying the isolation, we can target specific attacks according what is **trusted** or **untrusted** (assuming -2 and -3 is trusted)



Architectures for Different Hardware Threats

- **Multiple chips** connected
  - Susceptible of replacement or physical probing
  - Some accessible (e.g., Memory) some don't (e.g., processor)
  - In secure processor design memory and external wiring is untrusted
    - Modifications will be required to fix it

- **Processor is trusted**
  - Too small to probe (5nm) with a reasonable cost
  - An external bus is orders of magnitude easier to probe

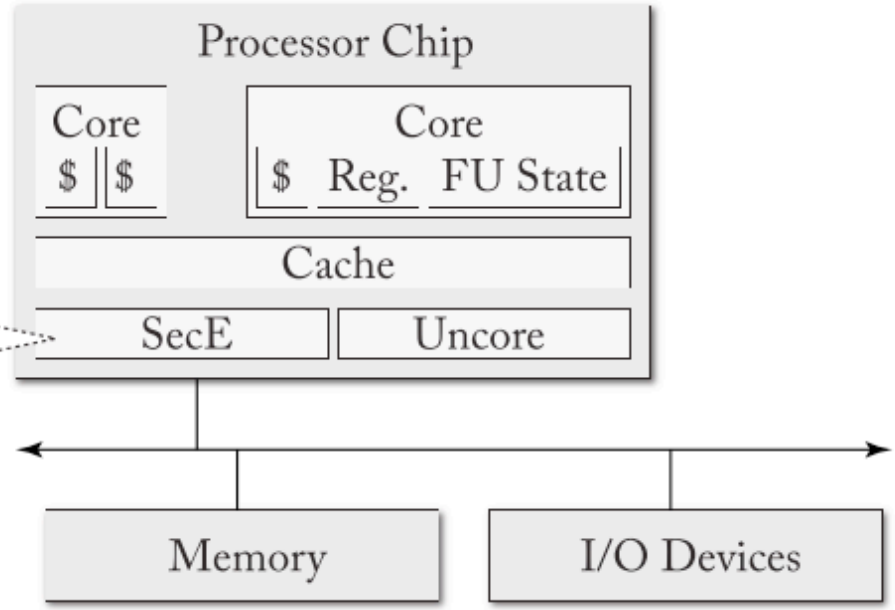
- **3D and 2.5D** can make the system more resilient to hardware threats
  - **Chip-let** designs (fewer external buses and chips in motherboard)
  - Memory integration (e.g., Apple M1)

Hardware Trusted Compute Base

- As custom logic or dedicated processors

Custom logic or hardware state machine:
- Most academic proposals

Code running on dedicated processor:
- Intel ME = ARC processor or Intel Quark processor
- AMD PSP = ARM processor



Examples of Secure Processor Architectures

- Academic
  - XOM, AEGIS, NoHype, …
  - Initially targeting protecting software from hardware attacks (e.g., modification of off-chip memories)
  - Protection against rogue OS added later and currently against rogue hypervisor too
  - Some consider all system potentially rogue and compute without decrypting (e.g., homomorphic cryptography)
  - Most focused in single-core systems

Examples of Secure Processor Architectures

- Commercial
  - 1970 IBM Logical Partitions
  - Reconsidered in 2000s with IBM/Toshiba/Sony Cell Broadband Engine (PS3) (Security processor vault) and follows with ARM TrustZone, **Intel SGX**, **AMD SEV**, …
- The pragmatic approach (processor) is flexible but also a weak point
  - The bugs in the software they run is vulnerable. The approach of security though obscurity amplifies the problem
  - Hardware solutions are too inflexible

Secure Processor Architecture Assumptions

- Chip Assumptions
  - It is the trust boundary for the hardware TCB
  - Everything in the chip is trusted (and untrusted out of it)

- Size TCB Assumption
  - Small software means less bugs, easy to verify and easier to audit
  - Small Hardware, ""

- Open TCB Assumption
  - Apply Kerckhoffs's Principle → no secrets int the TCB: has to be public. The only secret should be the cryptographic keys!

Limitations of Secure Architectures

- Physical Realization Threats
  - Assume the manufacture is correct
  - Hardware Trojans might be added post-design in the foundry
  - Trojan detector can be included in the design too
- Supply Chain Threats
  - Current systems integrates many IP in the design/manufacture phase that can be integrated in late stages of the production
  - Use PUF (Physical Unclonable Functions) to verify that the system is compliant with the specification
- IP Protection and Reverse Engineering
  - Certain component might need to be "non-public"
  - Split-manufacturing (BEOL and FEOL in separate foundries)
- Side and cover attacks
  - Information leak trough unintended channels
- Alternatives to HW: **Full Homomorphic Encryption (FHE)**
  - Perform operation over cyphertext without leaking any information
  - Still not practical: currently very slow. Protects only the data:
  - If FHE is completive, TCB is no longer needed? There is no way to leak information with non-trusted hardware or software