Trusted Execution Environments (TEEs)

Shweta Shinde

Motivating Example

Input

Bank's Web Server

Private Key

Signature

Sensitive

What can go wrong: Attack Surface

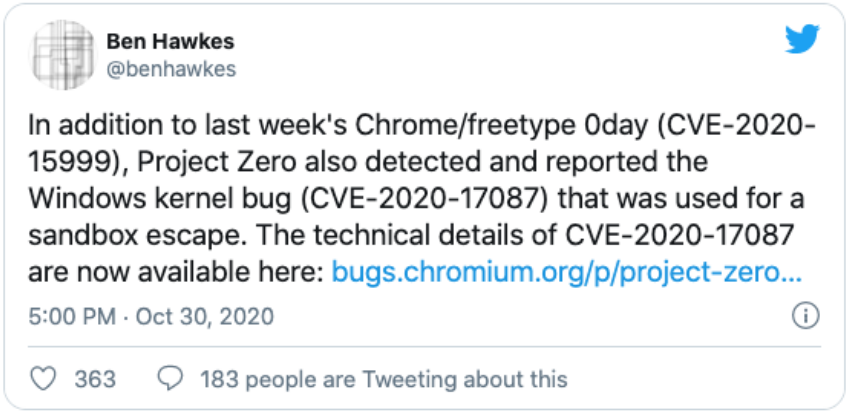
- Signature Verification
  - bad crypto (design / implementation)
- Webserver <-> client
  - broken communication channel
  - MITM, broken protocol (design/implementation)
- Web server
  - software bugs
- Browser
  - software bugs
- Other avenues of attack?

Google patches two new zero-day flaws in Chrome

The last three weeks have seen a bumper crop of patches for zero-day bugs across software from Google, Apple and Microsoft

Amer Owaida

12 Nov 2020 - 05:25PM



Windows kernel bug (CVE-2020-17087) that was used for a sandbox escape. The technical details of CVE-2020-17087

Team IDs Real-world Vulnerabilities In Popular Browser During Premier Hackathon

Monday, April 6, 2020

WELCOME TO PWN2OWN 2020 - THE SCHEDULE AND LIVE RESULTS

March 18, 2020 | Dustin Childs

Welcome to Pwn2Own 2020! This year's contest will be the first where all attempts occur remotely. We have contestants from around the world ready to demonstrate some amazing research, and we have ZDI researchers ready to run and verify their attempts. Our returning partner Microsoft and sponsor VMware will also be online with us, as well as other affected vendors. We might not be in the same room, but we're virtually together and ready for a fantastic contest.

The schedule for today is posted below, and we'll be updating this blog throughout the day with results and updates. We'll post a full summary of today's events (including videos of the attempts) tomorrow morning.

As always, we started the contest with a random drawing to determine the order of attempts. We have four attempts scheduled for today and four queued up for tomorrow. The full schedule for Day One is below (all times Pacific [UTC -7:00]). We will update this schedule with results as they become available.

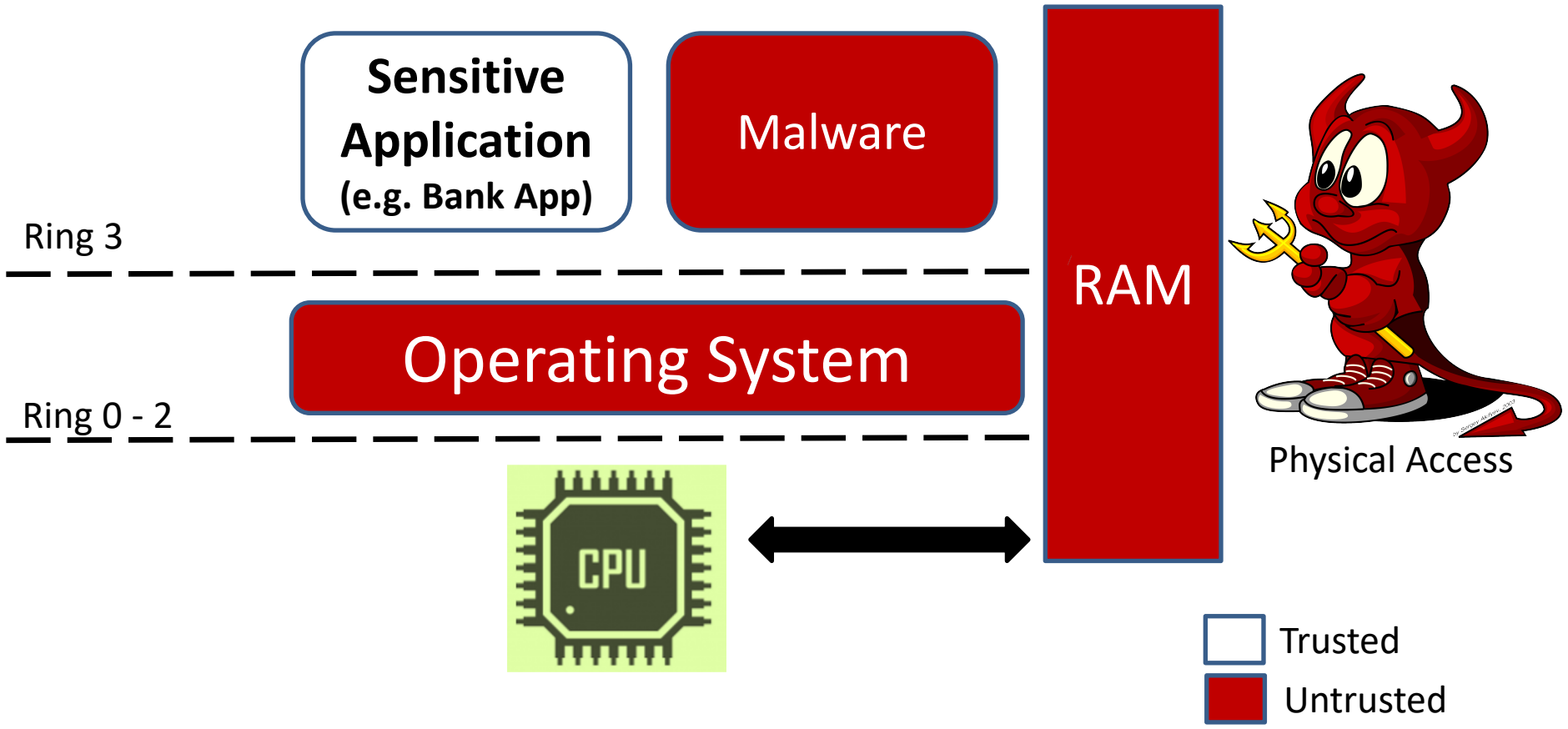
Day One – March 18, 2020

1000 - The Georgia Tech Systems Software & Security Lab (@SSLab_Gatech) team of Yong Hwi Jin (@jinmo123), Jungwon Lim (@setuid0x0_), and Insu Yun (@insu_yun_en) targeting Apple Safari with a macOS kernel escalation of privilege.

Apple Safari with a macOS kernel escalation of privilege.

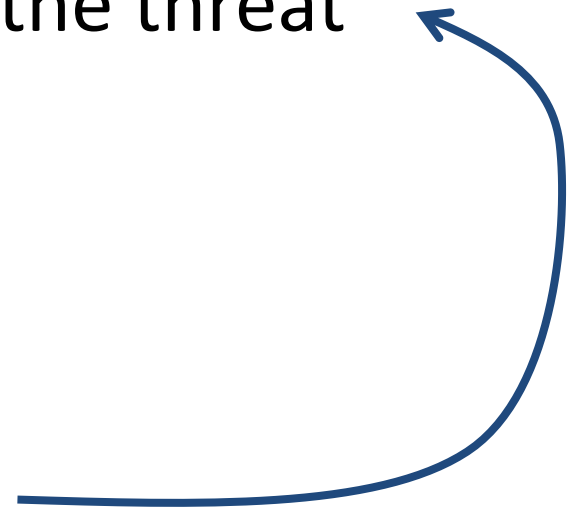
A Much Larger Attack Surface

- Threats to your bank application



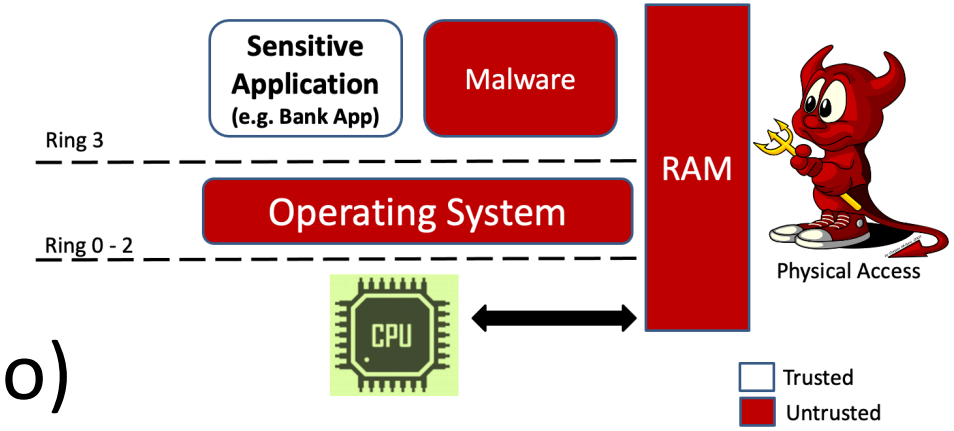
This Class: the good, the bad, the ugly

- Look at a new threat model and its ramifications
- Learn how to build systematic defenses against the threat
- Understand the gaps in our defenses
- Leverage the gaps to circumvent the defenses



Possible Defenses

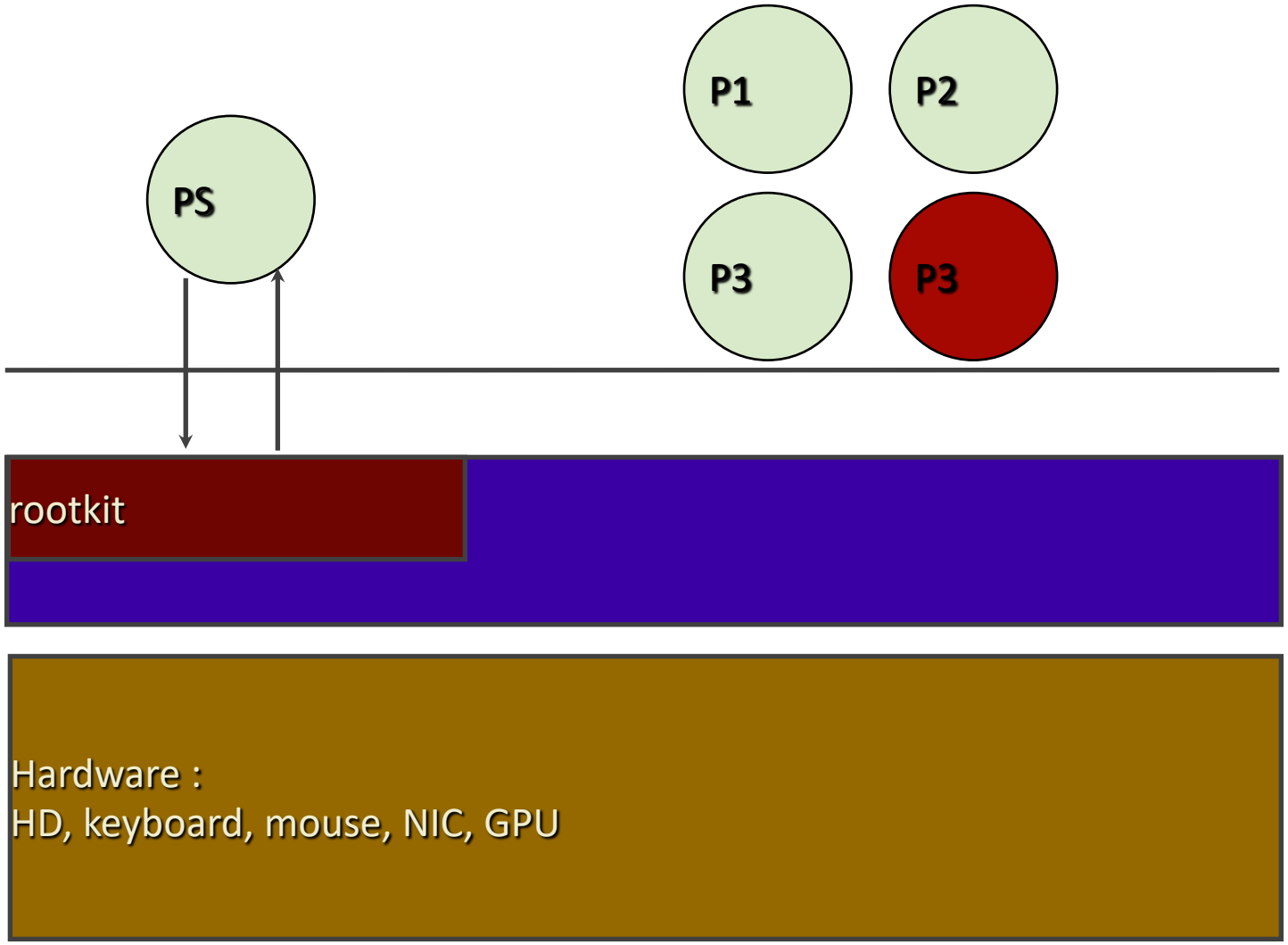
- Binary analysis (e.g., Fuzzing)
- Binary hardening (e.g., CFI, SFI)
- Memory safe languages (e.g., Rust, Go)
- Verification (e.g., seL4)
- Encrypted computation (e.g., homomorphic encryption)
  - How to protect the private keys?
- Virtualization (e.g., XEN)
  - Assumes hypervisor is uncompromised



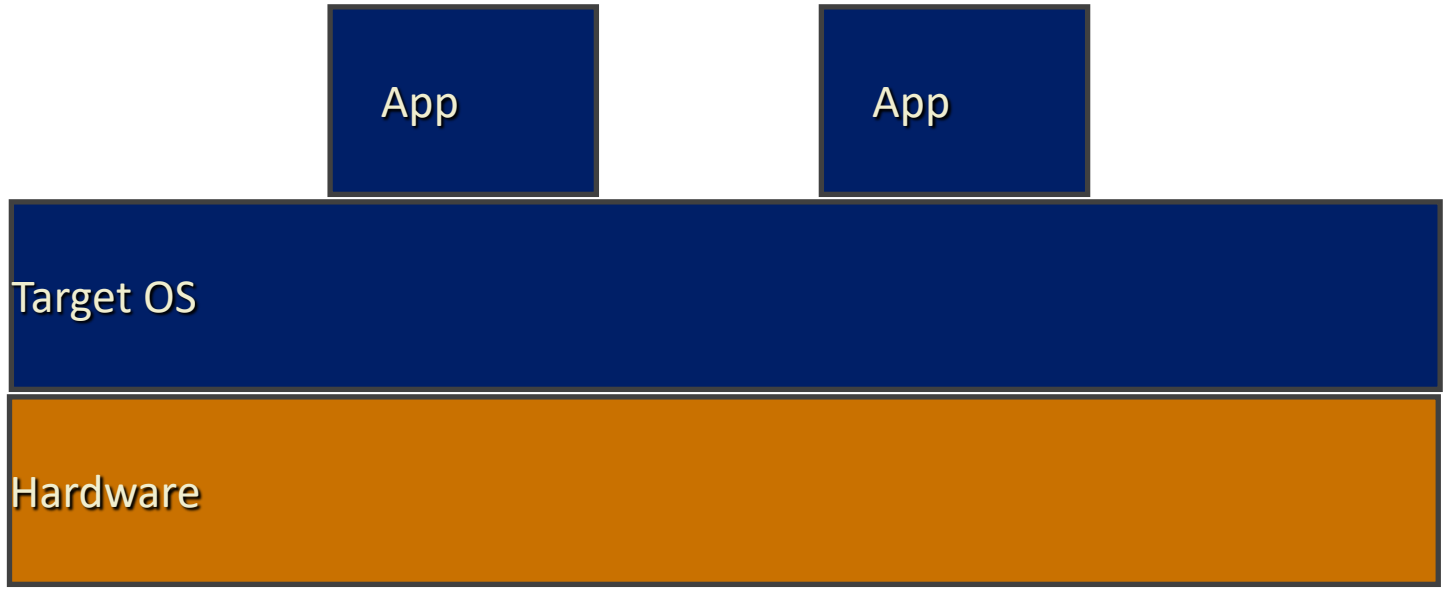
These approaches are not scalable and/or complete

How can one compromise the confidentiality & integrity of sensitive computation?

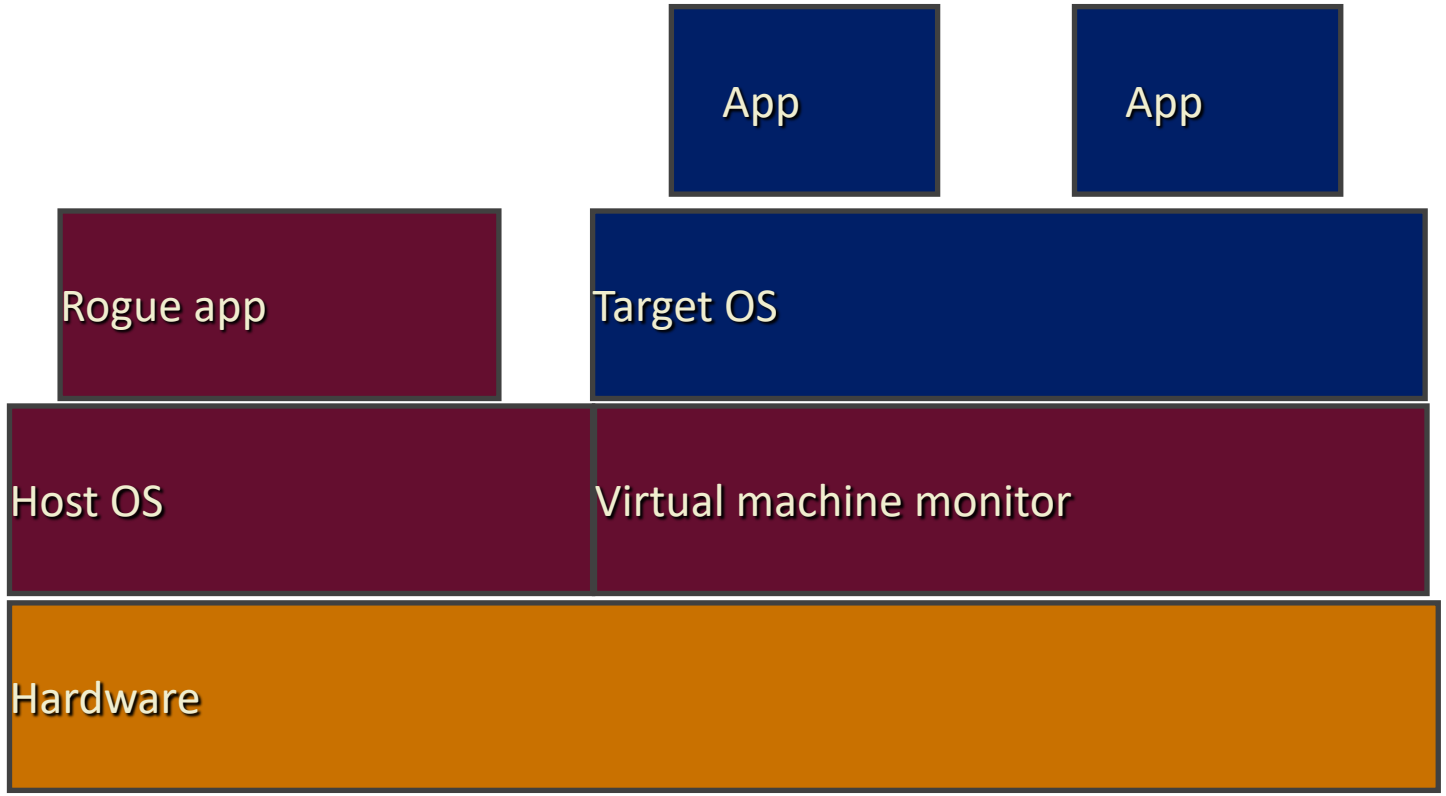
Kernel rootkit



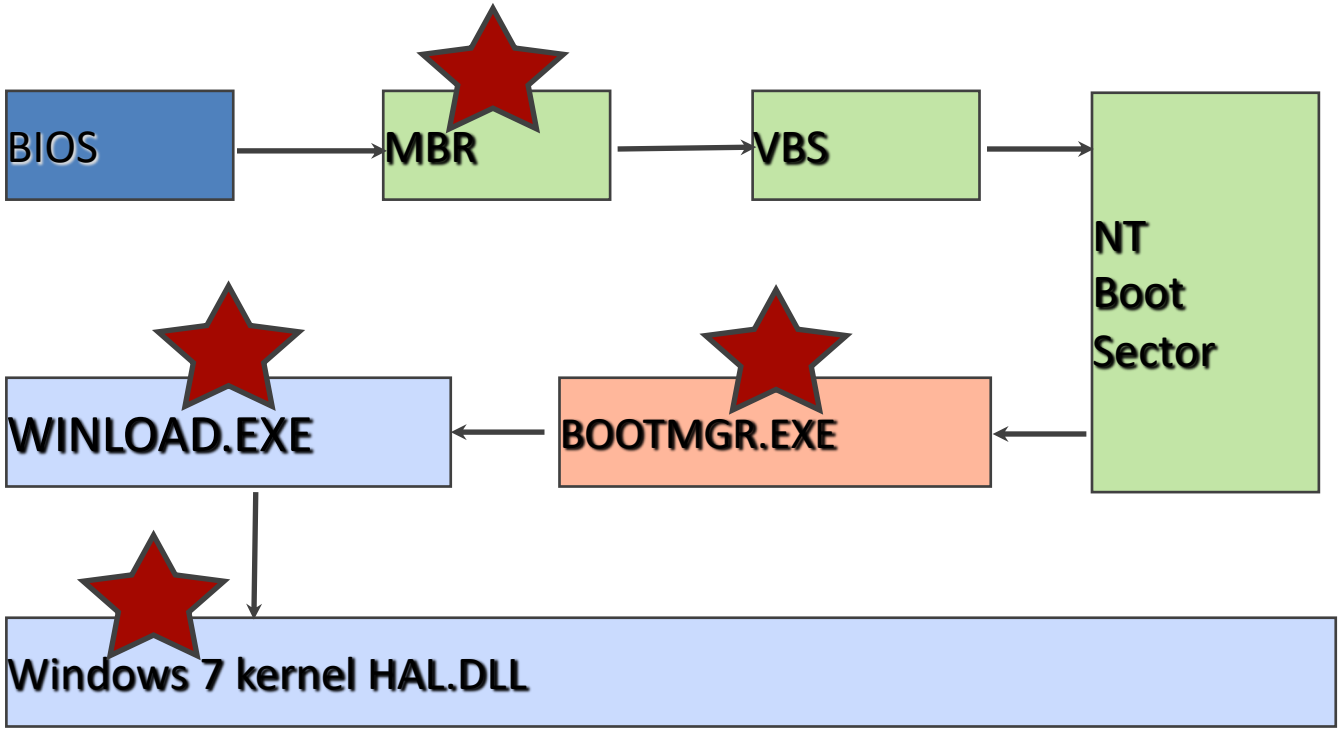
Hypervisor rootkit



Hypervisor rootkit



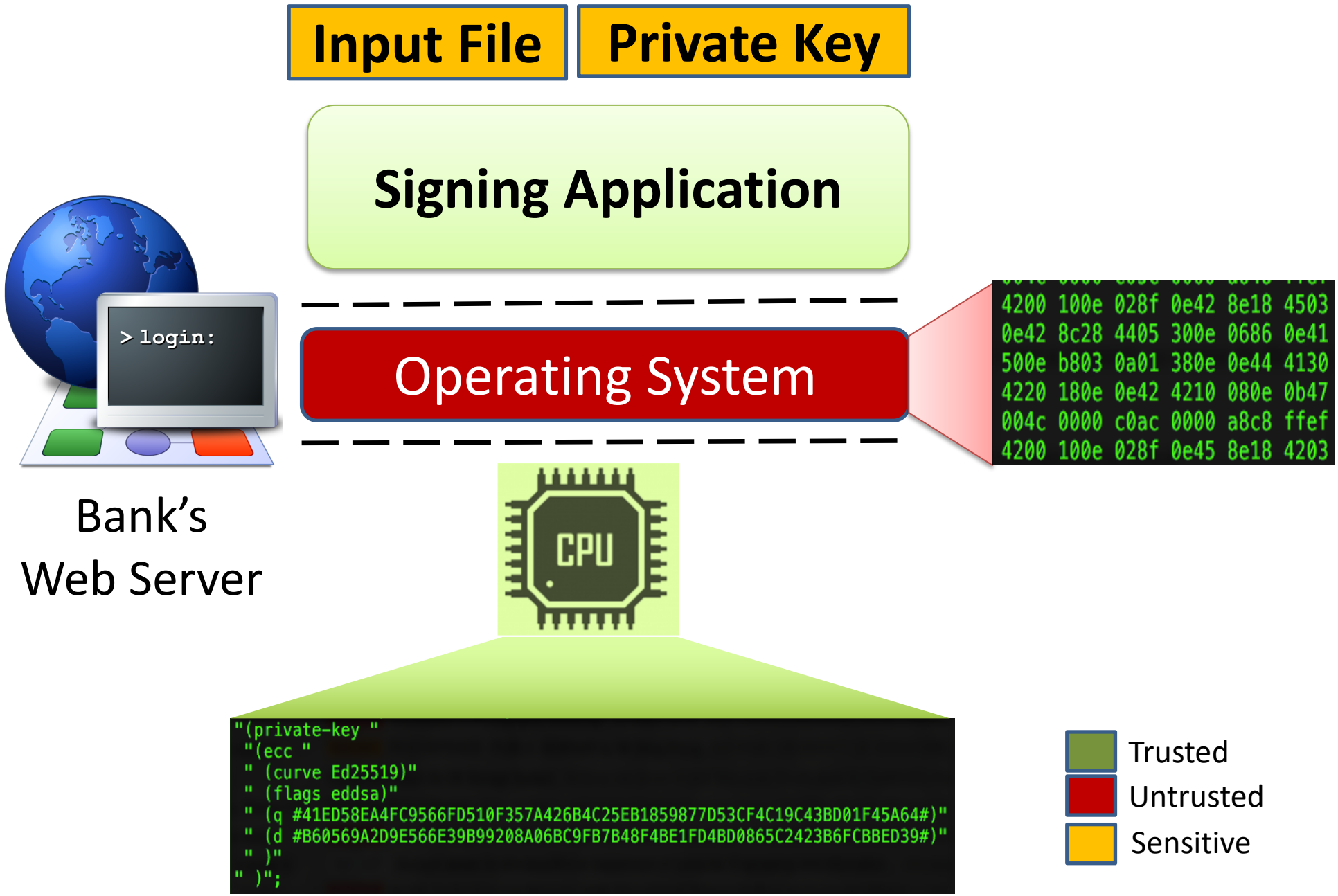
MBR/Bootkit


BIOS
MBR
VBS
NT
Boot
Sector
WINLOAD.EXE
BOOTMGR.EXE
Windows 7 kernel HAL.DLL

All software is susceptible to attacks

How to run computation on data with confidentiality and integrity?

Make hardware the root of trust
and
imagine a world where …


Input File
Private Key
Signing Application
> login:
Operating System
4200 100e 028f 0e42 8e18 4503
0e42 8c28 4405 300e 0686 0e41
500e b803 0a01 380e 0e44 4130
4220 180e 0e42 4210 080e 0b47
004c 0000 c0ac 0000 a8c8 ffef
4200 100e 028f 0e45 8e18 4203
Bank's
Web Server
CPU
"(private-key "
"(ecc "
" (curve Ed25519)"
" (flags eddsa)"
" (q #41ED58EA4FC9566FD510F357A426B4C25EB1859877D53CF4C19C43BD01F45A64#)"
" (d #B60569A2D9E566E39B99208A06BC9FB7B48F4BE1FD4BD0865C2423B6FCBBED39#)"
")"
")";
Trusted
Untrusted
Sensitive

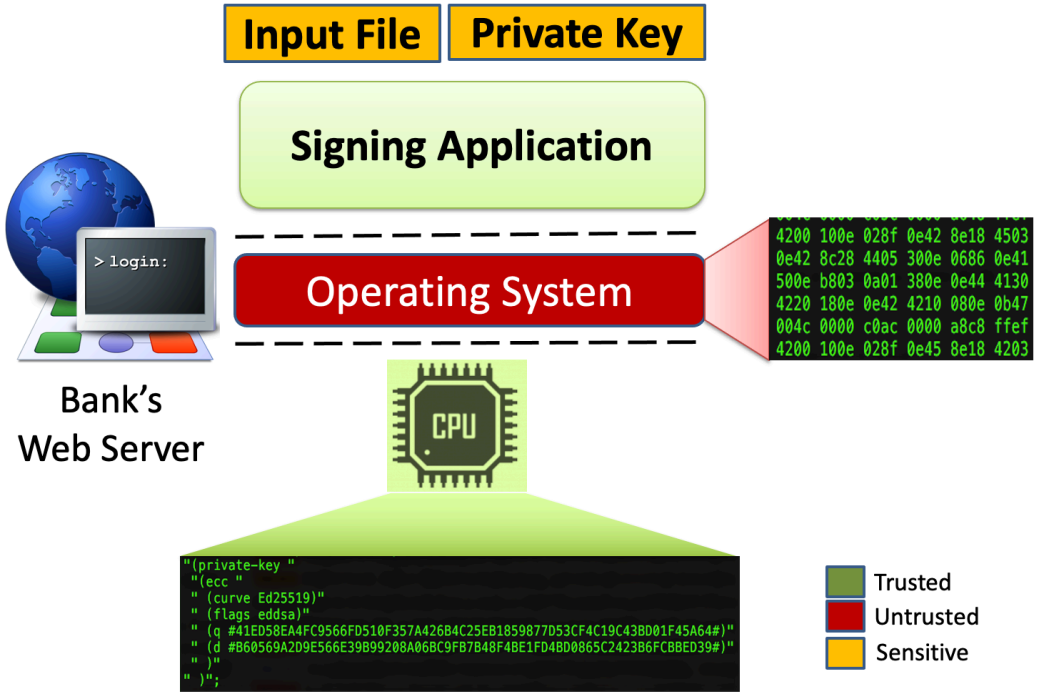
Hurdles in achieving this dream

- How to bootstrap trust?
  - Ensure that the CPU is genuine (e.g., Intel Inside)
  - Ensure that we are running on a real machine (vs virtualized)

- How and where to store secrets if everything is malicious?
- What about attacks during execution?

- How to load an application if the OS is malicious?
- What about OS services?

Break for questions



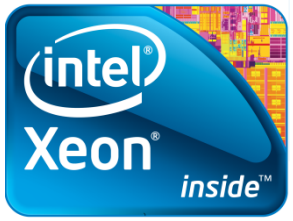
This is possible (yay!) if we design things carefully*

Trusted Computing Hardware

- A hardware root-of-trust vs. assumed software trust
- Why trust hardware?
  - Tamper-resistant from all software
  - Highest level of privilege, can monitor everything
  - Slow to iterate but can be verified

Evolution of Trusted Computing

ARM TRUSTZONE
System Security



2004	2005	2006	2014	2016
TPM 1.2	TrustZone	Intel TXT	TPM 2.0	Intel SGX

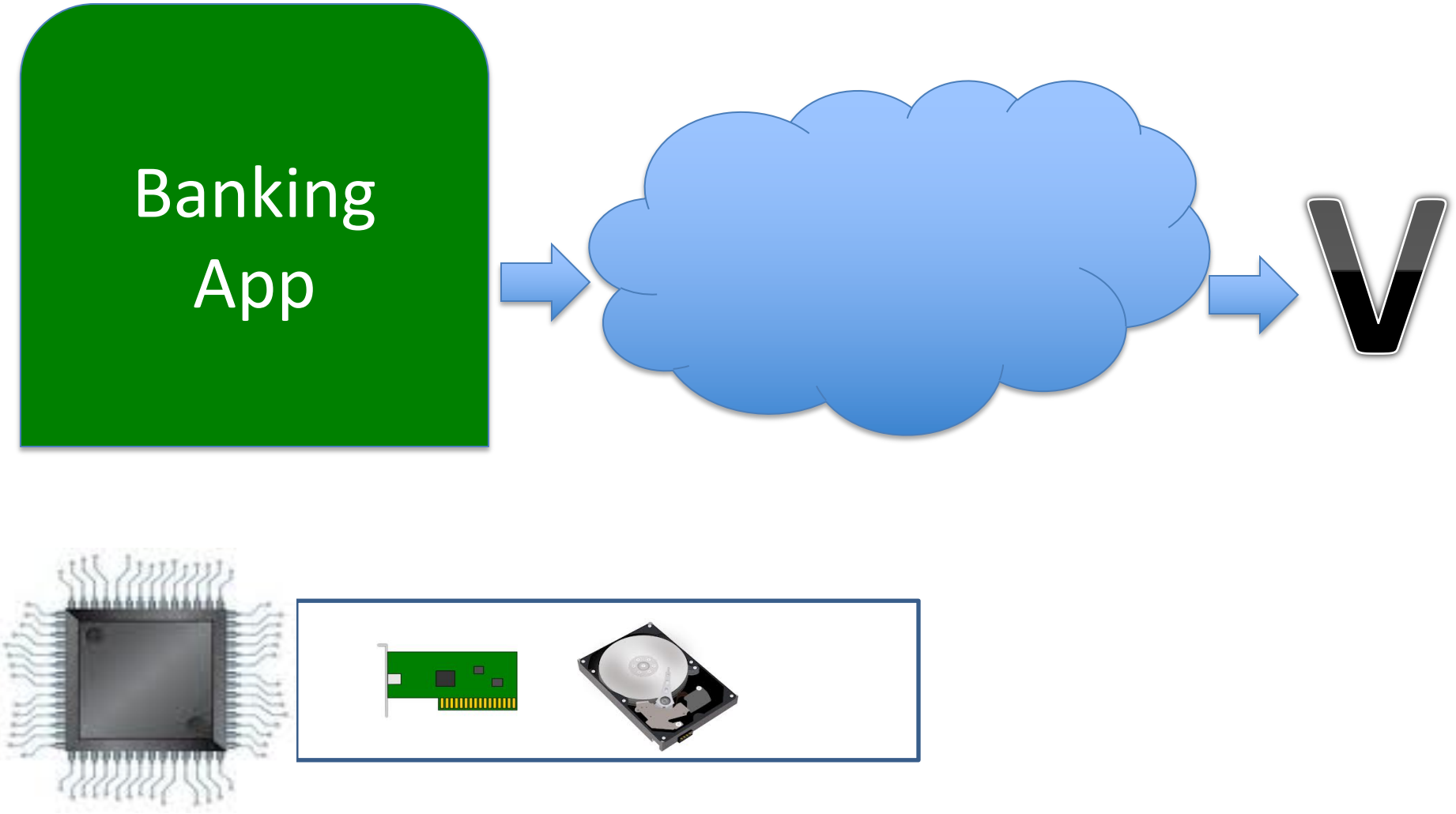
Trusted Computing Primitives

- Remote Attestation
- Trusted Boot
- Sealed Storage
- Isolated Execution

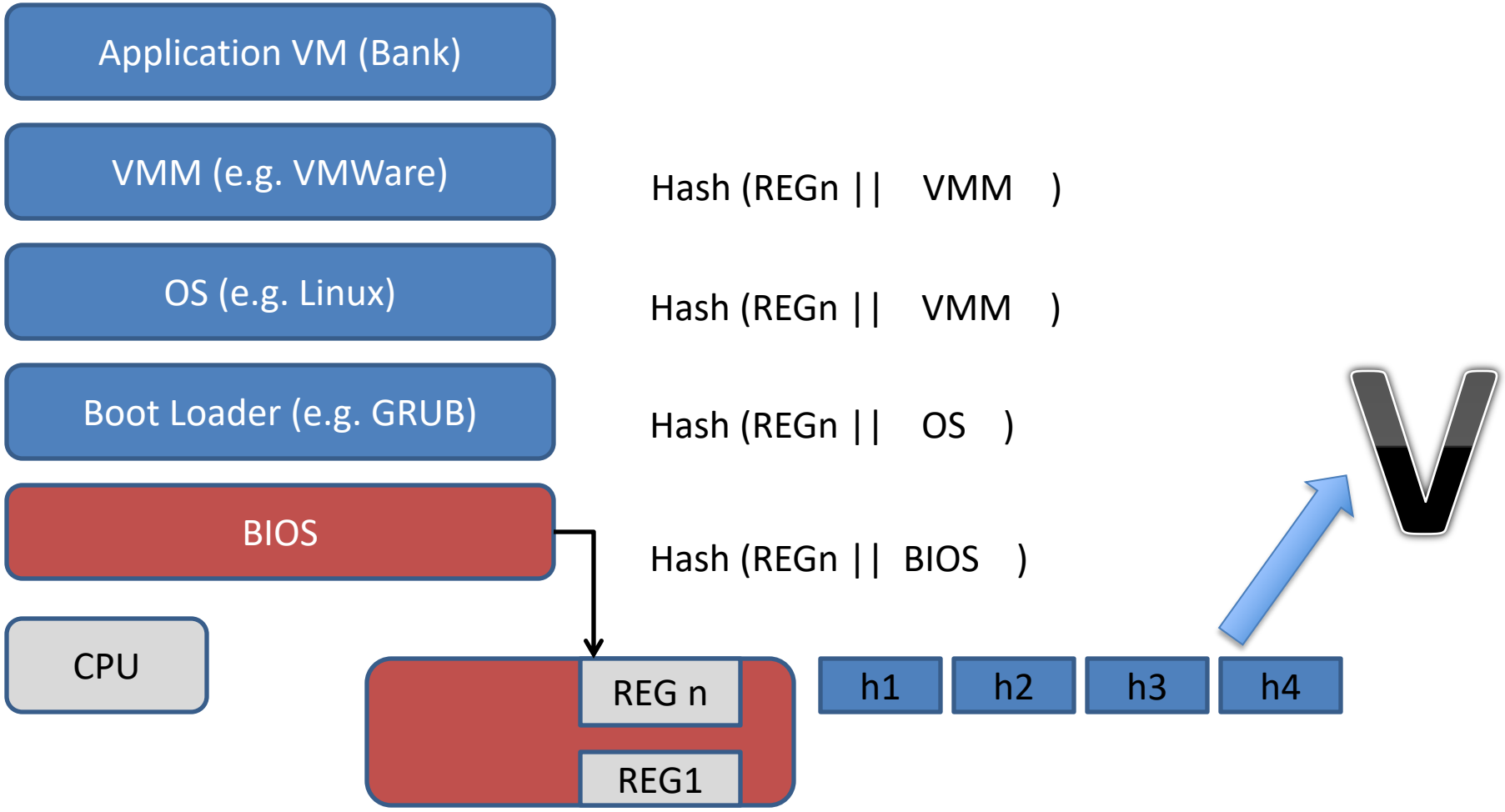
Trusted Computing Primitives (I): Remote Attestation

- **Goal**: Prove to remote party what software is loaded on my machine

Trusted Computing Primitives (I): Remote Attestation



Trusted Computing Primitives (I): Remote Attestation



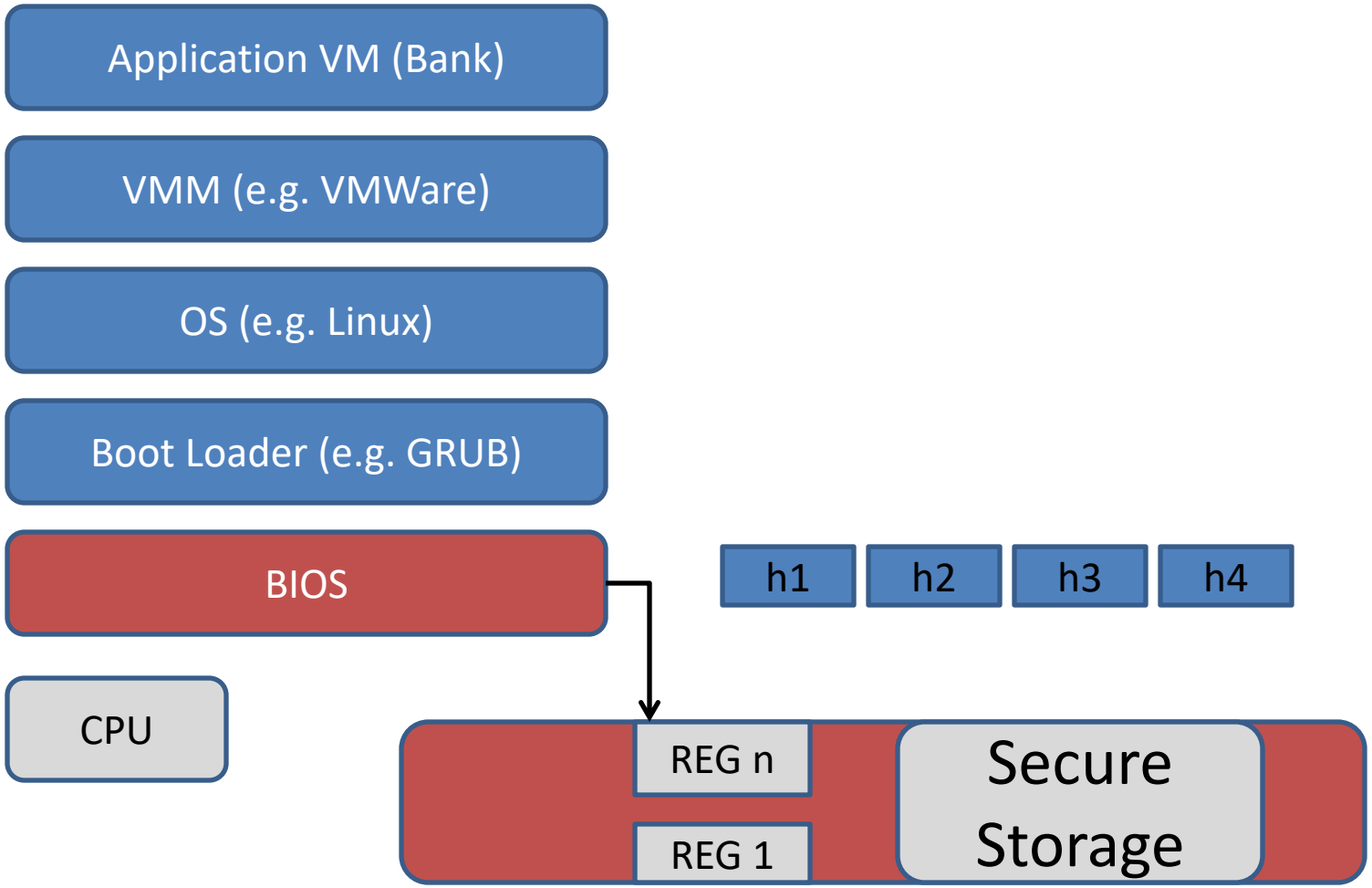
Each CPU has a signing key

Trusted Computing Primitives (II): Trusted Boot

- **Goal:** To perform a measured launch of an OS kernel/VMM

- **Example:** Bank allows money transfer only if customer's machine runs an "up-to-date" OS with patches

Trusted Computing Primitives (II): Trusted Boot

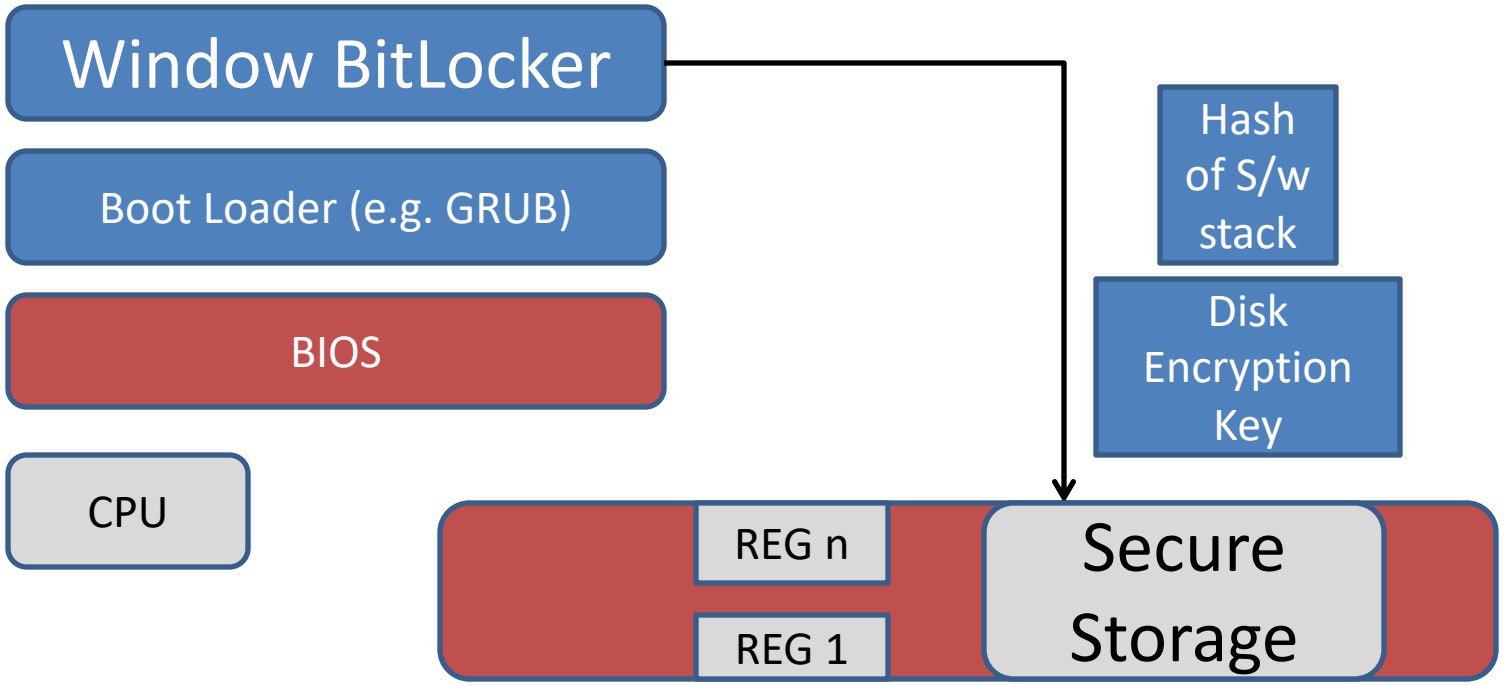


Trusted Computing Primitives(III): Sealed Storage

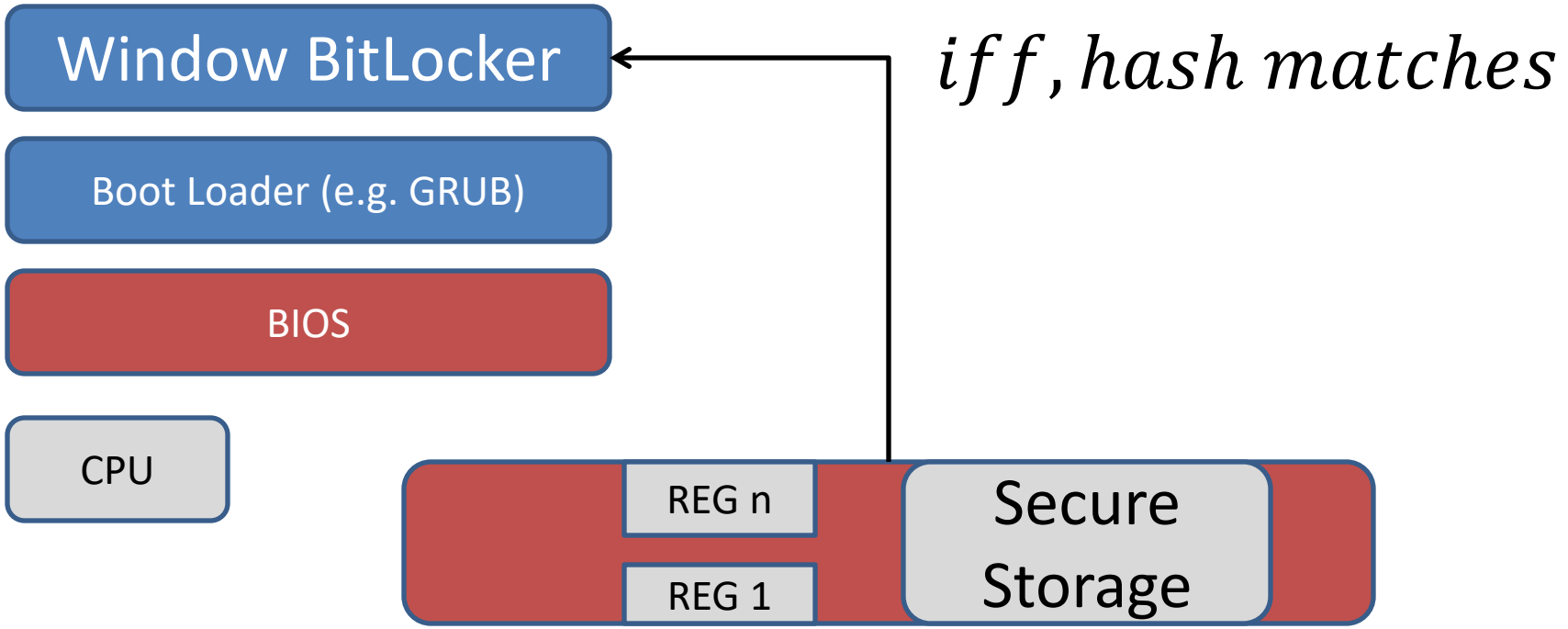
- **Goal:** Protect private information by binding it to platform. Data can be released only to a particular combination of SW & HW

Trusted Computing Primitives(III): Sealed Storage

Use CPU Measurements & Secure Storage for Disk Encryption Systems?



Trusted Computing Primitives (III): Sealed Storage



Use Case: Full Volume Encryption

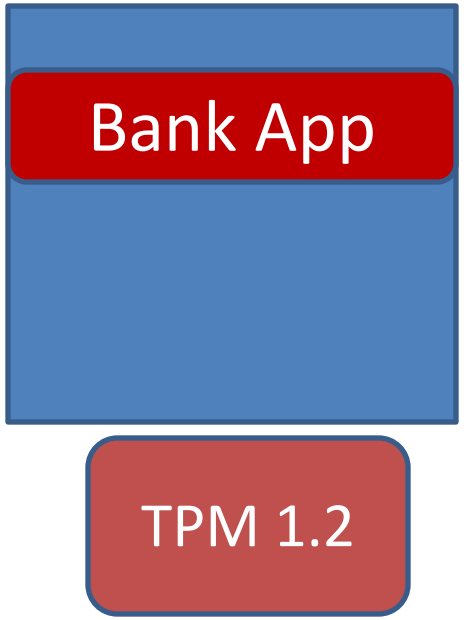
- Encryption at the block level underneath file system
- Everything in the volume is encrypted.
- BitLocker was first used by Microsoft since Windows Vista
- BitLocker takes advantage of TEEs
  - Top level root key sealed in hardware
  - Root key encrypts disk encryption key, which encrypts sector data
- CPU protects disk encryption key by encrypting it
- CPU releases key only after comparing hash of early (unencrypted) boot files with previous hash

Trusted Computing Primitives (IV): Isolated Execution

- Goal: Execute sensitive code in isolation from other malicious programs running on the same machine

Isolated Execution with Late Launch / DRTM

- One Approach: Intel TXT "SKINIT/SENTER"

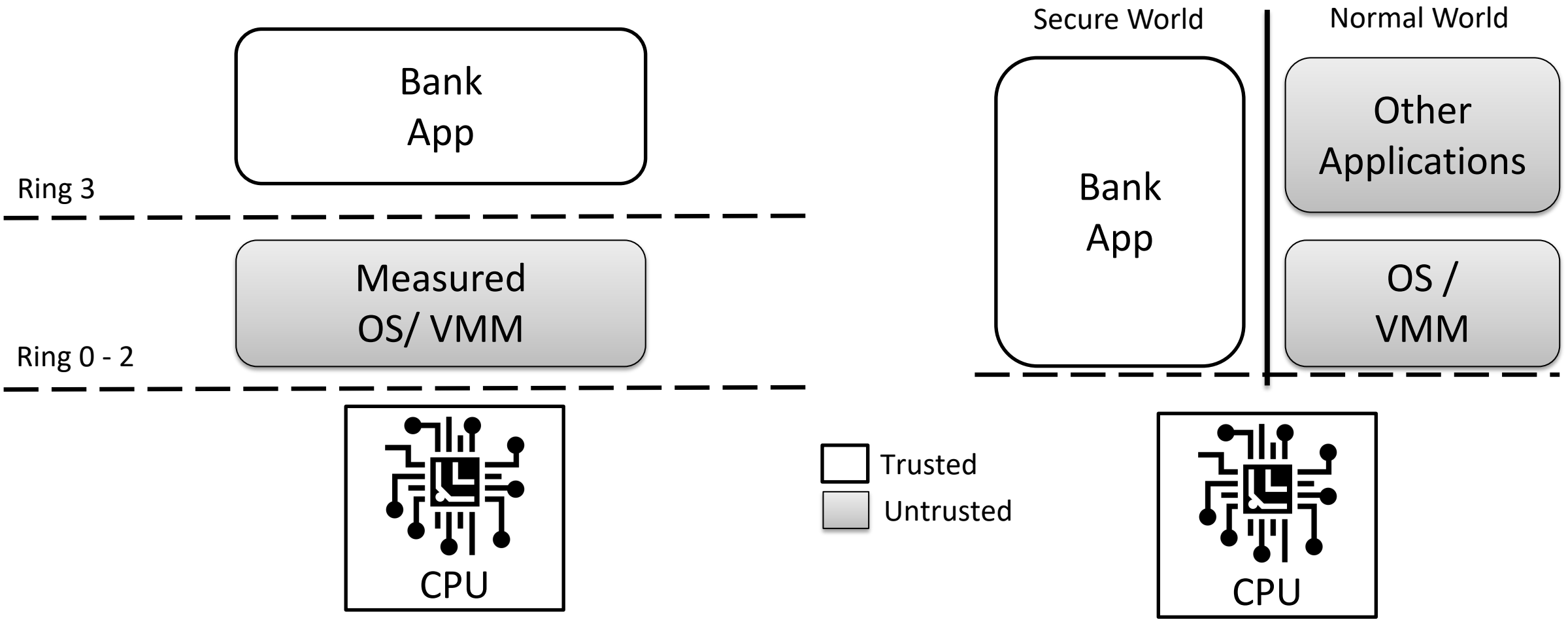


On SENTER:

- TPM hashes memory region in PCR17
- Checks if valid
- Executes the trusted region code

[1] http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf

Isolated Execution Models

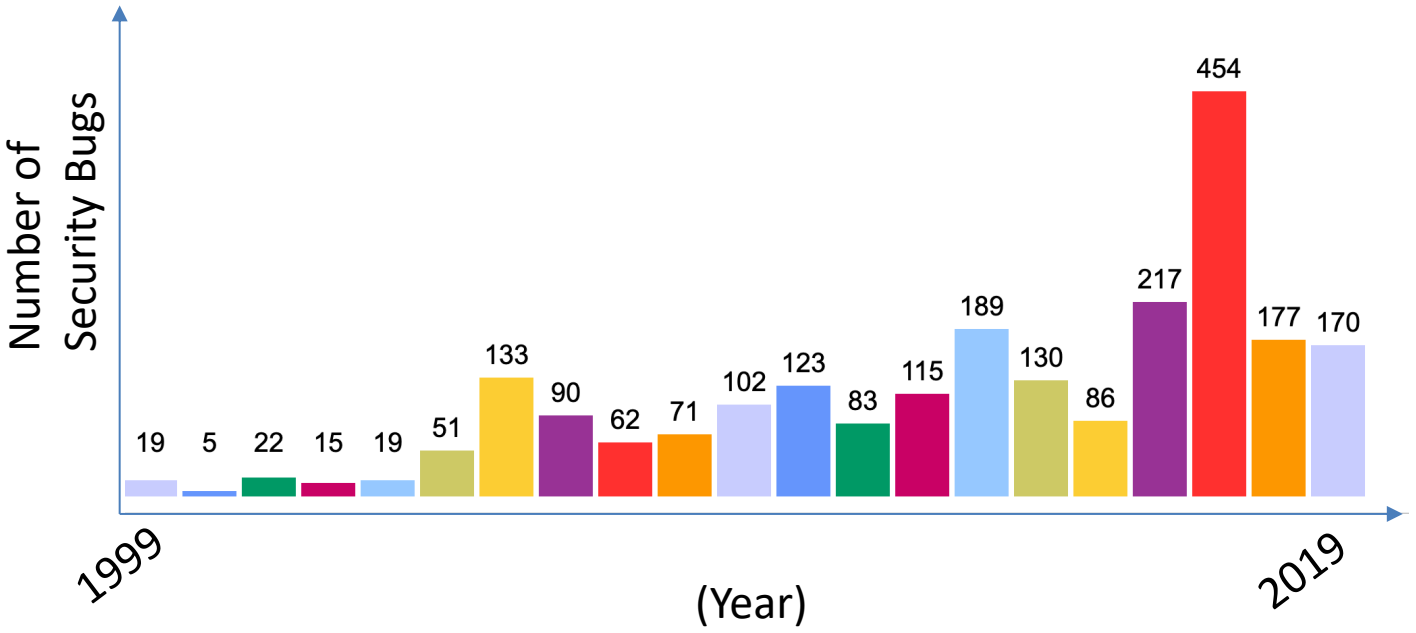


Break for questions

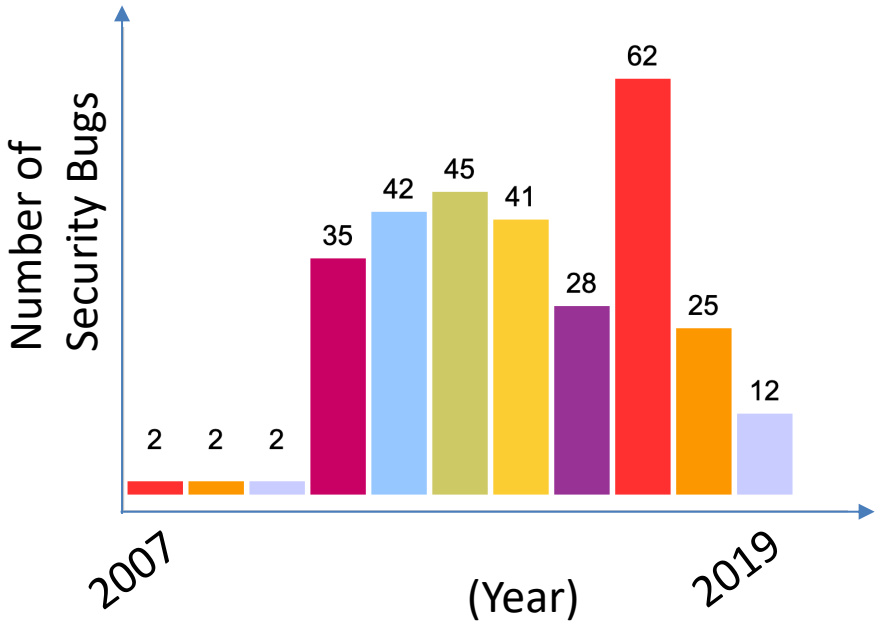
How much code are we trusting?

We measure the OS but still assume it to be bug-free

Operating System (Linux Kernel)
27 Million Lines

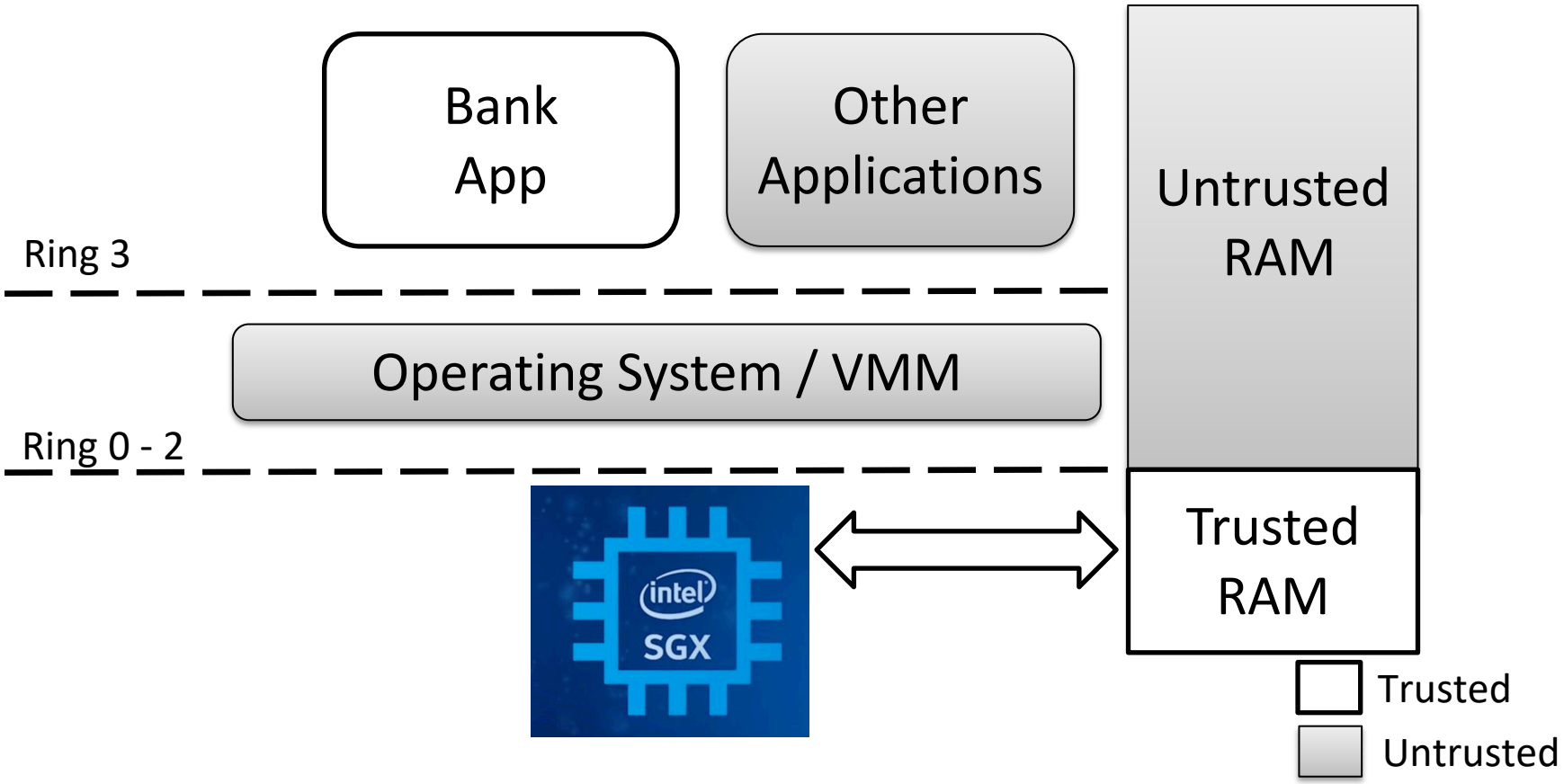


Hypervisor (XEN)
0.5 Million Lines

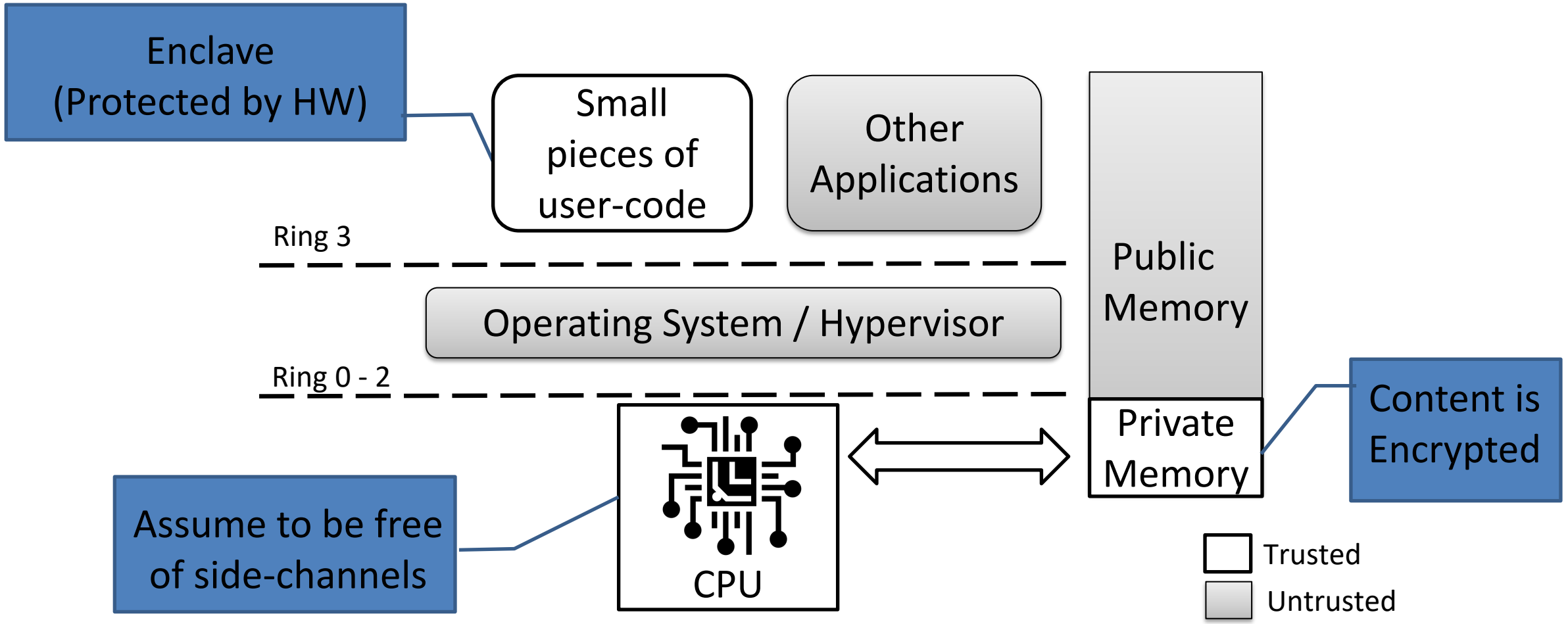


Another approach for achieving Isolated Execution

SGX Isolated Execution



Intel SGX Enclaves



Design Choice: Untrusted OS/VMM

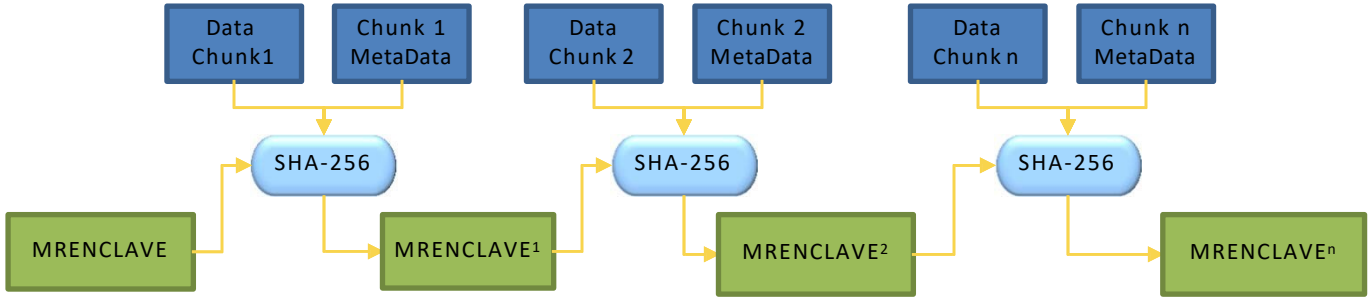
Enclave Measurement

When building an enclave, Intel® SGX generates a cryptographic log of all the build activities

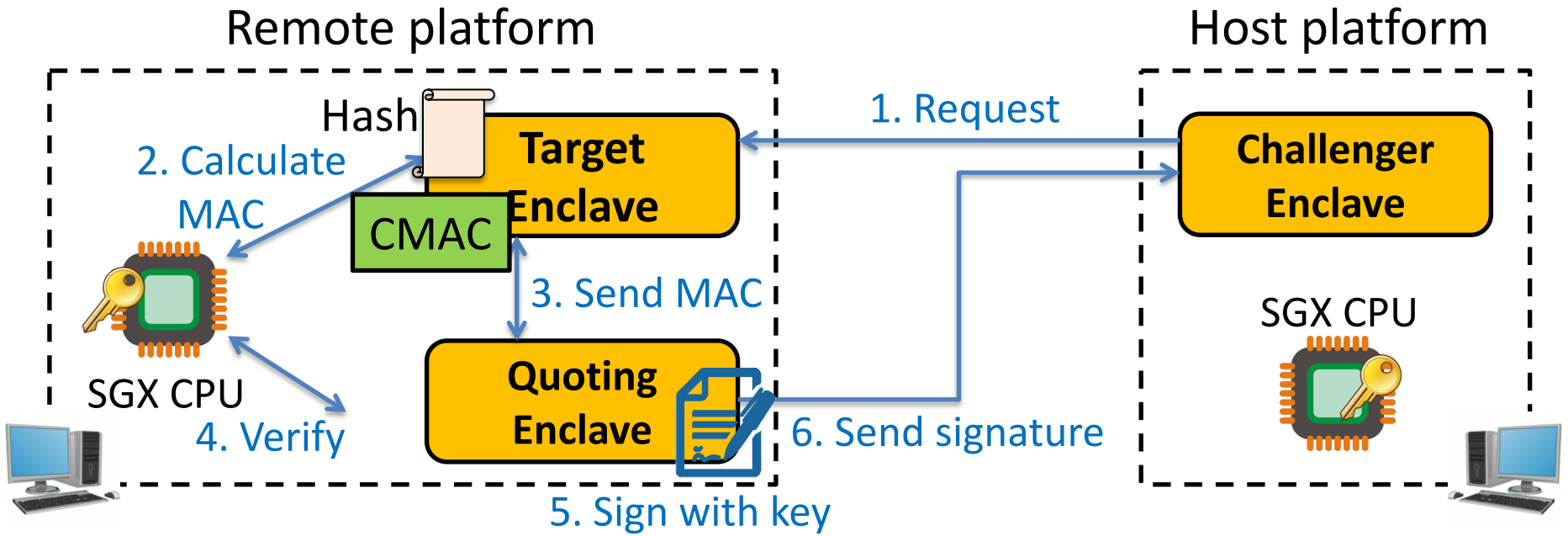
- Content: Code, Data, Stack, Heap
- Location of each page within the enclave
- Security flags being used
- Order in which it was built

MRENCLAVE ("Enclave Identity") is a 256-bit digest of the log

- Represents the enclave's software trusted computing base (TCB)



SGX Remote Attestation



- Attest an application on remote platform
- Check the identity of enclave (**hash of code/data pages**)
- Can establish a "**secure channel**" between enclaves

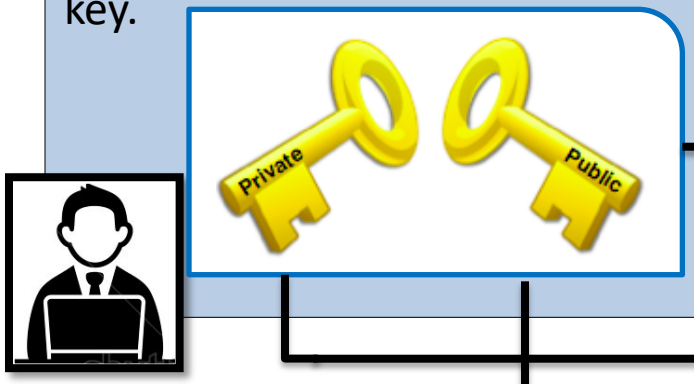
SGX Sealed Storage

- Cryptographically protect data when it is stored outside enclave

SGX Sealing Actors

Sealing Authority – Developer

Enclave that Signs the enclave **certificate** with its RSA-3072 Private key.



Enclave certificate.

- Sealing Authority **Public key** (Used by HW to verify **Certificate**).
- Enclave identity aka **MRENCLAVE** (Used by HW to verify enclave Integrity).
- SVN – Security Version Number.
- Other enclave attributes.

Sealing Authority signature

Sealing Identity.

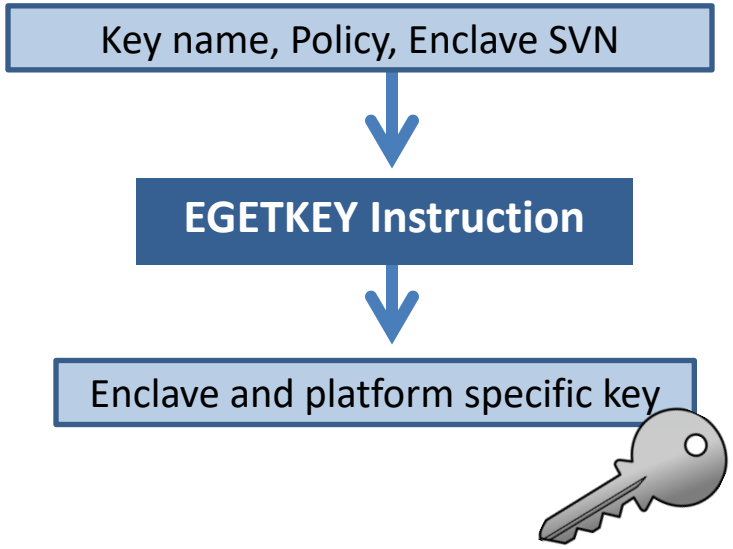
- Created on enclave initialization.
- Accessible only by the TCB.
- **Used to seal enclave data.**

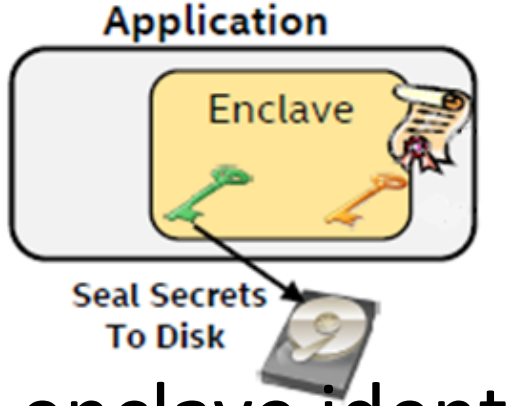
Includes:

- sealing authority **Public Key** **hash**.
- Product ID.
- SVN – Security Version Number.

SGX Sealing Process

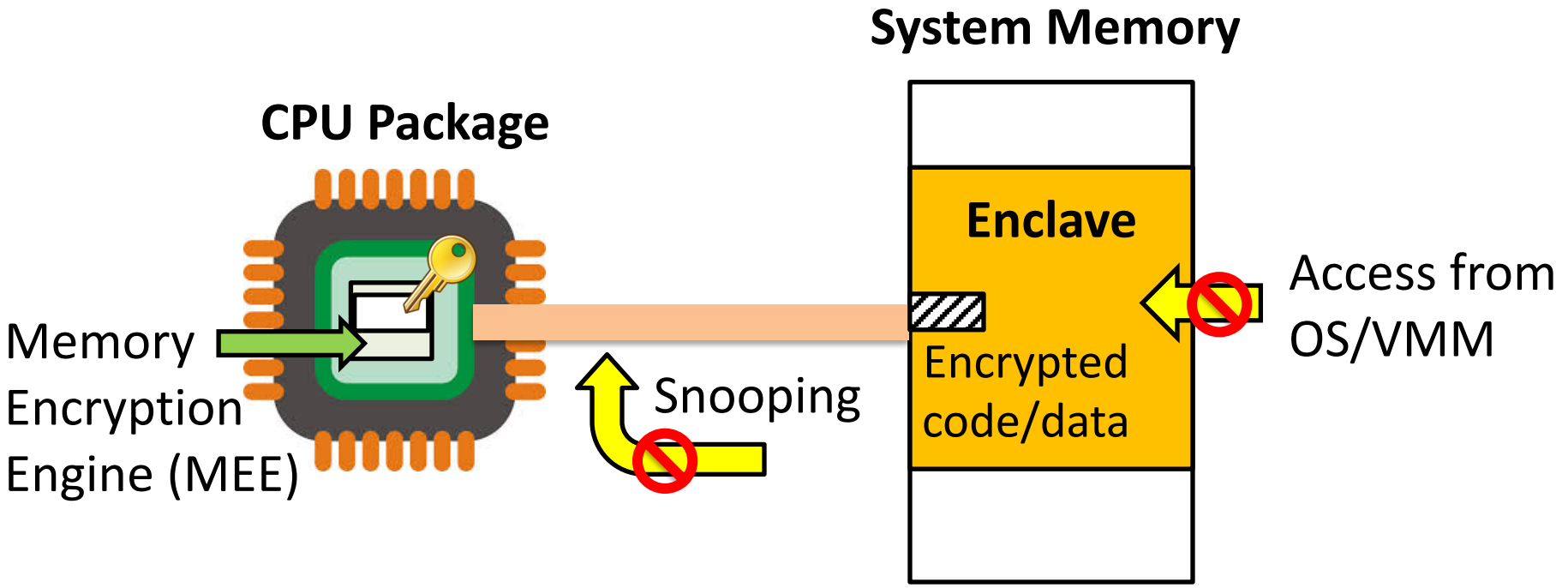
- 1. Enclave calls EGETKEY
- 2. Sets sealing key
- 3. Seals data with key
- 4. Writes sealed data to untrusted storage





Data sealed under processor key, enclave identity...

SGX: Isolated Execution



- Application keeps its data/code inside the "enclave"
  - Smallest attack surface by reducing TCB (App + processor)
  - Protect app's secret from untrusted privilege software (e.g., OS, VMM)

Architecture Overview

Enclave

Trusted Execution Environment embedded in application

Provides confidentiality and/or integrity of it's own code/data

EPC (Enclave Page Cache)

Trusted Memory for enclave

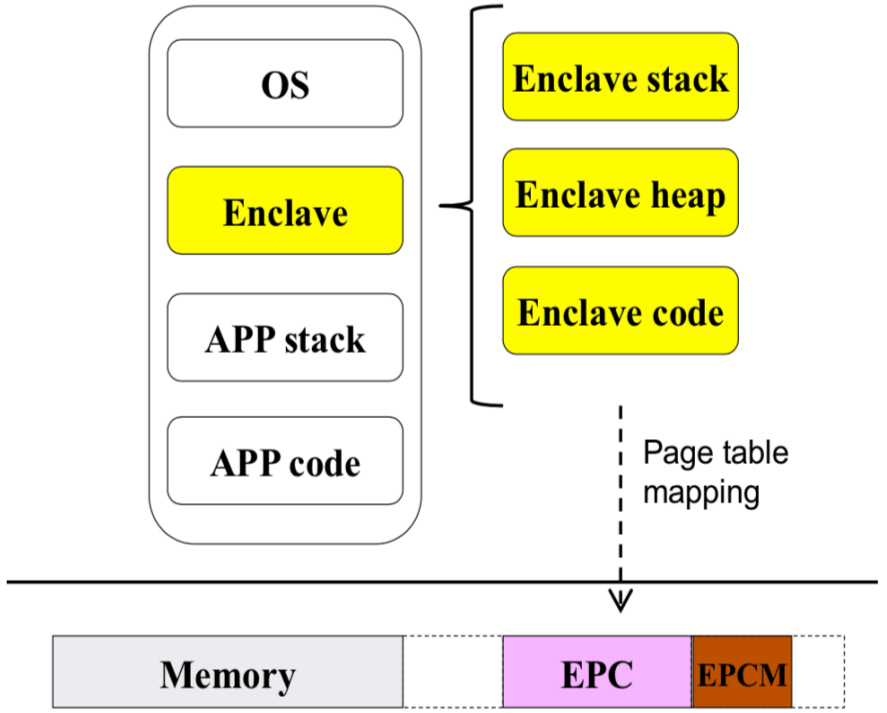
Typically reserved by BIOS

Limited size (ex, 32/64/128M)

EPCM (Enclave Page Cache Map)

Used by HW to track EPC

(not-visible to SW)

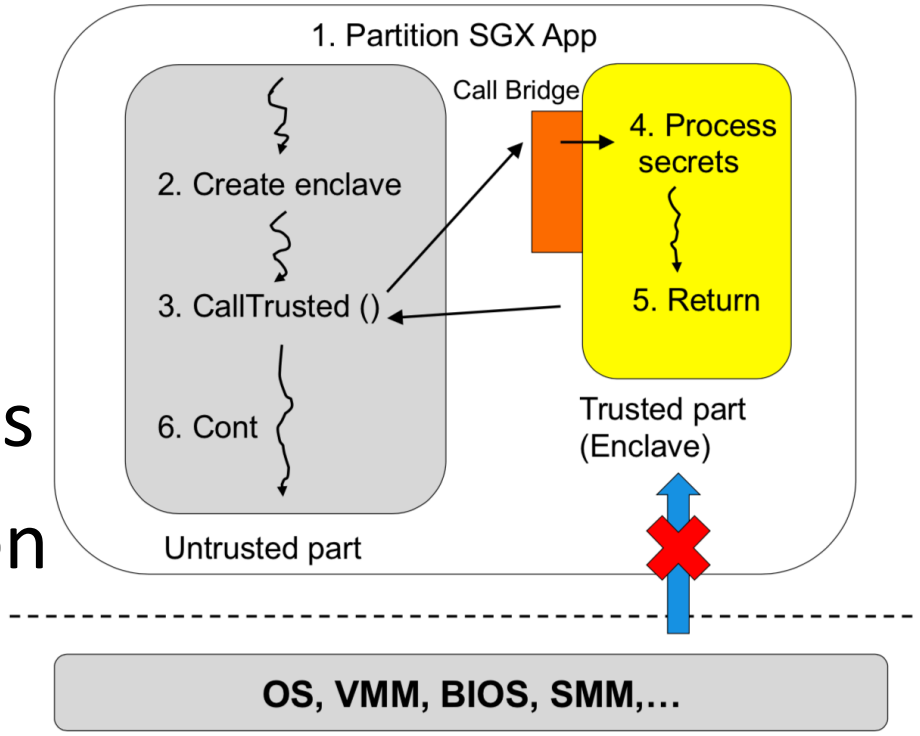


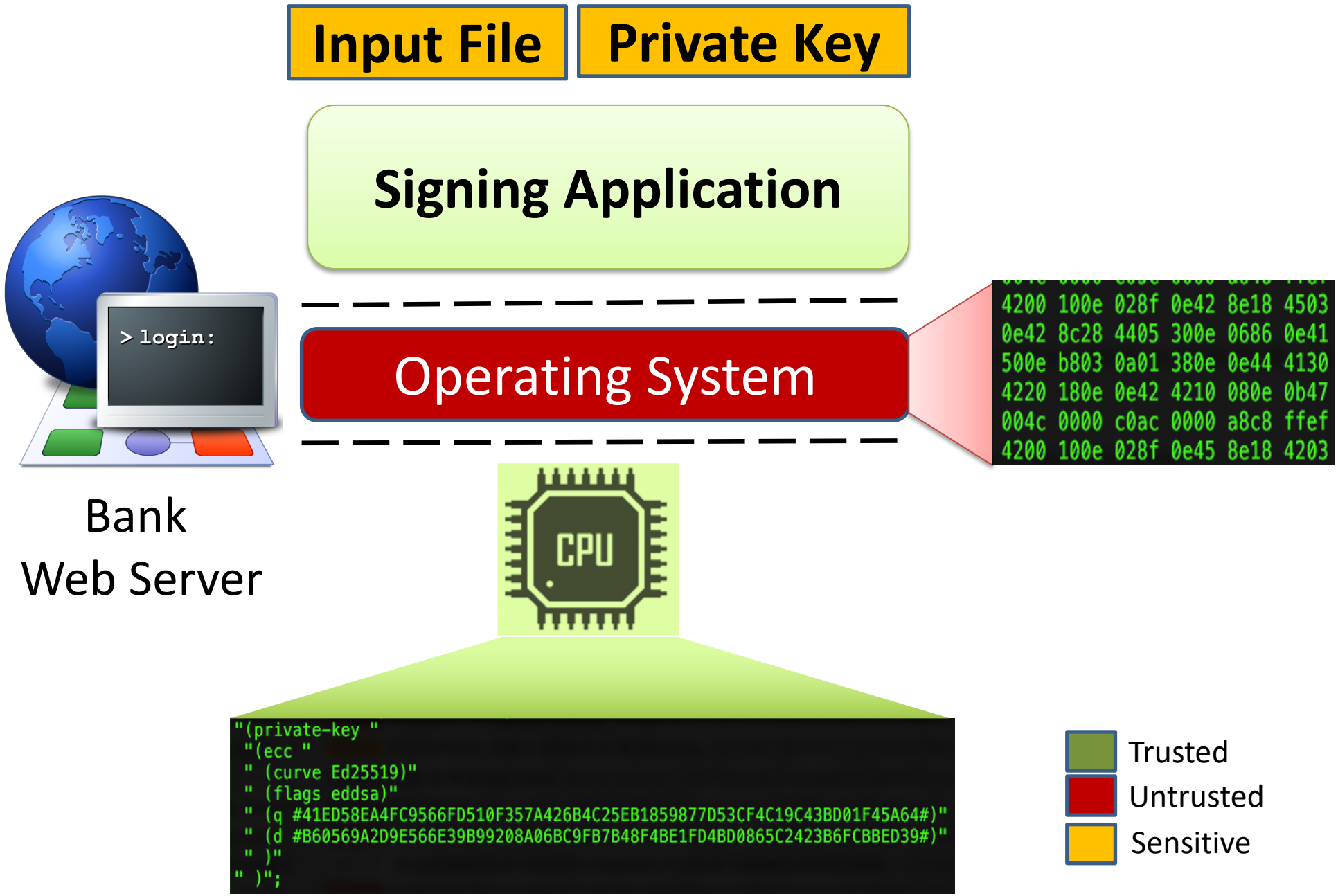
Design Choice: Untrusted OS

- Against the typical setting used for decades
- What should we protect?
  - **Creation of the enclave**
  - **Code & data inside the enclave**
  - **Data generated by the enclave**
  - **Services like timers, random numbers**
  - Start and end of enclave
  - Enclave-enclave communication
  - Memory management of the enclave
  - IO & System calls by the enclave

Controlled Entry & Exit Points

- Define & partition app
  - untrusted and trusted
- App creates enclave
- Calls a trusted function
- Enclave code executes
- Trusted function returns
- App continues execution




Input File
Private Key
Signing Application
> login:
Operating System
4200 100e 028f 0e42 8e18 4503
0e42 8c28 4405 300e 0686 0e41
500e b803 0a01 380e 0e44 4130
4220 180e 0e42 4210 080e 0b47
004c 0000 c0ac 0000 a8c8 ffef
4200 100e 028f 0e45 8e18 4203
Bank
Web Server
CPU
"(private-key "
"(ecc "
" (curve Ed25519)"
" (flags eddsa)"
" (q #41ED58EA4FC9566FD510F357A426B4C25EB1859877D53CF4C19C43BD01F45A64#)"
" (d #B60569A2D9E566E39B99208A06BC9FB7B48F4BE1FD4BD0865C2423B6FCBBED39#)"
")"
")";
Trusted
Untrusted
Sensitive

Applications of TEEs

- Server-side apps
  - webserver, database
- Client-side apps
  - hardware security module (HSM)
  - Enterprise Rights Management (DRM)
  - Faster encrypted computation
- Securing remote execution
  - Data analytics (e.g., ML) in the cloud
  - Distributed computing (e.g., Tor, consensus protocols)

BREAK

Are TEEs (e.g., SGX) sufficient?

Covert Channels

- Definition: "An unintended channel of communication between two untrusted programs"

- E.g., Shared Cache Latency
  - Sender
    - Send bitval 1: Perform random memory access
    - Send bitval 0: Do nothing
  - Receiver
    - Rcv bitval 1: If long read time for a fixed memory location
    - Rcv bitval 0: If short read time for a fixed memory location

- Many traditional channels: Caches (all levels), Timing, Power, Disk, I/O, virtualization latency

Side Channels

- Definition: "An channel of information leakage from an trusted to an untrusted program"
- E.g. Learning encryption keys of co-located VMs
- $E := x^e \ (mod\ \mathbf{N})$

```
SquareMult(x, e, N):
  let e_n, ..., e_1 be the bits of e
  y ← 1
  for i = n down to 1 {
    y ← Square(y)                (S)
    y ← ModReduce(y, N)          (R)
    if e_i = 1 then {
      y ← Mult(y, x)             (M)
      y ← ModReduce(y, N)        (R)
    }
  }
  return y
```

Timing Differences

Cache Access Patterns

Typical Defenses

- Decouple the relationship between secrets and observations

- Determinism: Pad to normalize the channel for
- Indistinguishability: Add jitter, randomization, or noise

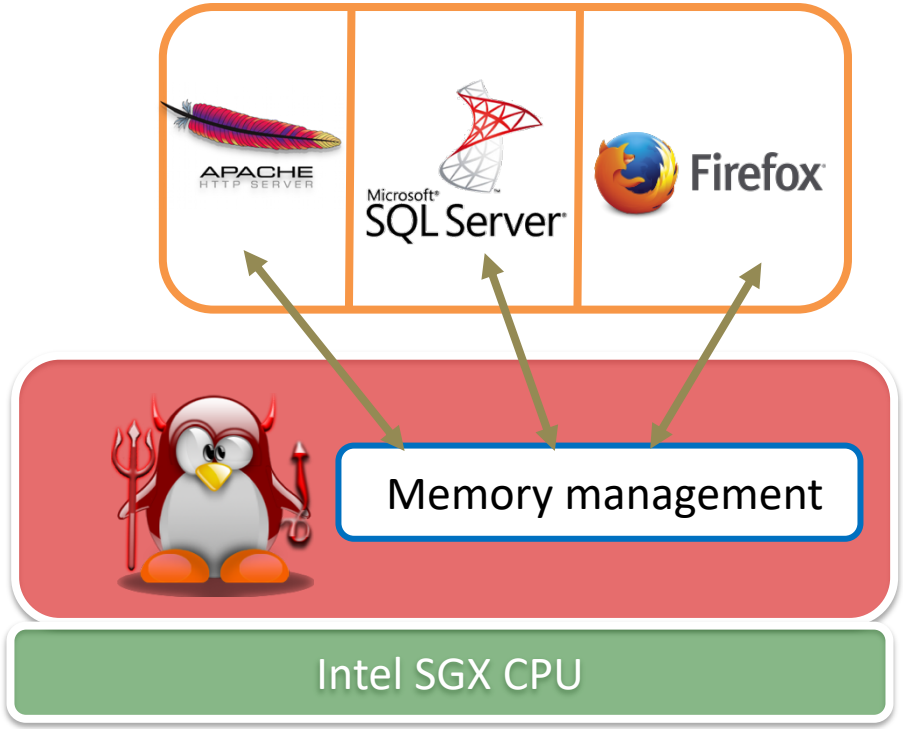
- Clear the state every time irrespective of the secret
- Disallow access to the channel

Threat to TEE Guarantees

- TEEs do not directly protect enclaves against side channels ☹
- Enclaves may leak information via traditional channels
- Added Threat:
  - Recent channels via shared microarchitecture (e.g., spectre):
  - private caches, branch predictors, TLBs
- Leaking the CPU key has larger ramifications
  - e.g., Compromise remote attestation
- More bad news: TEEs create new side-channels

Untrusted Memory Manager

- OS manages all enclave resources
- New sources of side channels: system calls, file I/O
- Recent attack channels: Page faults

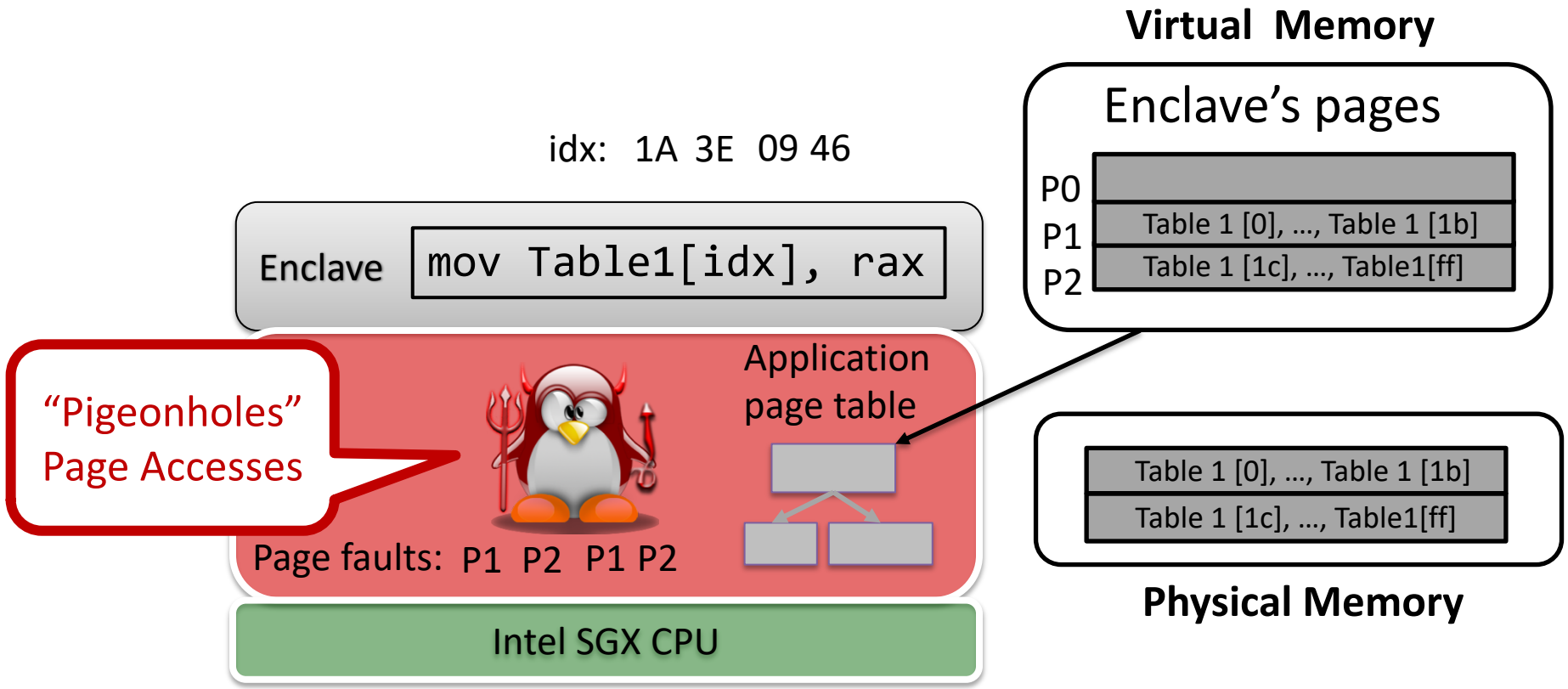


Page-fault side-channel in TEEs

- Observation: The untrusted OS controls the memory management of the enclave
- OS decides how much memory to allocate for each enclave
- OS is notified of memory events (e.g., page faults)

- In SGX, the page fault reports only the page number, not the offset within the page

Attack Example I: AES Encryption



OS can extract bits of private AES key

Attack Example II: ECDSA Signing

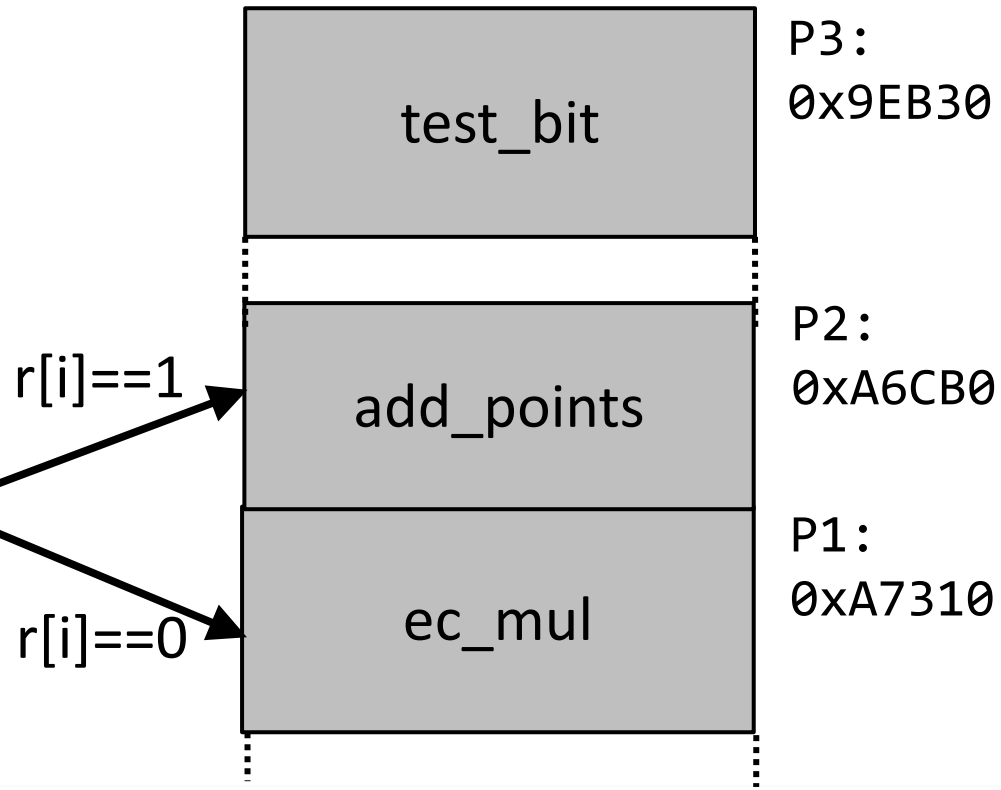
ECDSA Signing Routine

```
ec_mul(r, G) {
res = O
nbits = |r|

for (i = nbits-1; i>=0; i--):
  res = dup_point(res)

  if (test_bit(r[i])):
    res = add_points(res, G)

return res }
```

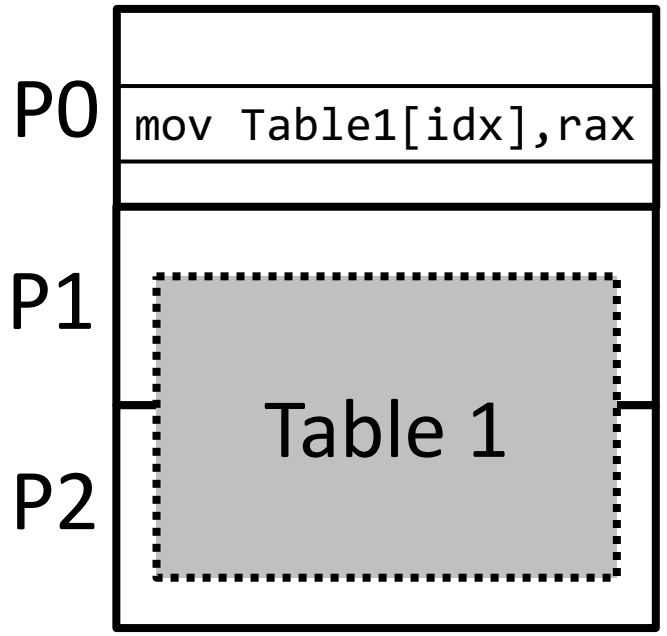


Input dependent code page access reveals all the bits of ECDSA keys!

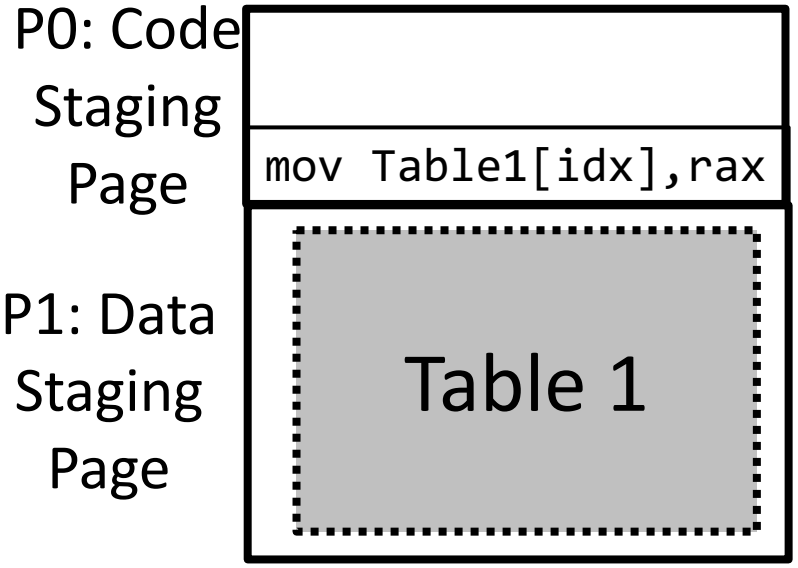
Defenses against the PF side-channel

- Remove secret-dependent page-faults
  - Fit the table/branches in the same page
  - Always leads to a page fault
  - May not fit in one page
- Make the page fault pattern deterministic
  - Always leads to the same sequence of page faults
- Detect the attack in real-time and stop the leakage

Defense Example: Determinising Data Accesses

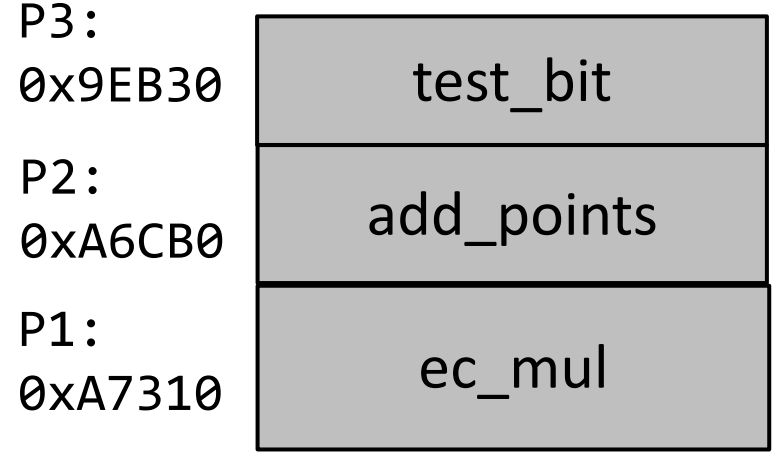


Page-fault Profile: P0 P1 P0 P2 P0 P1 P0 P2



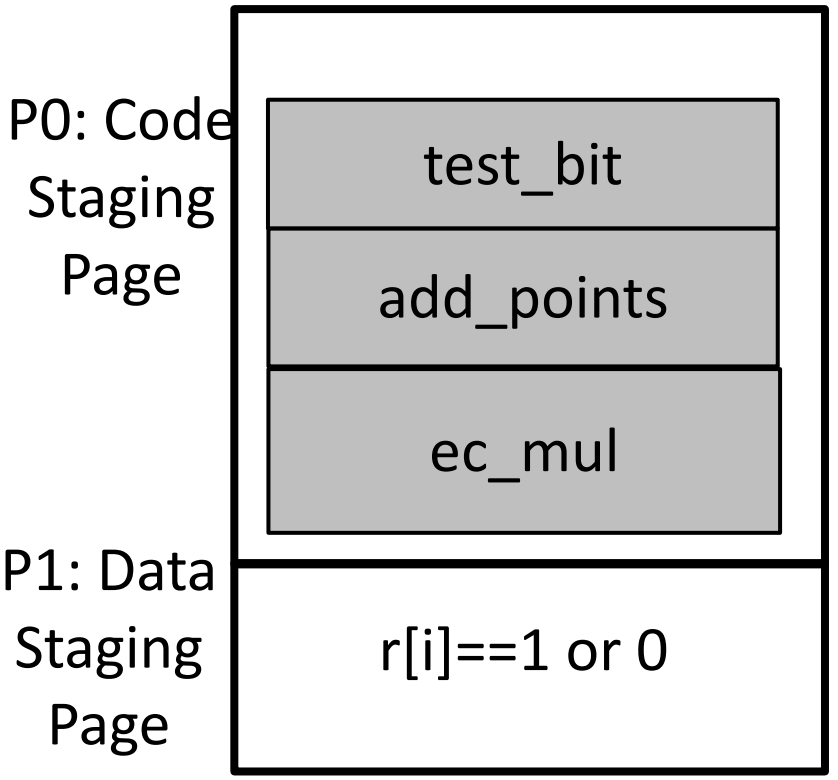
Page-fault Profile: P0 P1 P0 P1 P0 P1 P0 P1

Example: Determinising Code Accesses



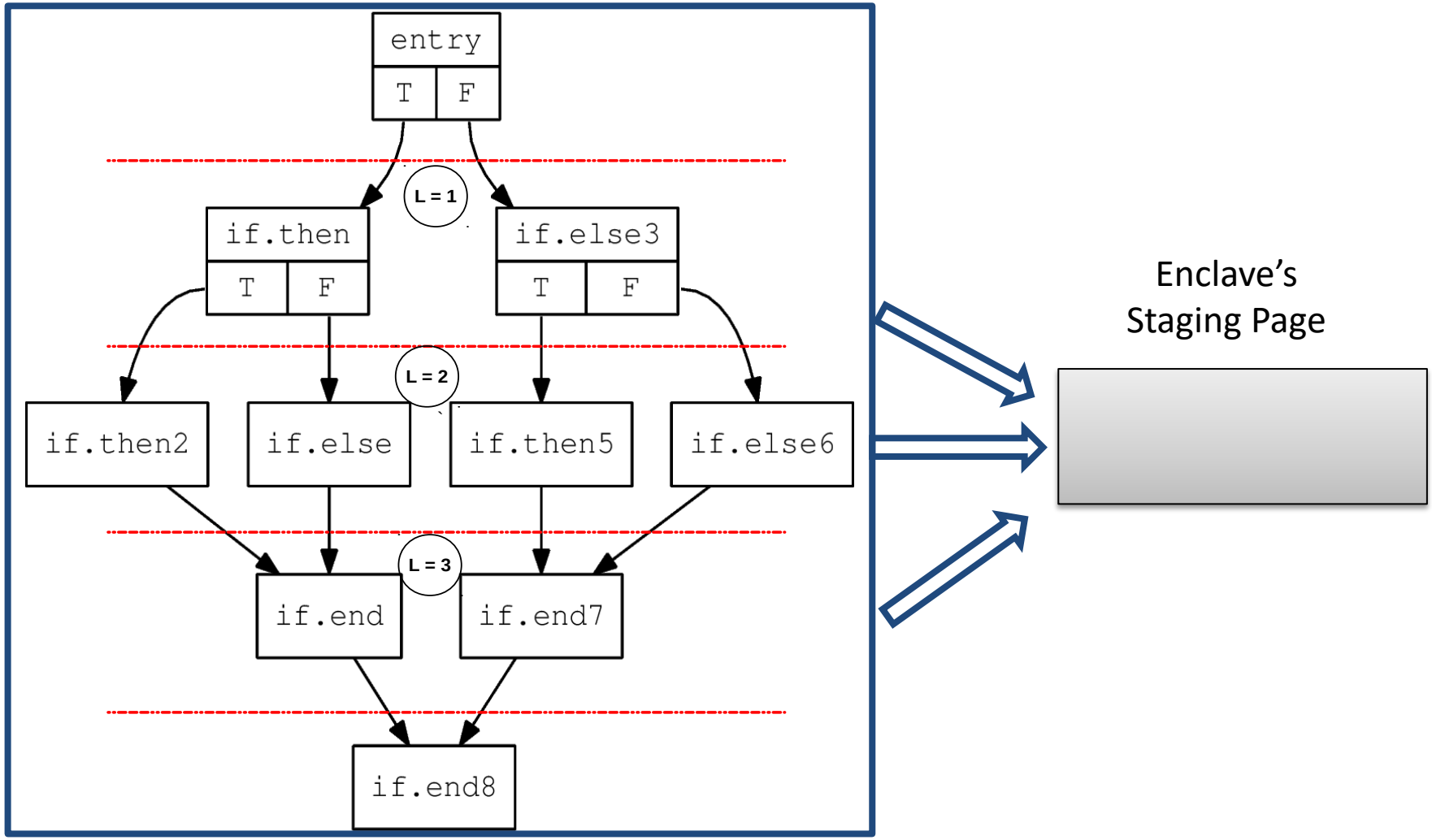
Page-fault Profile r =1 P1 P2 P1 P3 P1 P1 P2

Page-fault Profile r =0 P1 P2 P1 P3 P1



Page-fault Profile: P0 P1 P0

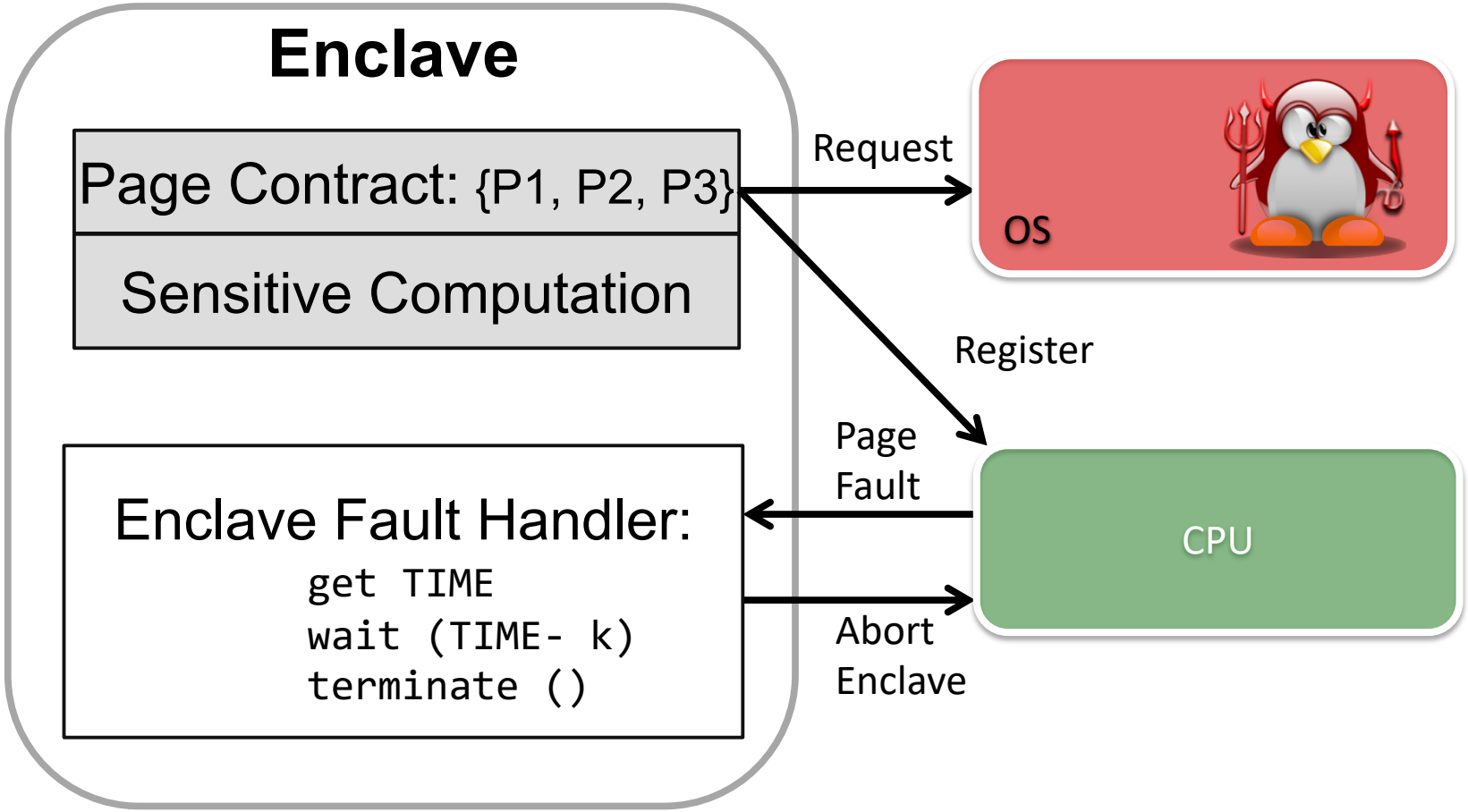
Determinising Page Layout (multi-page)



Multiplexed Execution

- **Fetch** all the input-dependent code / data to a staging page
- **Execute** the dependent-logic within the staging page via a multiplexer
- **Save** the updated values from data page

Hardware Approach: Contractual Execution



Summary

- 4 Primitives of Trusted Computing
  - Remote Attestation
  - Trusted Boot
  - Sealed Storage
  - Isolated Execution

- Using these primitives to understand Intel's TEE
- Identifying and fixing gaps in TEEs

Shameless Plugs

- Join Secure & Trustworthy Systems (SECTRS) group
  - Open positions for PhD & Master students.
  - Email me for more info

- Next Semester: 252-2603-00L Seminar on Systems Security

Thank you!

Questions

- About the material covered in this lecture

- About the lab
  - Now is a good time to ask
  - More Q&A in the exercise and lab sessions
  - Discuss on Moodle