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: 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)
Trust Domain eXtensions (TDX)

Also AMD SME and SEV

Stanford | Center for Professional Development

SGX / TDX: Goals

Extension to Intel processors that support:

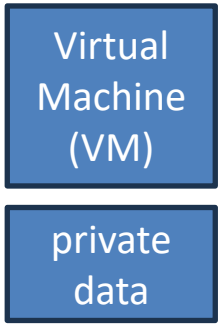
- **Enclaves**: running code and memory isolated from the rest of system (code outside of enclave cannot read enclave memory)
- **Attestation**: prove to a remote system what code is running in the enclave
- **Minimum TCB**: only processor is trusted, nothing else!

  RAM and peripherals are untrusted

  ⇒ Memory controller must encrypt all writes to RAM (TME)

Why enclaves 1: cloud computing

Client



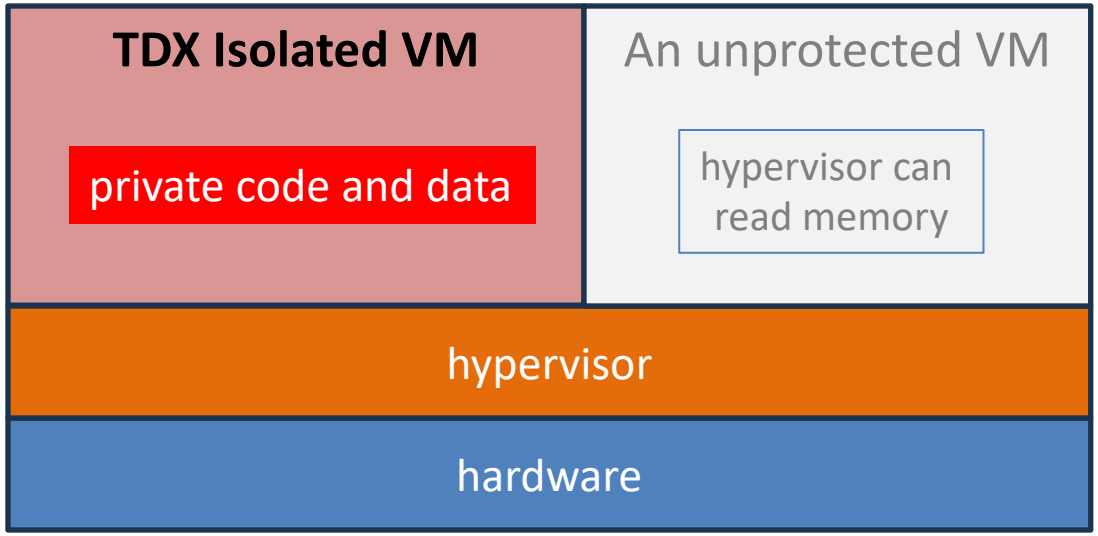
Cloud
(AWS, GCP, …)

Goal: move data & VM to the cloud, without cloud seeing them in the clear

- Simple solution: encrypt data and VM, key stays with Client
- The problem: now cloud cannot search or compute on data

  ⇒ defeats the purpose of cloud computing

A solution: a HW enclave

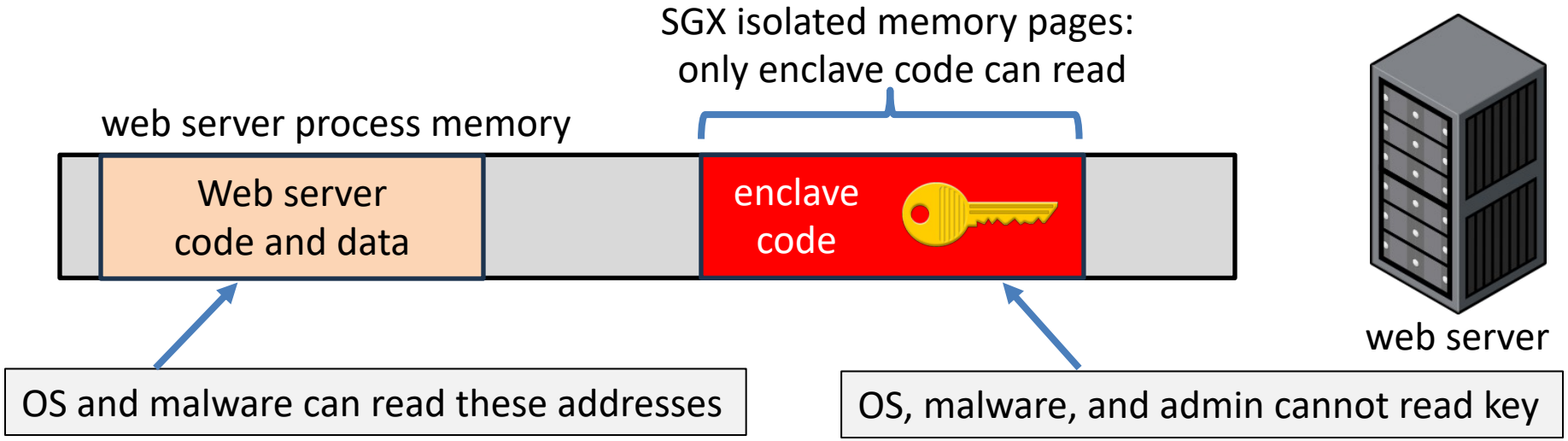


Cloud
(AWS, GCP, …)

Goal: no one can read the memory of the isolated VM;
not the hypervisor, and not even a malicious admin

How does this work? Will see in a minute.

Why enclaves 2: protecting keys



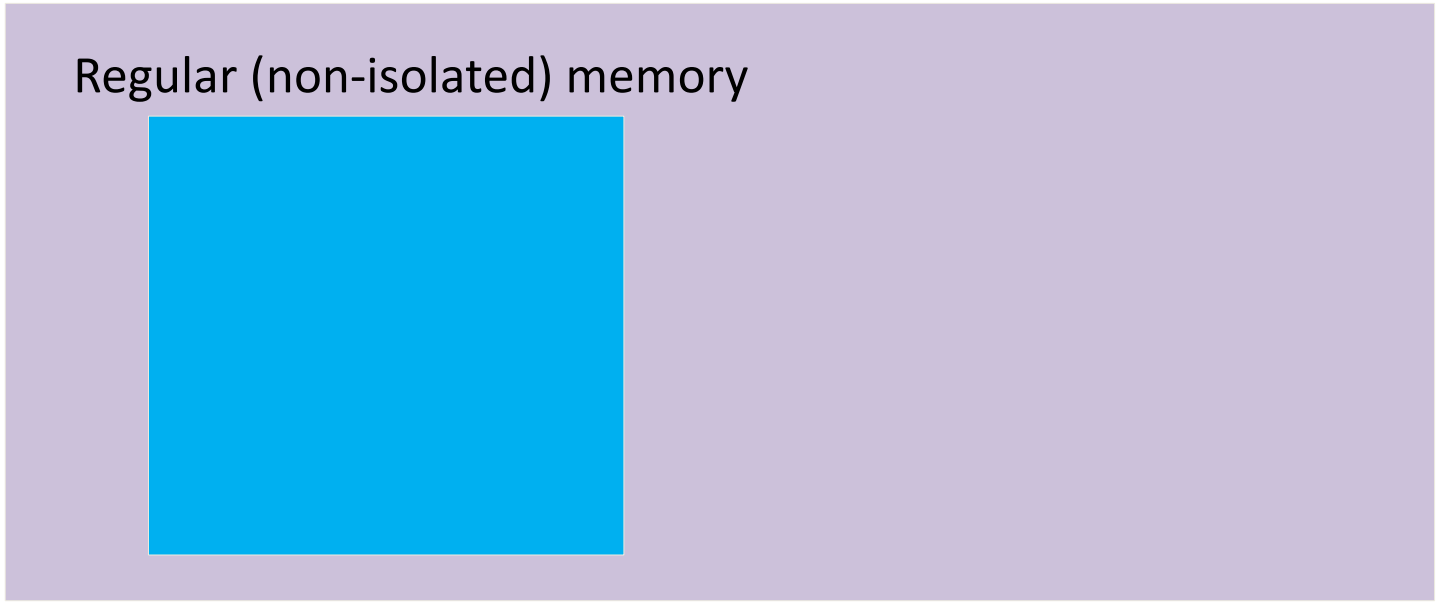
Storing a Web server HTTPS secret key:

secret key is only available in the clear inside an enclave

⇒ malware cannot extract the key

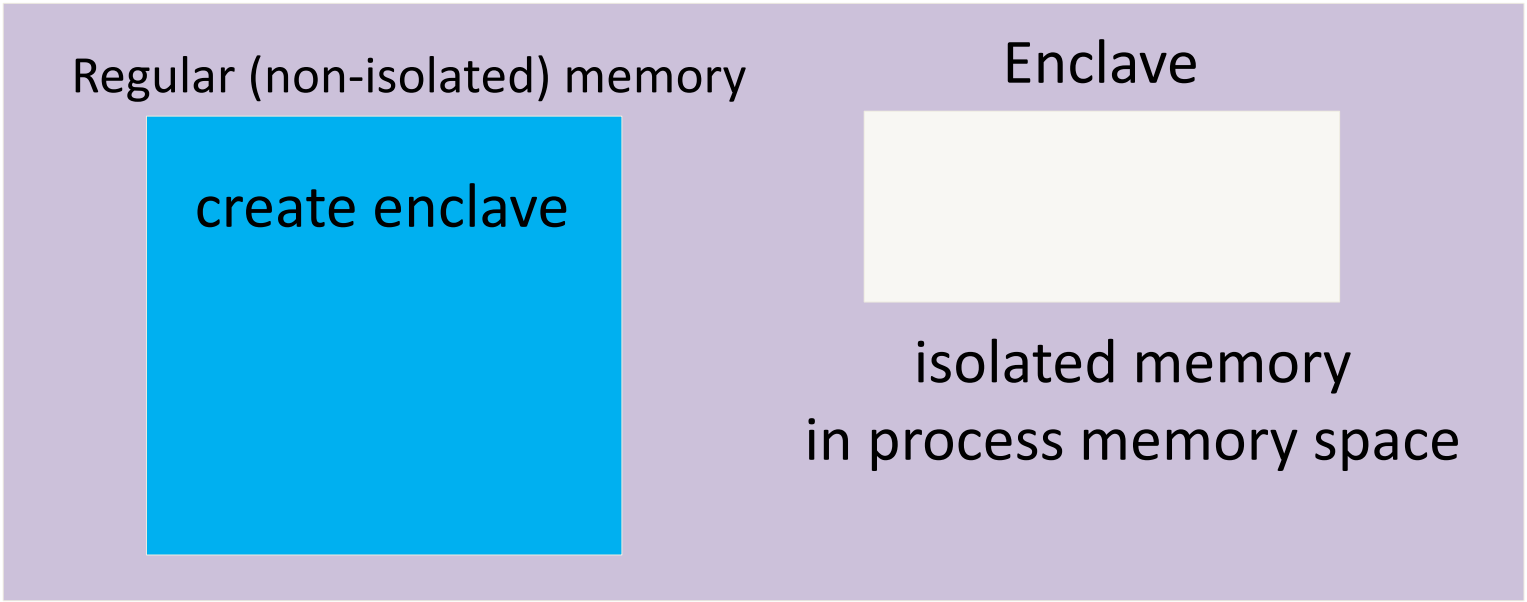
Intel SGX: how does it work?

An application defines part of itself as an enclave



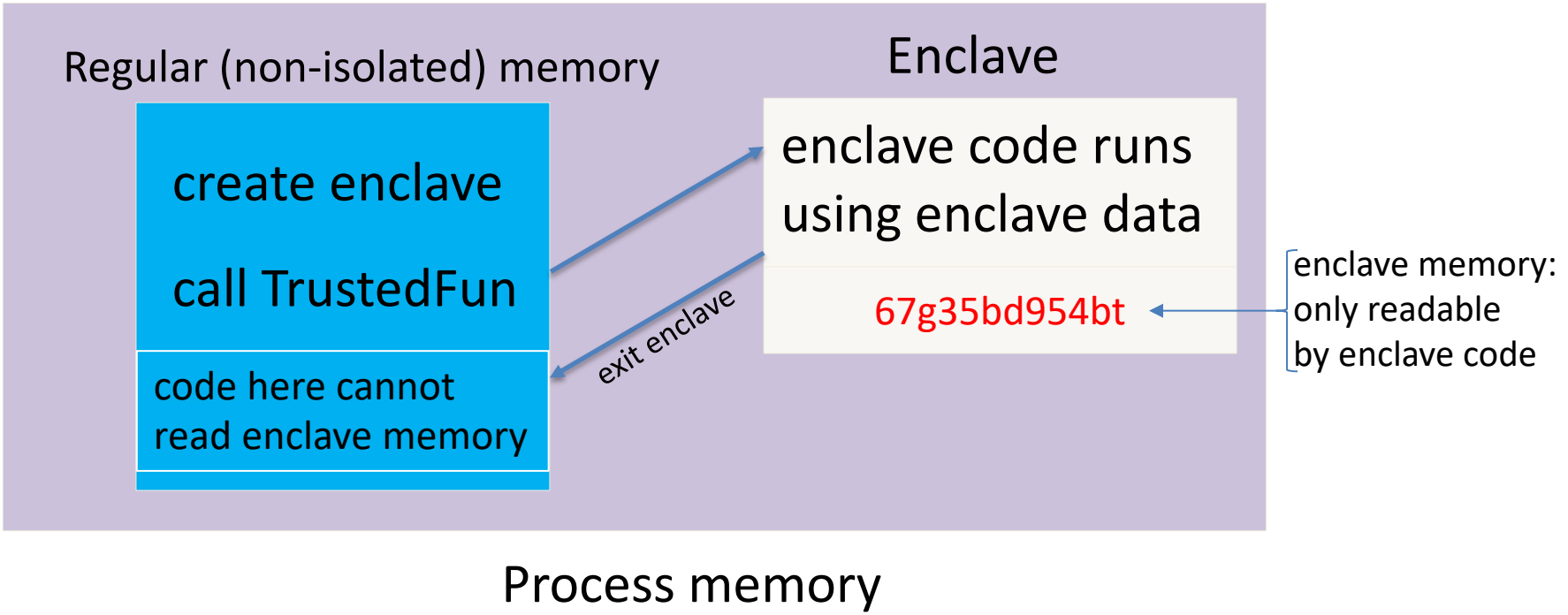
How does it work?

An application defines part of itself as an enclave



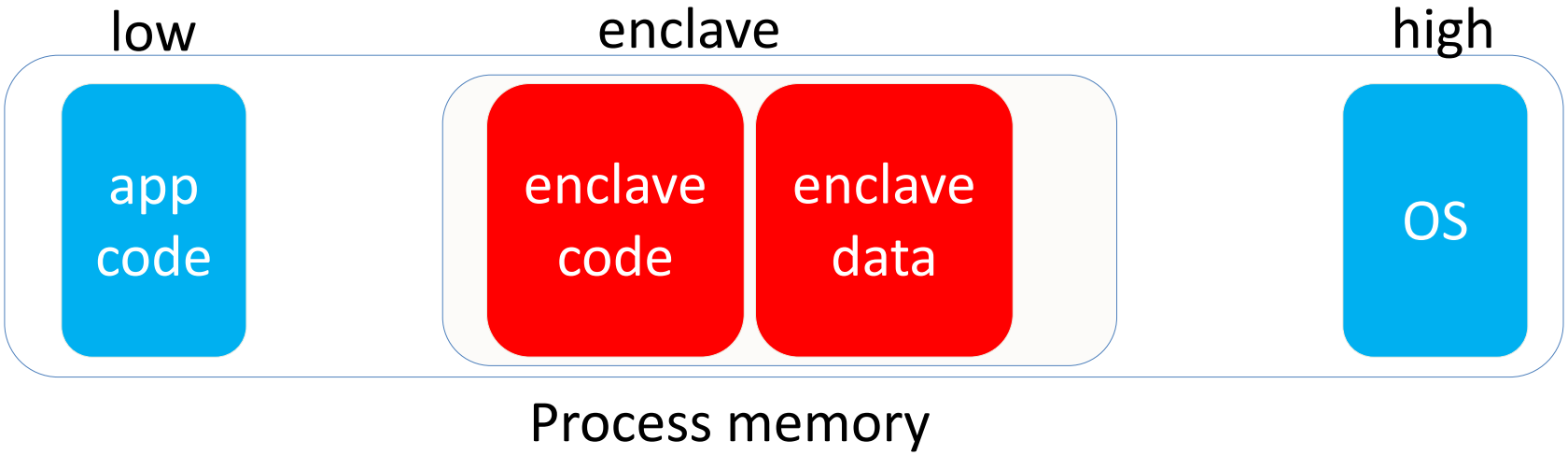
How does it work?

An application defines part of itself as an enclave



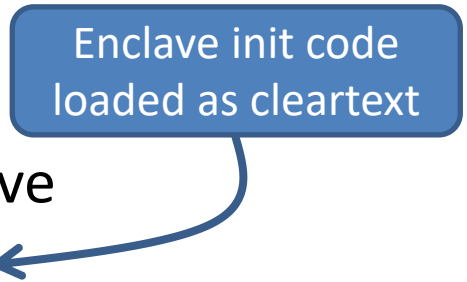
How does it work?

Part of process memory holds the enclave:



- Processor prevents access to cached enclave data outside of enclave.
- Enclave code and data are written **encrypted to RAM (TME)**

Creating an enclave: new instructions

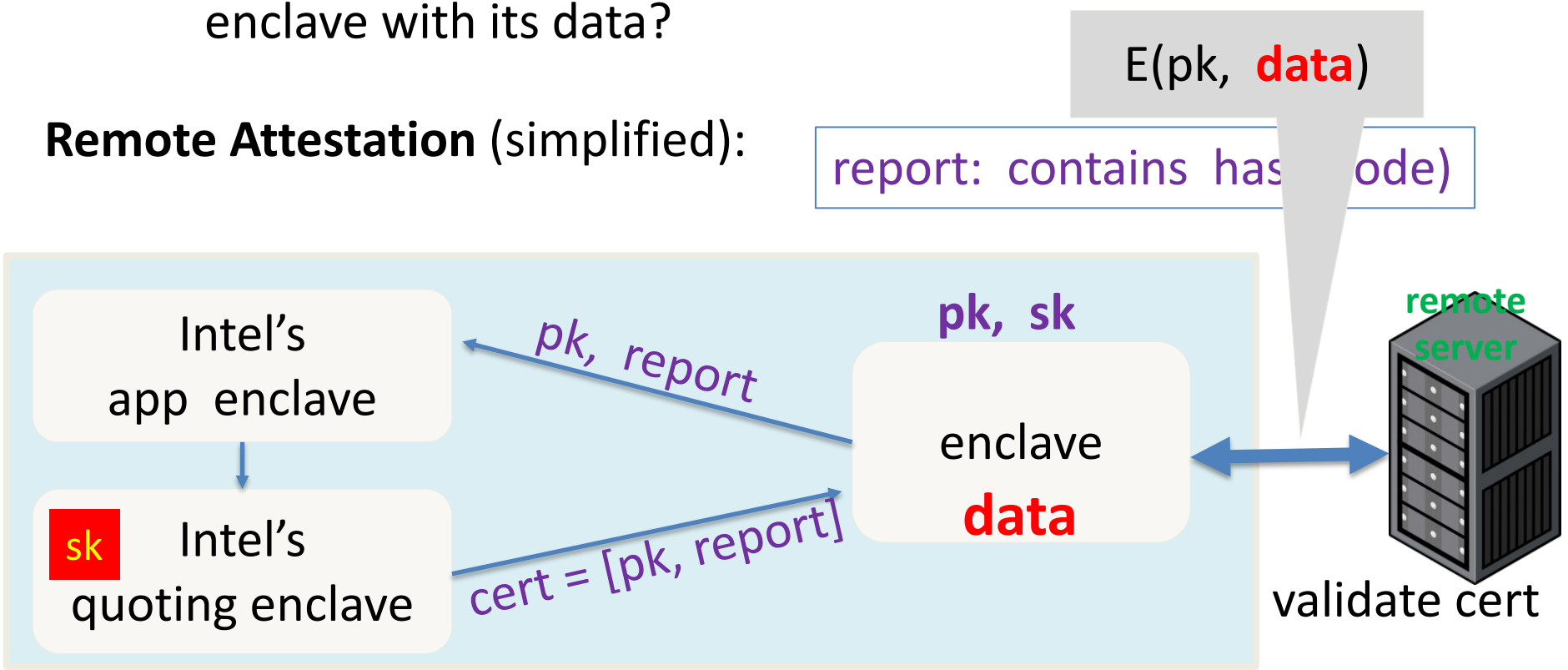


- **ECREATE**: establish memory address for enclave
- **EADD**: copies memory pages into enclave
- **EEXTEND**: computes hash of enclave contents (256 bytes at a time)
- **EINIT**: verifies that hashed content is properly signed
  if so, initializes enclave (signature = RSA-3072)
- **EENTER**: call a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

When to send secret data to enclave: attestation

The problem: How does a remote system know when it can trust an enclave with its data?

Remote Attestation (simplified):



SGX Summary

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 main processor
  ⇒ Memory controller encrypts all writes to RAM
- **Not suitable for legacy applications:** must split app into parts
  - Requires lots of code rewriting … not suitable for legacy apps

TDX Briefly: an easy-to-use encalve

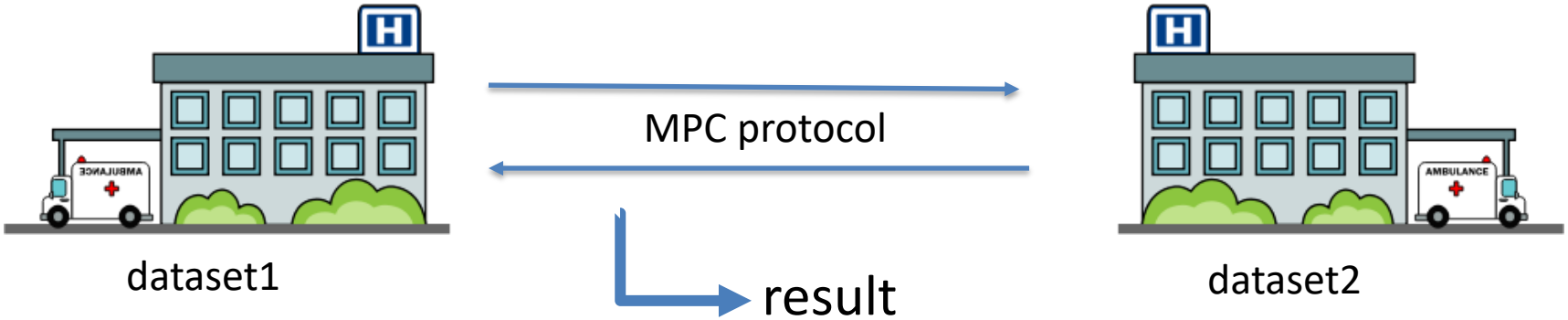
TDX: puts an entire VM in an enclave (e.g., an entire web server)

- Support for attestation and minimal TCB (as with SGX)
- Isolated VMs are managed by a new ***Intel TDX Module***
  - The TDX module is implemented in signed code by Intel
  - It is loaded into an isolated region of physical memory
  - Creates, manages, and attests to isolated VMs

https://cdrdv2.intel.com/v1/dl/getContent/690419

One more example application

Data science on federated data:

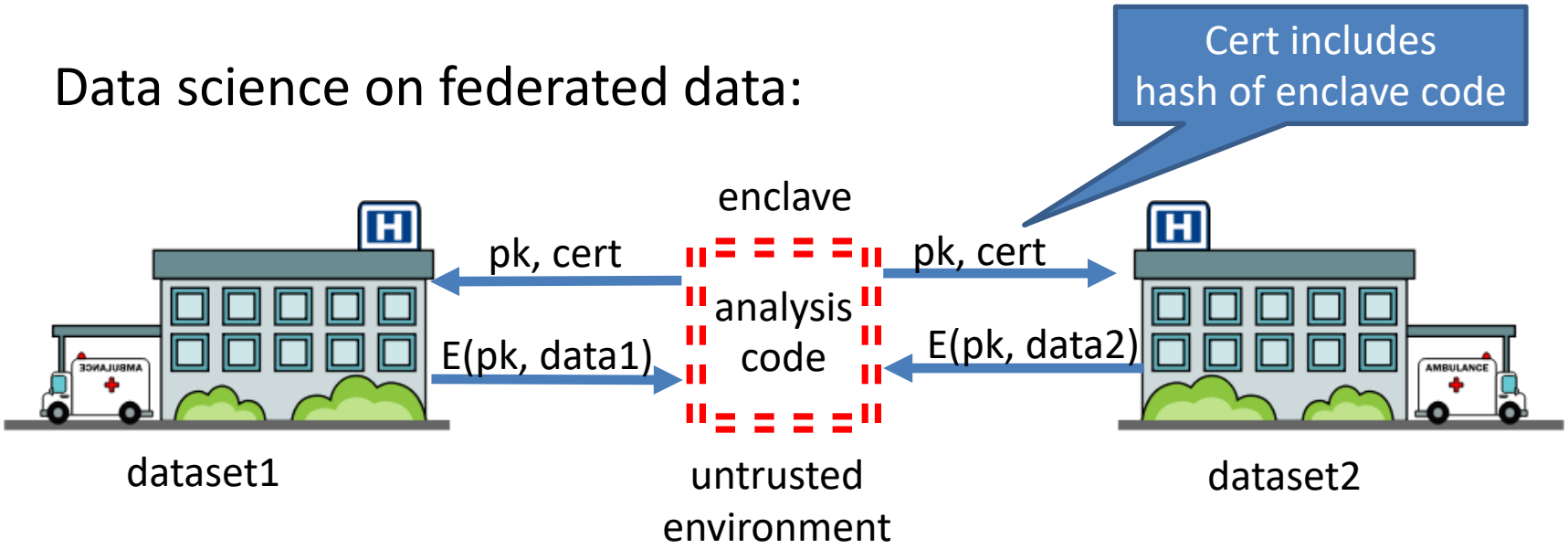


Can we run analysis on union(dataset1, dataset2) ??

cryptographic solutions (e.g., MPC) work for simple computations

An example application

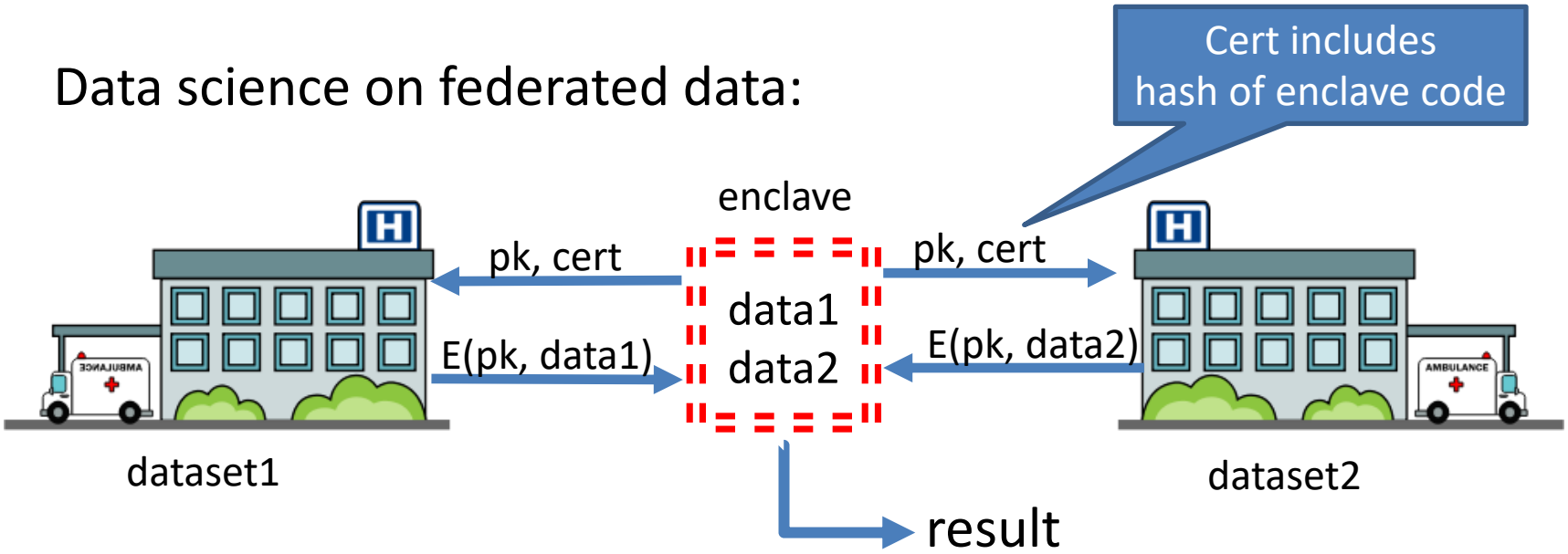
Data science on federated data:



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

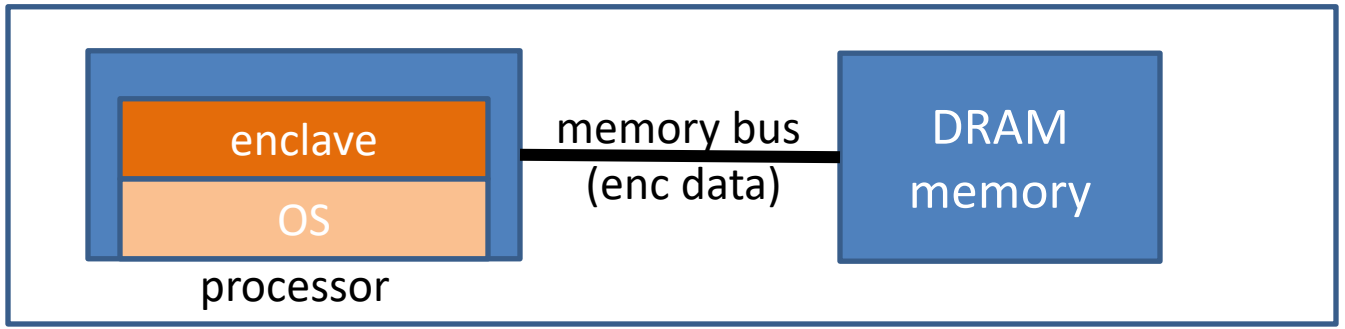
An example application

Data science on federated data:



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

SGX insecurity: (1) side channels

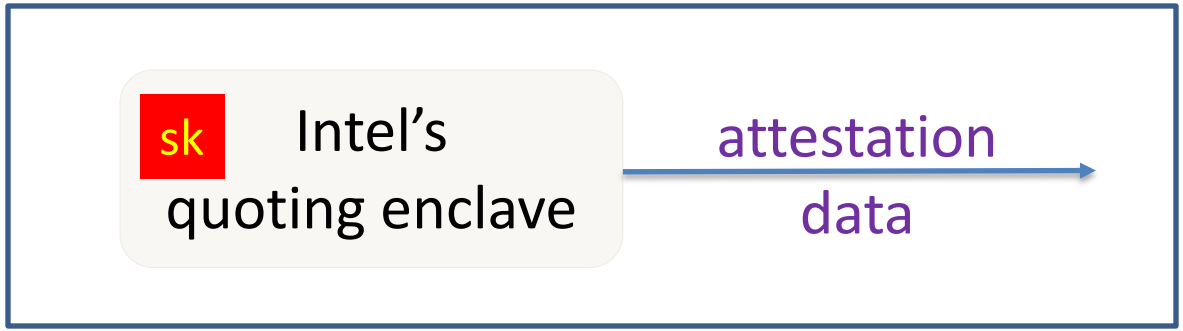


Attacker controls the OS. OS sees lots of side-channel info:

- 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 some 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:

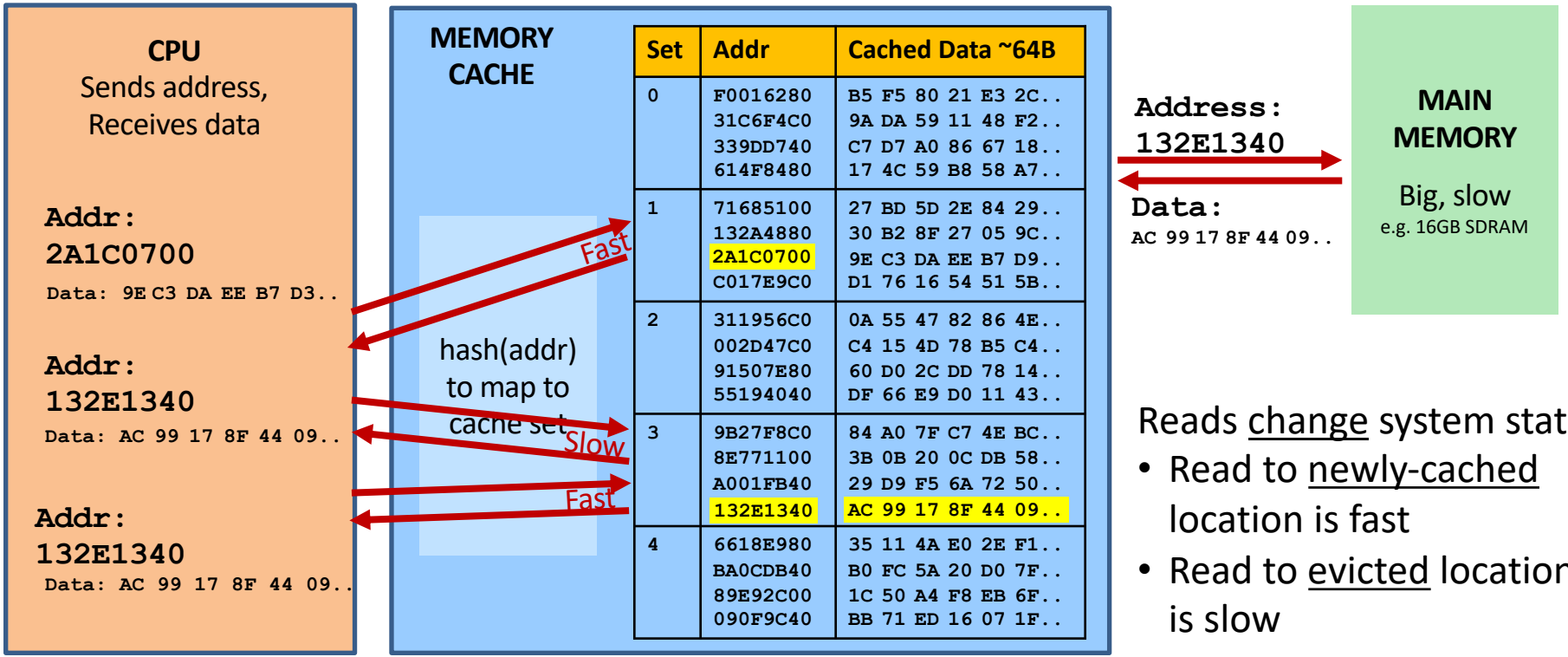
- Pentium 4 reached 3.8 GHz in 2004
- Memory latency is slow and not improving much

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



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 `x=1001,1002,…` (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)

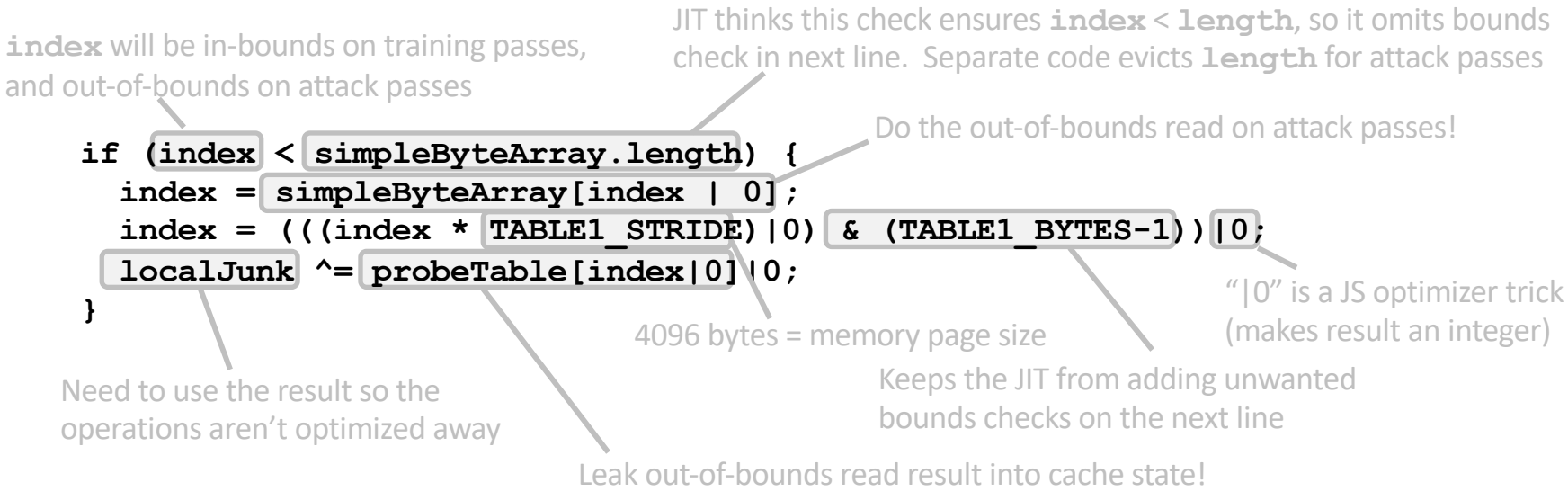
```
array2[ 0*4096]
array2[ 1*4096]
array2[ 2*4096]
array2[ 3*4096]
array2[ 4*4096]
array2[ 5*4096]
array2[ 6*4096]
array2[ 7*4096]
array2[ 8*4096]
array2[ 9*4096]
array2[10*4096]
array2[11*4096]
  ...
```

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…



Can evict length and 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 ];
```

LFENCE: stops speculative execution.

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

Put LFENCES everywhere?	⇒ Abysmal performance
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 non-trivial for all Spectre variants:

- Software must deal with microarchitectural complexity
- Mitigations are hard to test:
  - an active area of research (see Prof. Caroline Trippel's work)

More ideas needed !

... but there is more

More speculative execution attacks:

- **Meltdown**
- Rogue inflight data load (**RIDL**) and **Fallout**
- **ZombieLoad**
- **Micro-op caches** (June 2020)
- **Pointer prefetching in Apple's M1** (March 2024)

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 few years

… likely more will be found in coming years

THE END