CS155, Spring 2016

Trusted Computing and SGX

Acknowledgments: Lecture slides are from the Computer Security course taught by Dan Boneh and John Mitchell at Stanford University. When slides are obtained from other sources, a reference will be noted on the bottom of that slide. A full list of references is provided on the last slide.

TCG: Background

TCG consortium. Founded in 1999. Lots of companies.

Goals:

- **Hardware protected (encrypted) storage**:
  - Only "authorized" software can decrypt data
  - e.g.: protecting key for decrypting file system

    ⇒ only "authorized" software can boot

- **Attestation**: Prove to remote server what software started on my machine.

TCG: changes to the PC

Extra hardware: Trusted Platform Module (**TPM**) chip (33Mhz)

- Available on many laptops

Software changes:

Hardware layer: BIOS, EFI (UEFI)

Software: OS and apps

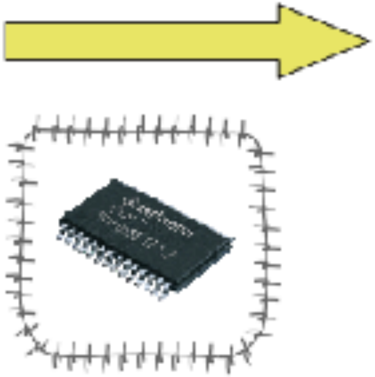
Trusted Computing

What is the TPM?

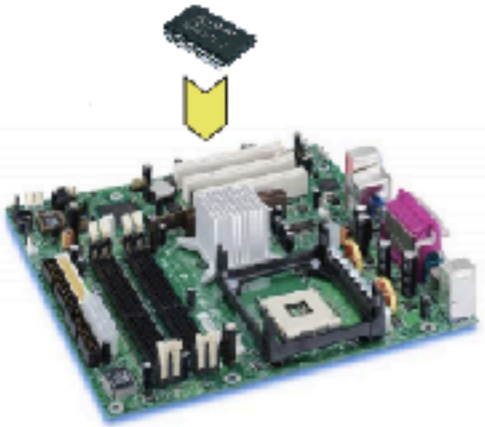
Integrating Trust and Security into Computing Platforms using a Security Chip

Standard Processor System

- Easy to program
- Easy to change
- Easy to attack



TPM- Security Module

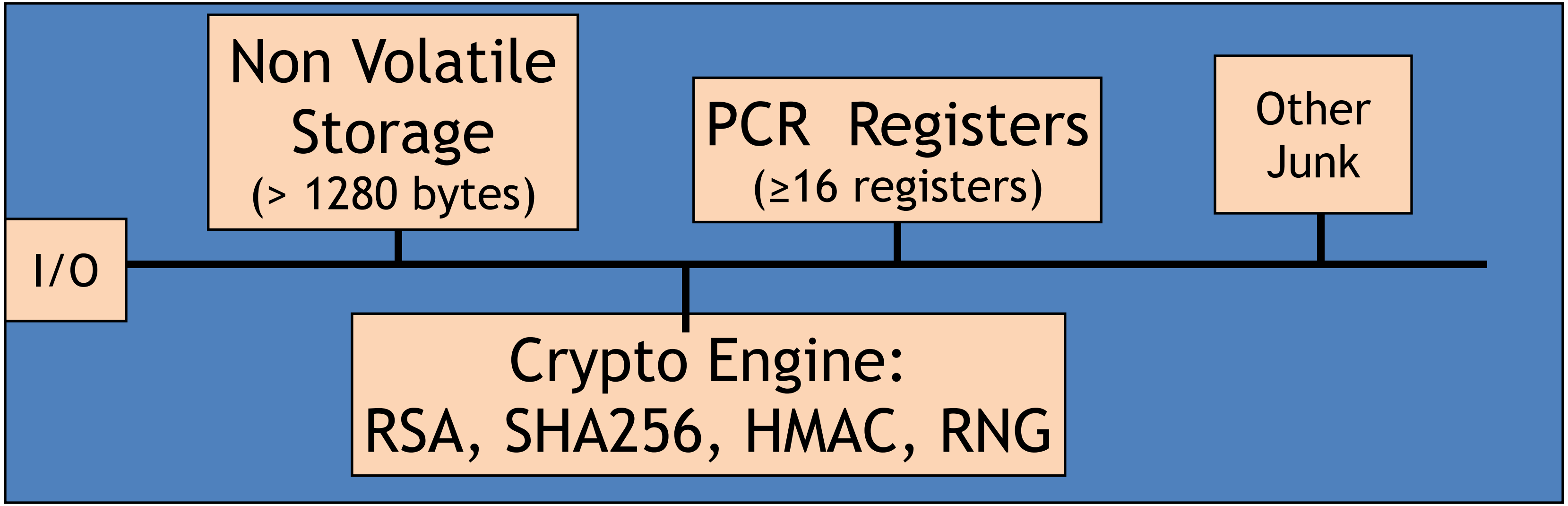


Trusted platform

- Shielded and encapsulated chip
- Controlled interface to external
- Trusted software in a protected hardware

=> Security functions, protected against manipulations

Components on TPM chip



RSA: 1024, 2048 bit modulus

SHA256: Outputs 32 byte digest

Non-volatile storage

1. Endorsement Key (EK) (2048-bit RSA)
   - Created at manufacturing time. Cannot be changed.
   - Used for “attestation” (described later)

2. Storage Root Key (SRK) (2048-bit RSA)
   - Used for encrypted storage. Created after running
     TPM_TakeOwnership(OwnerPassword, …)
   - Can be cleared later with TPM_ForceClear from BIOS

3. OwnerPassword (160 bits) and persistent flags

Private: EK, SRK, and OwnerPwd never leave the TPM

PCR: the heart of the matter

PCR: Platform Configuration Registers

- Many PCR registers on chip (at least 16)
- Contents: 32-byte SHA256 digest (+junk)

Updating PCR #n :

- TPM_Extend(n,D): PCR[n] ⟵ SHA256 (PCR[n] ll D)

- TPM_PcrRead(n): returns value(PCR(n))

PCRs initialized to default value (e.g. 0) at boot time

Using PCRs: the TCG boot process (SRTM)

On power-up: TPM receives a TPM_Init signal from LPC bus.

BIOS **boot block** executes:

- Calls TPM_Startup (ST_CLEAR) to initialize PCRs to 0
  [can only be called once after TPM_Init]
- Calls PCR_Extend(n, <BIOS code>)
- Then loads and runs BIOS post boot code

BIOS executes: Calls PCR_Extend(n, <MBR code>)

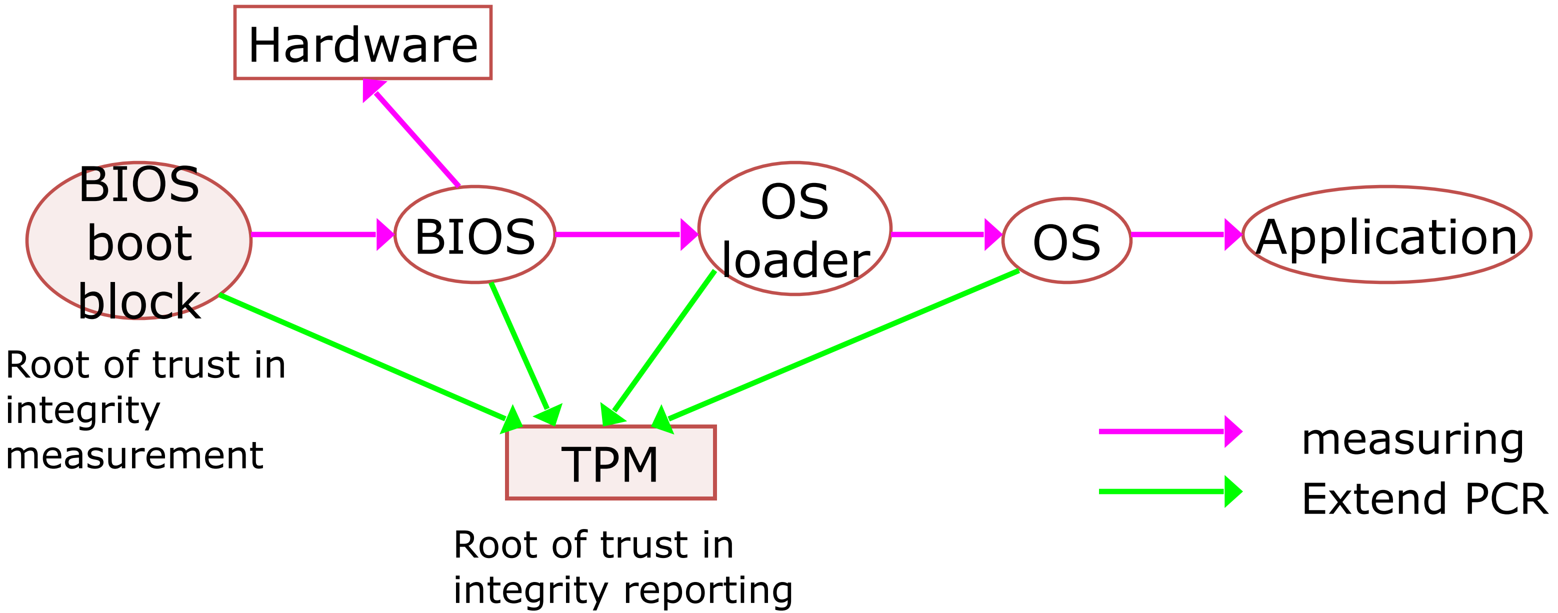
- Then runs MBR (master boot record), e.g. GRUB.

MBR executes: Calls PCR_Extend(n, <OS loader code, config>)

- Then runs OS loader

… and so on

In a diagram



Hardware

BIOS boot block

BIOS

OS loader

OS

Application

TPM

Root of trust in integrity measurement

Root of trust in integrity reporting

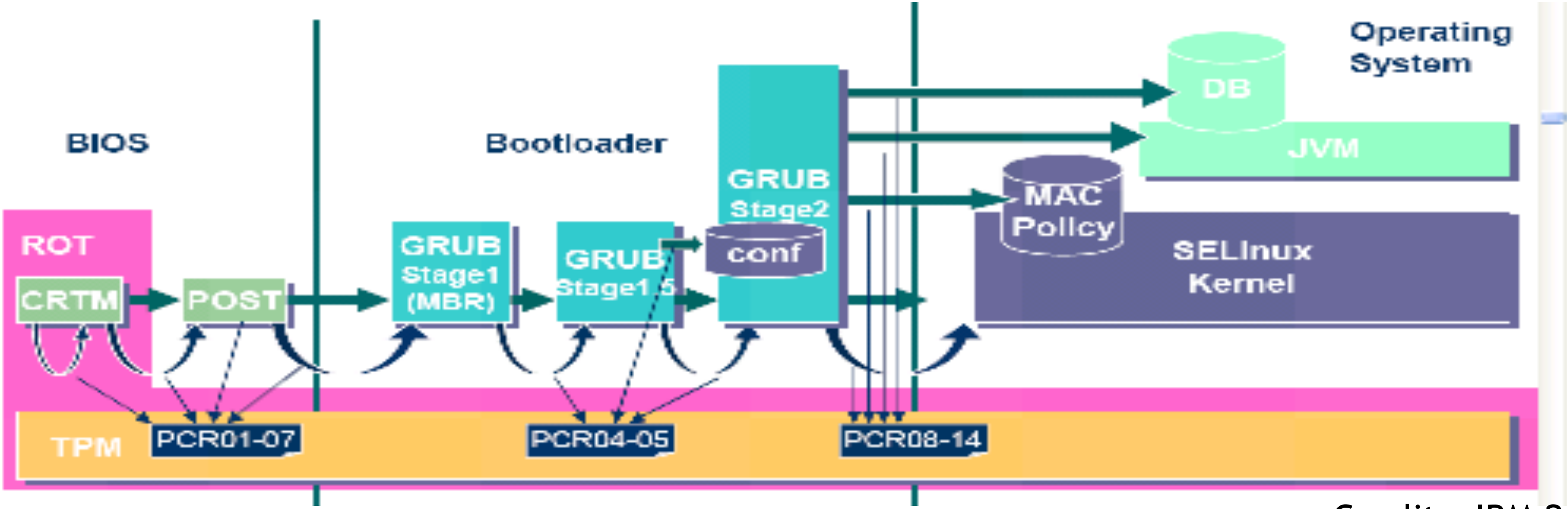
→ measuring

→ Extend PCR

After boot, PCRs contain hash chains of booted software

Collision resistance of SHA256 ensures commitment

Example: Trusted GRUB



Credit: IBM 2005

PCR # to use and what to measure is specified in GRUB config file

The main point

After boot completes, PCR registers measure the entire software stack that booted on the machine:

- BIOS and hardware configuration
- Boot loader and its configuration
- Operating system
- Running apps

Trusted Computing

Using PCRs after boot

Using PCRs after boot

Application: **encrypted (a.k.a sealed) storage.**

Setup step 1: TPM_TakeOwnership(OwnerPassword, …)

- Creates 2048-bit RSA Storage Root Key (SRK) on TPM
- Cannot run TPM_TakeOwnership again without OwnerPwd:
  - Ownership Enabled Flag ⟵ False
- Done once by IT department or laptop owner.

(optional) Step 2: TPM_CreateWrapKey / TPM_LoadKey

- Create more RSA keys on TPM protected by SRK
- Each key identified by 32-bit keyhandle

Implementing Protected Storage

TPM_Seal: Encrypt data using RSA key on TPM. (some) Arguments:

- keyhandle: which TPM key to encrypt with
- KeyAuth: Password for using key `keyhandle’
- PcrValues: PCRs to embed in encrypted blob (named by PCR num.)
- data block: at most 256 bytes [e.g. an AES key]

Returns encrypted blob.

Main point: blob can only be decrypted with TPM_Unseal when **PCR-reg-vals = PCR-vals** in blob. TPM_Unseal fails otherwise

Protected Storage

Embedding PCR values in blob ensures that only specific apps can decrypt data.

– Changing MBR or OS kernel will change PCR values

⇒ data cannot be decrypted

Sealed storage: applications

Lock software on machine:

- Suppose OS and apps are sealed with MBRs PCR value
- Any changes to MBR will prevent sealed OS from loading
- Prevents modifying or inspecting OS (or loading other OS)

Web server: seal server's SSL private key

- Goal: only unmodified Apache can access SSL key
- Problem: updates to Apache or Apache config

Example: BitLocker drive encryption

tpm.msc: utility to manage TPM (e.g TakeOwnership)

- Auto generates 160-bit OwnerPassword
- Stored on TPM and in file computer_name.tpm

Volume Master Key (VMK) encrypts disk volume key

- VMK is sealed (encrypted) under TPM SRK using
  - BIOS, extensions, and optional ROM (PCR 0 and 2)
  - Master boot record (MBR) (PCR 4)
  - NTFS Boot Sector and block (PCR 8 and 9)
  - NTFS Boot Manager (PCR 10), and
  - BitLocker Access Control (PCR 11)

BitLocker

Many options for **VMK** recovery: disk, USB, paper (enc. with pwd)

- Recovery needed after legitimate system change:
  - Moving disk to a new computer
  - Replacing system board containing TPM
  - Clearing TPM (with TPM_ForceClear)

At system boot (before OS boot)

- Optional: OS loader requests PIN or USB key from user
- TPM unseals VMK, only if PCR and PIN are correct

Trusted Computing

Attestation

Attestation: what it does

Goal: prove to remote party what software loaded on my machine

Good applications:

- Bank allows money transfer only if customer’s machine runs “up-to-date” OS patches
- Enterprise allows laptop to connect to its network only if laptop runs “authorized” software
- Gamers can join network only if their game client is unmodified

DRM: MusicStore sells content for authorized players only.

Attestation: how it works

Recall: EK private key on TPM.

- Cert for EK public-key issued by TPM vendor.

Step 1: Create Attestation Identity Key (AIK)

- Details not important here
- AIK Private key known only to TPM
- AIK public cert issued only if EK cert is valid

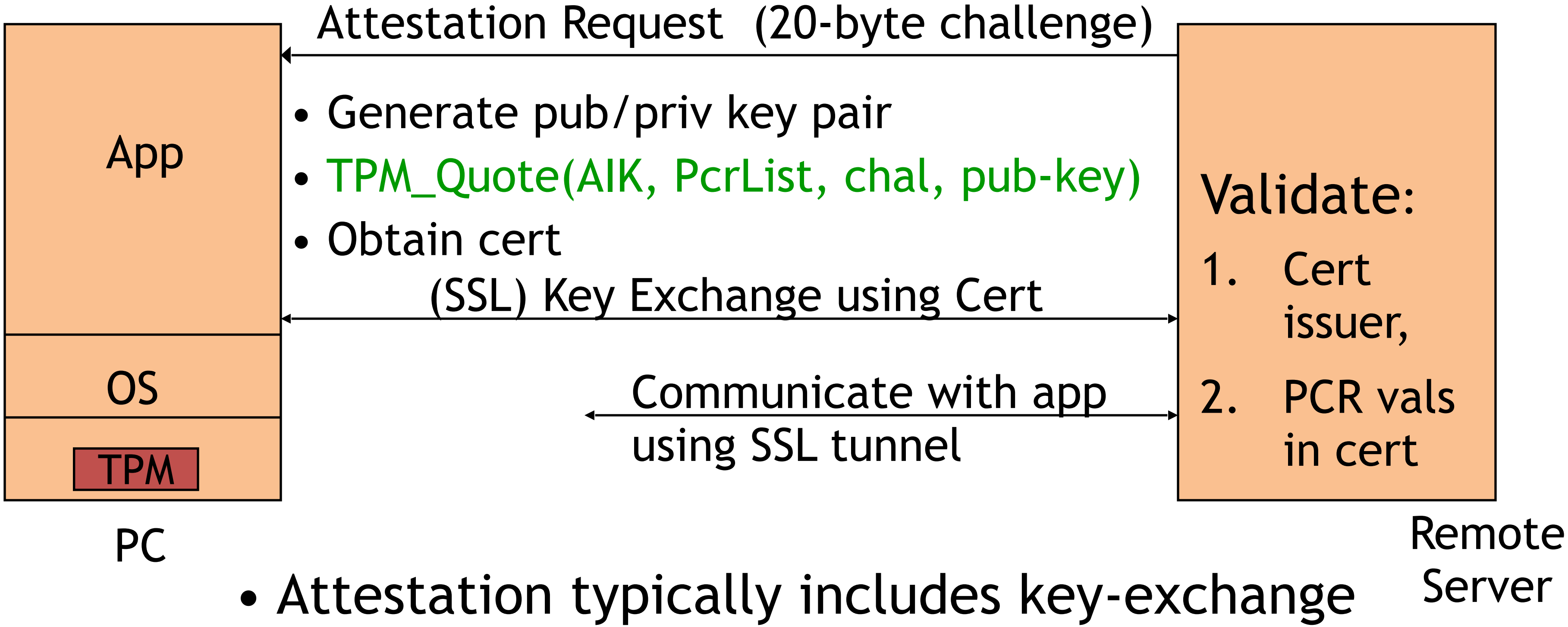
Attestation: how it works

Step2: sign PCR values (after boot) with TPM_Quote. Arguments:

- keyhandle: which AIK key to sign with
- KeyAuth: Password for using key `keyhandle’
- PCR List: Which PCRs to sign.
- Challenge: 20-byte challenge from remote server
  - Prevents replay of old signatures.
- Userdata: additional data to include in sig.

Returns signed data and signature.

Attestation: how it works



Attestation Request (20-byte challenge)

- Generate pub/priv key pair
- TPM_Quote(AIK, PcrList, chal, pub-key)
- Obtain cert

(SSL) Key Exchange using Cert

Communicate with app using SSL tunnel

App

OS

TPM

PC

Validate:

1. Cert issuer,
2. PCR vals in cert

Remote Server

- Attestation typically includes key-exchange
- App must be isolated from rest of system

Intel SGX

An overview

(Software Guard eXtensions)

The processor

Part of the trusted computing base (TCB):

- but is optimized for performance,
  … security may be secondary

Processor design and security:

- Important security features, such as hardware enclaves
- Some features can be exploited for attacks:
  - Speculative execution, transactional memory, …
  - An active area of research!

SGX: Goals

- Extension to Intel processors that support:
- **Enclaves**: running code and memory isolated from the rest of system
- **Attestation**: prove to local/remote system what code is running in enclave
- **Minimum TCB**: only processor is trusted
- nothing else: DRAM and peripherals are untrusted
- ⇒ all writes to memory are encrypted

Applications

- **Server side**:

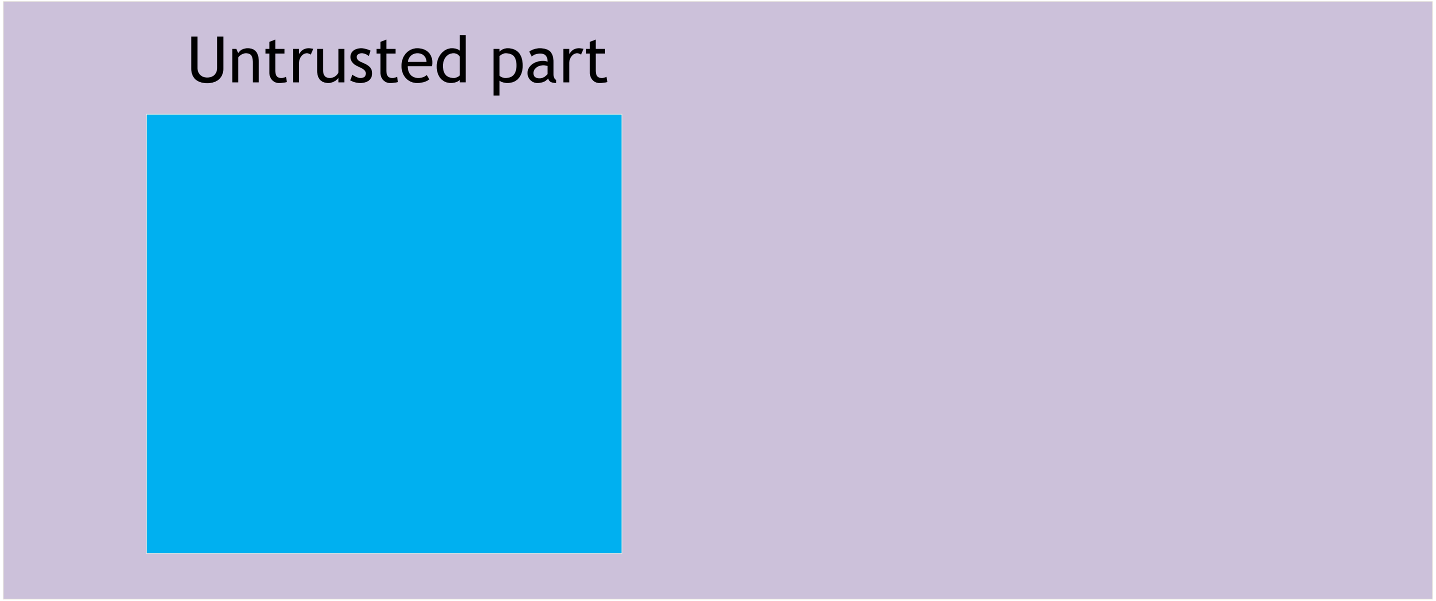
- Storing a Web server HTTPS secret key:
  - secret key only opened inside an enclave
  - ⇒ malware cannot get the key

- Running a private job in the cloud: job runs in enclave
  - Cloud admin cannot get code or data of job

- **Client side**:
- Hide anti-virus (AV) signatures:
  - AV signatures are only opened inside an enclave
  - not exposed to adversary in the clear

Intel SGX: how does it work?

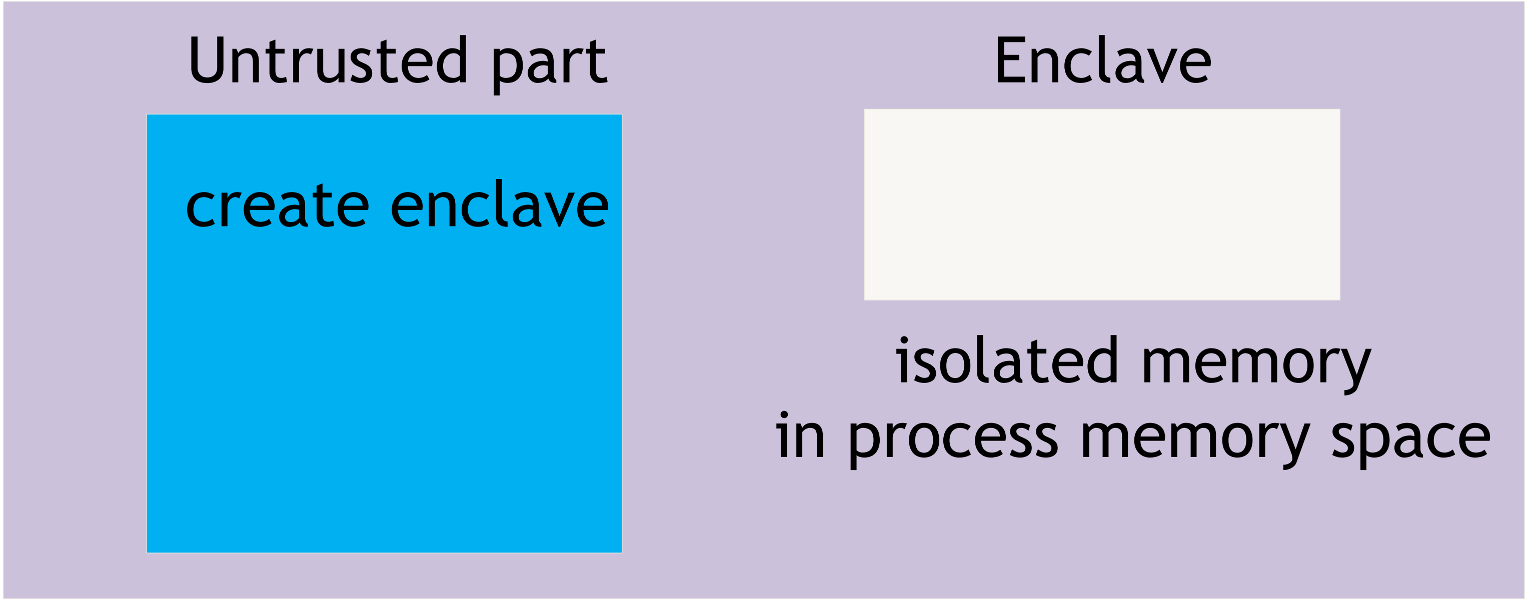
- An application defines part of itself as an enclave



Process memory

How does it work?

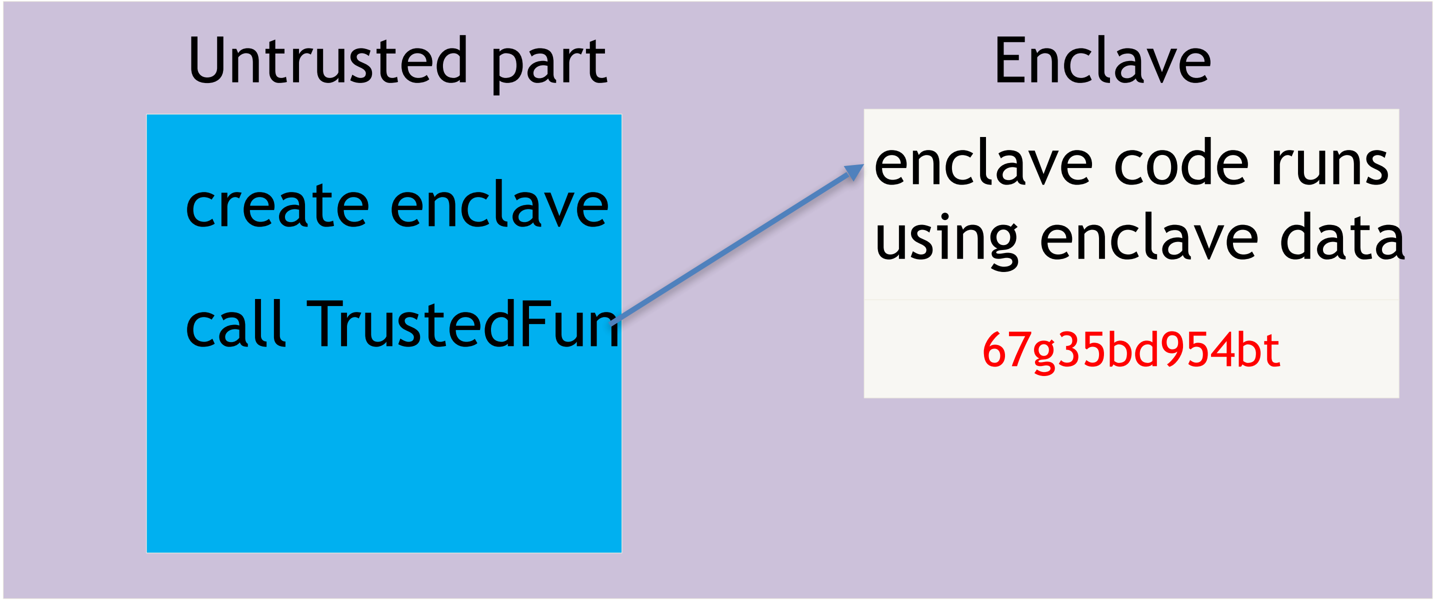
- An application defines part of itself as an enclave



Process memory

How does it work?

- An application defines part of itself as an enclave



Untrusted part

create enclave

call TrustedFun

Enclave

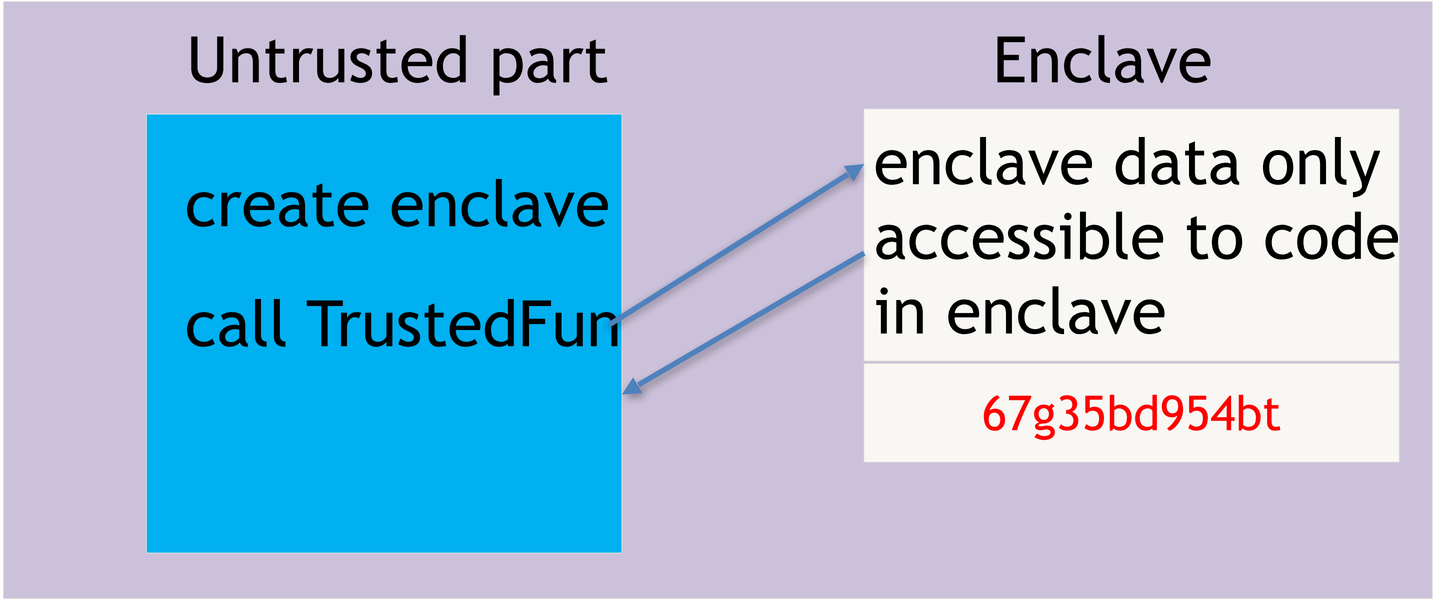
enclave code runs using enclave data

67g35bd954bt

Process memory

How does it work?

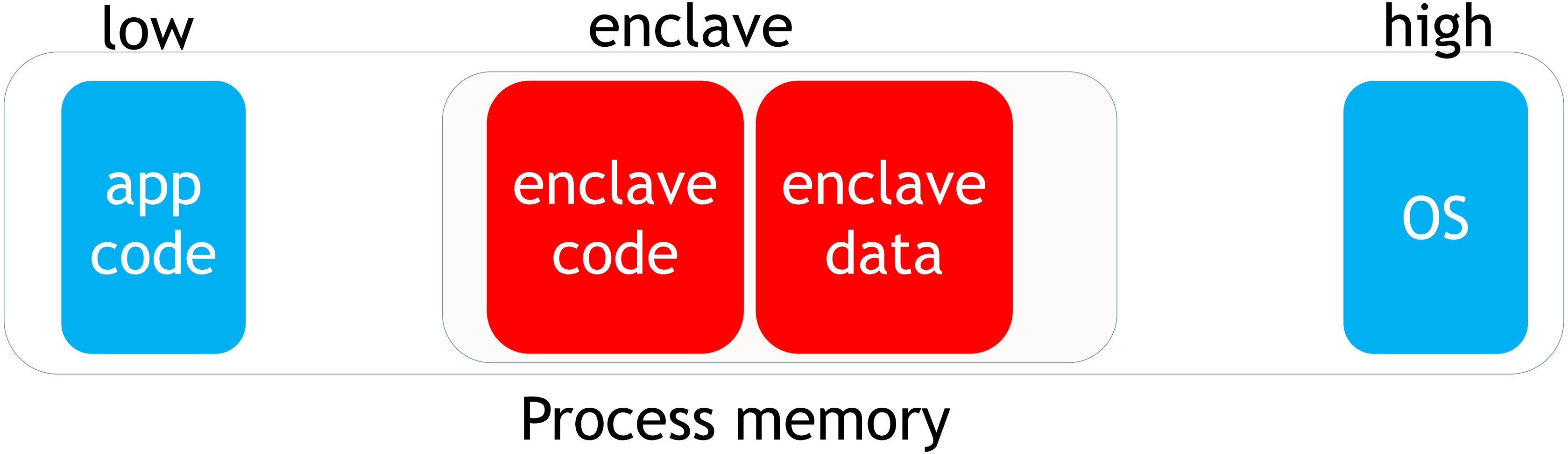
- An application defines part of itself as an enclave



Process memory

How does it work?

- Part of process memory holds the enclave:



- Enclave code and data are written encrypted to main memory
- Processor prevents access to cached enclave data outside of enclave.

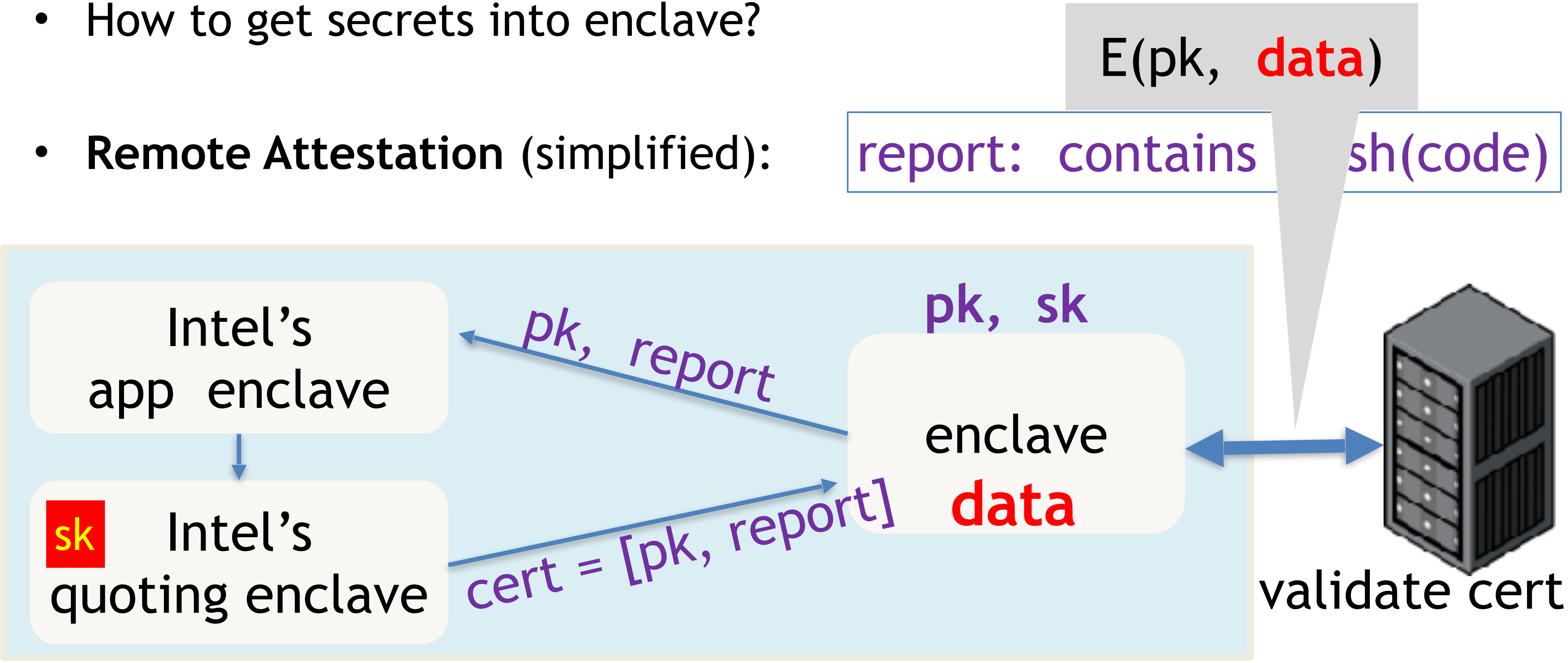
Creating an enclave: new instructions

- **ECREATE**: establish memory address for enclave
- **EADD**: copies memory pages into enclave
- **EEXTEND**: computes hash of enclave contents (256 bytes at a time)
- **EINIT**: verifies that hashed content is properly signed
  if so, initializes enclave (signature = RSA-3072)

- **EENTER**: call a function inside enclave
- **EEXIT**: return from enclave

Provisioning enclave with secrets: attestation

- The problem: enclave memory is in the clear prior to activation (EINIT)
- How to get secrets into enclave?
- **Remote Attestation** (simplified):



E(pk, data)

report: contains sh(code)

Intel's app enclave

sk Intel's quoting enclave

pk, report

cert = [pk, report]

pk, sk

enclave

data

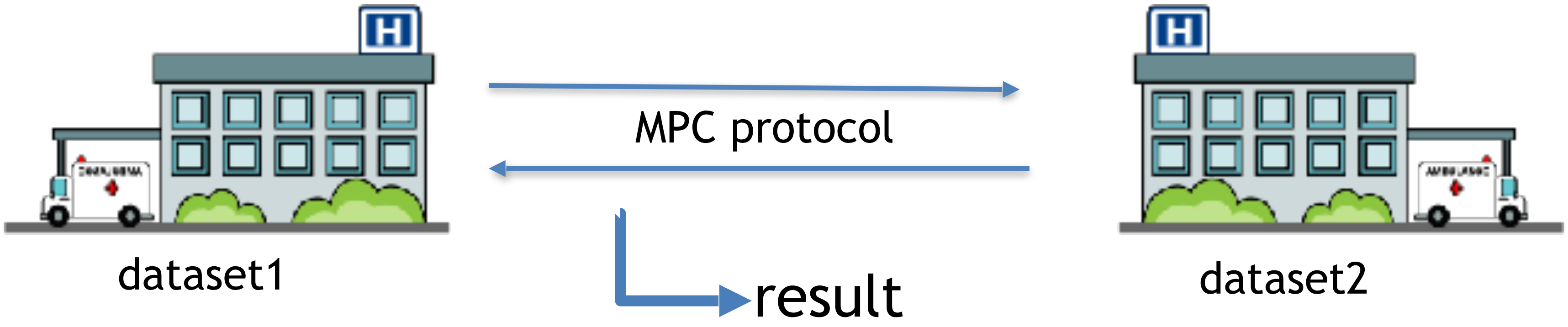
validate cert

Summary

- SGX: an architecture for managing secret data
- Intended to process data that cannot be read by anyone, except for code running in enclave
- Attestation: proves what code is running in enclave
- Minimal TCB: nothing trusted except for x86 processor
- Not suitable for legacy applications

An example application

Data science on federated data:



MPC protocol

dataset1

result

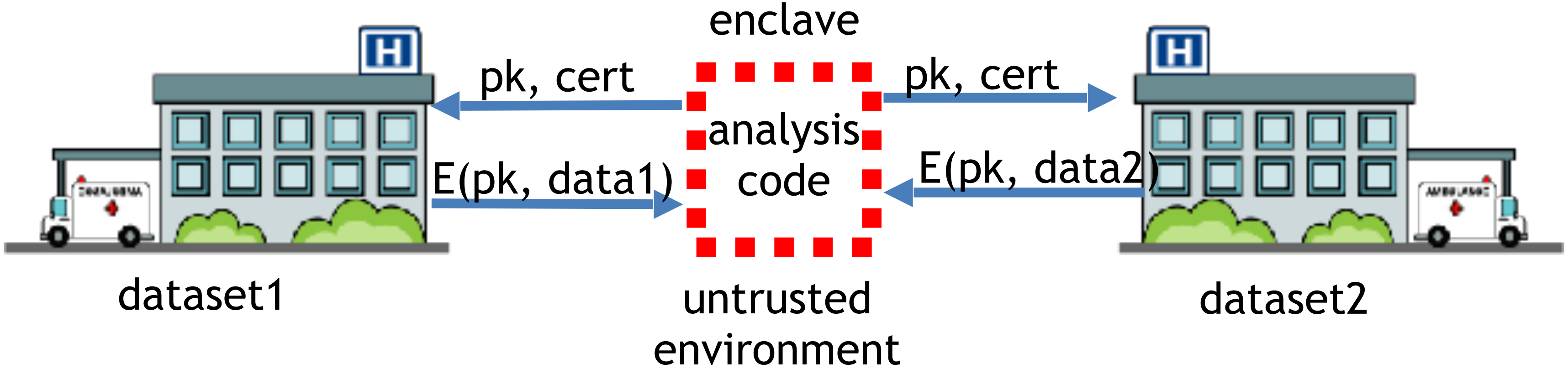
dataset2

Can we run analysis on union(dataset1, dataset2) ??

For simple computations, can use multiparty computation (MPC)

An example application

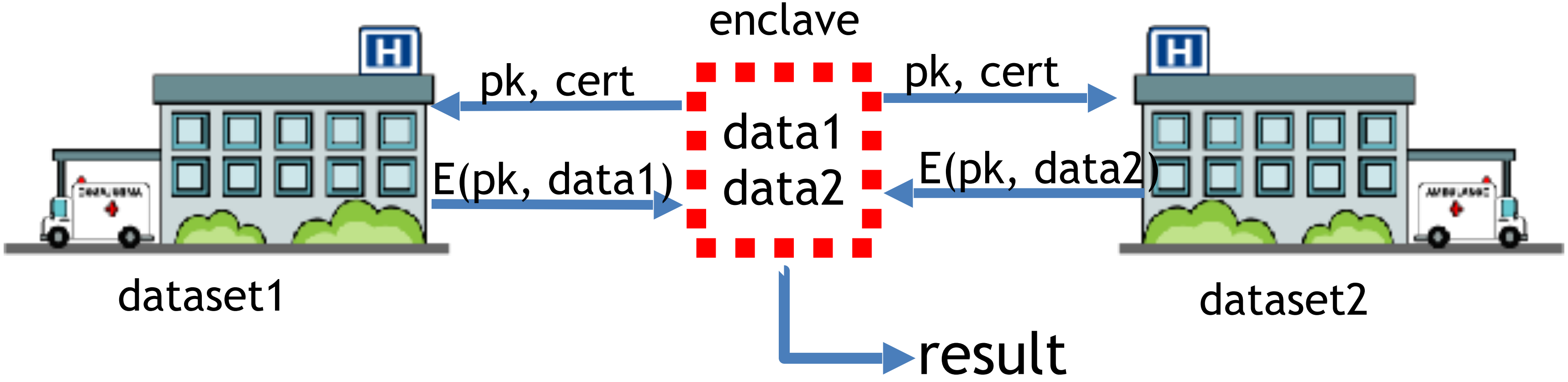
Data science on federated data:



For more complex analysis, can use (secure) hardware enclave

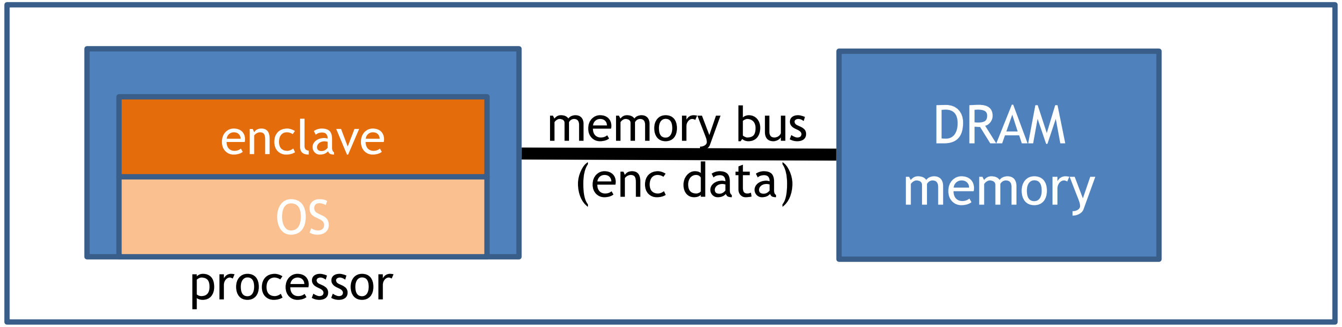
An example application

Data science on federated data:



For more complex analysis, can use (secure) hardware enclave

SGX insecurity: (1) side channels

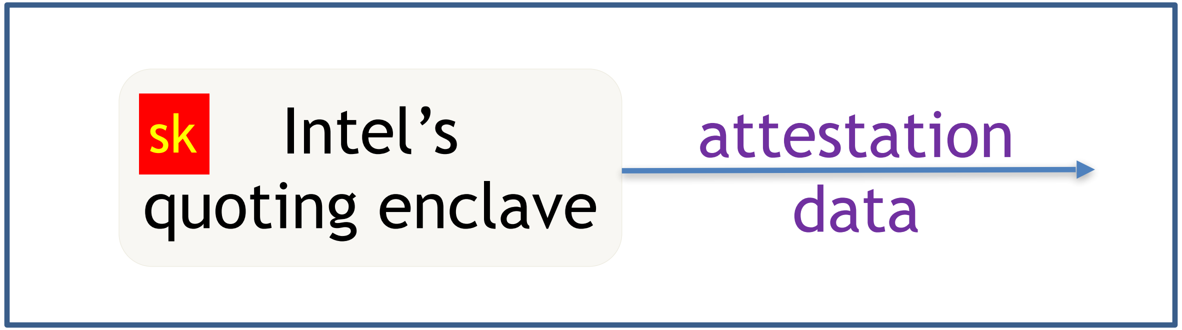


Attacker controls the OS. OS sees lots of side-channel info:

- Memory access patterns
- State of processor caches as enclave executes
- State of branch predictor

All can leak enclave data. Difficult to block.

SGX insecurity: (2) extract quoting key



Attestation: proves to 3rd party what code is running in enclave

- Quoting **sk** stored in Intel enclave on untrusted machines

What if attacker extracts **sk** from <u>some</u> quoting enclave?

- Can attest to arbitrary non-enclave code

… see Foreshadow attack and Intel's response

The Spectre attack

Speed vs. security in HW

[slides credit: Paul Kocher

Performance drives CPU purchases

Clock speed maxed out:

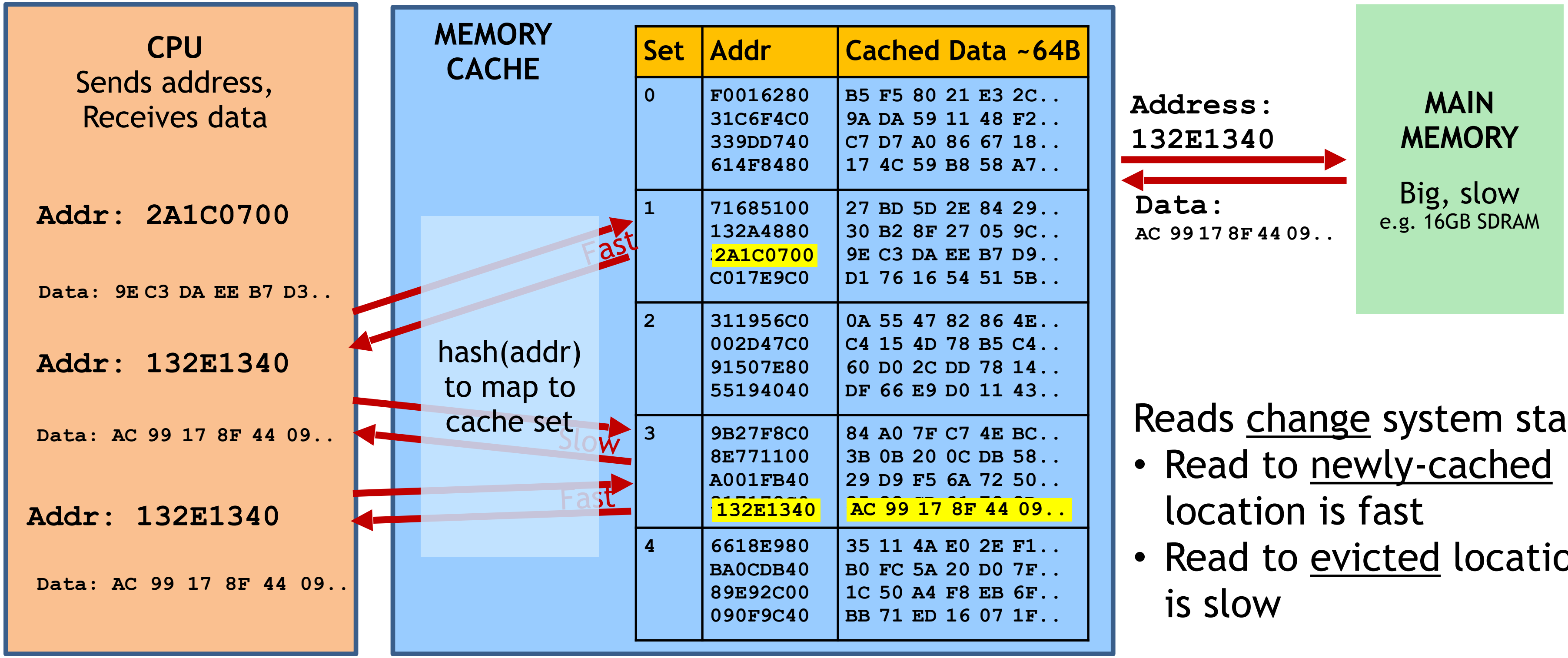
- Pentium 4 reached 3.8 GHz in 2004
- Memory latency is slow and not improving much

To gain performance, need to do more per cycle!

- Reduce memory delays → caches
- Work during delays → speculative execution

Memory caches (4-way associative)

Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



```
Addr:  2A1C0700

Data:  9E C3 DA EE B7 D3..

Addr:  132E1340

Data:  AC 99 17 8F 44 09..

Addr:  132E1340

Data:  AC 99 17 8F 44 09..
```

Set	Addr	Cached Data ~64B
0	F0016280 31C6F4C0 339DD740 614F8480	B5 F5 80 21 E3 2C.. 9A DA 59 11 48 F2.. C7 D7 A0 86 67 18.. 17 4C 59 B8 58 A7..
1	71685100 132A4880 2A1C0700 C017E9C0	27 BD 5D 2E 84 29.. 30 B2 8F 27 05 9C.. 9E C3 DA EE B7 D9.. D1 76 16 54 51 5B..
2	311956C0 002D47C0 91507E80 55194040	0A 55 47 82 86 4E.. C4 15 4D 78 B5 C4.. 60 D0 2C DD 78 14.. DF 66 E9 D0 11 43..
3	9B27F8C0 8E771100 A001FB40 132E1340	84 A0 7F C7 4E BC.. 3B 0B 20 0C DB 58.. 29 D9 F5 6A 72 50.. AC 99 17 8F 44 09..
4	6618E980 BA0CDB40 89E92C00 090F9C40	35 11 4A E0 2E F1.. B0 FC 5A 20 D0 7F.. 1C 50 A4 F8 EB 6F.. BB 71 ED 16 07 1F..

```
Address:
132E1340

Data:
AC 99 17 8F 44 09..
```

Reads change system state:

- Read to newly-cached location is fast
- Read to evicted location is slow

Speculative execution

CPUs can *guess* likely program path and do speculative execution

- Example:

```
if  (uncached_value == 1)      // load from memory
        a = compute(b)
```

- Branch predictor guesses if() is ‘true’ (based on prior history)
- Starts executing *compute(b)* speculatively

- When value arrives from memory, check if guess was correct:
  - **Correct:**   Save speculative work ⇒ performance gain
  - **Incorrect:**   Discard speculative work ⇒ no harm (?)

Architectural Guarantee

Register values eventually match result of in-order execution

Speculative Execution

CPU regularly performs incorrect calculations, then deletes mistakes

Is making + discarding mistakes the same as in-order execution?

The processor executed instructions that were not supposed to run !!

The problem: instructions can have observable side-effects

Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[ array1[x]*4096 ];
```

Suppose `unsigned int x` comes from untrusted caller

Execution without speculation is safe:

`array2[array1[x]*4096]` not eval unless `x < array1_size`

What about with speculative execution?

Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Before attack:

- Train branch predictor to expect if() is true (e.g. call with `x < array1_size`)
- Evict `array1_size` and `array2[]` from cache

Memory & Cache Status

```
array1_size = 00000008
```

Memory at `array1` base:

8 bytes of data (value doesn't matter)

Memory at `array1` base+1000:

`09 F1 98 CC 90...` (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]

 . . .
```

Contents don't matter
only care about cache *status*

Uncached Cached

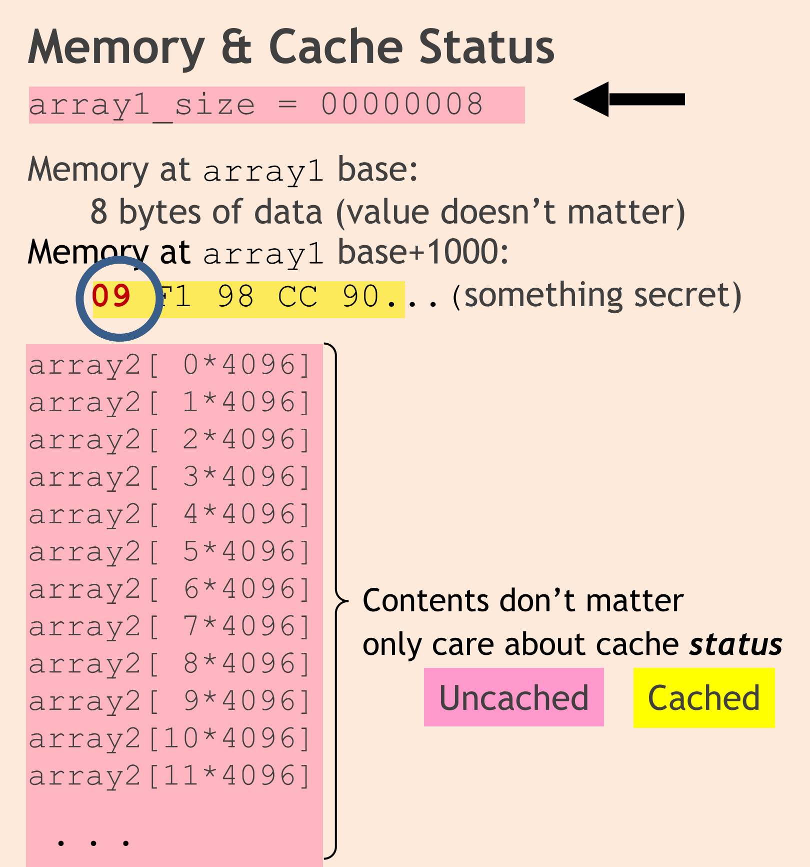
Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Speculative exec while waiting for array1_size:

- Predict that if() is true
- Read address (array1 base + x)
  (using out-of-bounds x=1000)
- Read returns secret byte = **09**
  (in cache ⇒ fast)



Memory & Cache Status

```
array1_size = 00000008
```

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
   . . .
```

Contents don't matter
only care about cache ***status***

Uncached Cached

Conditional branch (Variant 1) attack

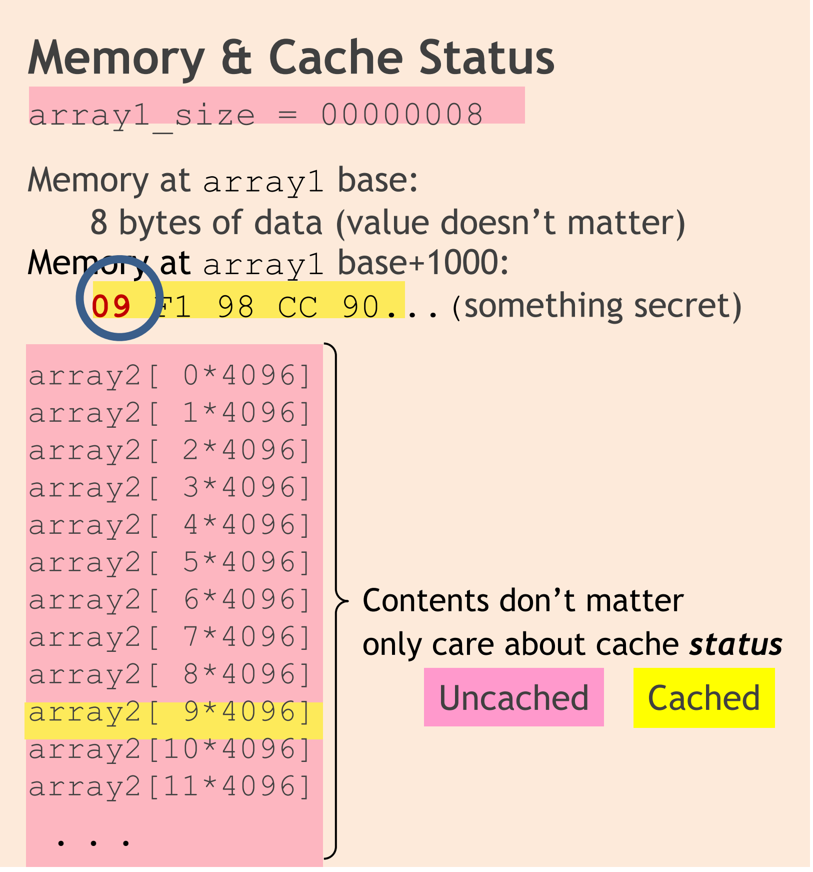
```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with x=1000

Next:

- Request mem at (array2 base + **09***4096)
- Brings array2[**09***4096] into the cache
- Realize if() is false: discard speculative work

Finish operation & return to caller



Memory & Cache Status

```
array1_size = 00000008
```

Memory at array1 base:

8 bytes of data (value doesn't matter)

Memory at array1 base+1000:

09 F1 98 CC 90... (something secret)

```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
  . . .
```

Contents don't matter only care about cache *status*

Uncached Cached

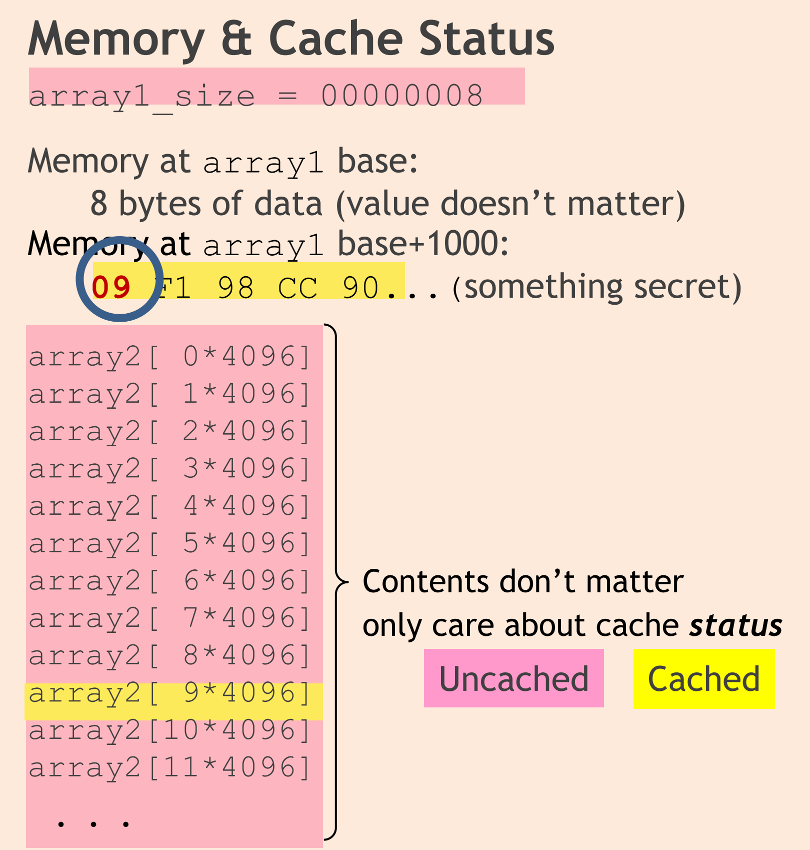
Conditional branch (Variant 1) attack

```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

Attacker calls victim with `x`=1000

Attacker:

- measures read time for `array2[i*`4096`]`
- Read for `i=`**09** is fast (cached),
  reveals secret byte !!
- Repeat with many `x`   (10KB/s)



Memory & Cache Status

`array1_size = 00000008`

Memory at `array1` base:
8 bytes of data (value doesn’t matter)

Memory at `array1` base+1000:
`09 F1 98 CC 90...` (something secret)

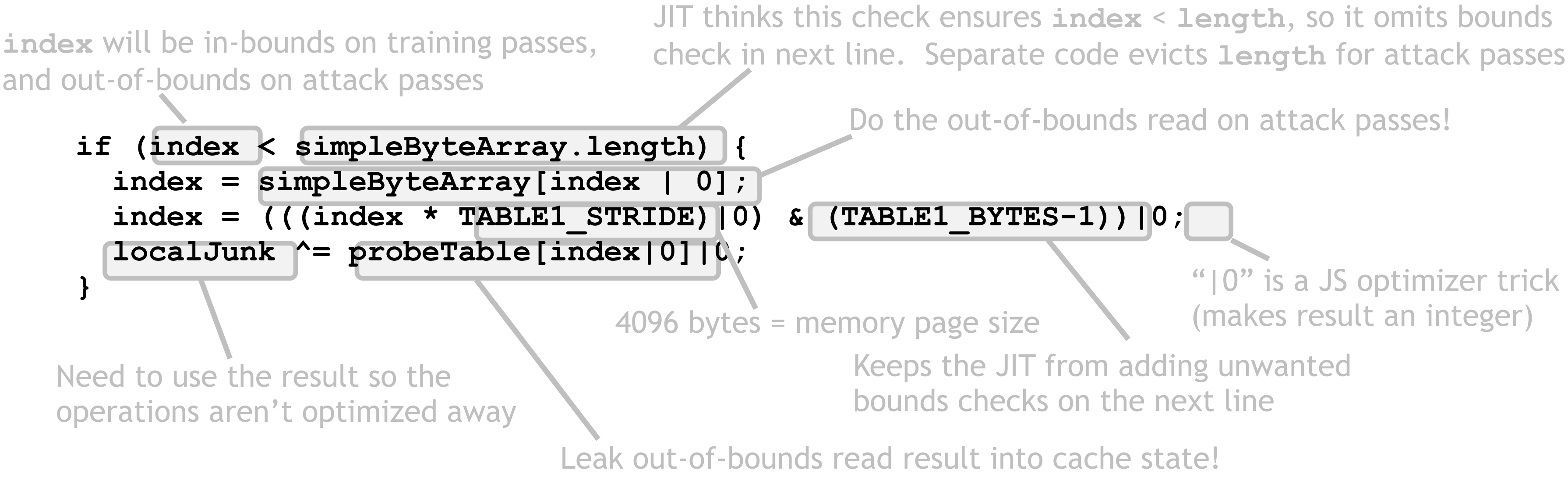
```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
  . . .
```

Contents don’t matter
only care about cache ***status***

Uncached   Cached

Violating JavaScript's sandbox

- Browsers run JavaScript from untrusted websites
  - JIT compiler inserts safety checks, including bounds checks on array accesses
- Speculative execution runs through safety checks…



```
if (index < simpleByteArray.length) {
  index = simpleByteArray[index | 0];
  index = (((index * TABLE1_STRIDE)|0) & (TABLE1_BYTES-1))|0;
  localJunk ^= probeTable[index|0]|0;
}
```

index will be in-bounds on training passes, and out-of-bounds on attack passes

JIT thinks this check ensures **index** < **length**, so it omits bounds check in next line. Separate code evicts **length** for attack passes

Do the out-of-bounds read on attack passes!

"|0" is a JS optimizer trick (makes result an integer)

4096 bytes = memory page size

Keeps the JIT from adding unwanted bounds checks on the next line

Need to use the result so the operations aren't optimized away

Leak out-of-bounds read result into cache state!

Can evict length/probeTable from JavaScript (easy)

… then use timing to detect newly-cached location in probeTable

Variant 2: indirect branches

Indirect branches: can go anywhere , e.g. `jmp[rax]`

- If destination is delayed, CPU guesses and proceeds speculatively
- Find an indirect jmp with attacker controlled register(s)
  … then cause mispredict to a useful 'gadget'

Attack steps:

- **Mistrain** branch prediction so speculative execution will go to gadget
- **Evict** address [rax] from cache to cause speculative execution
- **Execute** victim so it runs gadget speculatively
- **Detect** change in cache state to determine memory data

Non-mitigations

Can we prevent Spectre without a huge cost in performance?

Idea 1: fully restore cache state when speculation fails.

Problem: Insecure!

Speculative execution can have observable side effects beyond the cache state

```
if (x < array1_size) {
    y = array1[x];
    do_something_observable(y);
}
```

occupy a bus:
detectable from
another core,
or cause EM radiation

Variant 1 mitigation: Speculation stopping instruction (e.g. **LFENCE**)

- Idea: insert **LFENCE** on all vuln. code paths

```
if (x < array1_size)
    LFENCE          // processor instruction
    y = array2[ array1[x]*4096 ];
```

Variant 1 mitigation: Speculation stopping instruction (e.g. `LFENCE`)

- Claim: efficient, no performance impact on benchmark software

Insert LFENCEs manually?	Often millions of control flow paths Too confusing - speculation runs 188++ instructions, crosses modules Too risky – miss one and attacker can read entire process memory
Put LFENCES everywhere?	Abysmal performance - LFENCE is very slow Not in binary libraries, compiler-created code patterns
Insert by smart compiler?	Protect only known-bad bad patterns = unsafe – Microsoft Visual C/C++ `/Qspectre` unsafe for 13 of 15 tests ⇒ Protect all potentially-exploitable patterns

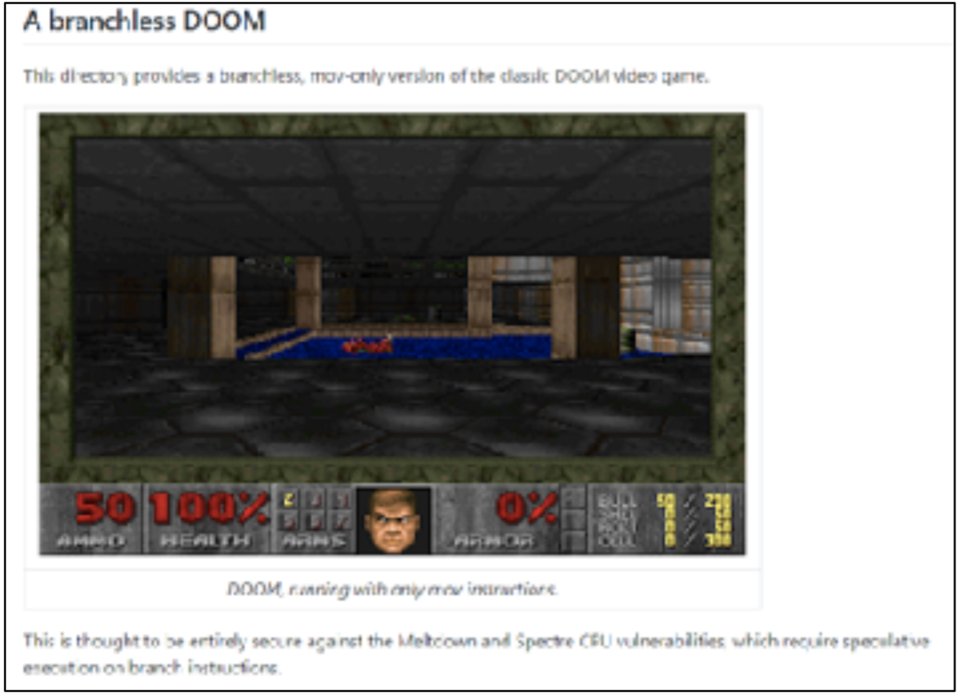
Transfer of blame (CPU -> SW): “you should have put an `LFENCE` there”

Mitigations: Indirect branch variant

Remove all branches?

DOOM with no branches:

- One frame every ~7 hours



A branchless DOOM

This directory provides a branchless, mov-only version of the classic DOOM video game.

DOOM, running with only mov instructions.

This is thought to be entirely secure against the Meltdown and Spectre CPU vulnerabilities, which require speculative execution on branch instructions.

Oops!   Variant 4: speculative store

Mitigations: summary

Mitigations are messy for all Spectre variants:

- Software must deal with microarchitectural complexity
- Mitigations for all variants are really hard to test:
  - formal models beginning to appear

More ideas desperately needed !

… but there is more

More speculative execution attacks:

- **Meltdown**
- Rogue inflight data load (**RIDL**) and **Fallout**
- **ZombieLoad**
- **Store-to-leak forwarding**

Enable reading unauthorized memory (client, cloud, SGX)

- Mitigating incurs significant performance costs

How to evaluate a processor?

Processors are measured by their performance on benchmarks:

- Processor vendors add <u>many</u> architectural features to speed-up benchmarks
- Until recently: security implications were secondary

⇒ lots of security issues found in last three years

… likely more will be found in coming years

THE END