CS 253 Cyber Security
Microarchitectural Security

ShanghaiTech University

- Last week we stopped at Acoustic Side-Channel as an example…

- Now, what indeed are Side-Channel Attacks?
  - Attacks based on information that can be gleaned from the physical implementation of a system, rather than breaking its theoretical properties

PART ONE

01

Classical Side-channel Attacks

Side-channels and Crypto

- Side-channels were initially most commonly discussed in the context of cryptosystems

- Timing attacks
- Power analysis
- Old but good overview: http://www.nicolascourtois.com/papers/sc/sidech_attacks.pdf

- If you do something different for secret key bits 1 vs. 0, attacker can learn something...

Timing Side-Channel Attacks

- Famous example: RSA
- RSA needs to do perform key-based modular exponentiations
- Naïve implementations may look like this…

```
BigInt modexp(BigInt a, BigInt e, BigInt n) {
   BigInt result = 1;
   for (int i = bitlen(e) - 1; i >= 0; i--) {
      // always do square
      result = (result * result) % n;

      if (bit_test(e, i)) {
         // only multiply when the bit is 1
         result = (result * a) % n;
      }
   }
   return result;
}
```


* Intel's Secure Coding Guide cites this example as a textbook illustration of timing leakage in RSA implementations.

Timing Side-Channel Attacks

- If we have a e={1,0,1,1,0,1}, total time will be:
  6*T1 + 4*T2
- If we have a e={0,0,0,0,0,0}, total time will be:
  6*T1 + 0*T2
- By measuring time, we know the how many *1*s in *e*.

```
BigInt modexp(BigInt a, BigInt e, BigInt n) {
   BigInt result = 1;
   for (int i = bitlen(e) - 1; i >= 0; i--) {
      // always do square
      result = (result * result) % n;          } T1

      if (bit_test(e, i)) {
         // only multiply when the bit is 1
         result = (result * a) % n;            } T2
      }
   }
   return result;
}
```

Power Side-Channel Attacks

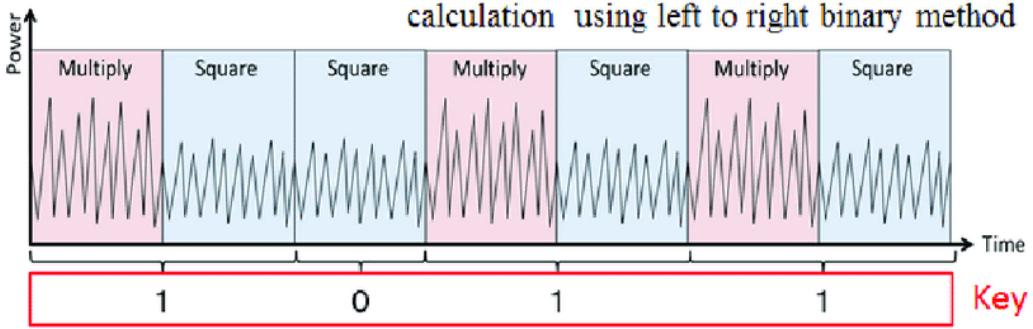
$$C = P^e \bmod N$$
$$P = C^d \bmod N$$

C: Cipher Text
P: Plain Text
e: Public Key
d: Private Key
N: modulo

(a) RSA crypto algorithm

```
input : X, N, d = (d_{k-1}, d_{k-2}, ....., d_0)
output: Z = X^d mod N
Z ← 1;
For i = k − 1 down to 0 do
   Z ← Z × Z mod N;     //Square
     if (d_i = 1) then
     Z ← Z × X mod N;   //Multiply
   end
end
return Z;
```

(b) Modular exponentiation (X^d mod N) calculation using left to right binary method



(c) Power dissipation model during modular exponentiation

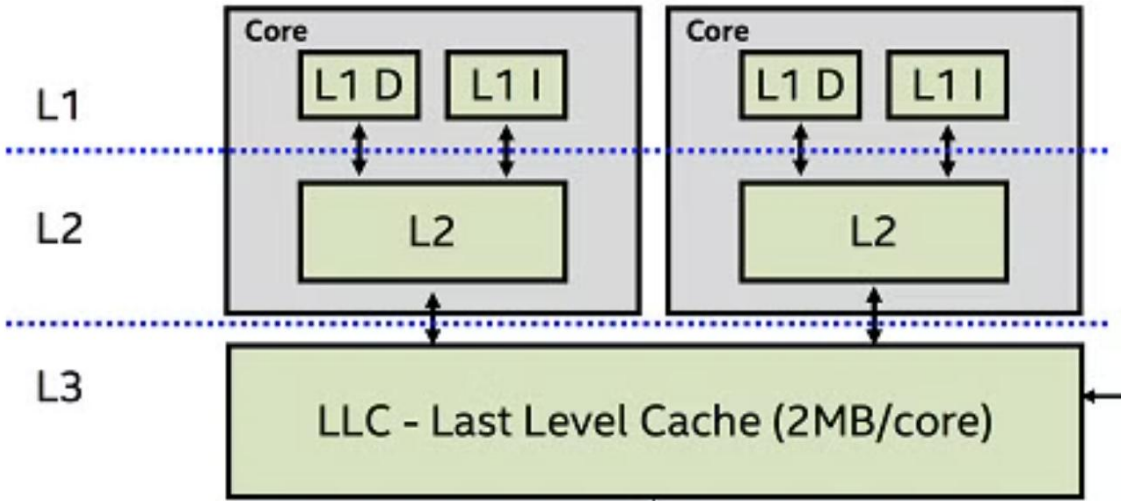
PART TWO

02

Cache Side-Channel Attacks

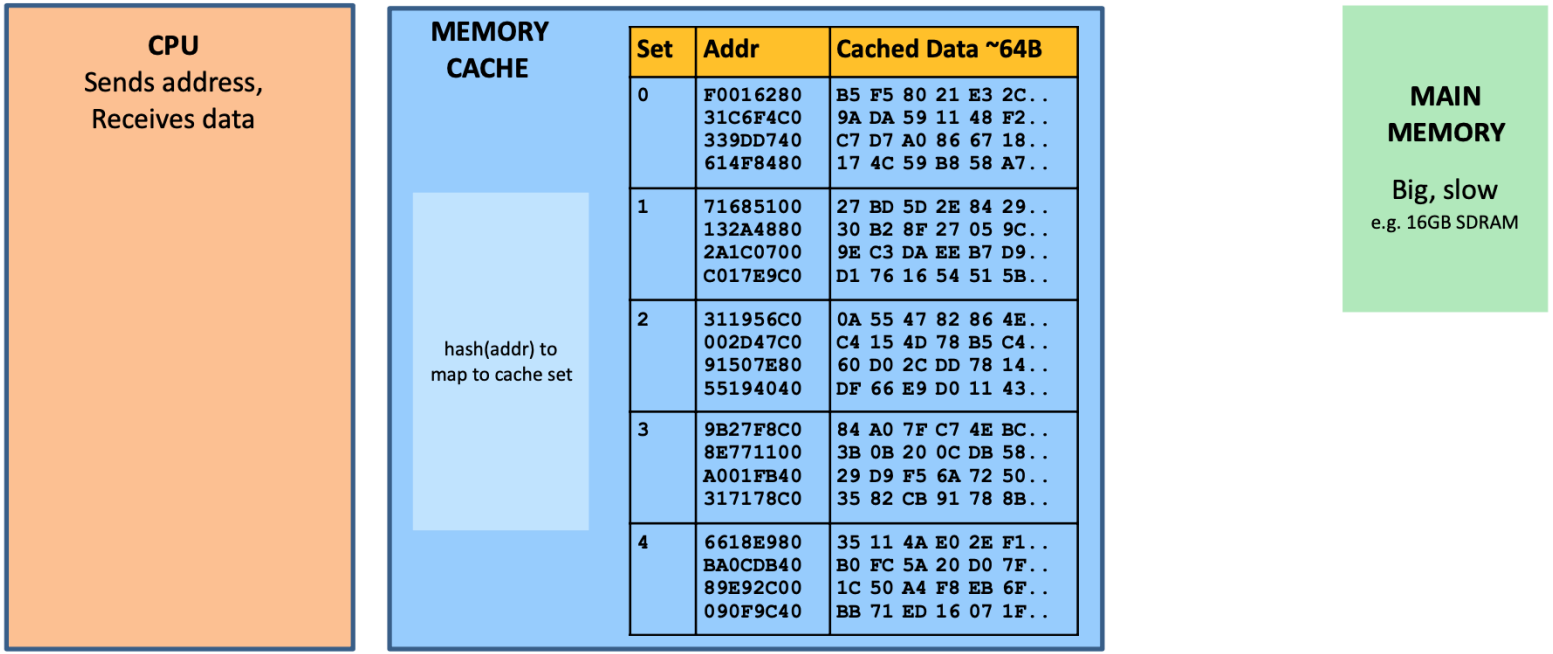
Quick Recap: Memory and Cache

- Main memory is large and slow
- Processors have faster, smaller caches to store more recently used memory closer to cores
- Caches organized in hierarchy: closer to the core are faster and smaller



Memory and Cache

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



Memory and Cache

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

CPU
Sends address,
Receives data

Addr: 2A1C0700

MEMORY CACHE

hash(addr) to map to cache set

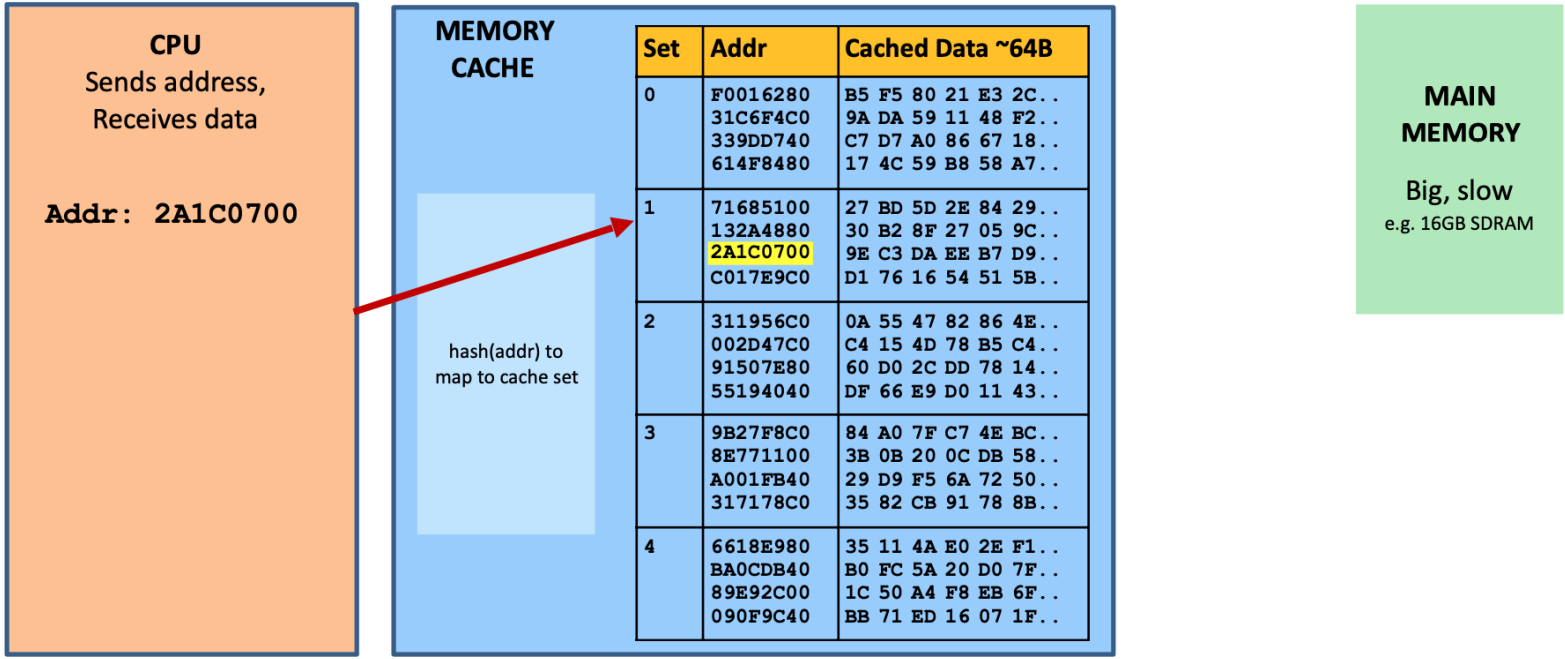
Set	Addr	Cached Data ~64B
0	F0016280 31C6F4C0 339DD740 614F8480	B5 F5 80 21 E3 2C.. 9A DA 59 11 48 F2.. C7 D7 A0 86 67 18.. 17 4C 59 B8 58 A7..
1	71685100 132A4880 2A1C0700 C017E9C0	27 BD 5D 2E 84 29.. 30 B2 8F 27 05 9C.. 9E C3 DA EE B7 D9.. D1 76 16 54 51 5B..
2	311956C0 002D47C0 91507E80 55194040	0A 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 317178C0	84 A0 7F C7 4E BC.. 3B 0B 20 0C DB 58.. 29 D9 F5 6A 72 50.. 35 82 CB 91 78 8B..
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..

MAIN MEMORY

Big, slow
e.g. 16GB SDRAM

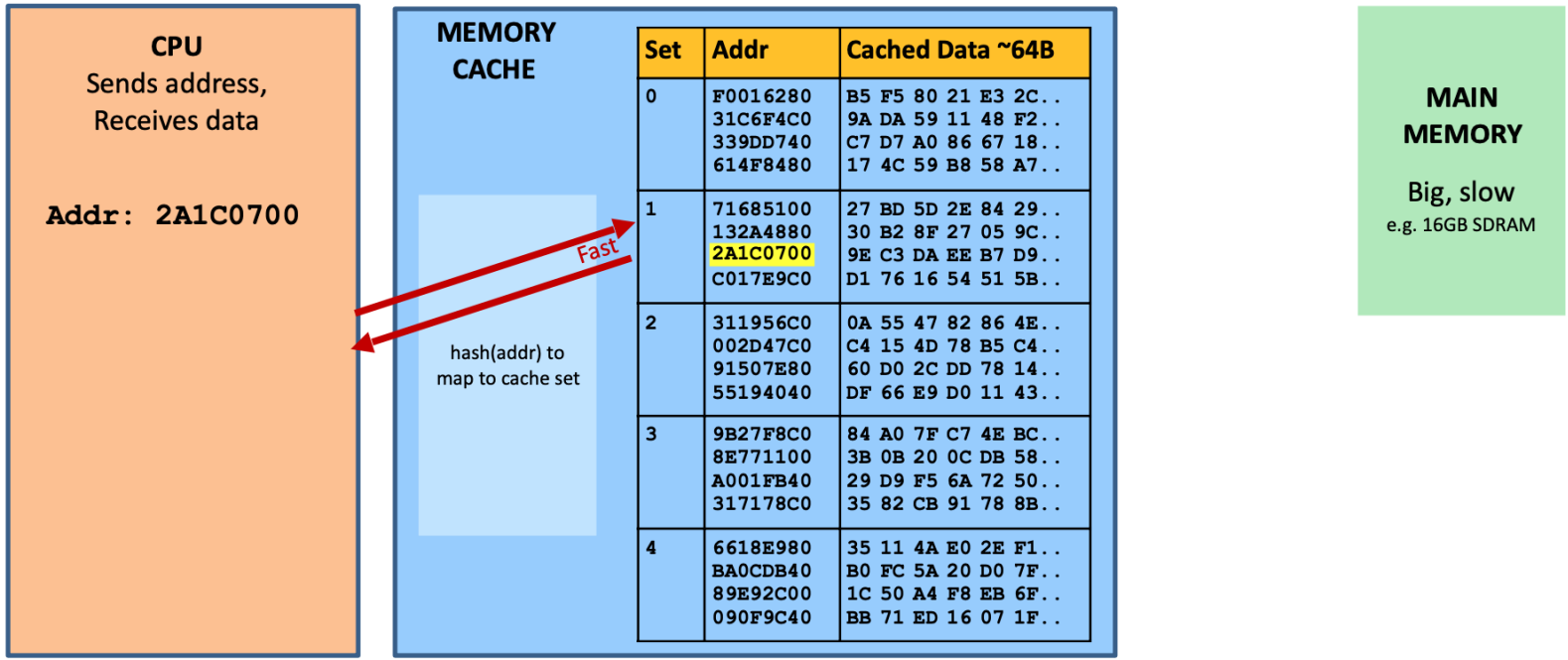
Memory and Cache

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



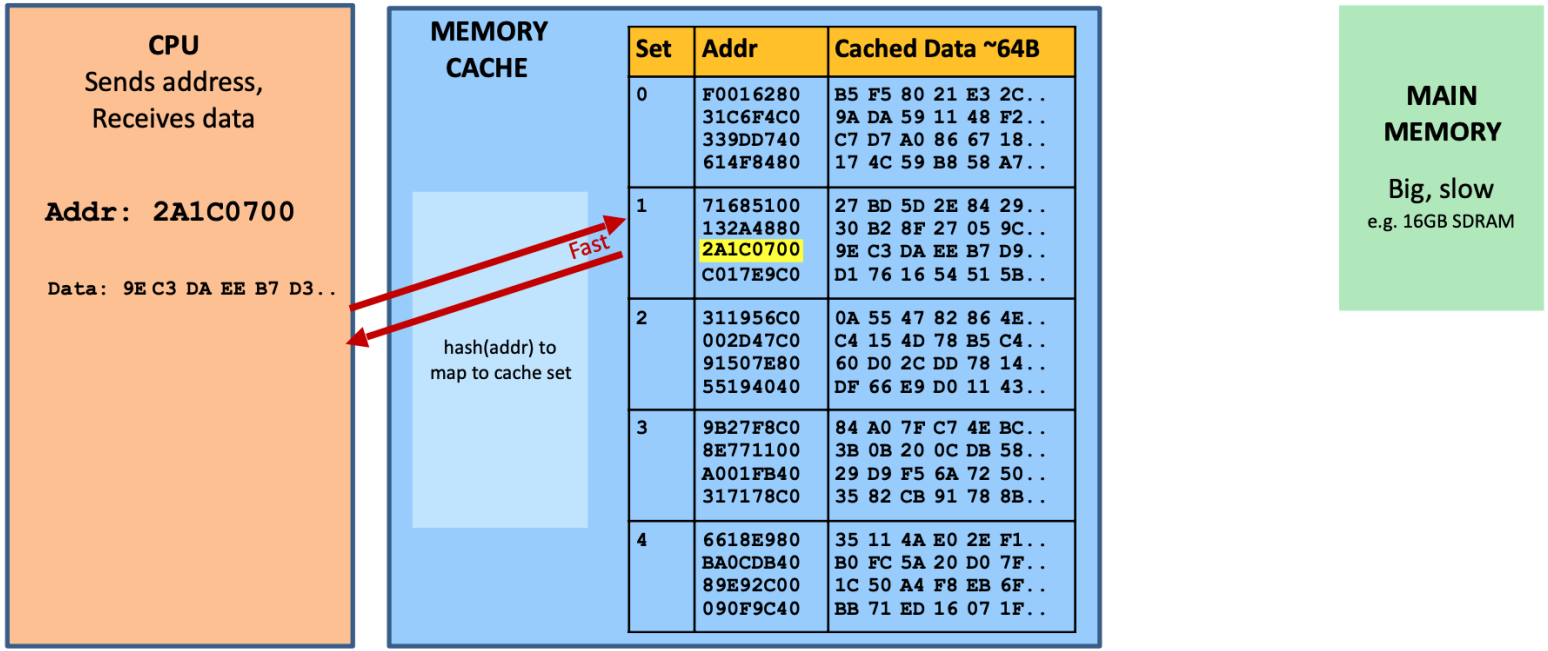
Memory and Cache

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



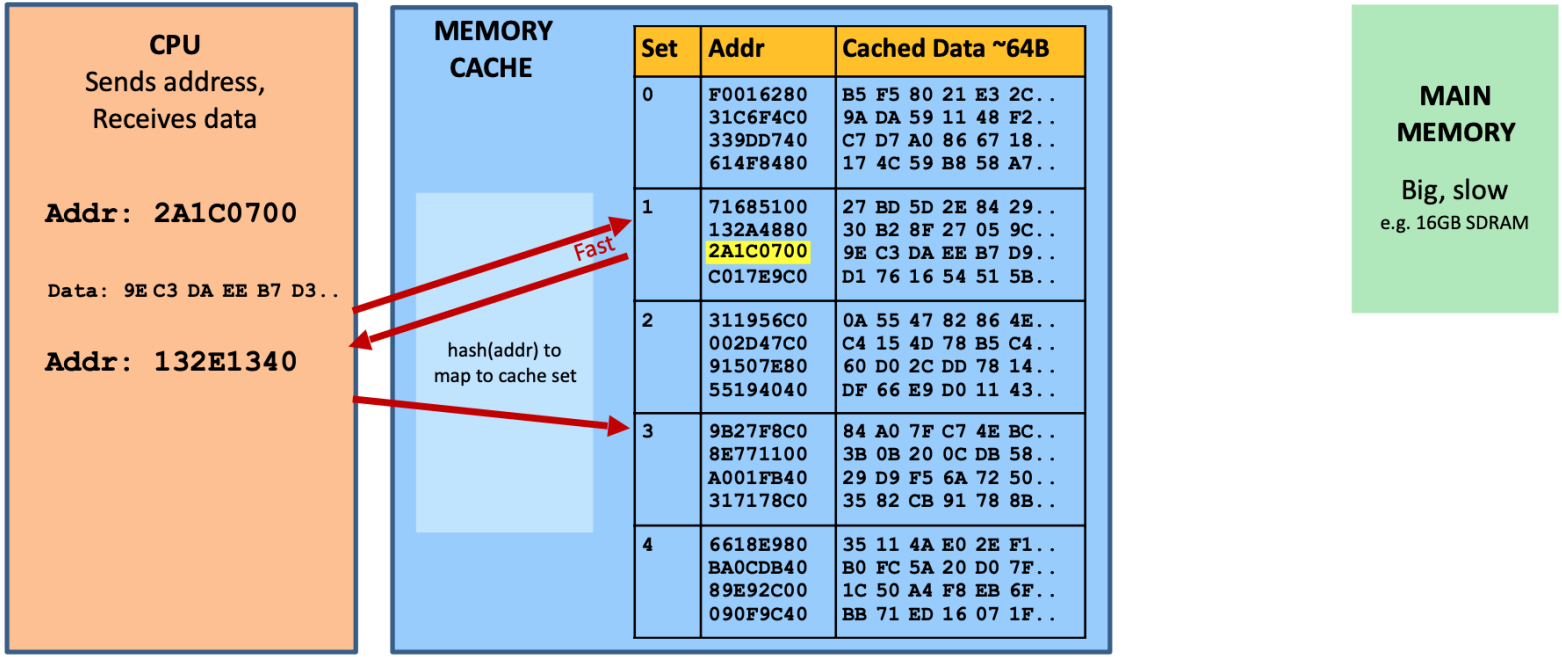
Memory and Cache

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



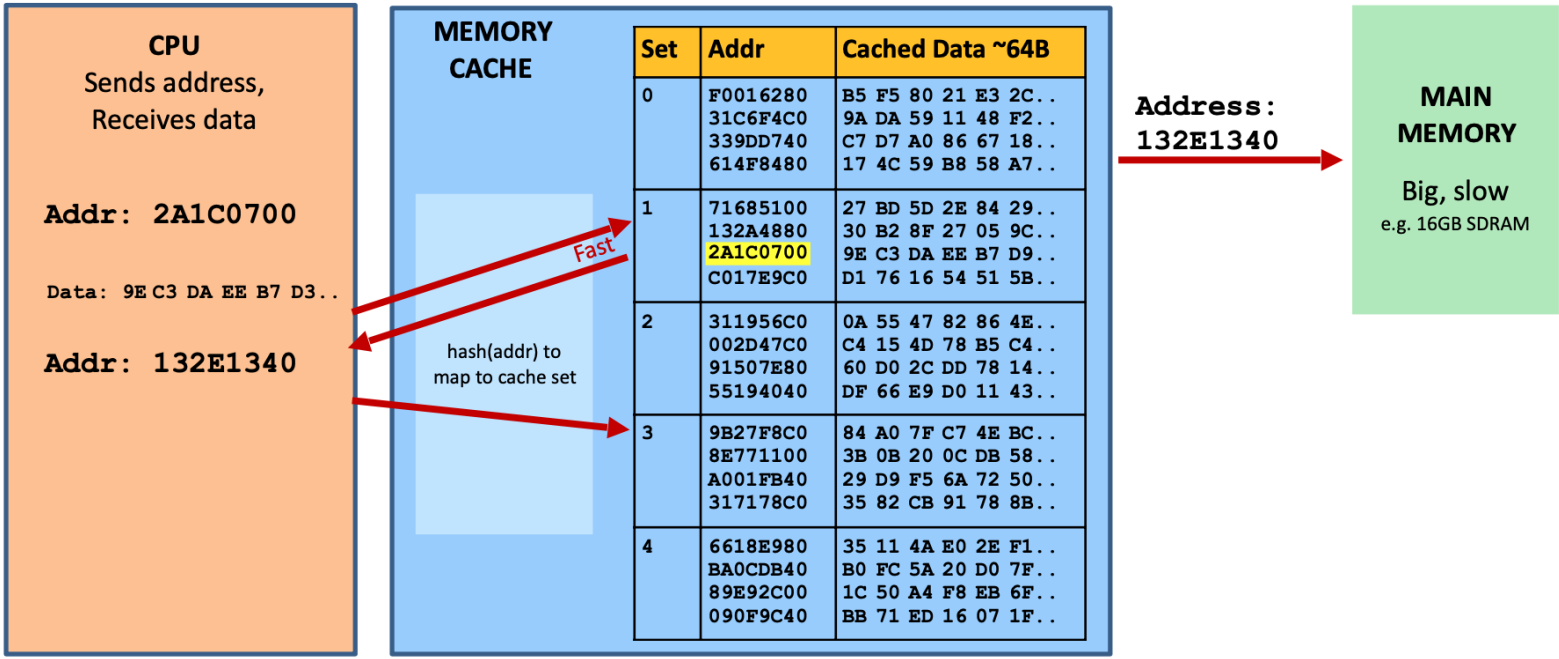
Memory and Cache

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



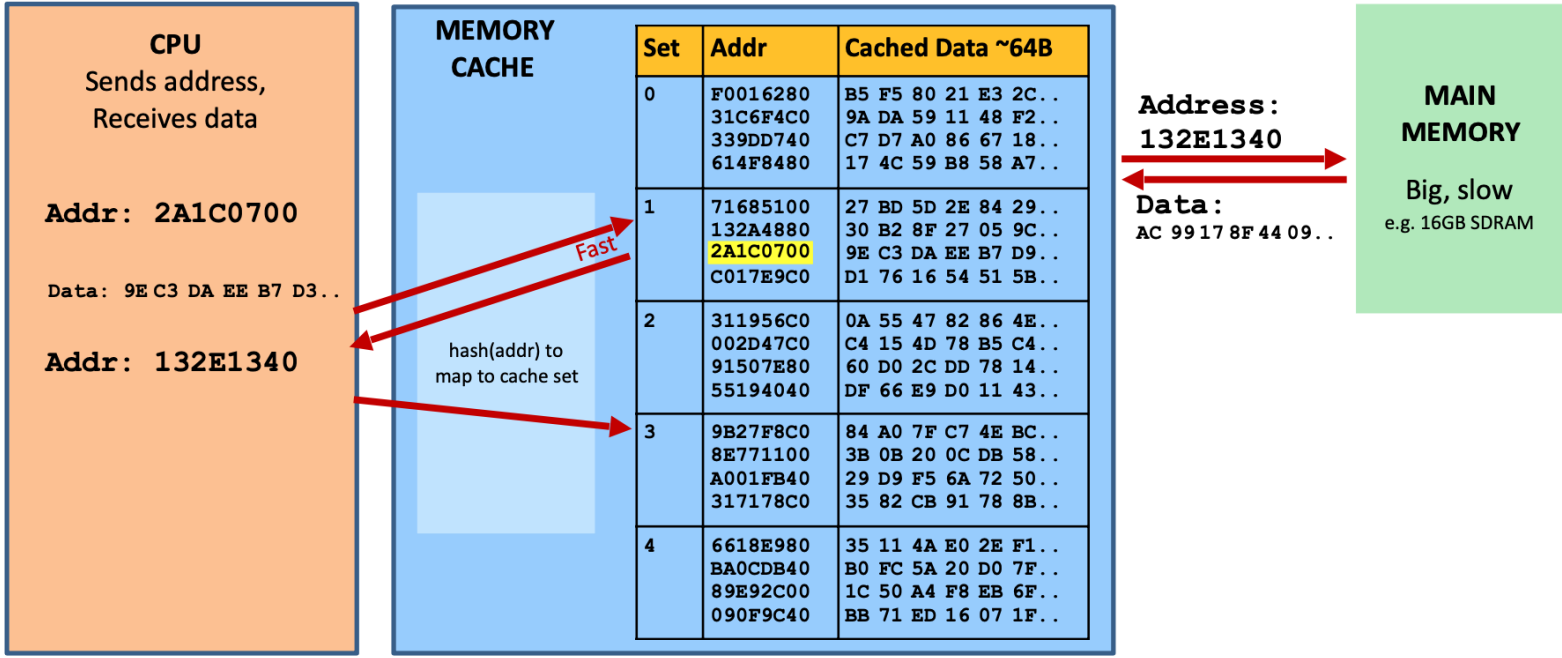
Memory and Cache

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



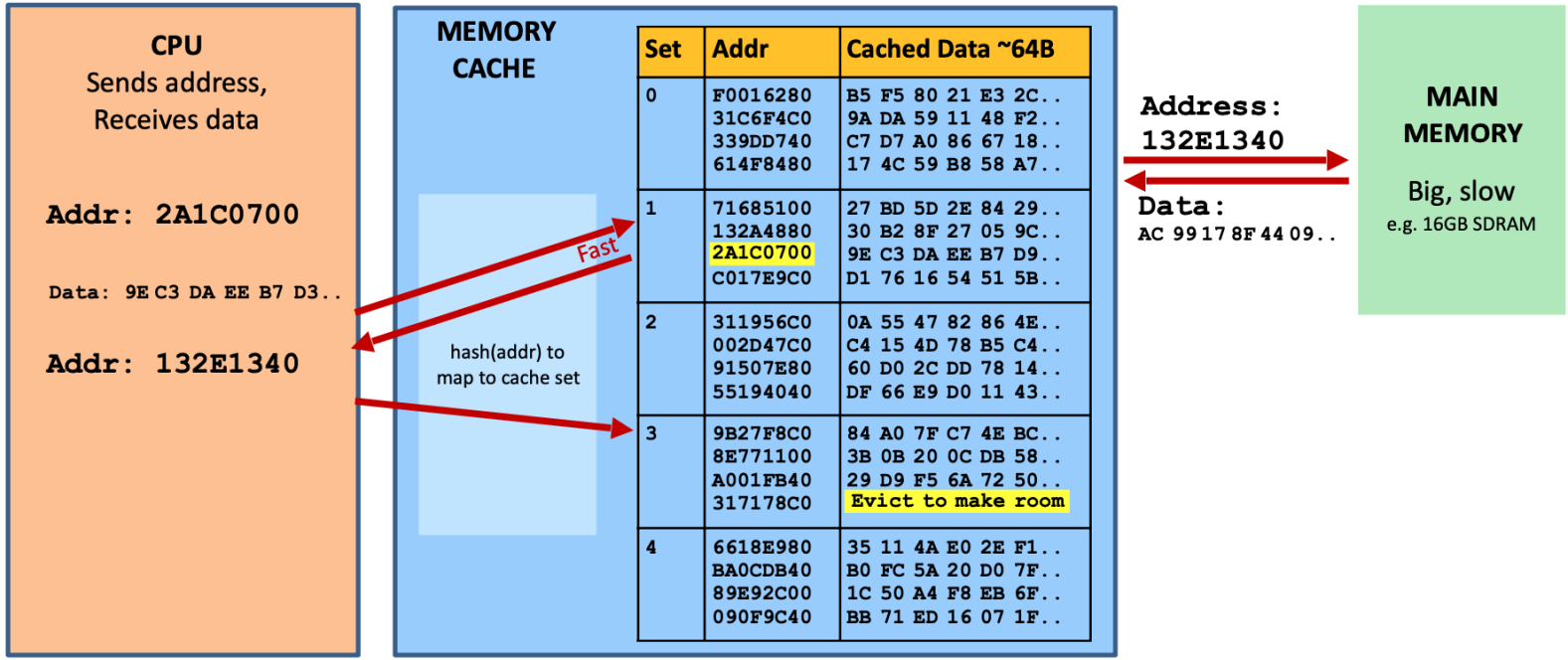
Memory and Cache

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



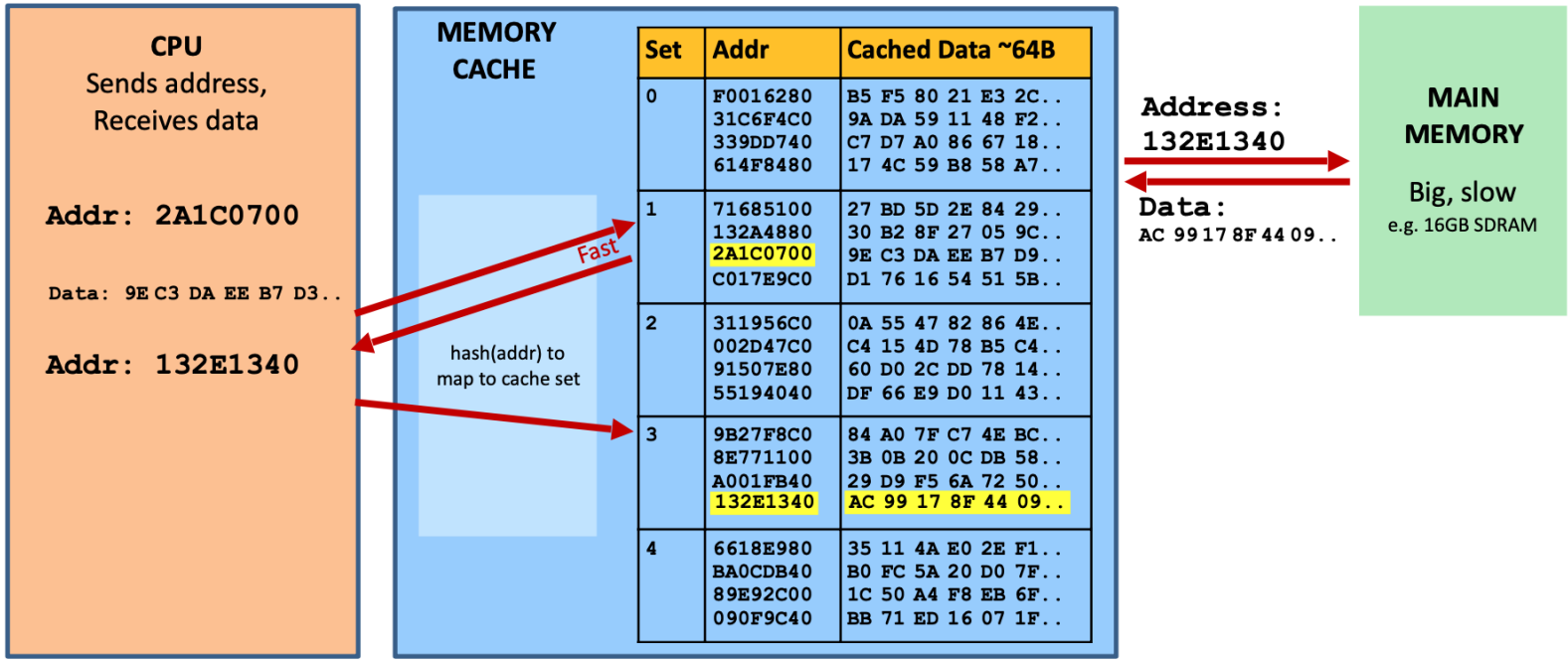
Memory and Cache

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



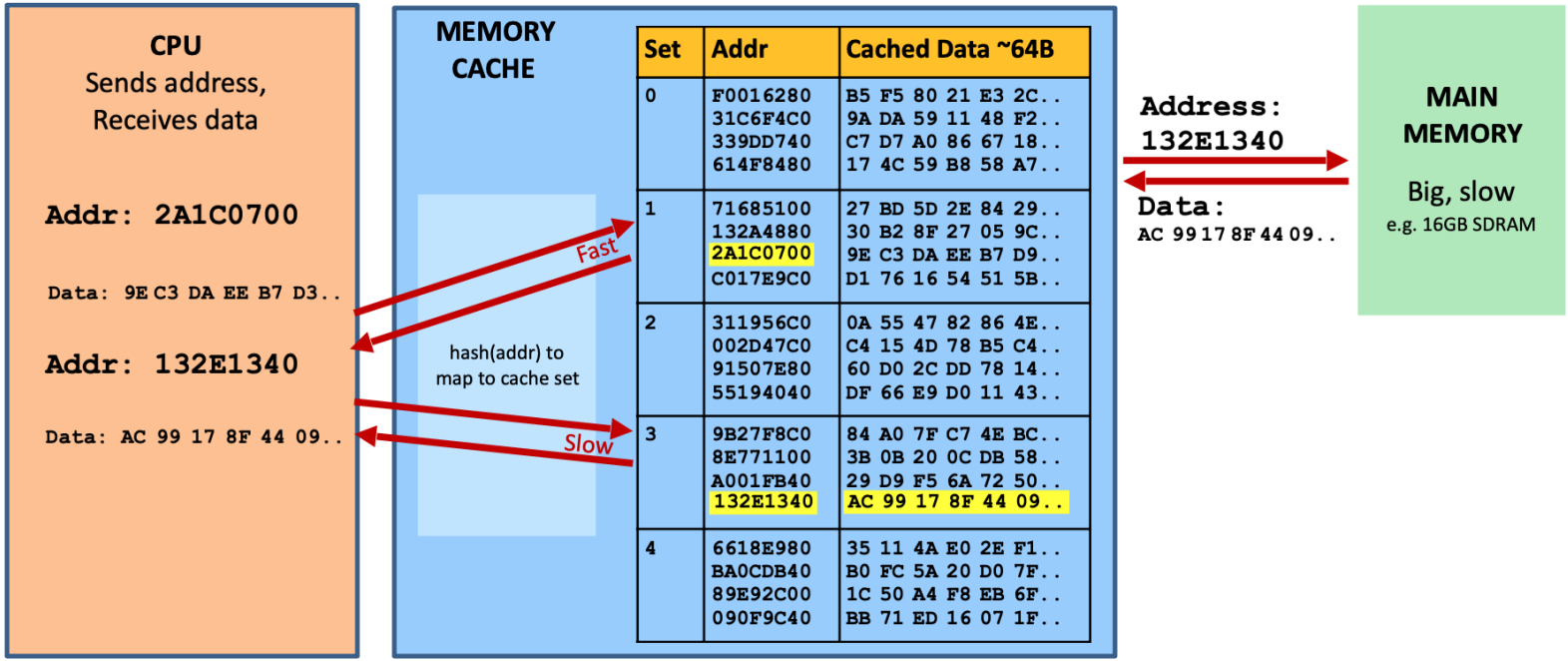
Memory and Cache

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



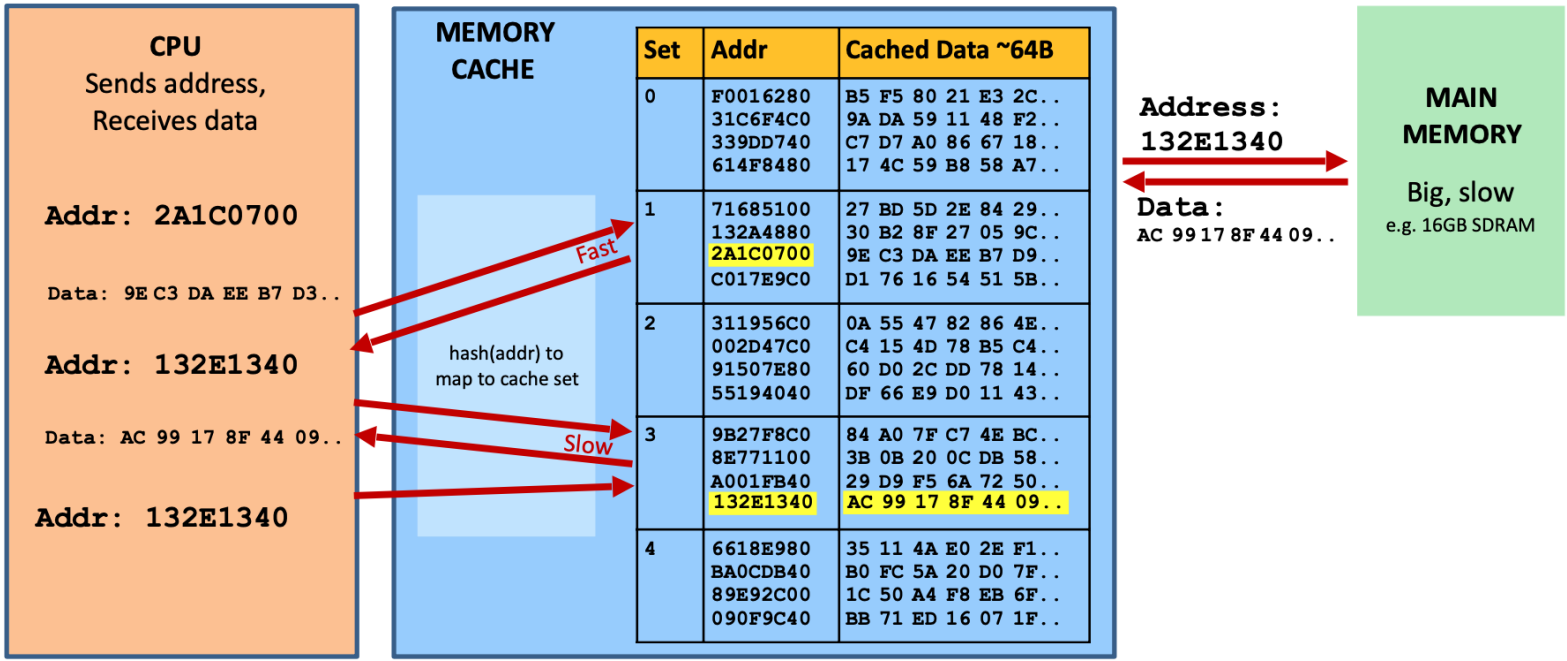
Memory and Cache

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



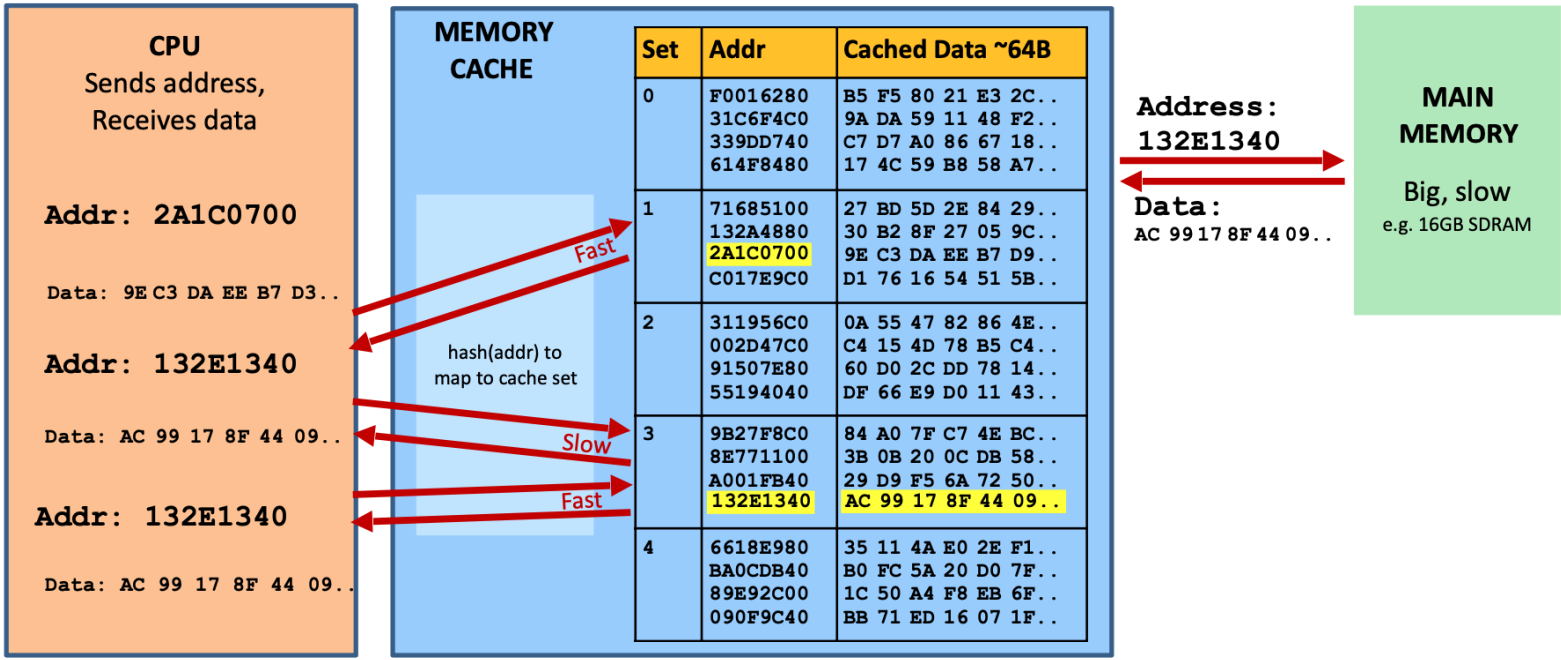
Memory and Cache

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



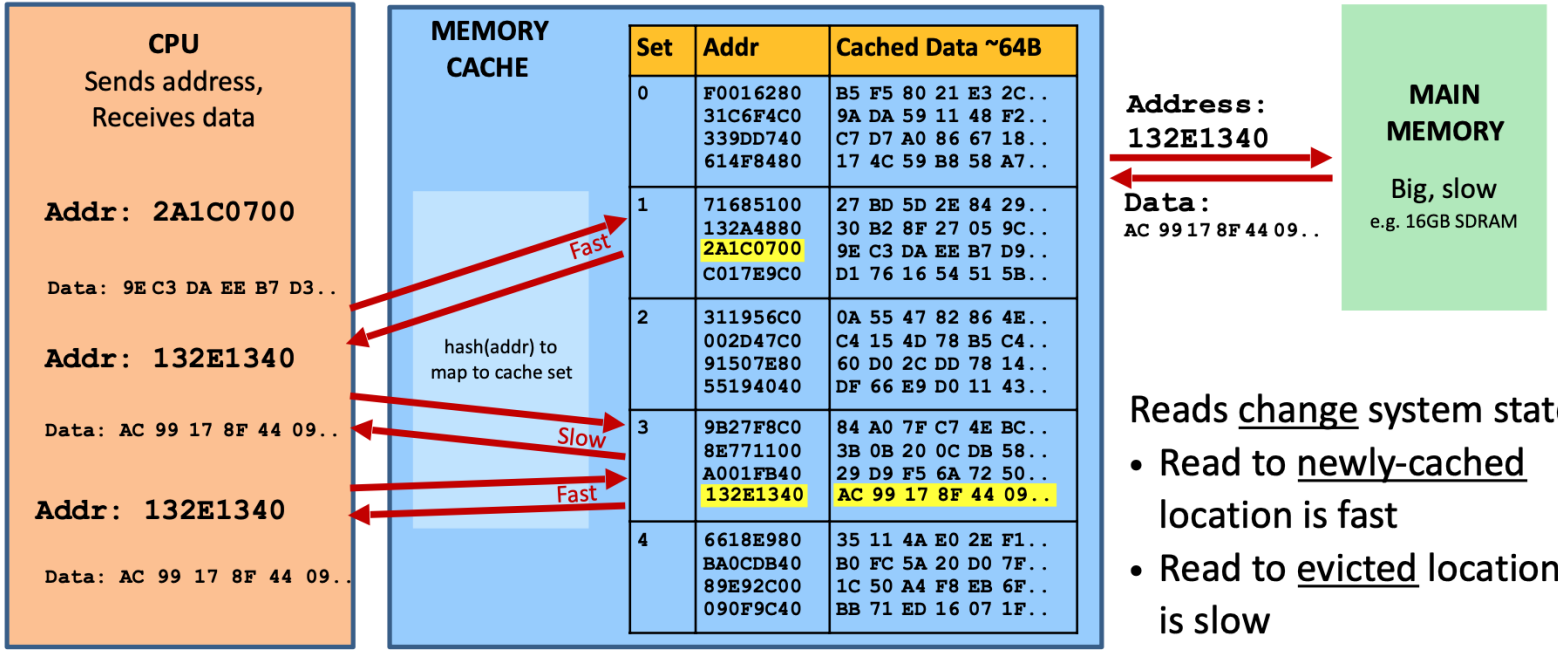
Memory and Cache

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



Memory and Cache

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

Cache timing side channel attacks

- Caches are a shared system resource
- Not isolated by process, VM, or privilege level
- An attacker who can run code on same physical hardware can abuse this shared resource to learn information from another process or VM

Cache timing attack threat model

- Attacker and victim are isolated (separate processes) on same physical system
- Attacker is able to invoke (directly or indirectly) functionality exposed by victim
- Examples?
- Attacker does not have direct access to contents of victim memory
- Attacker will use shared cache access to infer information about victim memory contents

Cache timing attack vector

- Many algorithms have memory access patterns that are dependent on sensitive memory contents
  - Examples?
- If attacker can observe access patterns they can learn secrets

Cache timing attack options

- **Prime**: Place a known address in the cache by reading it
- **Evict**: Access memory until address is no longer cached (force capacity misses)
- **Flush**: Remove an address from the cache (clflush on x86)
- **Mesaure**: Precisely (down to the cycle) how long it takes to do something (rdtsc on x86)
- **Attack form**: Manipulate cache into known state, make victim run, infer what changed after run

Three basic techniques

- Evict and time
  - Evict things from the cache and measure if victim slows down as a result
- Prime and probe
  - Place things in the cache, run the victim, and see if you slow down as result
- Flush and reload
  - Flush a particular line from the cache, run the victim, and see if your accesses are still fast

PART THREE

03

Spectre Attack

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

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 <u>without</u> 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

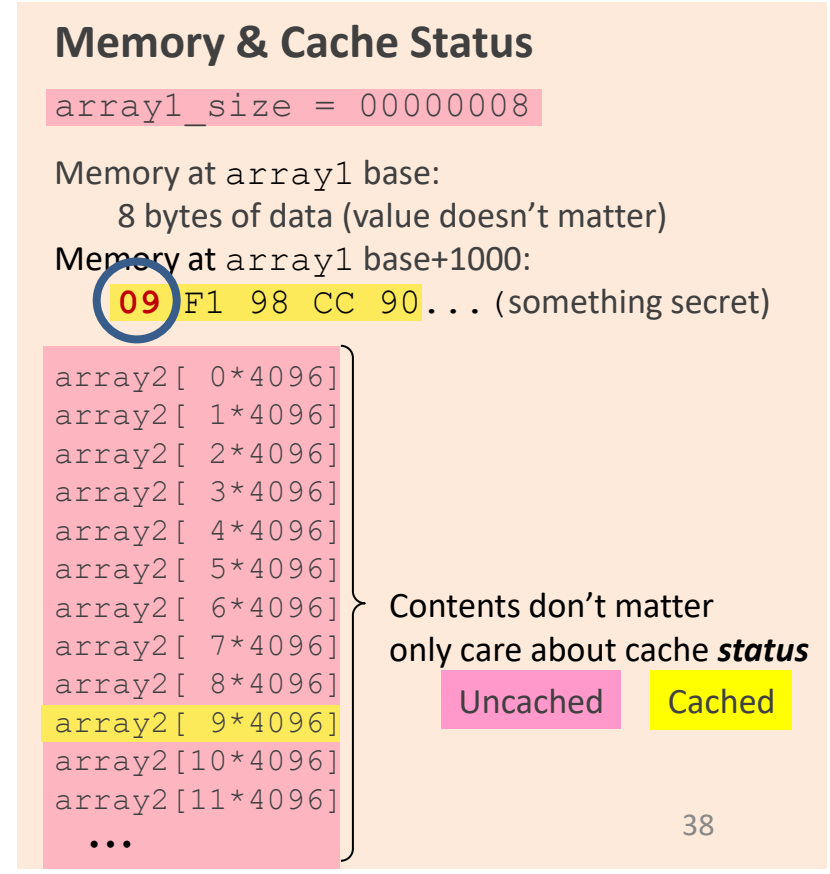
Conditional branch (Variant 1) attack

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

Attacker calls victim with x=1000

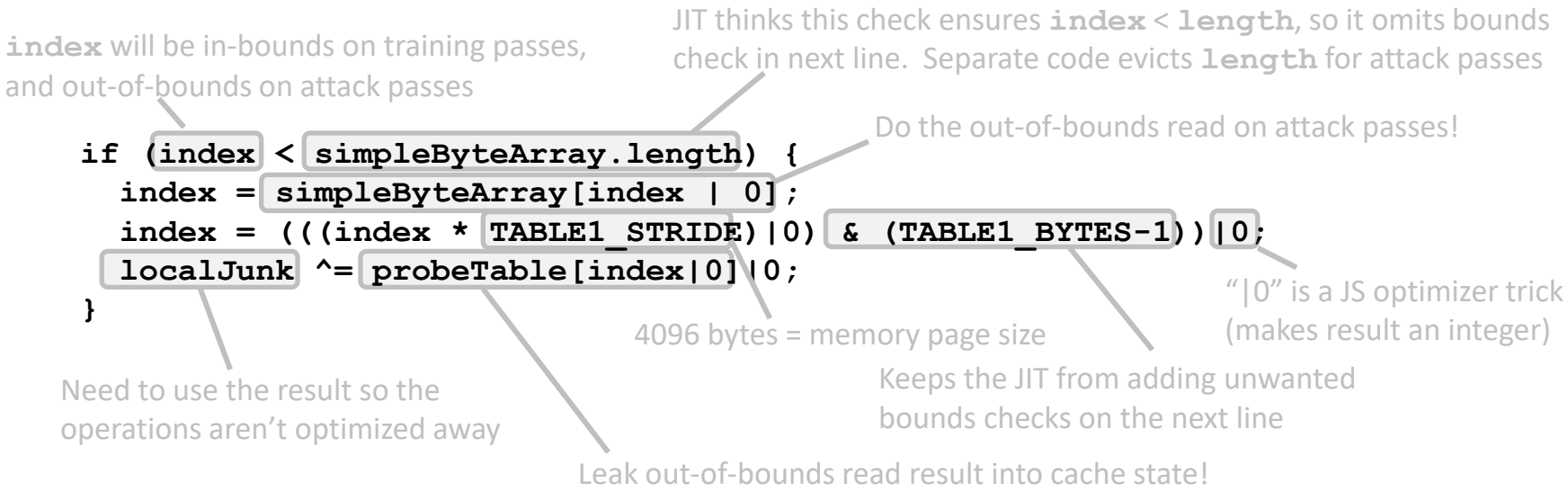
Attacker: (another process or core)

- 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)



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

... 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, TDX)

- 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