CS 155: Computer Security

Processor security

Logistics

Homework 1 due Thursday (4/23)

The processor

Part of the trusted computing base (TCB)

… but it is optimized for performance (security may be secondary)

Today:

- How the processor can enable secure systems: hardware enclaves
- How the processor can be exploited for attacks: speculative execution

Outline

1. Hardware enclaves
    a. Intel SGX
    b. Intel TDX
2. Spectre attack
    a. Background
    b. Attack
    c. Mitigations

Outline

1. **Hardware enclaves**
    a. Intel SGX
    b. Intel TDX
2. Spectre attack
    a. Background
    b. Attack
    c. Mitigations

Goals of Intel SGX and TDX

Extension to Intel processors support enclaves, which enable

- Isolation: Running code and memory isolated from the rest of the system
  ↳ Code outside the enclave cannot read enclave memory
- Attestation: Prove to a remote system what code is running in the enclave
- Minimal TCB: Only processor is trusted, nothing else
  ↳ Not the OS, RAM, or peripherals
  ↳ Memory controller must encrypted all writes to RAM
  (total memory encryption, TME)

Enclave use case #1: Cloud computing

Goal: Move data and VM to the cloud
↳ Cloud should not be able to see them in the clear

Simple solution: Encrypt data and upload to cloud, key stays with the client

Problem: Cloud cannot search or compute on data
↳ Defeats the purpose of cloud computing

Client

Cloud

Enclave use case #1: Cloud computing

Goal: Move data and VM to the cloud
↳ Cloud should not be able to see them in the clear

Simple solution: Encrypt data and upload to cloud, key stays with the client

Problem: Cloud cannot search or compute on data
↳ Defeats the purpose of cloud computing

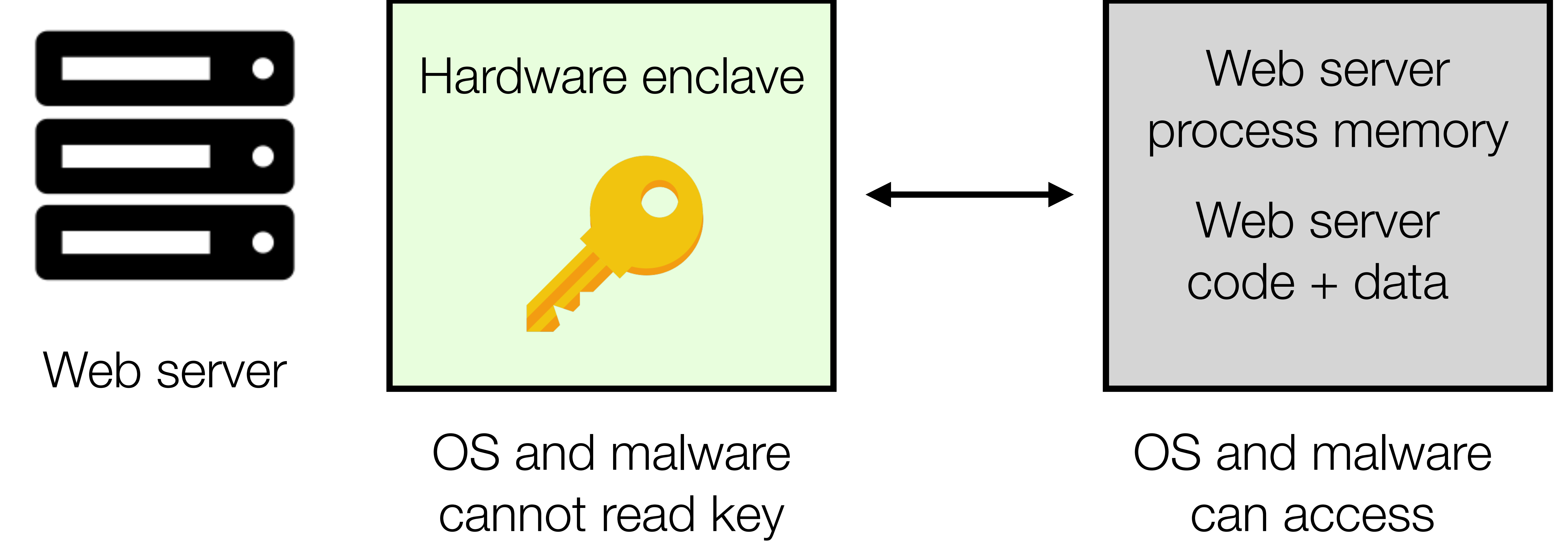
Run client code inside a hardware enclave

Client

Cloud

Enclave use case #2: Protecting keys

Goal: Protect a web server HTTPS secret key
↳ Secret key is only available inside an enclave
↳ Malware cannot extract the key



Enclave use case #3: Analytics on federated data

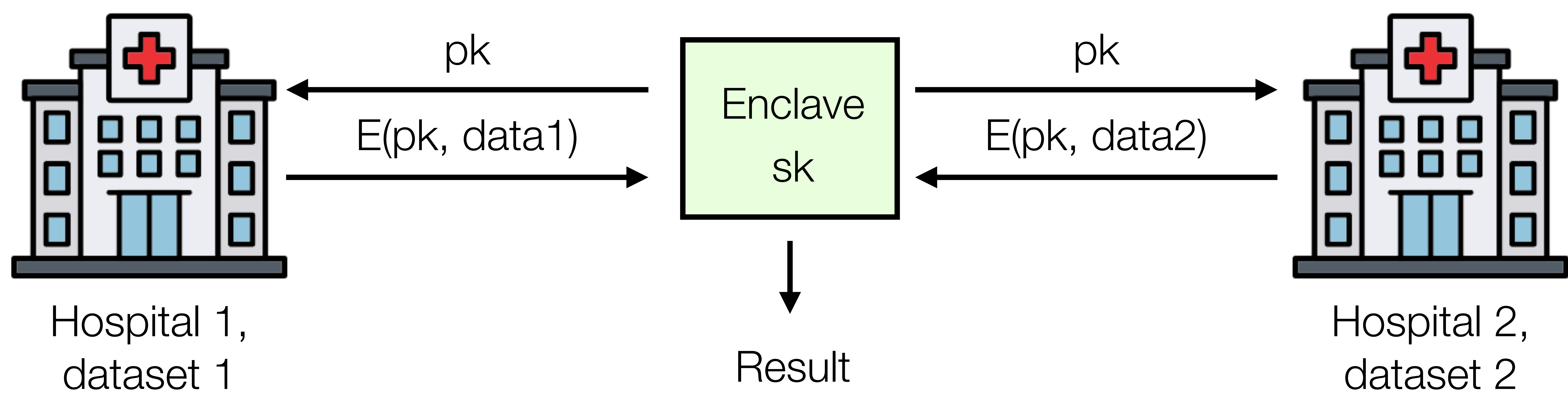
Can we run analysis on the union of dataset 1 and 2?

Cryptographic solutions (multi-party computation) are feasible for simple computations

Enclave use case #3: Analytics on federated data

Can we run analysis on the union of dataset 1 and 2?

Cryptographic solutions (multi-party computation) are feasible for simple computations

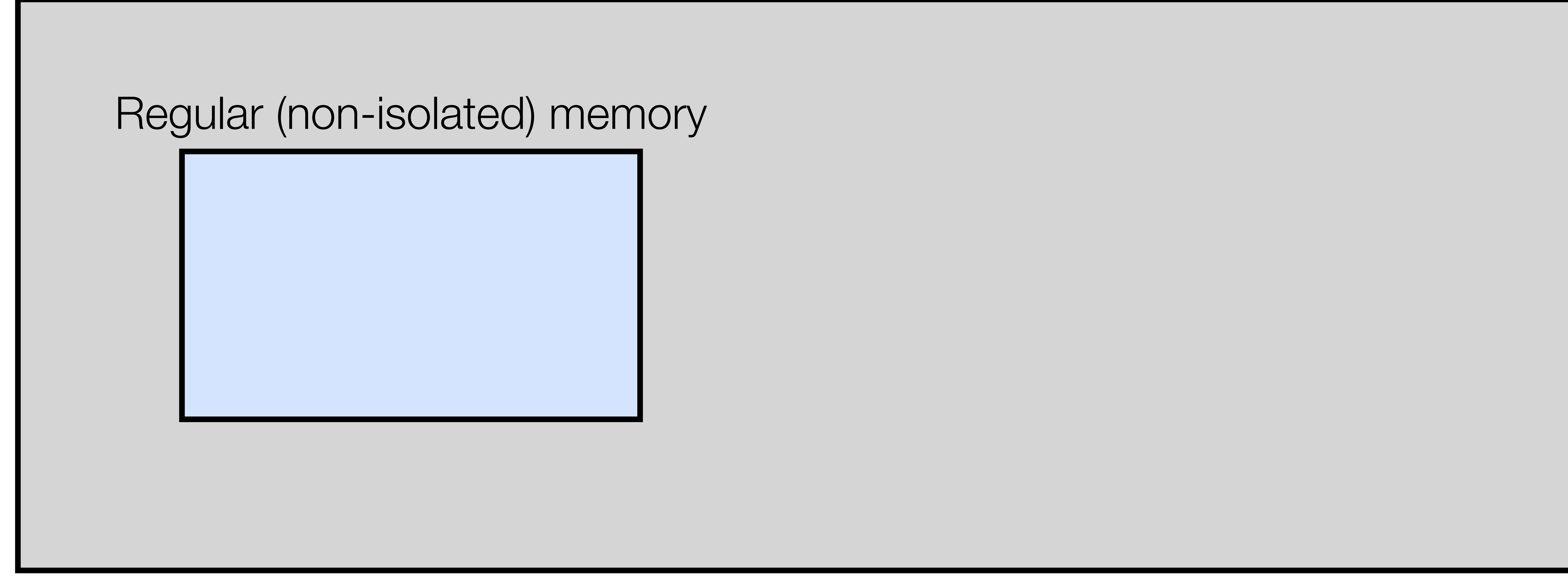


Enclaves can make more complex computations feasible with privacy

Intel SGX: How does it work?

Break application into two parts:

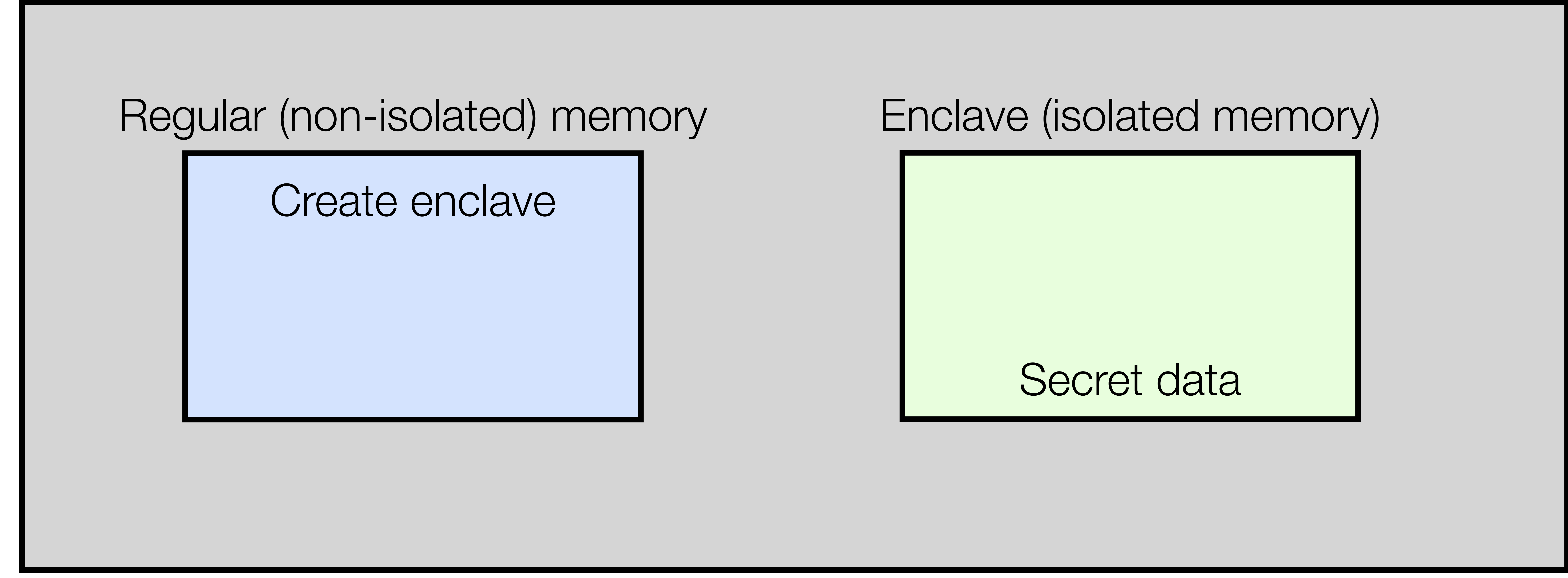
- One part runs inside the enclave
- The other part runs outside the enclave



Intel SGX: How does it work?

Break application into two parts:

- One part runs inside the enclave
- The other part runs outside the enclave



Intel SGX: How does it work?

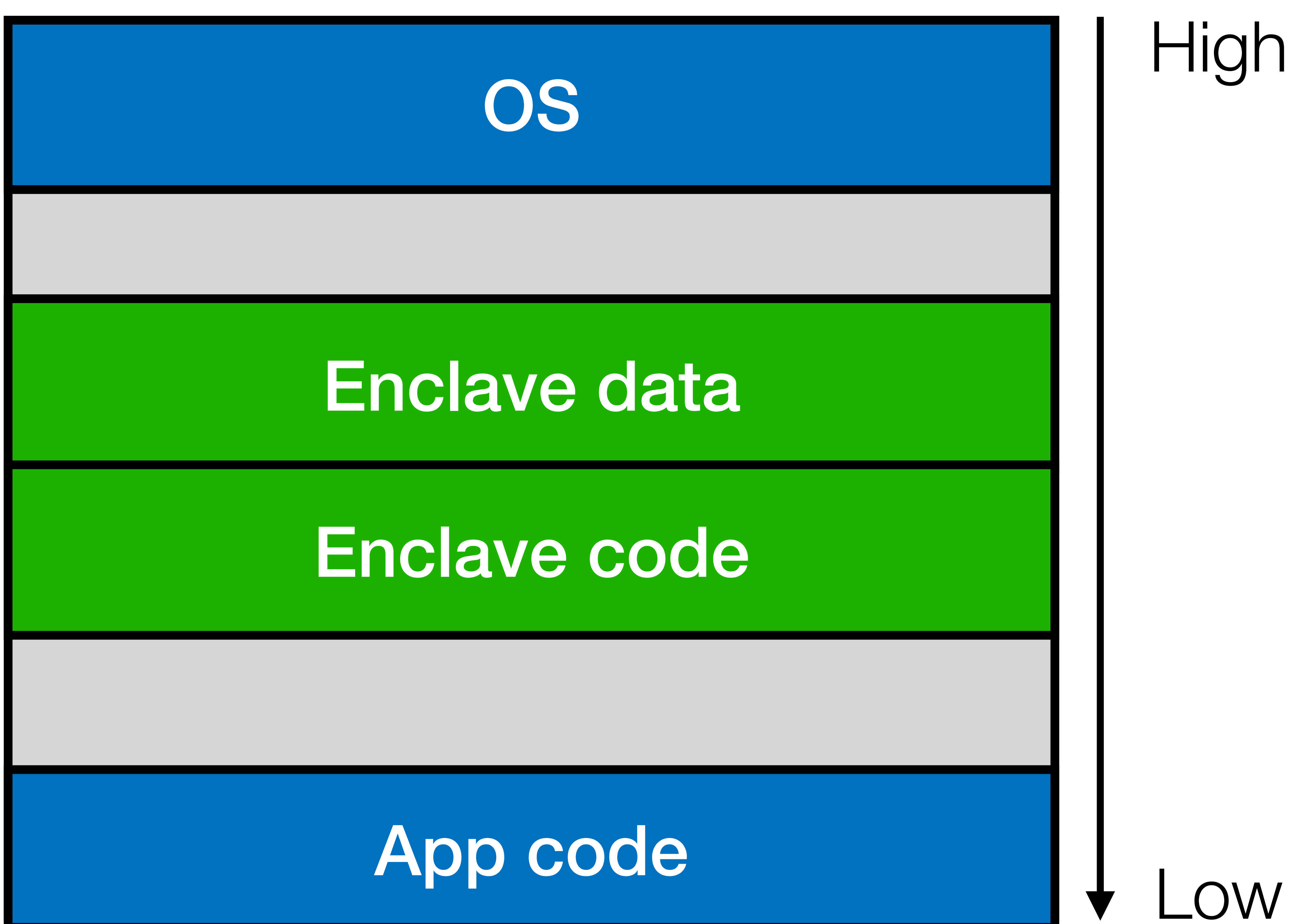
Break application into two parts:

- One part runs inside the enclave
- The other part runs outside the enclave

Intel SGX: How does it work?

Part of process memory allocated for the enclave

- Processor prevents access to cached enclave data outside of enclave
- Enclave code and data are written encrypted to RAM: total memory encryption (TME)

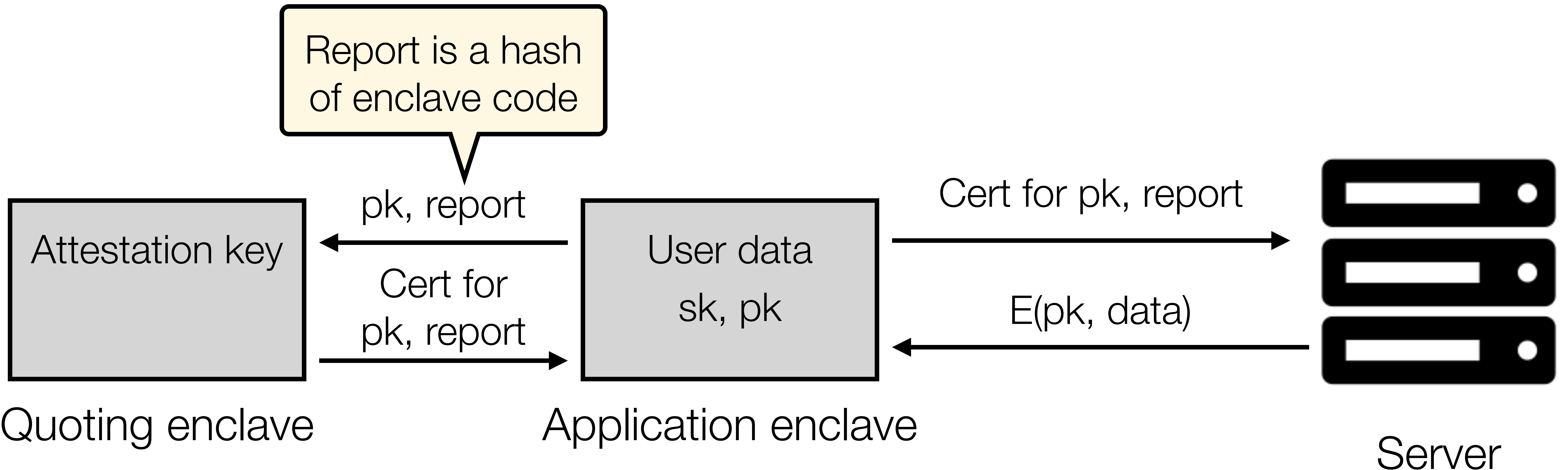


Creating an enclave: new instructions

- `ECREATE:` establish memory region for enclave
- `EADD:` copies memory pages into enclave
- `EEXTEND:` computes hash of enclave contents (256B at a time)
- `EINIT:` verifies that hashed content is properly signed; if so, initializes enclave
- `EENTER:` calls a function inside enclave
- `EEXIT:` return from enclave

Enclave attestation (simplified)

Problem: How does a remote system know when to trust an enclave with its data?
↳ How to know you're talking with a real SGX enclave?



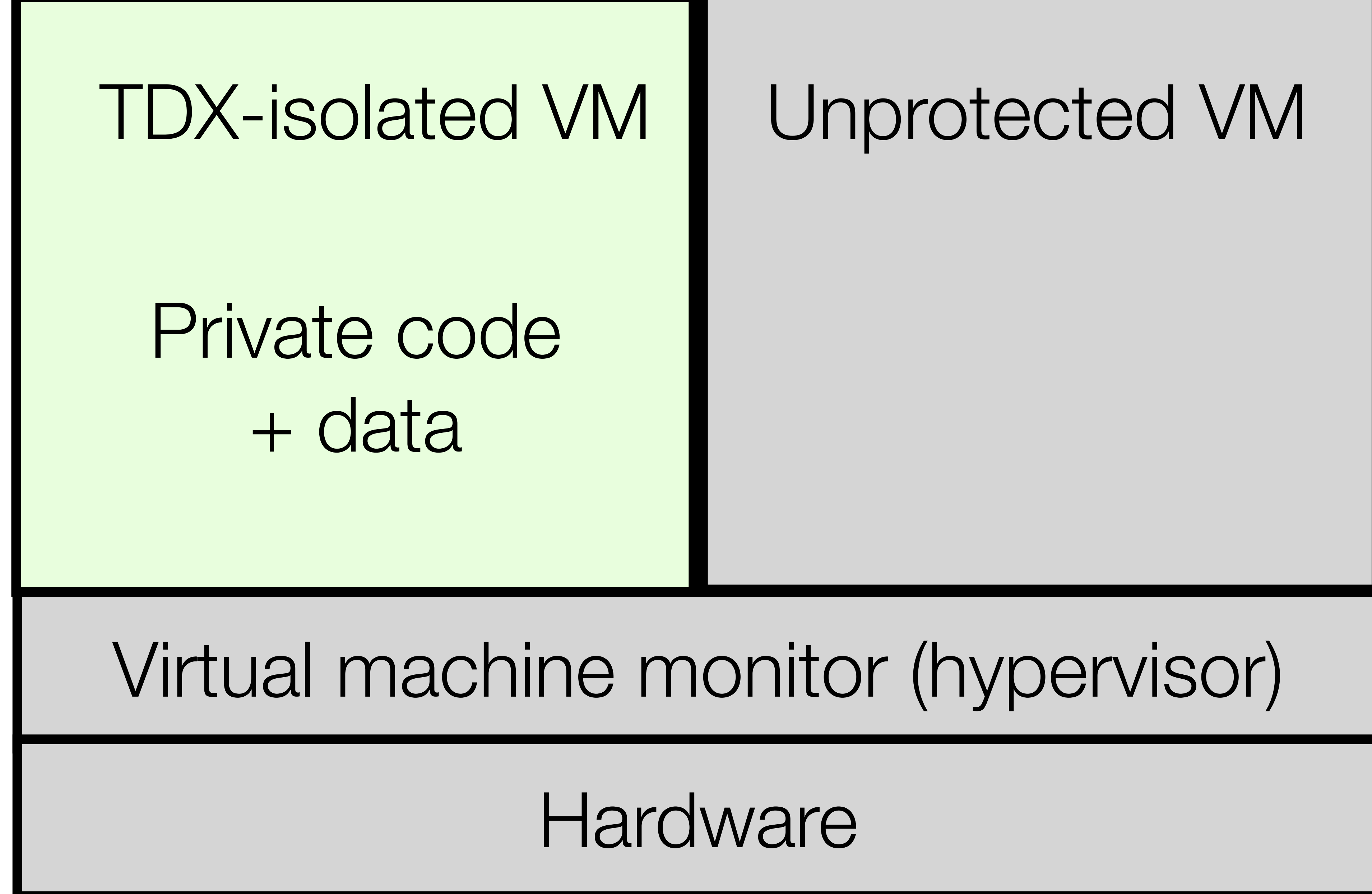
Intel SGX summary

- 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: only requires trusting the main processor (not even the OS!)
  ↳ Memory controller encrypts all writes to RAM
- **Not suitable for legacy applications:** must split application into parts
  ↳ Requires lots of code rewriting

Intel TDX

TDX: run an entire VM in an enclave (e.g., an entire web server)
↳ Don't need to partition application

Goal: Hypervisor cannot read the memory of the isolated VM



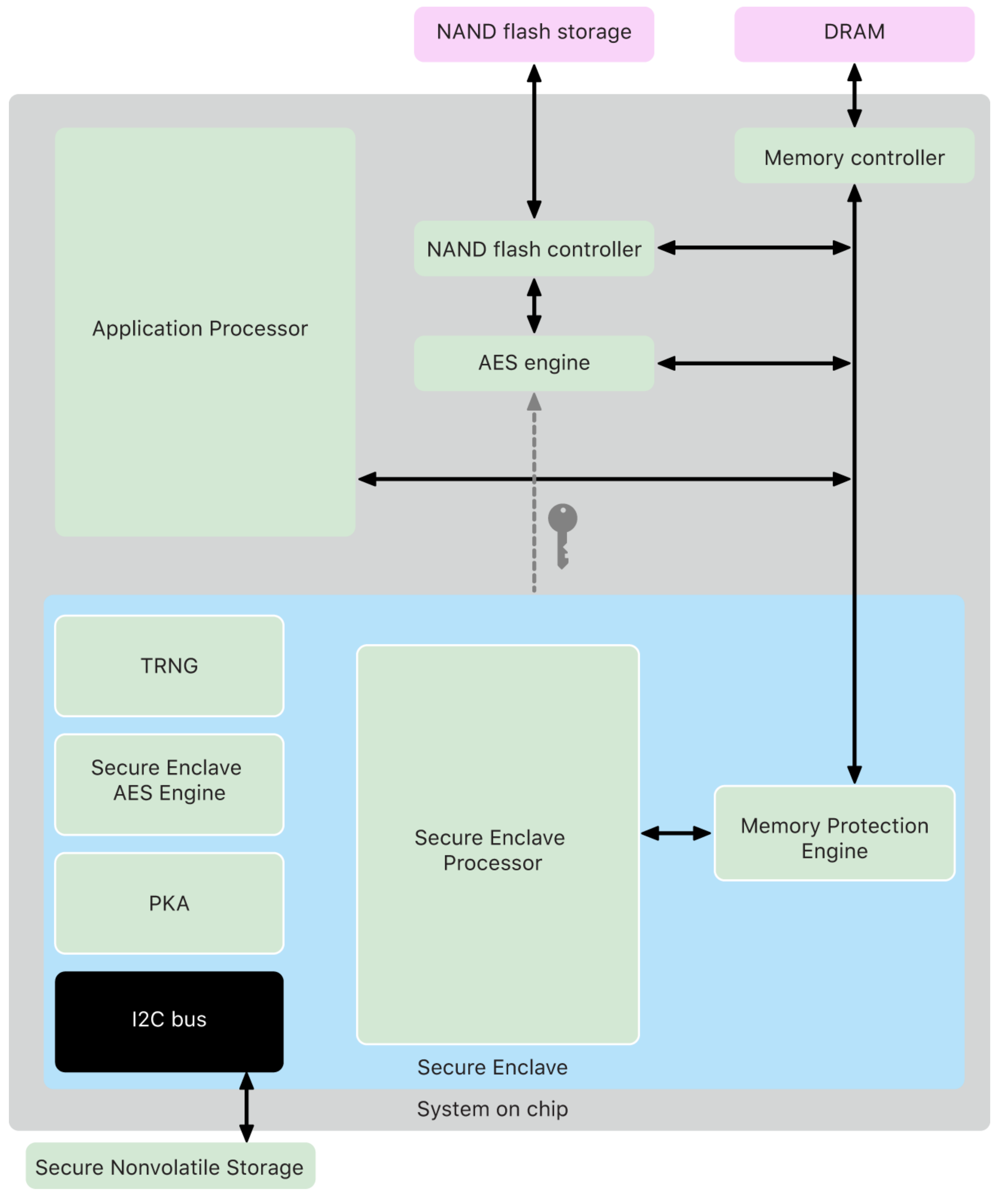
Intel TDX

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

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

Enclaves on iPhones

- Root of trust at boot time
- Manages sensitive operations
  ↳ Signing and decryption
  ↳ Login with FaceID, TouchID, PIN
- Part of memory dedicated to the secure enclave
- Secure nonvolatile storage holds long-term secrets and counters for sensitive actions (e.g., authentication)

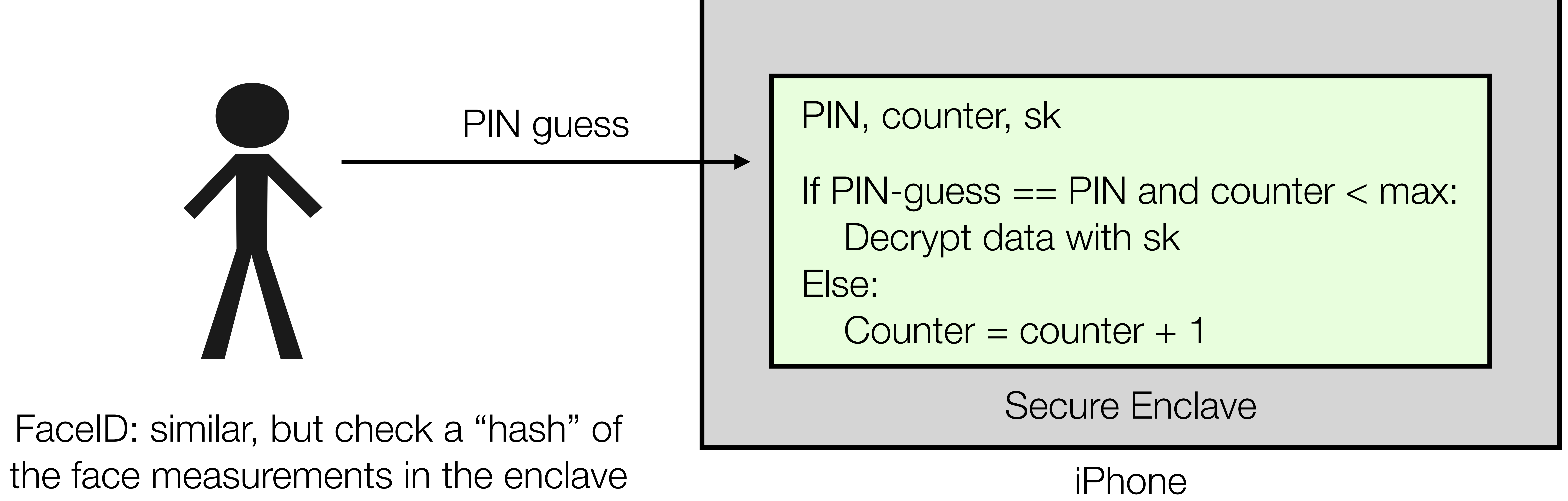


https://support.apple.com/guide/security/the-secure-enclave-sec59b0b31ff/web

iPhone: Protecting data at rest (simplified)

Data encrypted at rest

Goal: Can only decrypt if enter the right PIN

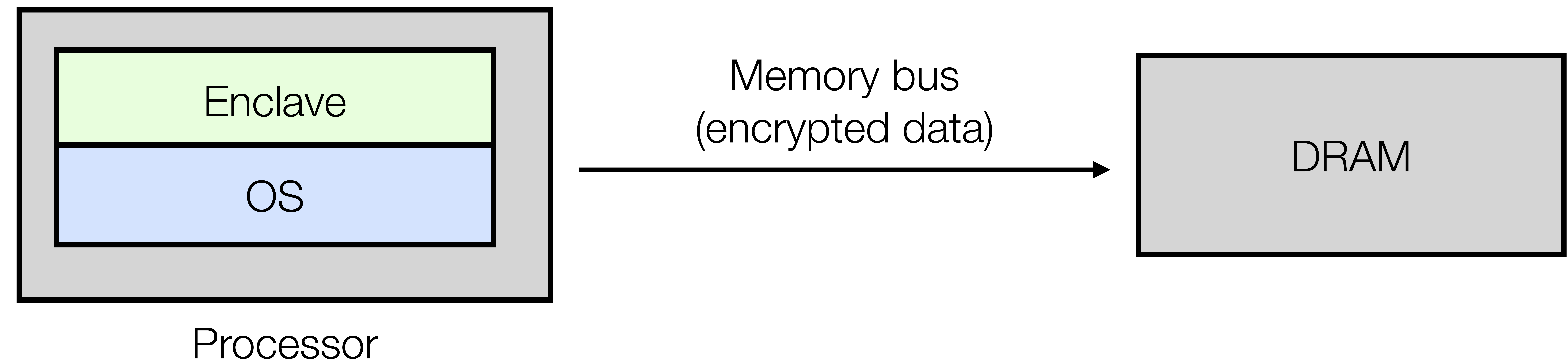


Enclave limitation: side channels

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

- Memory access patterns
- State of processors as enclave executes
- State of branch predictor

Can leak enclave data, and difficult to block!



Enclave limitation: extracting attestation 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 enclaves?
↳ Can attest to arbitrary non-enclave code
↳ Foreshadow attack and Intel's response

Outline

1. Hardware enclaves
    a. Intel SGX
    b. Intel TDX
2. **Spectre attack**
    a. Background
    b. Attack
    c. Mitigations

Slides credit: Paul Kocher

Performance drives CPU purchases

Clock speed maxed out:

- 2004: Pentium 4 reached 3.8 GHz
- 2026: Apple M5 is 4.6 GHz

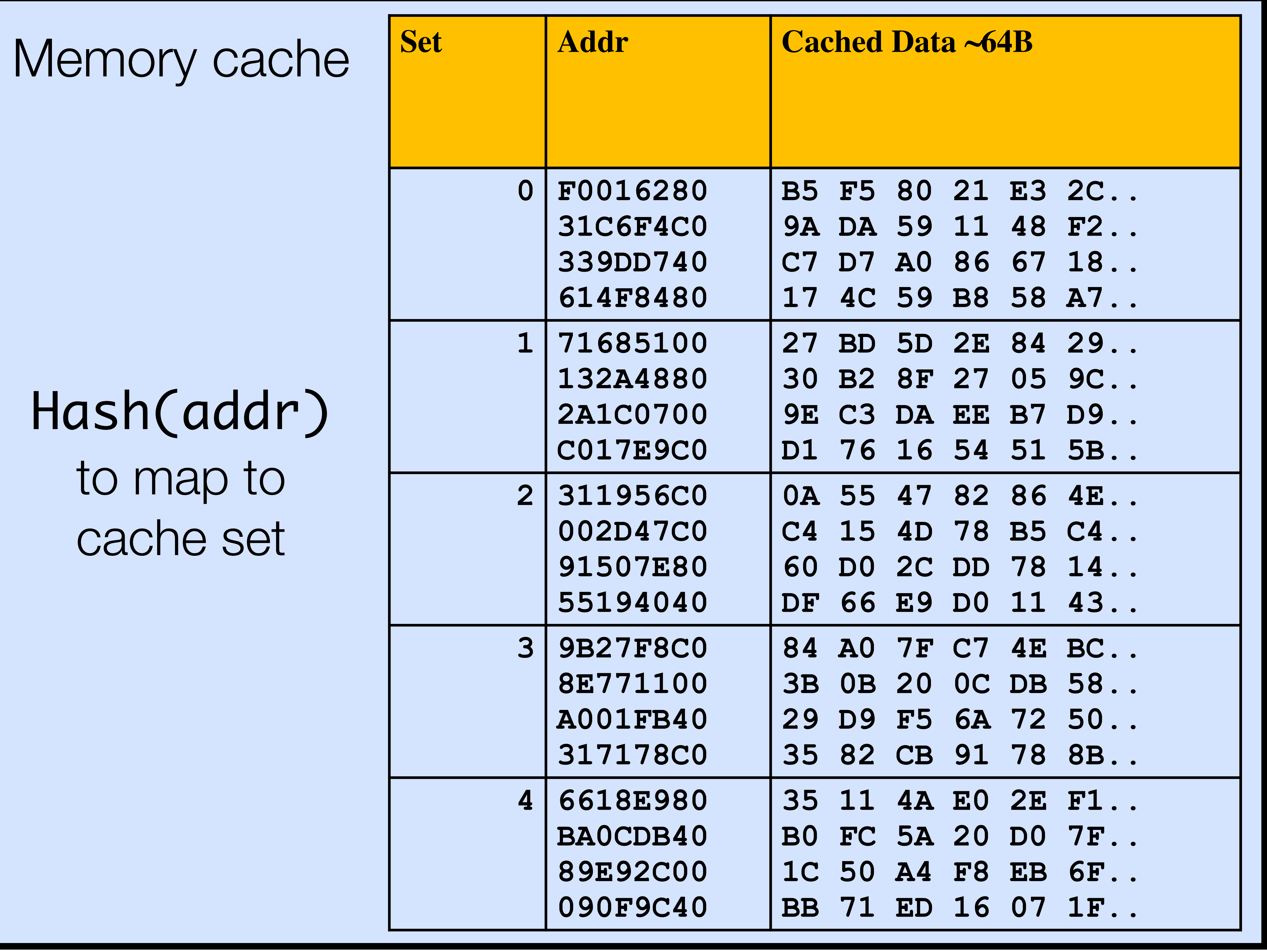
To gain performance, need to do more per cycle

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

Memory caches (4-way associative)

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

Processor



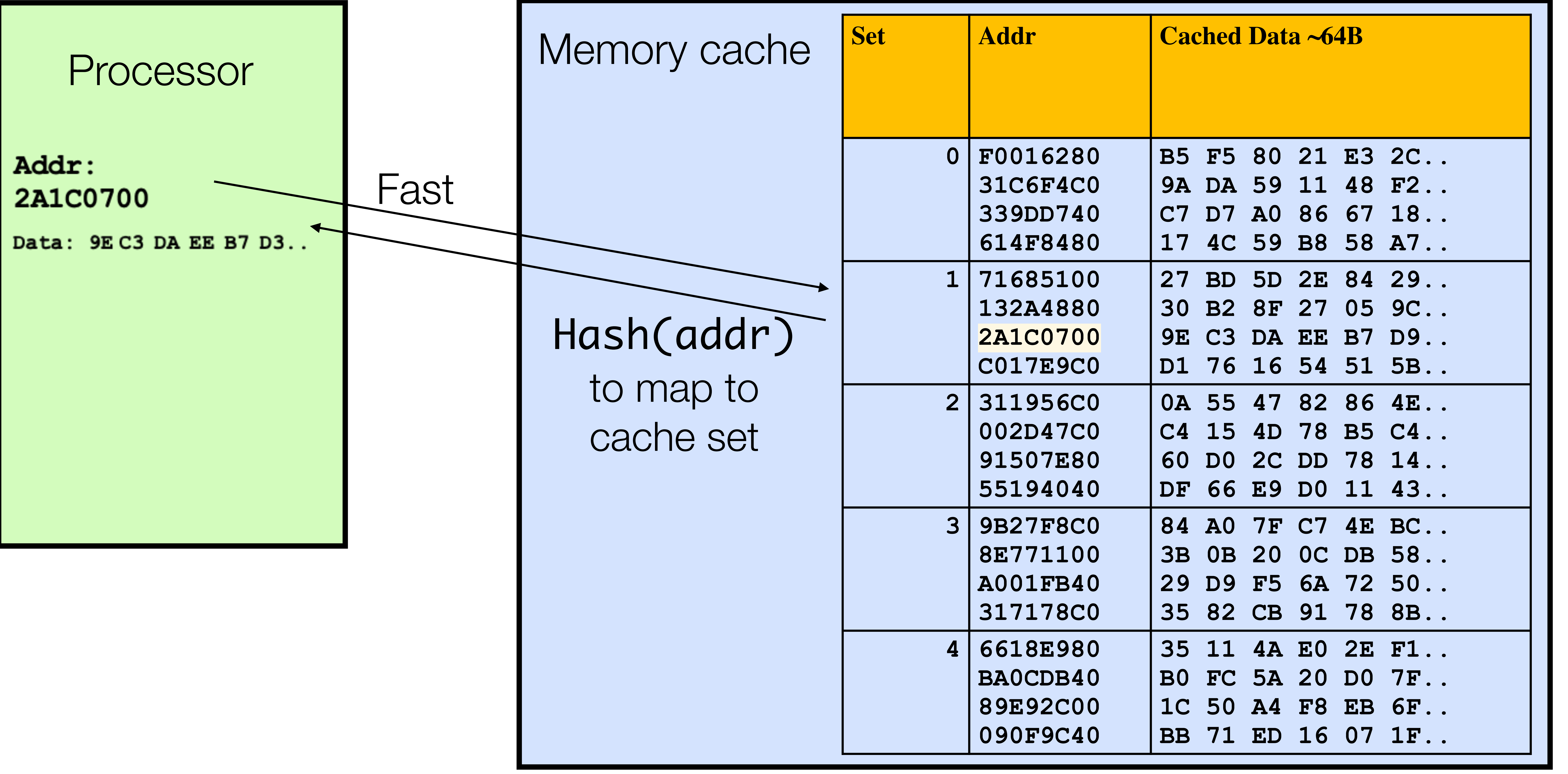
Set	Addr	Cached Data ~64B
0	F0016280	B5 F5 80 21 E3 2C..
	31C6F4C0	9A DA 59 11 48 F2..
	339DD740	C7 D7 A0 86 67 18..
	614F8480	17 4C 59 B8 58 A7..
1	71685100	27 BD 5D 2E 84 29..
	132A4880	30 B2 8F 27 05 9C..
	2A1C0700	9E C3 DA EE B7 D9..
	C017E9C0	D1 76 16 54 51 5B..
2	311956C0	0A 55 47 82 86 4E..
	002D47C0	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..
	317178C0	35 82 CB 91 78 8B..
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..

Main memory
(big, slow)

Reads change system state:
- Read to newly cached location is fast
- Read to evicted location is slow

Memory caches (4-way associative)

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



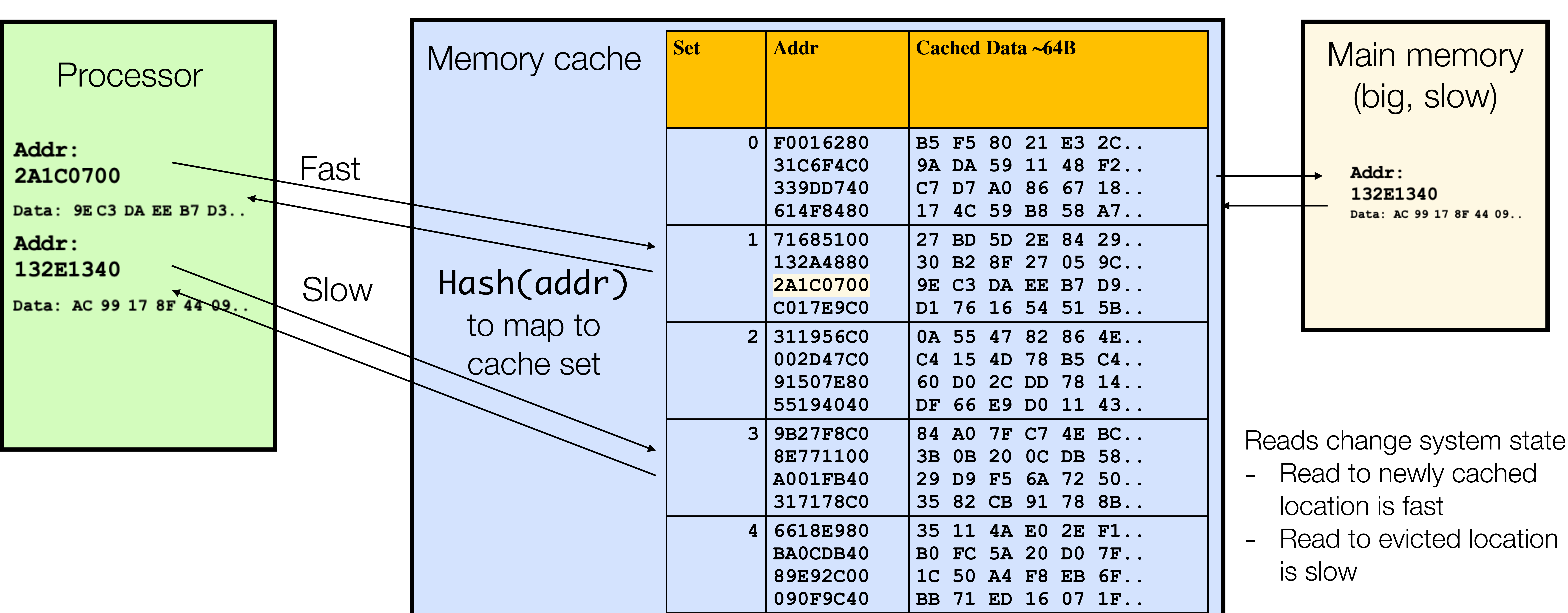
Set	Addr	Cached Data ~64B
0	F0016280	B5 F5 80 21 E3 2C..
	31C6F4C0	9A DA 59 11 48 F2..
	339DD740	C7 D7 A0 86 67 18..
	614F8480	17 4C 59 B8 58 A7..
1	71685100	27 BD 5D 2E 84 29..
	132A4880	30 B2 8F 27 05 9C..
	2A1C0700	9E C3 DA EE B7 D9..
	C017E9C0	D1 76 16 54 51 5B..
2	311956C0	0A 55 47 82 86 4E..
	002D47C0	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..
	317178C0	35 82 CB 91 78 8B..
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..

Main memory
(big, slow)

Reads change system state:
- Read to newly cached location is fast
- Read to evicted location is slow

Memory caches (4-way associative)

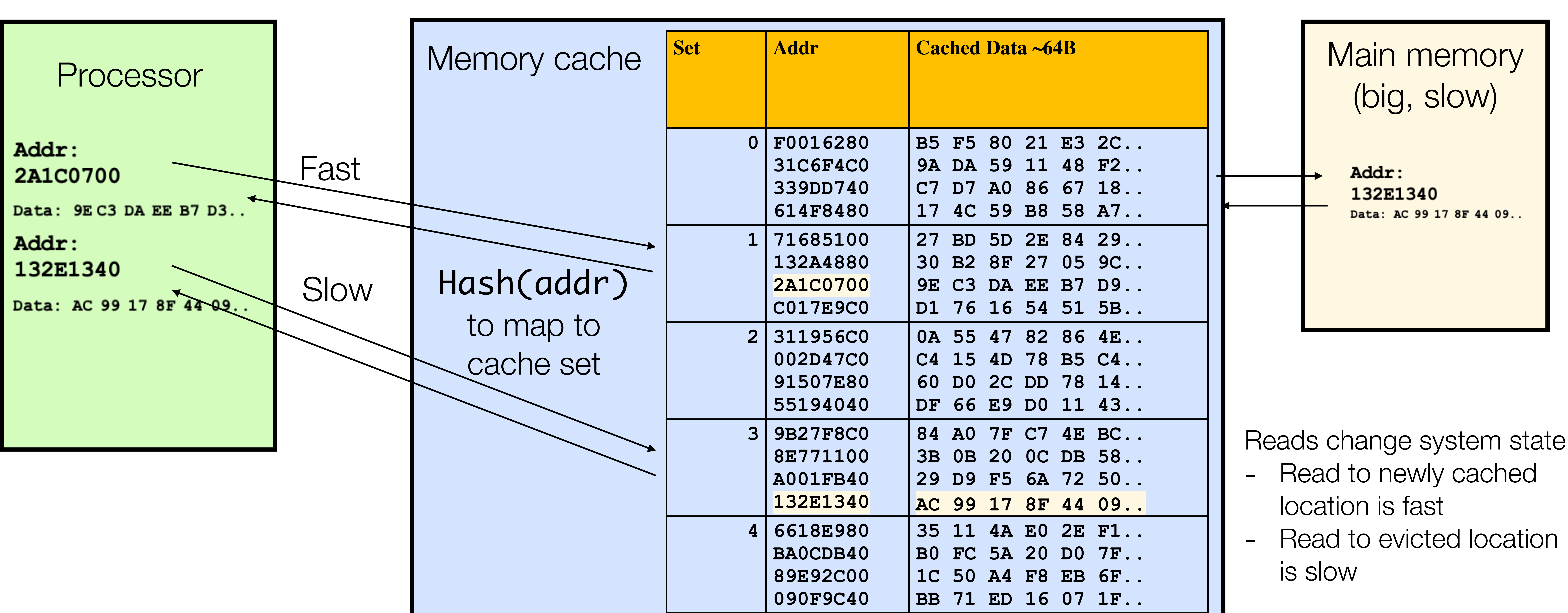
Caches hold fast (local) copy of recently accessed 64B chunks of memory



Reads change system state:
- Read to newly cached location is fast
- Read to evicted location is slow

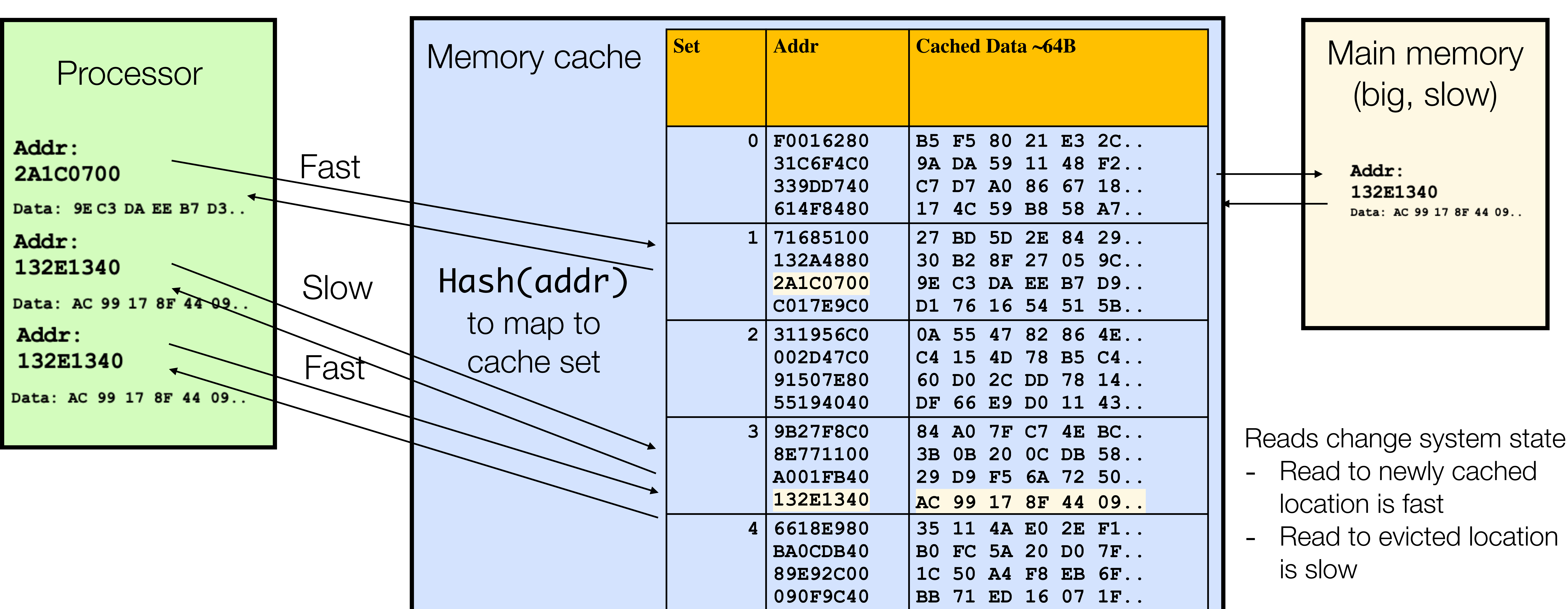
Memory caches (4-way associative)

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



Memory caches (4-way associative)

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



Speculative execution

CPU can guess likely path and perform **speculative execution**

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

Branch predictor guesses `if` is true (based on prior history)

Start executing `compute(b)` speculatively

When value arrives from memory, check if guess was correct:

- Correct: Save speculative work $\Longrightarrow$ performance gain
- Incorrect: Discard speculative work $\Longrightarrow$ no harm ???

Speculative execution

Architectural guarantee

Register values eventually match result of in-order execution

Speculative execution

CPU regularly performs incorrect calculations, then deletes mistakes

Is making and 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 `x` comes from untrusted caller

Execution ***without*** speculation is safe:
↳ `array2[array1[x]*4096]` is never executed unless `x < array1_size`

What about ***with*** speculation?

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 and cache status

`array1_size = 8`

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

Memory at `array1` base+100:
`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]
```

Contents don't matter, only 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 execution 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 and cache status

`array1_size = 8`

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

Memory at `array1` base+100:
`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]
```

Contents don't matter, only 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 cache
- Realize `if()` is false: discard speculative work

Proceed to next instruction

Memory and cache status

`array1_size = 8`

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

Memory at `array1` base+100:
`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]
```

Contents don't matter, only 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 (on 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 and cache status

`array1_size = 8`

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

Memory at `array1` base+100:
`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]
```

Contents don't matter, only 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` and `probeTable`

… 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 misprediction to a useful ‘gadget’

Attack steps:

1. **Mistrain** branch prediction so speculative execution will go to gadget
2. **Evict** address `[rax]` from cache to cause speculative execution
3. **Execute** victim so it runs gadget speculatively
4. **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 may have observable side effects beyond cache state
↳ TLB state, branch predictor, variations in power consumption, …

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

Variant 1 mitigation: Speculation-stopping instruction

`LFENCE`: stops speculative execution

Idea: insert `LFENCE` on all vulnerable code paths

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

Variant 1 mitigation: Speculation-stopping instruction

What if we put `LFENCE`s everywhere?
↳ Abysmal performance

Idea: Have a smart compiler insert `LFENCE`s
↳ Must protect against all potentially exploitable patterns
↳ Supported in LLVM, along with other mitigations
↳ Protects all LLVM-based compilers

Transfer of blame from CPU to software: “You should have put an `LFENCE` there”

Mitigations takeaways

Mitigations are nontrivial 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!

More speculative execution attacks

- Meltdown

- Rogue inflight data load (RIDL) and Fallout

- ZombieLoad
- Micro-op caches
- Pointer prefetching in Apple's M1

Enable reading unauthorized memory (client, cloud, TDX)
↳ Mitigations incur 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

Many security issues found in the last few years:
↳ Likely many more will be found in the coming years

Next time: Start of web security