Comp 590-184: Hardware Security and Side-Channels

Lecture 13: Hardware Security Modules

March 3, 2026

Andrew Kwong

THE UNIVERSITY of NORTH CAROLINA at CHAPEL HILL

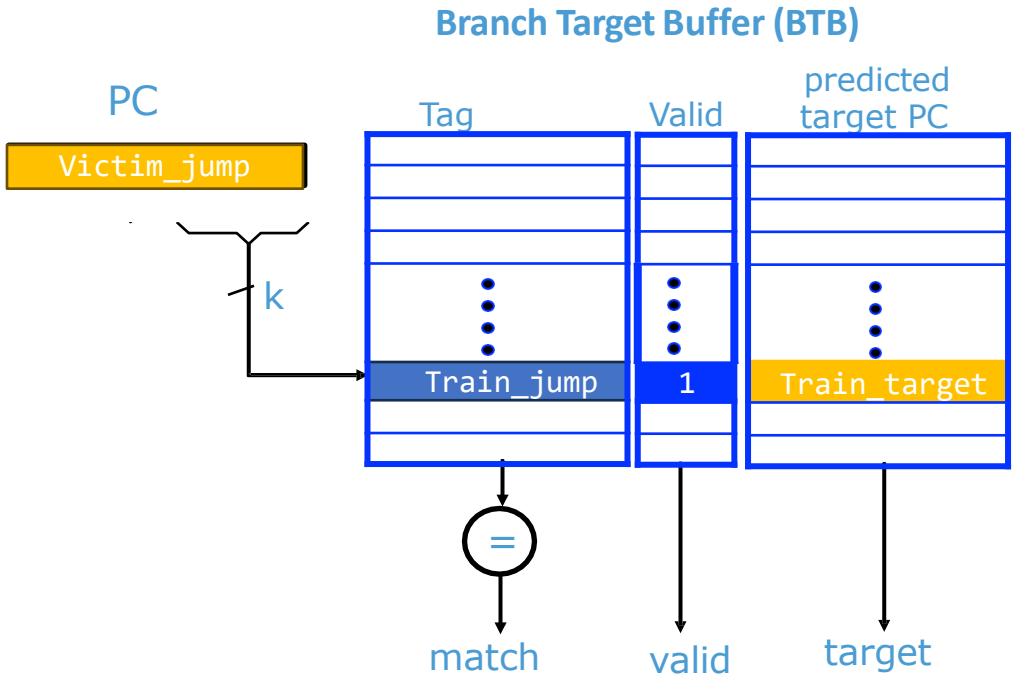
Slides adapted from Mengjia Yan

Agenda

- Continue transient execution mitigations
- Talk about hardware security modules

Recall Spectre v2 (BTB Injection)

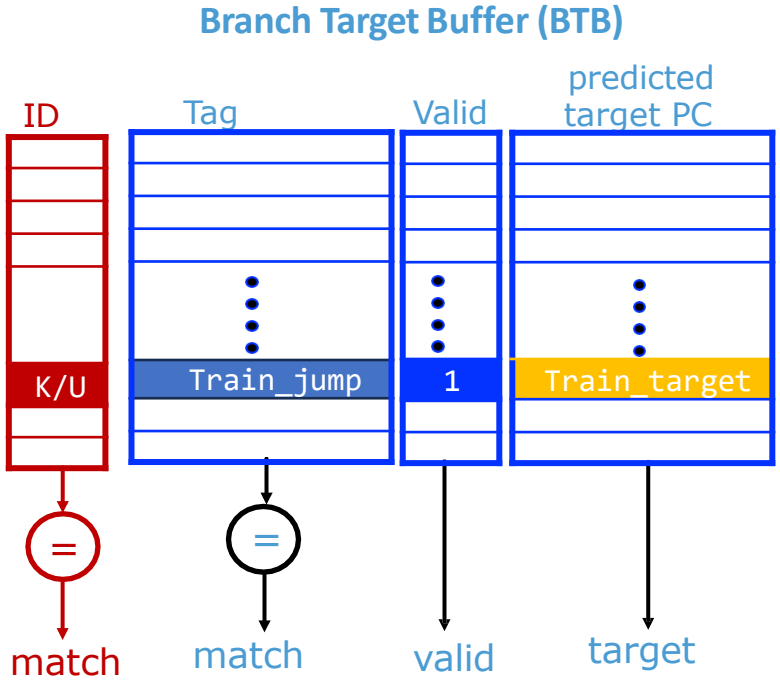
```
; Attacker code
Train_jump:
    jmp Train_target
    …
; ----CONTEXT SWITCH---
; Victim code
Victim_jump:
    jmp rax
    …
Train_target:
    secret = array1[x]
    y = array2[secret*4096]
    …
```



Deployed Hardware Fixes: eIBRS

eIBRS stands for Enhanced Indirect Branch Restricted Speculation

- Intention: isolate BTB entries across privilege levels.



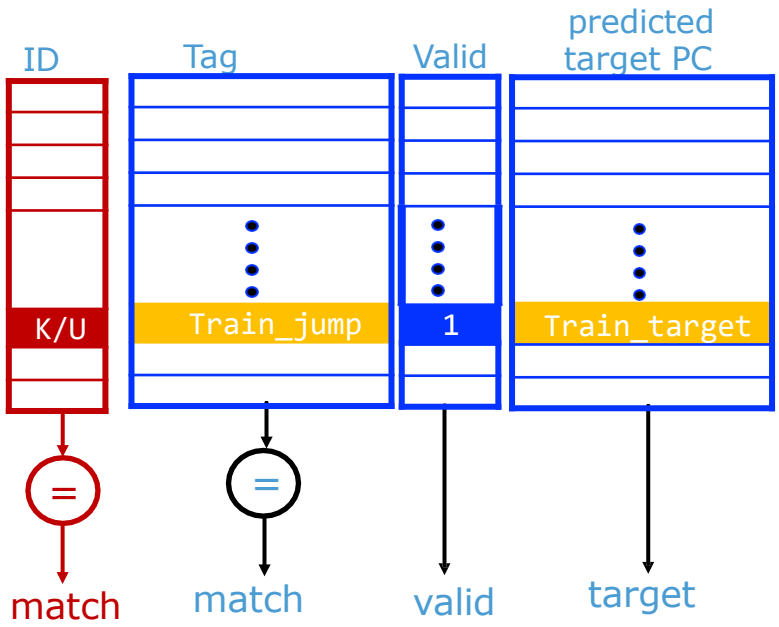
Intel. Indirect Branch Restricted Speculation. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-restricted-speculation.html

Examine the Security Properties

What do we mean by isolation?

- Property #1:
  - Kernelspace indirect branches **do not use** branch target inserted by userspace code.
- Property #2:
  - Userspace code **does not interfere** with Kernelspace indirect branch predictions.

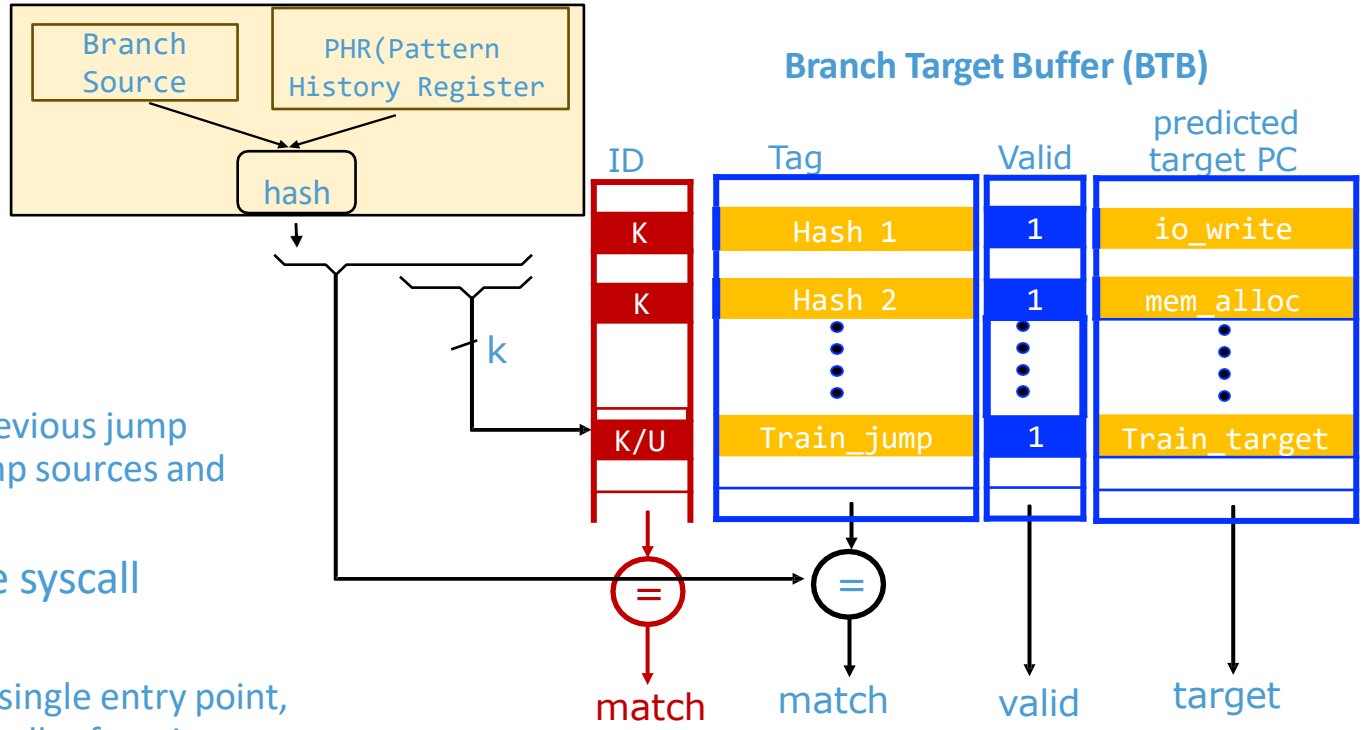
Does eIBRS achieve property #2? If not, counterexamples?



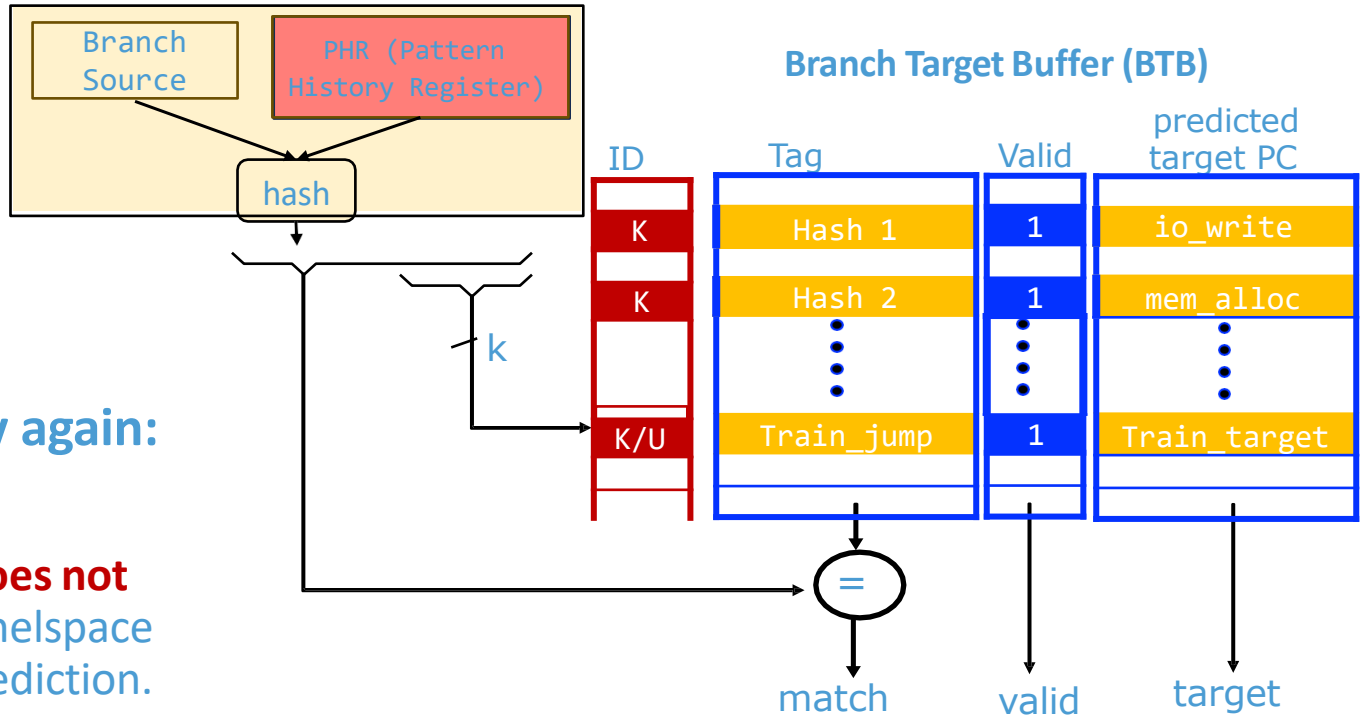
Same-mode misprediction

Surprise 1: How Does BTB Actually Work?

- PHR
  - History information of previous jump instruction, including jump sources and targets
- How does PHR improve syscall performance?
  - E.g., System calls share a single entry point, but will jump to many handler functions



Branch History Injection



Look at the property again:

- Property #2:
  - Userspace code **does not interfere** with Kernelspace indirect branch prediction.

Bypassing eIBRS

- Attacker controls PHR
- Can redirect control flow to any legitimate branch in the kernel
- Exploitable?

Exploit

- EBPF
  - User uploads hardened, verified code into the kernel
    - Accessed via indirect jump!
      - Normally operates on bpf_socket struct pointer passed via rdi
      - Speculatively operates on stack-saved user registers referenced by rdi
  - Attacker uploads an EBPF module that is a disclosure gadget and a legitimate branch target in the kernel

Surprise 2: Consequences due to Retpoline

Before retpoline	`jmp *%rax`

After retpoline	`1. call load_label`
	
	`2.capture_ret_spec:`
	`3. pause ; LFENCE`
	`4. jmp capture_ret_spec`
	
	`5.load_label:`
	`6. mov %rax, (%rsp)`
	`7. RET`

Listing 3 Linux implementation for the Spectre v2 mitigation before version 5.14 on Intel processors depending on eIBRS hardware support. The shown example is taken from the indirect jump in charge to execute the correct syscall handler stored in the `sys_call_table`.

```
1 do_syscall_64:
2     ;...
3     mov     rax, [sys_call_table + rax*8]
4     call    __x86_indirect_thunk_rax
```

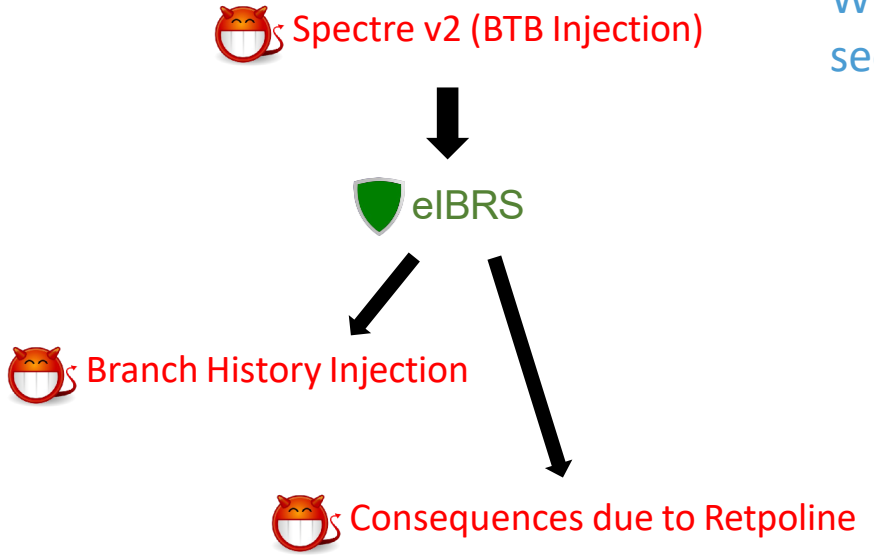
```
1 ;with eIBRS support
2 __x86_indirect_thunk_rax:
3     jmp     rax
```

Perfect victim branch for BTB attack

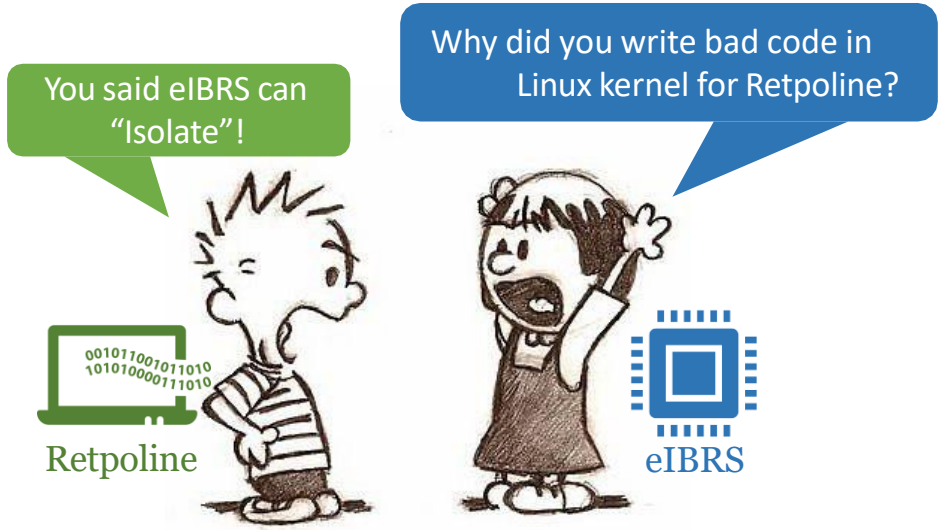
```
1 ;without eIBRS support (retpoline)
2 __x86_indirect_thunk_rax:
3     call    B
4 A:  pause
5     lfence
6     jmp     A
7 B:  mov     [rsp], rax
8     ret
```

Google. Retpoline: a software construct for preventing branch-target-injection https://support.google.com/faqs/answer/7625886

Summary: The Cat-and-Mouse Game



Why did hardware designers fail to make eIBRS secure?



Solution

- Goal: communicate security property achieved by hardware defenses
  - The bad example: eIBRS -> unclear what exactly “isolation” mean…
- Alternative approaches:
  - Approach 1: Show SW people all the HW implementation details

  - Approach 2: define new SW-HW contracts

SW-HW Contracts for Secure Speculation

Contract #1: Make Speculation Invisible

- Idea: make speculative executed instructions' microarchitecture effects invisible by the attacker
- Examine program examples

```
if (false)
  sec = ld x
  dummy = ld sec
```

```
sec = ld x

if (false)
  dummy = ld sec
```

```
sec = ld x
dummy = ld sec

if (false)
  ……
```

Secure if using invisible speculation?

Do they follow constant-time programming?

Speculative Non-interference

- New contract for security
- Property:
  - **if** the SW does not leak under the constant-time programming model
  - **then** the HW should ensure no secrets leaked under speculation

Speculative Non-interference

- Some notations
  - P: a deterministic program
  - M_{pub}: public memory and inputs
  - M_{sec}: secret memory and inputs
  - O: microarchitecture observation (traces)
- Property:
  - **if** the SW does not leak under the constant-time programming model
  - **then** the HW should ensure no more secrets leaked under speculation

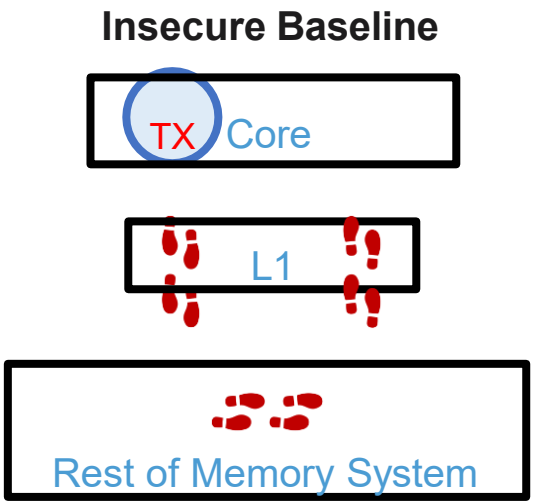
$$\forall\ P, M_{pub}, M_{sec}, M'_{sec},$$
$$\text{IF}\quad O_{seq}(P, M_{pub}, M_{sec}) = O_{seq}(P, M_{pub}, M'_{sec})$$
$$\text{THEN}\quad O_{spec}(P, M_{pub}, M_{sec}) = O_{spec}(P, M_{pub}, M'_{sec})$$

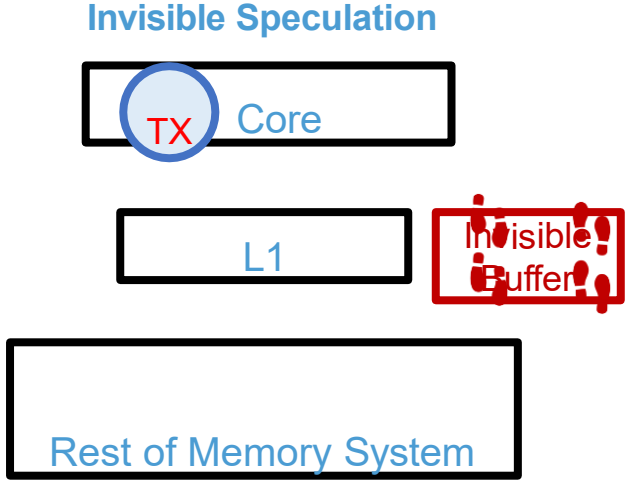
Execute program **sequentially**, monitor memory addresses.

Execute program **speculatively**, monitor memory addresses.

Defense #1: InvisiSpec

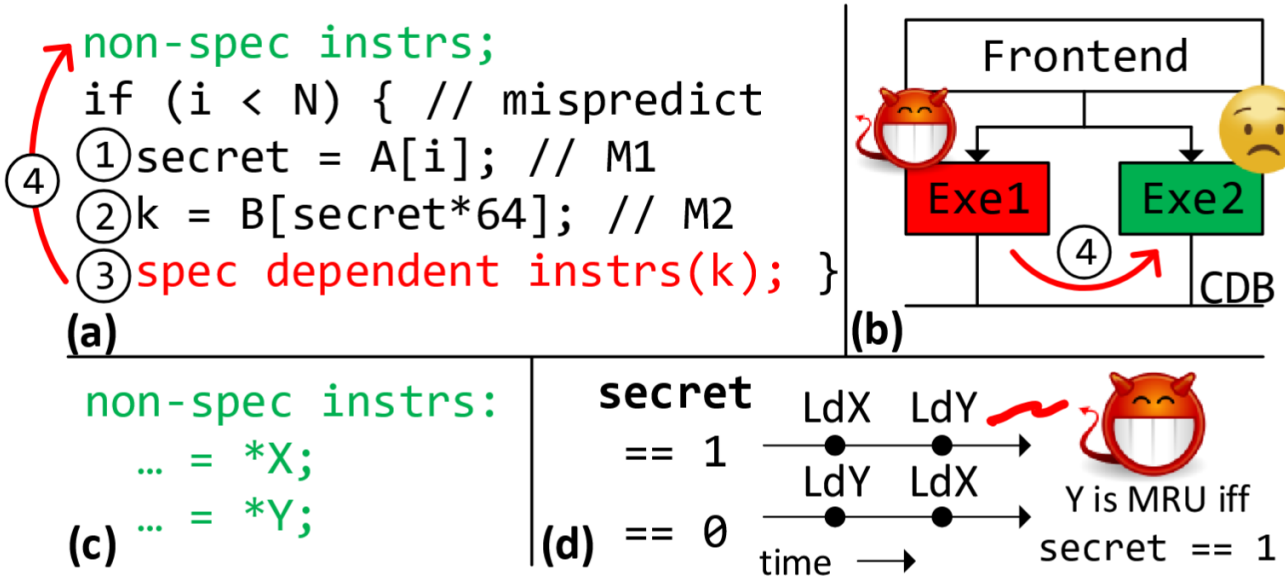
```
if (false)
  dummy = ld sec //Tx
```





Speculative interference attack

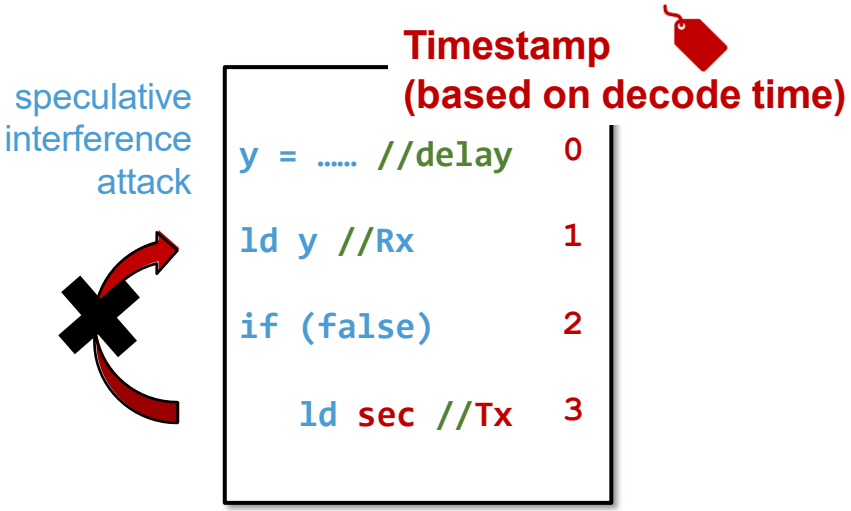
- 2 ideas
  - mis-speculated younger instructions can change the timing of older, bound-to-retire instructions, including memory operations.
  - changing the timing of a memory operation can change the order of that memory operation relative to other memory operations, resulting in persistent changes to the cache state

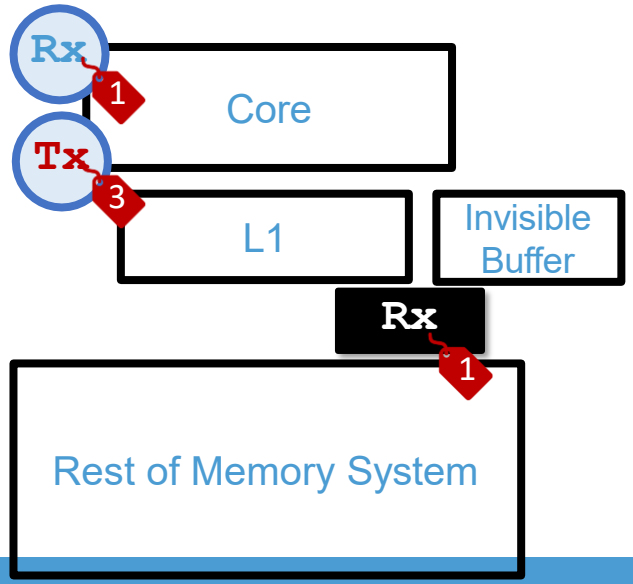

non-spec instrs;
if (i < N) { // mispredict
1 secret = A[i]; // M1
2 k = B[secret*64]; // M2
3 spec dependent instrs(k); }
4
(a)
Frontend
Exe1
Exe2
4
CDB
(b)
non-spec instrs:
… = *X;
… = *Y;
(c)
secret
== 1
== 0
(d)
LdX LdY
LdY LdX
time
Y is MRU iff
secret == 1

Defense #2: GhostMinion

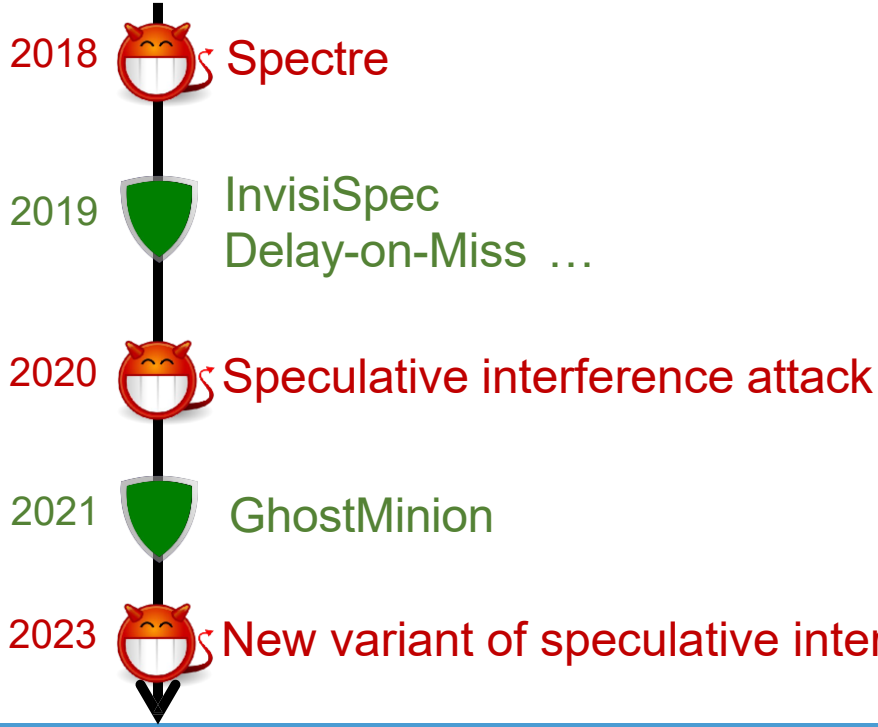
#1: Invisible Speculation

#2: Prioritize Older Instructions through Timestamps





Summary: The Cat-and-Mouse Game



Summary of SW-HW Contracts

$$\forall\ P, M_{pub}, M_{sec}, M'_{sec},$$
$$\text{IF}\quad O_{seq}(P, M_{pub}, M_{sec}) = O_{seq}(P, M_{pub}, M'_{sec})$$
$$\text{THEN}\quad O_{spec}(P, M_{pub}, M_{sec}) = O_{spec}(P, M_{pub}, M'_{sec})$$

Describe what SW needs to achieve

Describe what HW needs to achieve for **only** the SW that satisfies the IF statement

- The payoff: we can check security properties for SW and HW independently

Hardware Security Modules

- Hardware can offer security primitives we cannot achieve with only software

Secure Processors/HSM

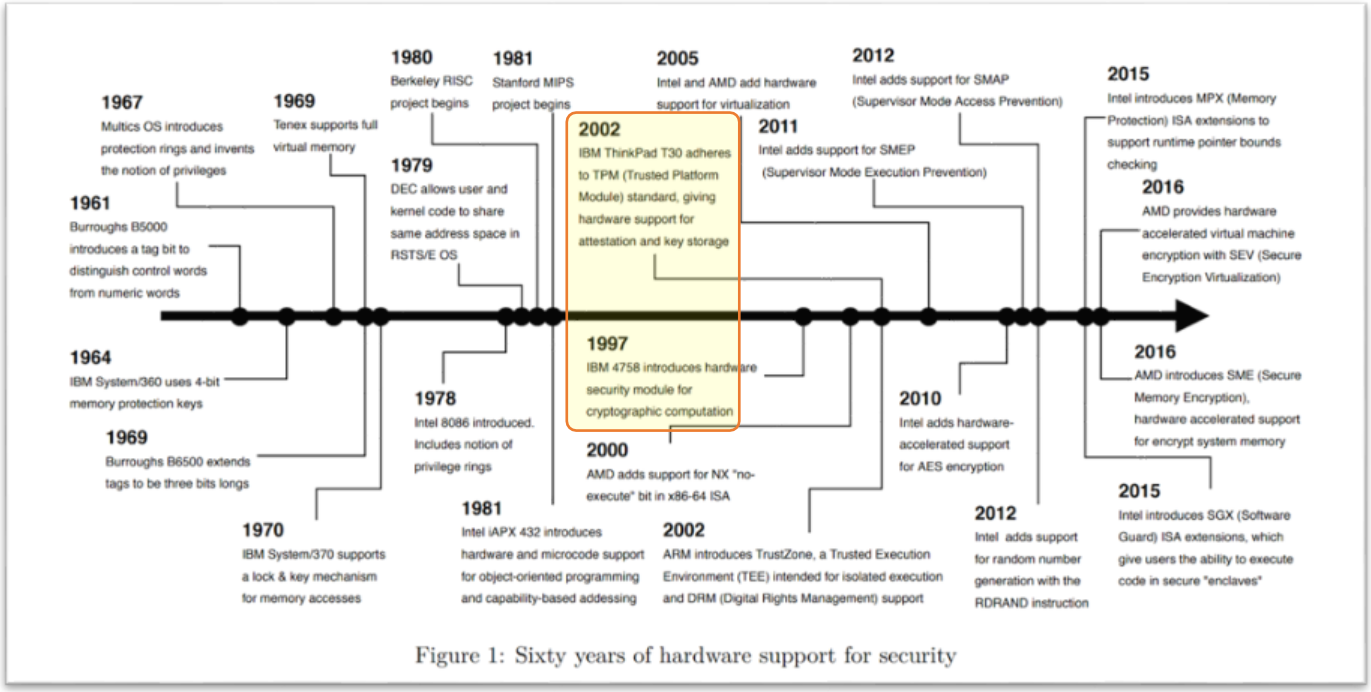
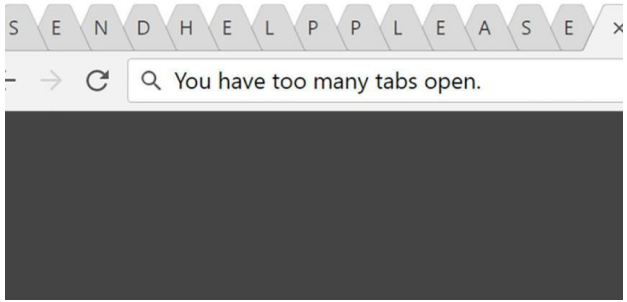
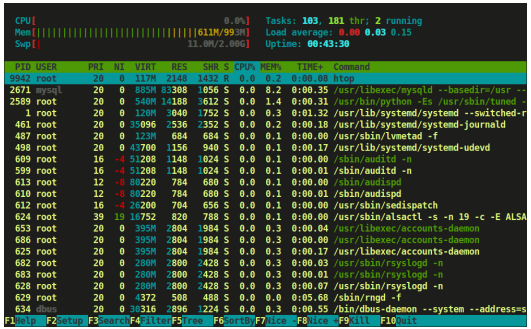


Figure 1: Sixty years of hardware support for security

Introduction to Security for Computer Architecture Students. Adam Hastings, Mohammed Tarek, Simha Sethumadhavan.
https://www.cs.columbia.edu/~simha/ch1_supplement.pdf

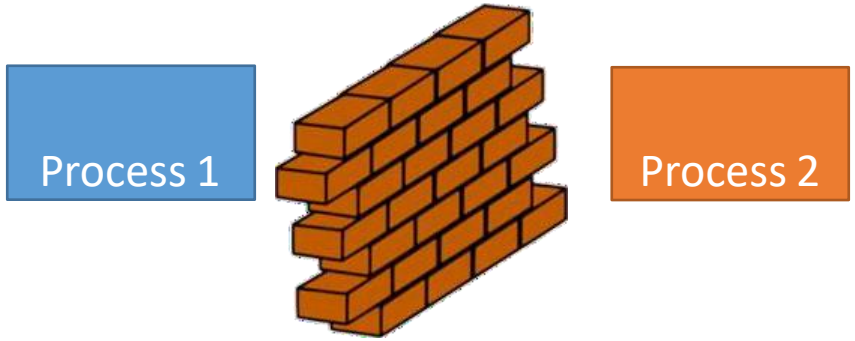
Security Context #1



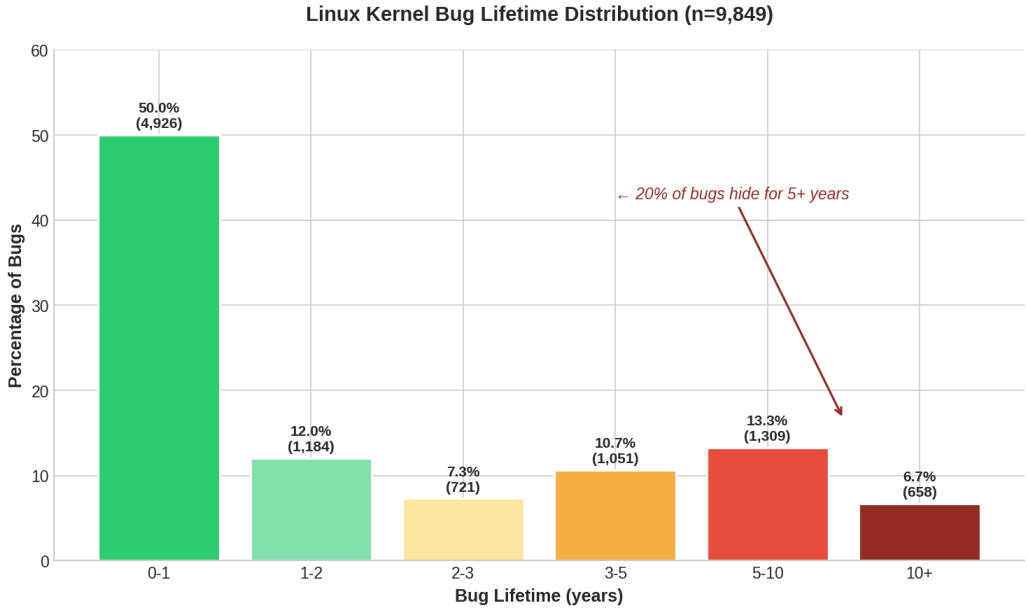


- Problems:
  - Running random applications together with security-sensitive applications
  - Software can be buggy (or sometimes malicious)

Isolation

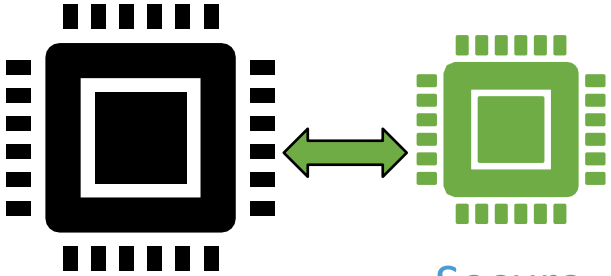


OS/Hypervisor enforced isolation

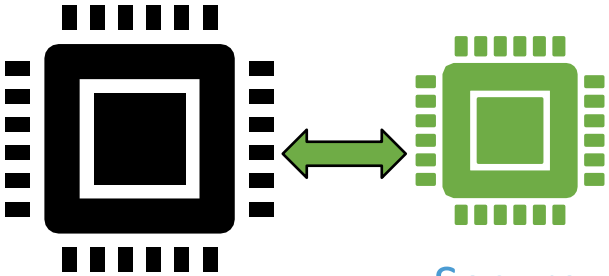


Can we do better than software-based isolation?

Physical Isolation



Secure Co-Processors

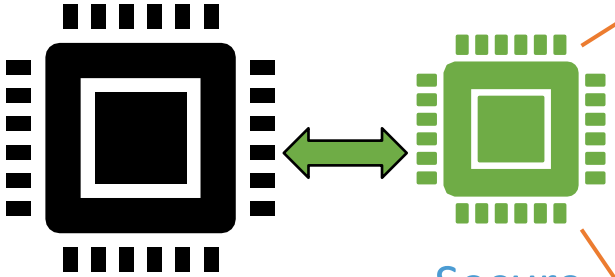


- Before IBM 4758 (1999):
  - Crypto accelerators (AES, RSA, etc.)
  - Store crypto keys inside the accelerator
  - Want to run more applications on the co-processor
- IBM 4758 (1999) -- 4765 (2012)
  - **Programmable** secure co-processor
  - Idea: create a virtual locker room

Secure Co-Processors

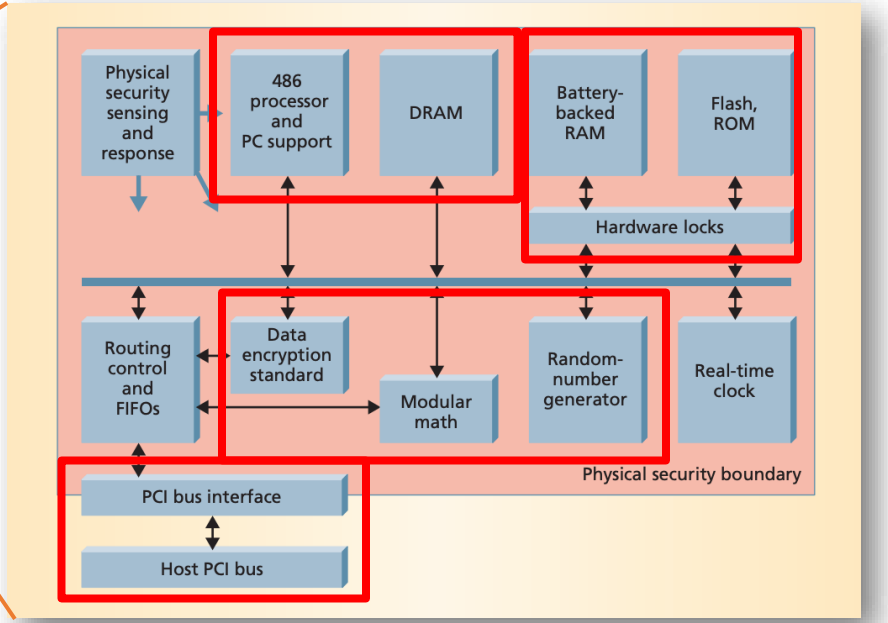
General-purpose processor, rather than ASIC, with isolated DRAM.

Hardware lock, resilient against physical attacks to modify firmware



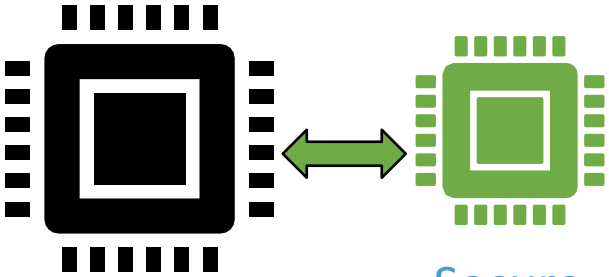
Normal Processor

Secure Processor



Narrow interface, only interact with external worlds via APIs
(keys do not leave the co-processor)

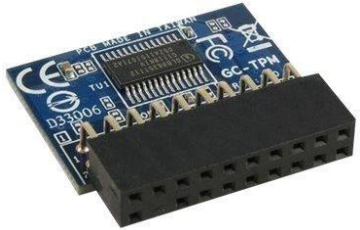
Secure Co-Processors

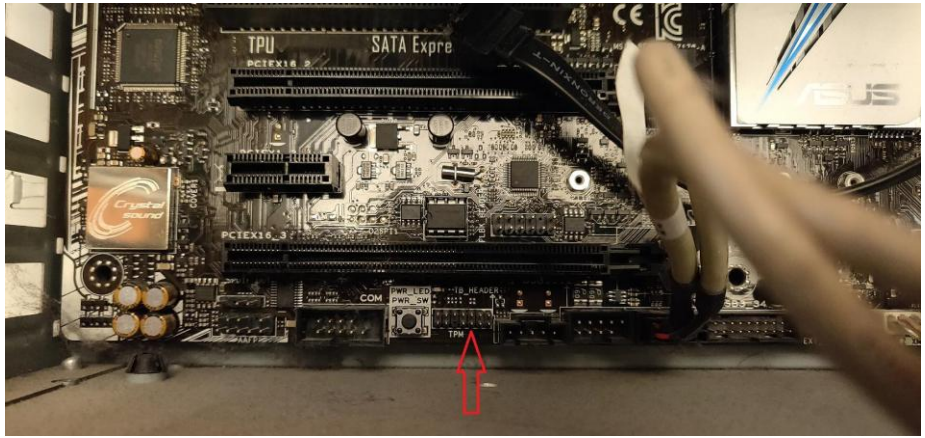


- Before IBM 4758 (1999):
  - Crypto accelerators (AES, RSA, etc.)
  - Store crypto keys inside the accelerator
  - Want to run more applications on the co-processor
- IBM 4758 (1999) -- 4765 (2012)
  - **Programmable** secure co-processor
  - Idea: create a virtual locker room
  - Problem?
  - The SOFTWARE! Bad programmability.
  - Need to find a middle ground: run selected applications that offer strong security functionality

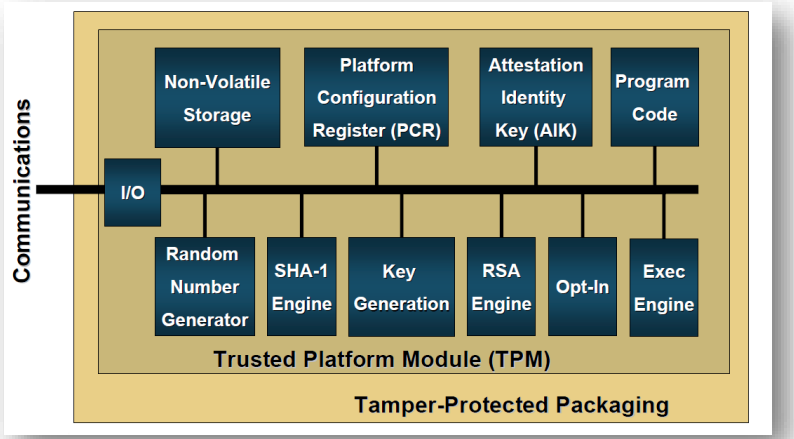
Trusted Platform Module (TPM)

- "*Commoditized* IBM 4758": Standard LPC interface attaches to commodity motherboards



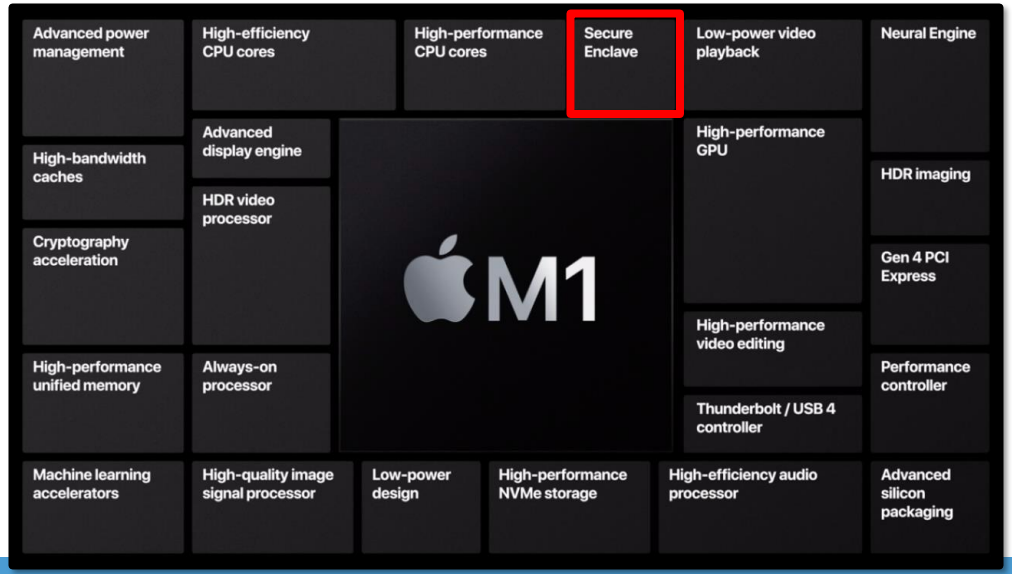


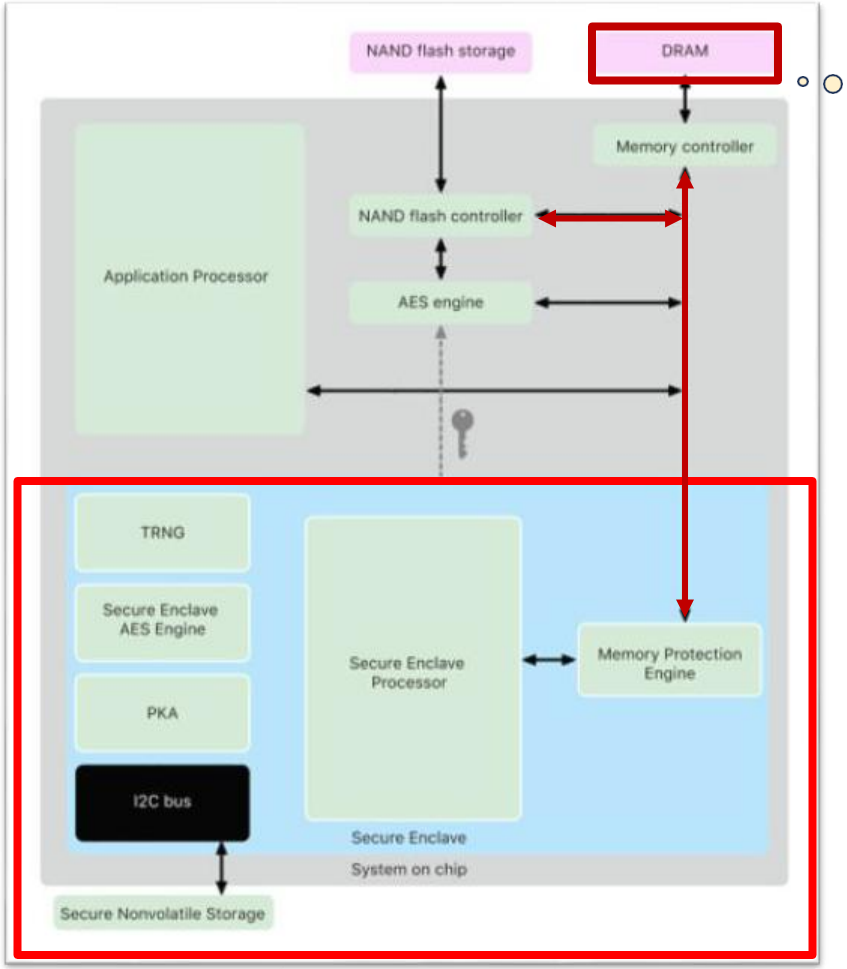
https://scotthelme.co.uk/upgrading-my-pc-with-a-tpm/



Apple Secure Enclave

- Advantage: one company controls both the hardware and the software
- Apple secure enclave runs a customized formally verified micro-kernel OS



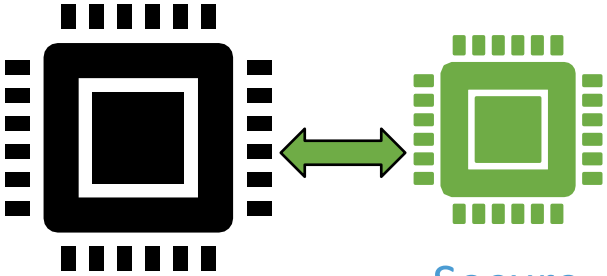


Encrypt enclave data and only decrypt at the memory protection engine

- Only run secure enclave functionality, no user code
- Block vulnerabilities due to software bugs (running L4 microkernel)
- Block uarch side channels

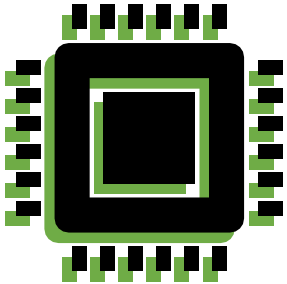
From Apple Platform Security (Page 5-96)

Make Physical Isolation More Flexible?



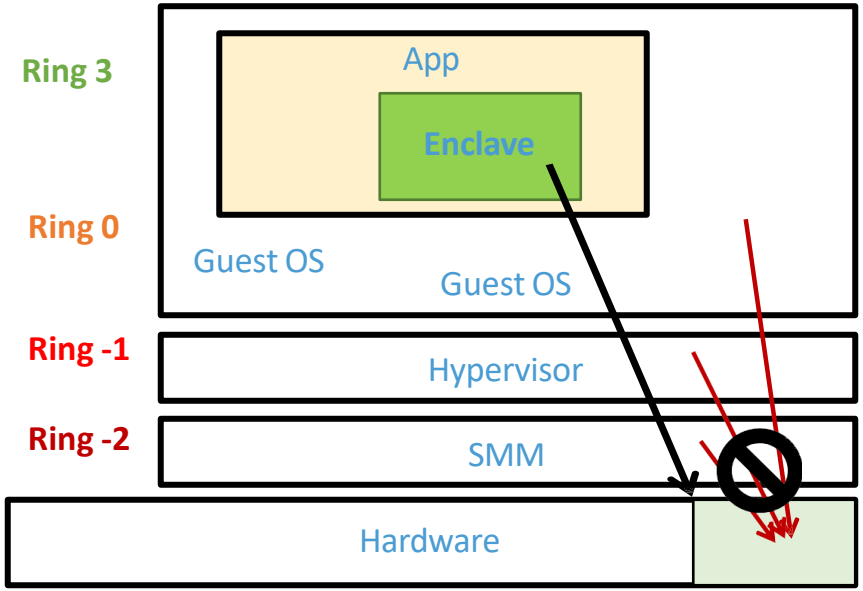
Normal Processor

Secure Processor

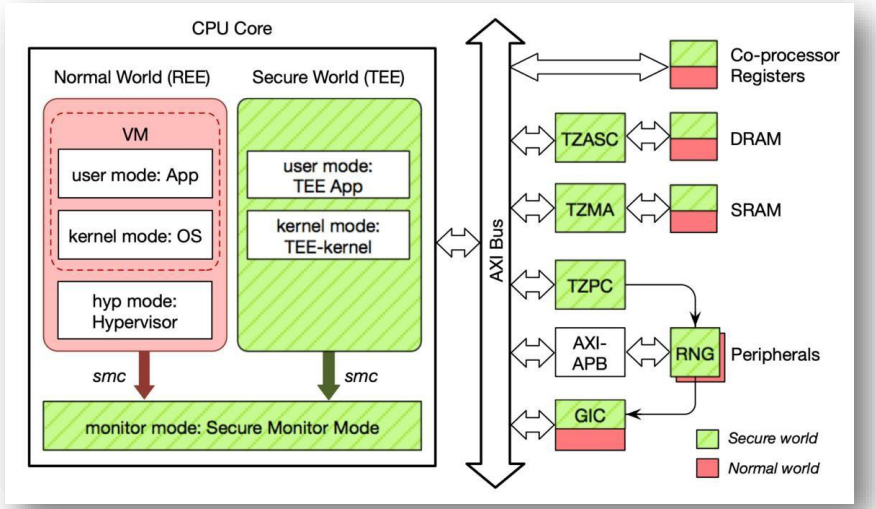


A Processor can switch between Normal and Secure Modes

The Trends (isolation with some sharing?)



Intel SGX model



ARM TrustZone

Security?

Usability?

Fixed Design (Static)

Flexible Design (Dynamic)