CS155

Processor security

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, memory encryption (TME), RDRAND, and others.
- Some features can be exploited for attacks:
  - Speculative execution, transactional memory, …

Intel SGX / TDX

An overview

(Software Guard eXtensions)

SGX: Goals

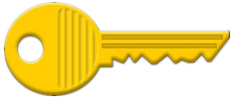
Extension to Intel processors that support:

- **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 (TME)

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

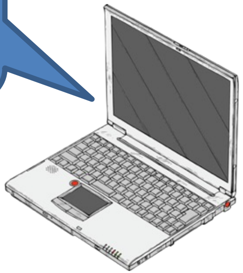
  Cloud admin cannot get code or data of job

Client side:

- Hide anti-virus (AV) signatures:

  AV signatures are only opened inside an enclave

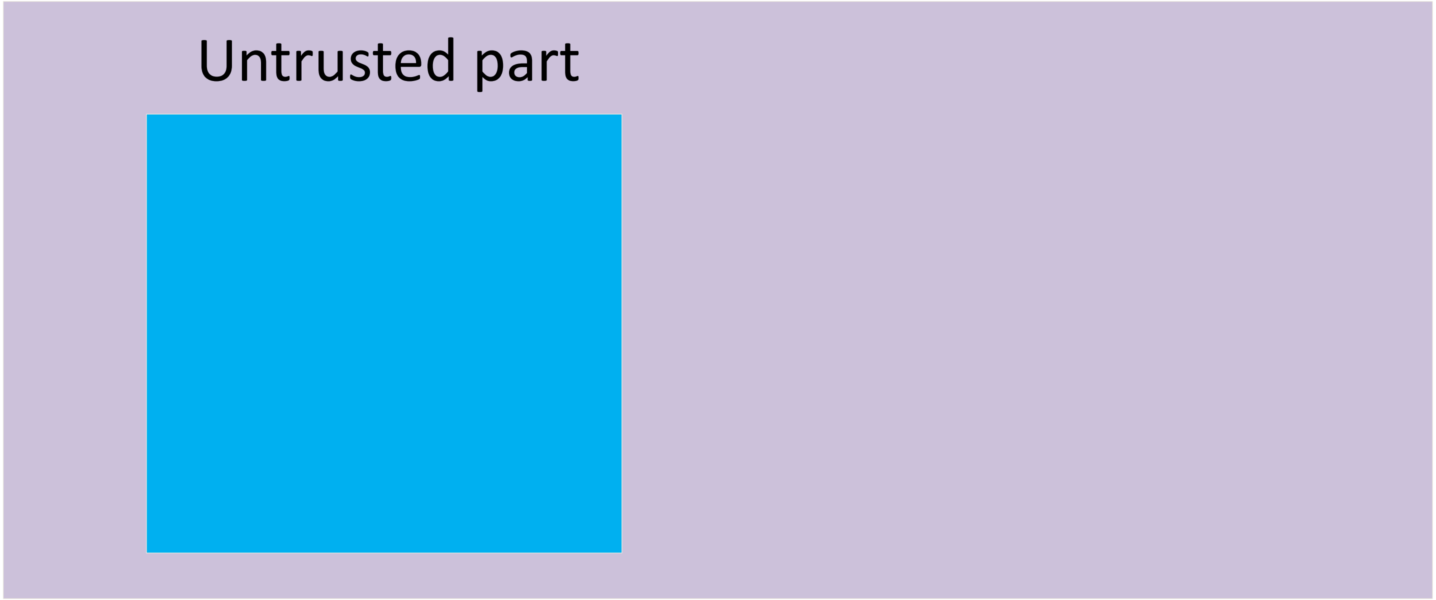
  not exposed to adversary in the clear



Deprecated in INTEL CORE processors

Intel SGX: how does it work?

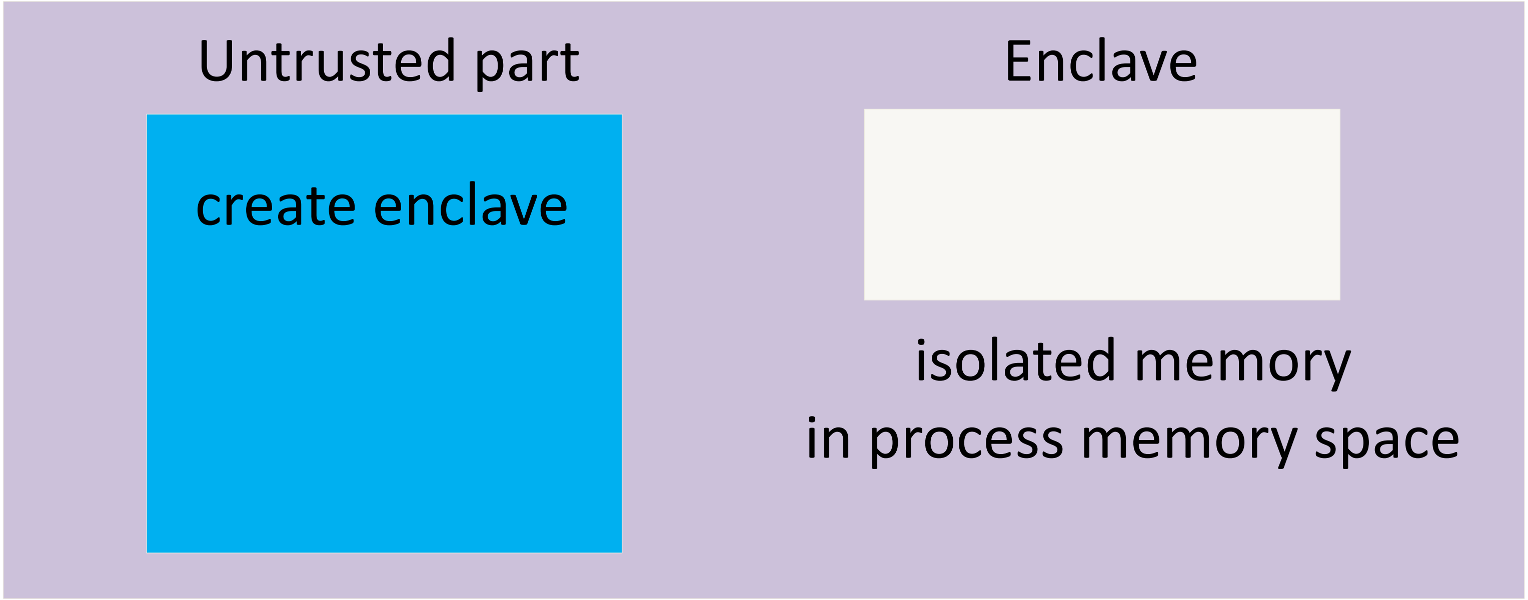
An application defines part of itself as an enclave



Process memory

How does it work?

An application defines part of itself as an enclave



Untrusted part

create enclave

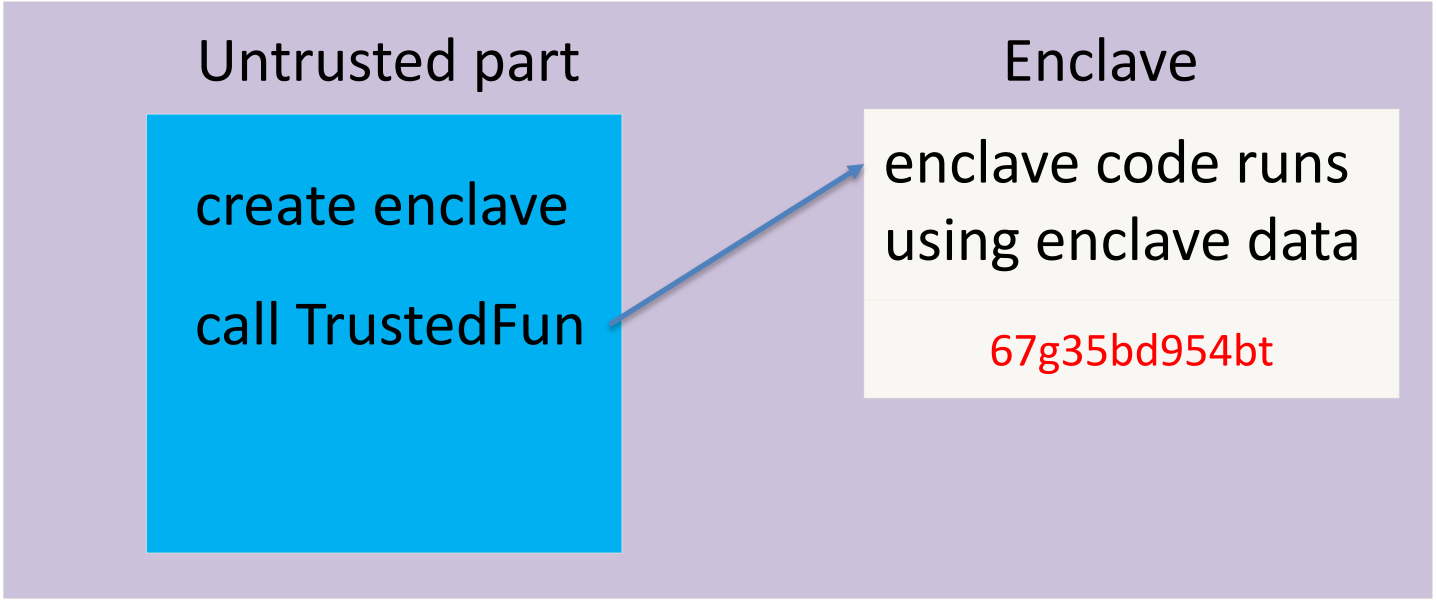
Enclave

isolated memory
in process memory space

Process memory

How does it work?

An application defines part of itself as an enclave



Untrusted part

create enclave

call TrustedFun

Enclave

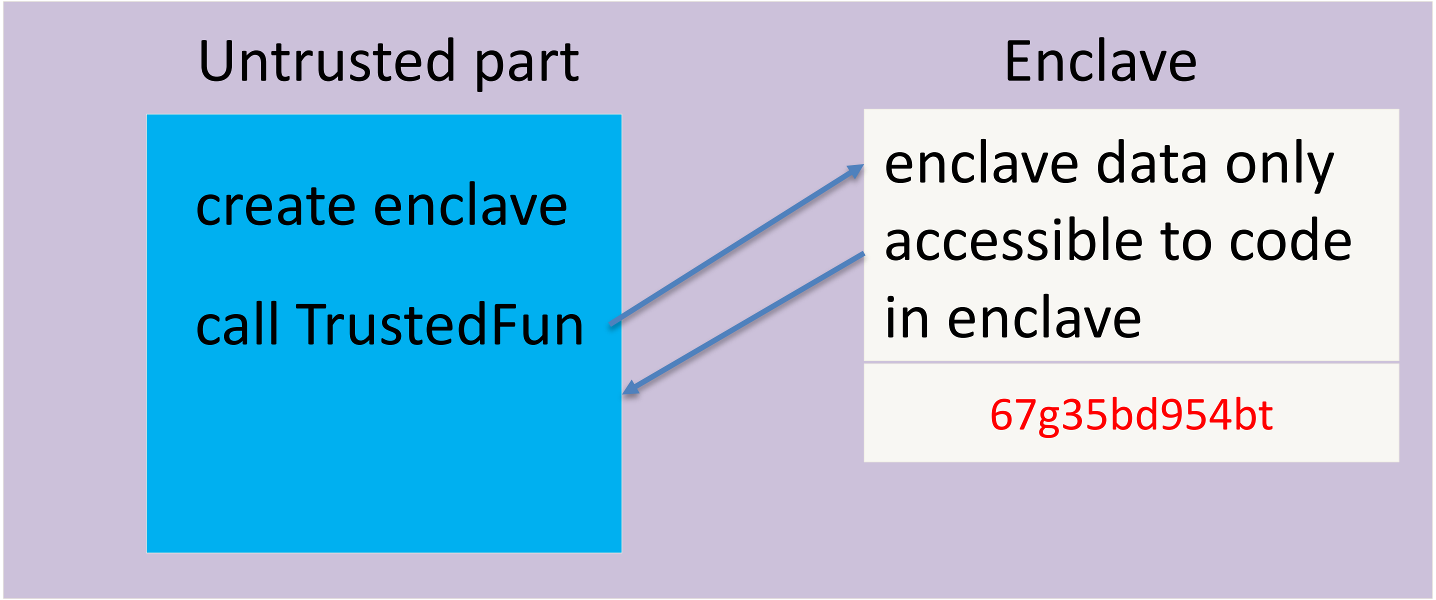
enclave code runs using enclave data

67g35bd954bt

Process memory

How does it work?

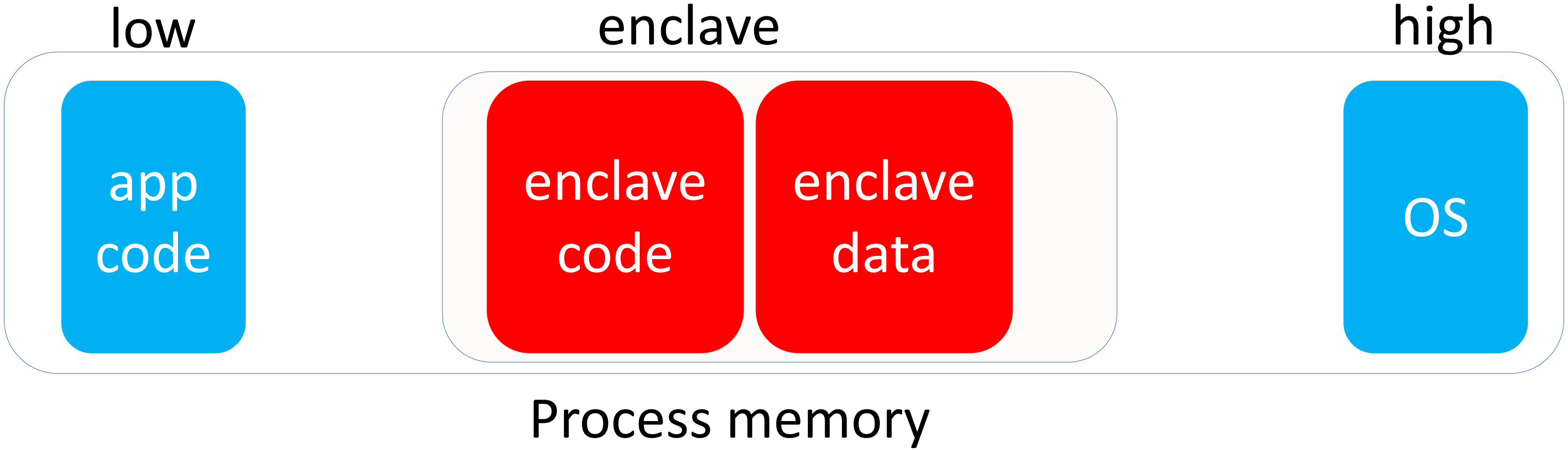
An application defines part of itself as an enclave



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

Enclave init code loaded as cleartext

- **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)

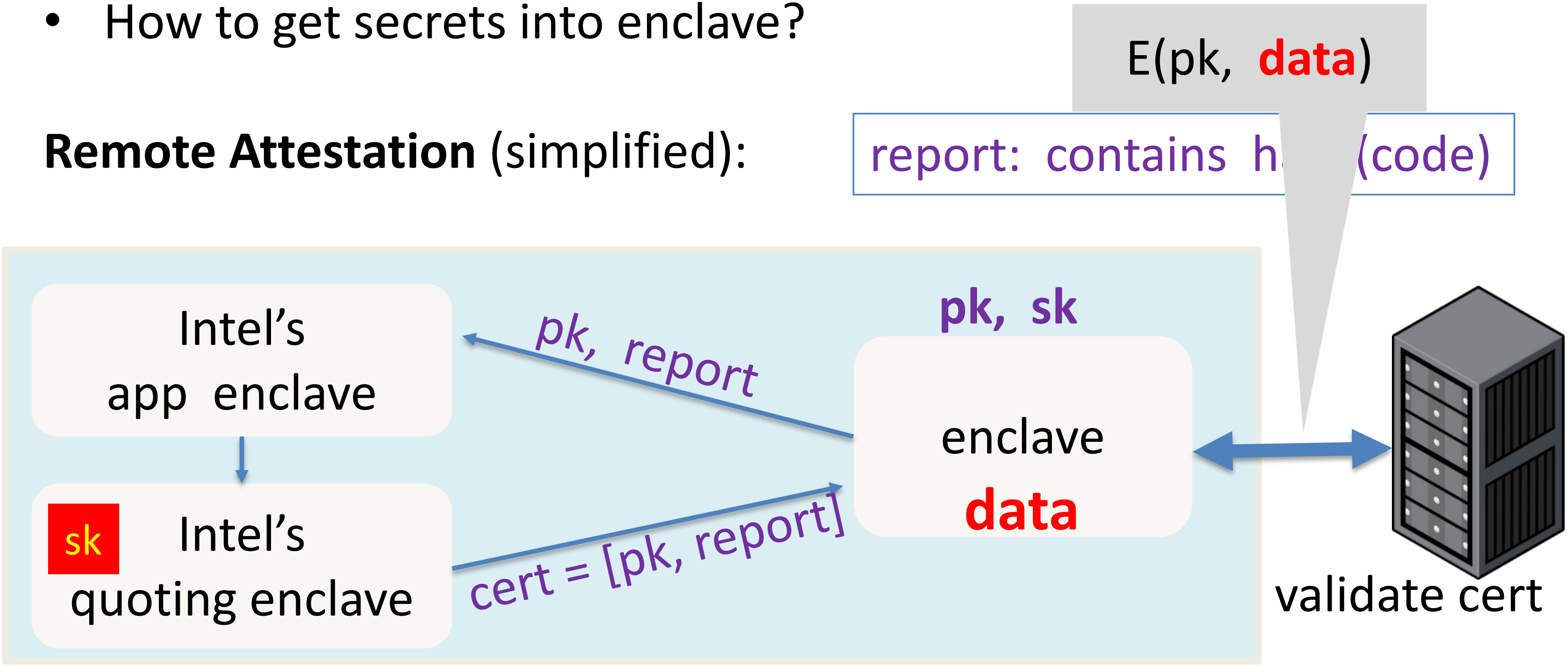
- **EENTER**: call a function inside enclave
- **EEXIT**: return from enclave

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):



Summary

SGX: an architecture for managing secret data

- Intended to process data that cannot be read by anyone, except for code running in enclave

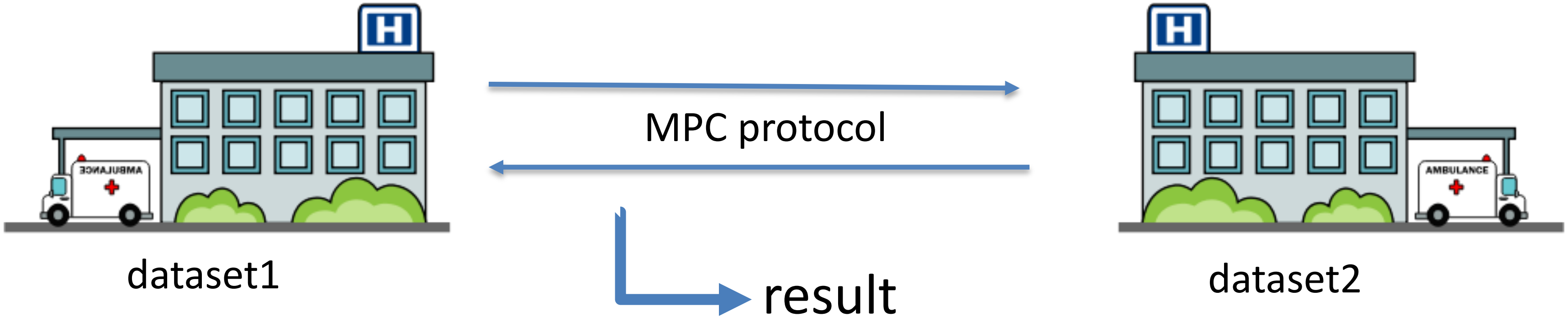
- Attestation: proves what code is running in enclave

- Minimal TCB: nothing trusted except for x86 processor

- Not suitable for legacy applications

An example application

Data science on federated data:

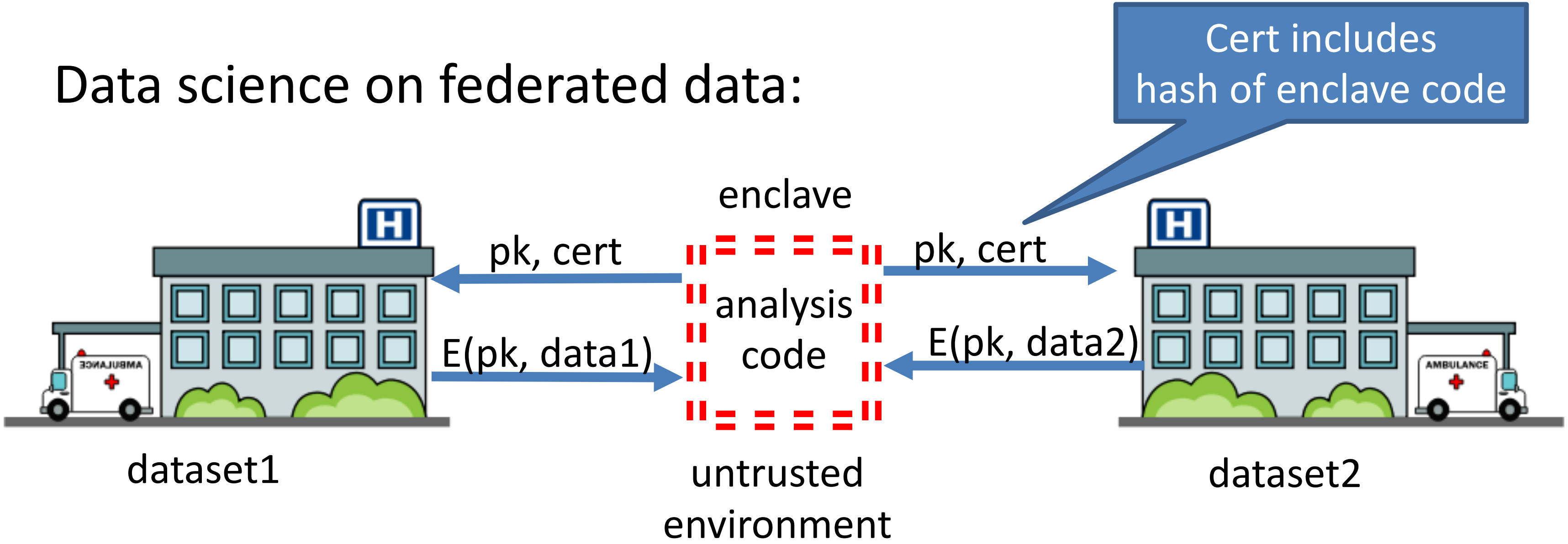


Can we run analysis on union(dataset1, dataset2) ??

For simple computations, can use multiparty computation (MPC)

An example application

Data science on federated data:



Cert includes hash of enclave code

enclave

pk, cert

pk, cert

analysis code

E(pk, data1)

E(pk, data2)

dataset1

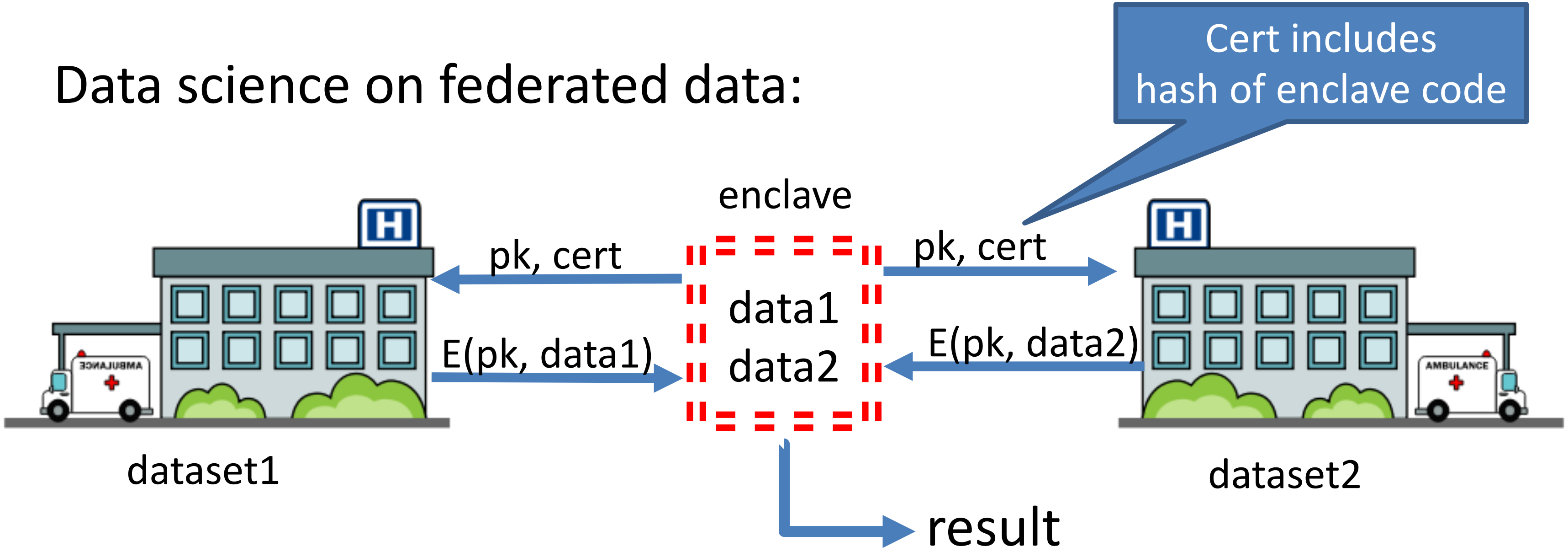
untrusted environment

dataset2

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

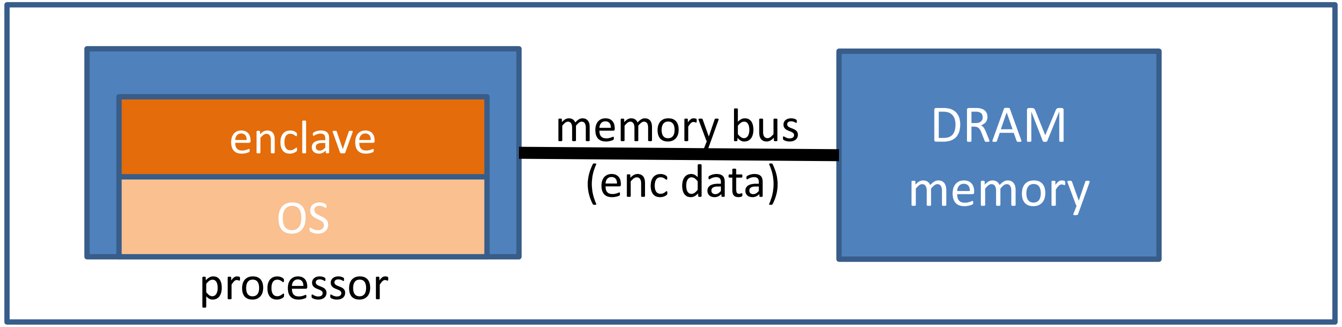
An example application

Data science on federated data:



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

SGX insecurity: (1) side channels

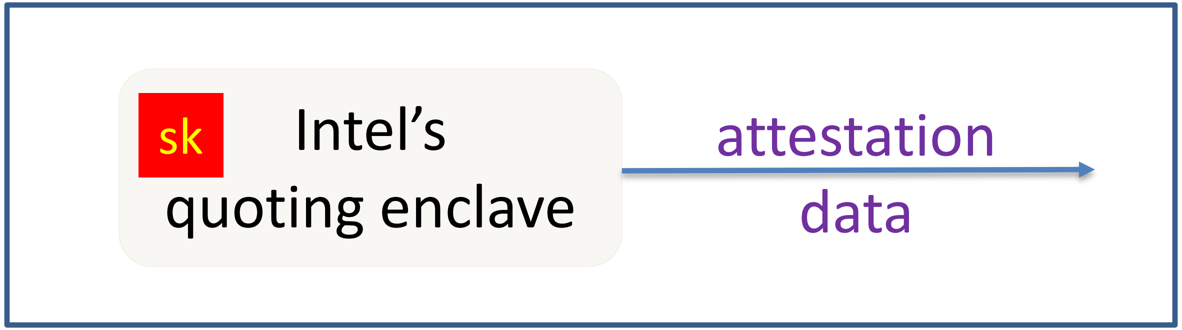


Attacker controls the OS. OS sees lots of side-channel info:

- 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:

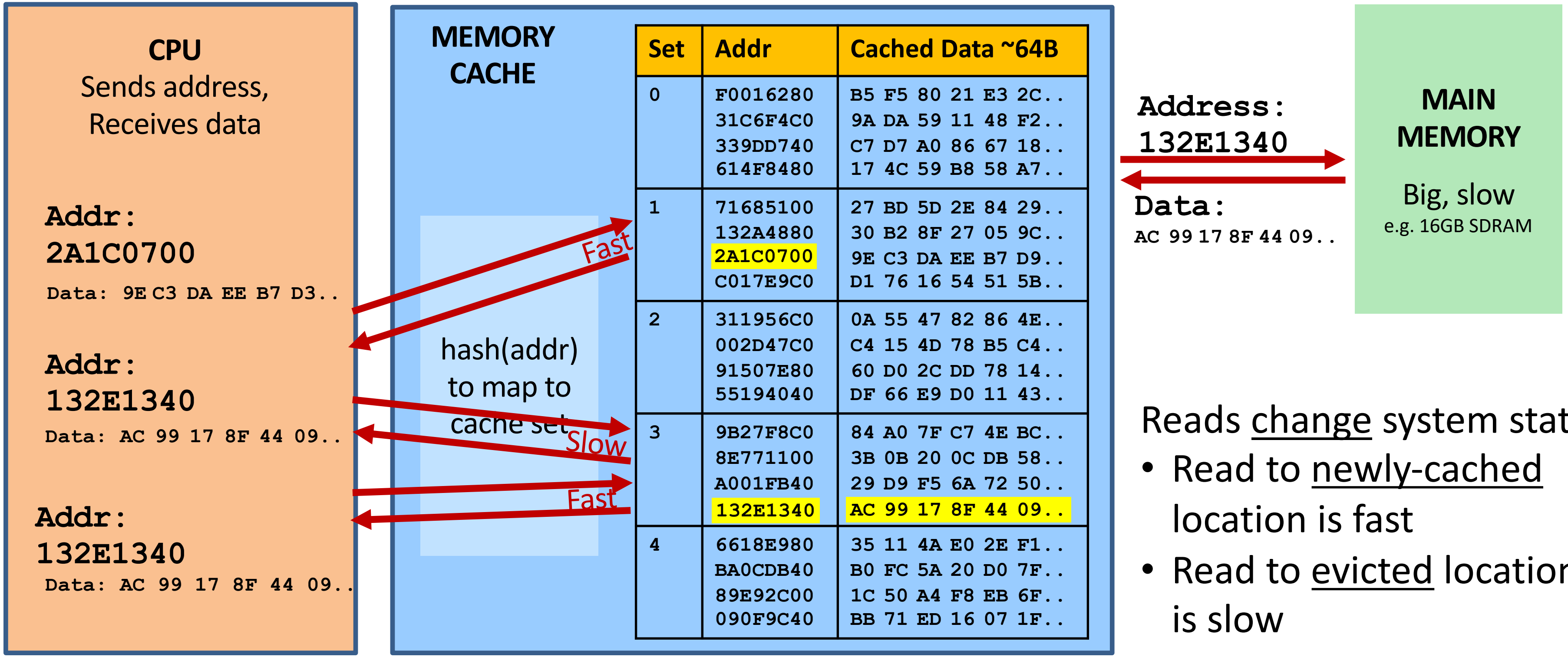
- Pentium 4 reached 3.8 GHz in 2004
- Memory latency is slow and not improving much

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



CPU
Sends address,
Receives data

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..

MEMORY CACHE

hash(addr) to map to cache set

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

Address: 132E1340

Data:
AC 99 17 8F 44 09..

MAIN MEMORY

Big, slow
e.g. 16GB SDRAM

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

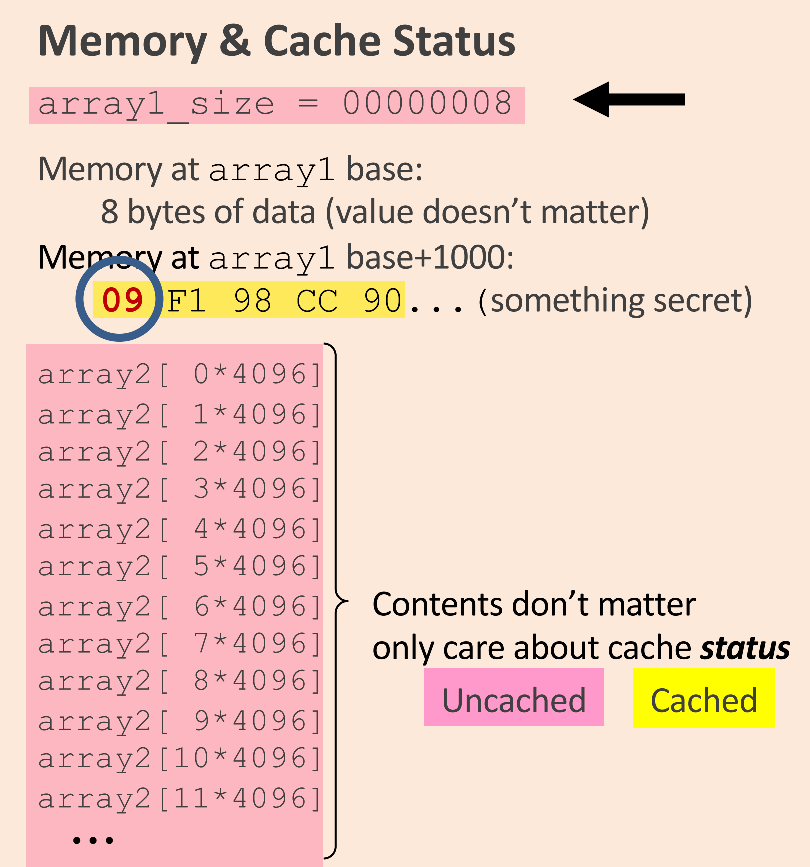
Conditional branch (Variant 1) attack

```
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

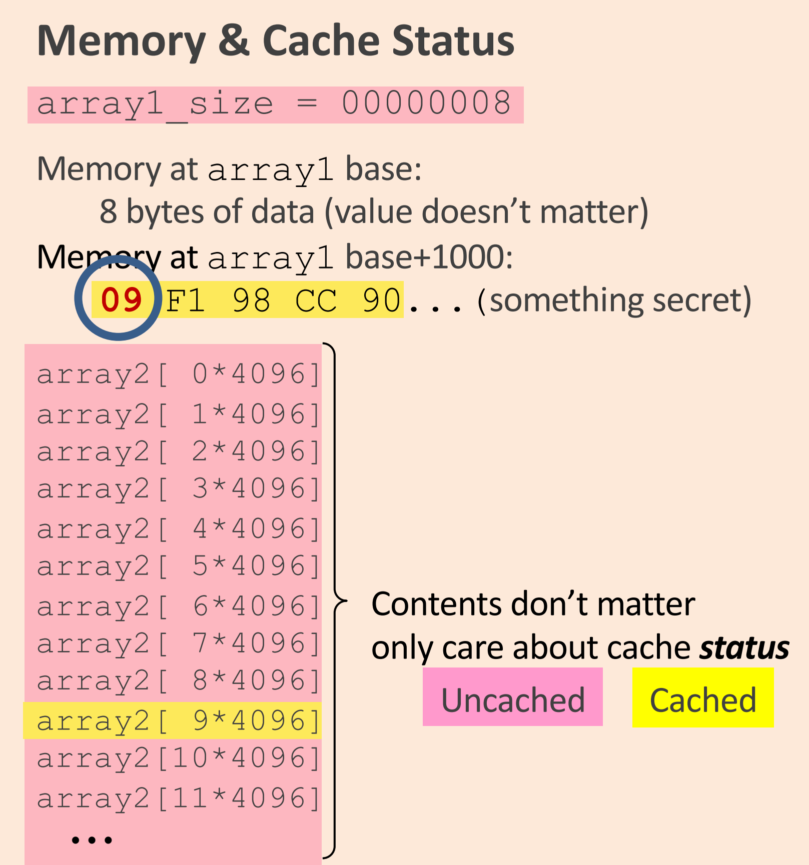
```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

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

proceed to next instruction



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

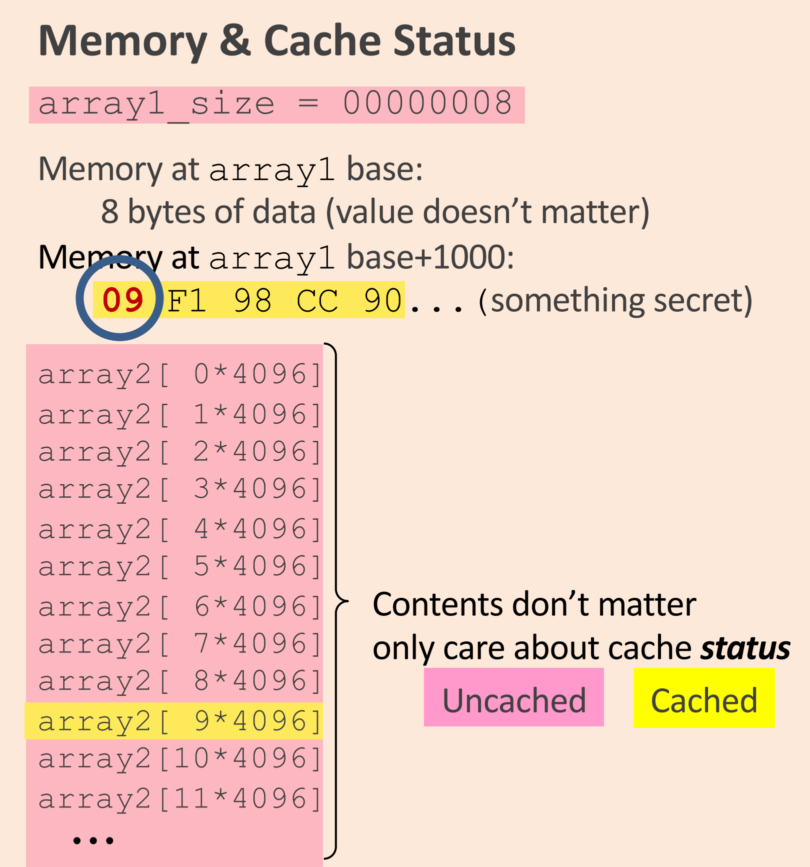
Conditional branch (Variant 1) attack

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

Attacker calls victim with `x`=1000

Attacker: (another process or core)

- `for i=0 to 255:`
  measure read time for `array2[i*4096]`
- When `i=09` read 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…

`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!

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

"|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' `y = array2[array1[x]*4096];`

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
✓ Insert by smart compiler?	Must protect against all potentially-exploitable patterns Supported in LLVM, along with other mitigations ⟹ protects all LLVM-based compilers

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

Mitigations: summary

Mitigations are messy for all Spectre variants:

- Software must deal with microarchitectural complexity
- Mitigations for all variants are really hard to test:
    - active area of research

More ideas needed !

… but there is more

More speculative execution attacks:

- **Meltdown**
- Rogue inflight data load (**RIDL**) and **Fallout**
- **ZombieLoad**
- **Store-to-leak forwarding**
- **Micro-op caches** (June 2020)

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 many architectural features to speed-up benchmarks

- Until recently: security implications were secondary

⇒ lots of security issues found in last four years

… likely more will be found in coming years

THE END