Side-Channel Attacks

CSE 127: Introduction to Security

Deian Stefan

Winter 2021

Some material from Dan Boneh, Stefan Savage, Nadia Heninger, Keegan Ryan

Last time

Isolation is key to building secure systems

- Basic idea: protect secrets so that they can't be accessed across a trust boundary
- Mechanisms: privilege separation, least privilege, complete mediation
- **Assumption:** We know what the trust boundaries are and can control access

How can attackers access protected data?

- Find a bug in an unprotected program
- Find a bug in the kernel, VMM, or runtime system providing protection
- Find a hardware bug that lets you bypass isolation

The power of abstraction in computer science

"All problems in computer science can be solved by another level of indirection." – David Wheeler

- Computer systems are often built on layers of abstraction
- Physics $\rightarrow$ hardware $\rightarrow$ operating system $\rightarrow$ applications
- An ideal abstraction allows each layer to treat the layer below as a black box with well-defined behavior

Side channels

Implementations have artifacts and side effects

- How long, how fast, how loud, how hot
- A side channel is a source of information beyond the output specified by an abstraction

Today

- Overview and history of side channels
- Cache side channels and countermeasures
- Spectre attacks

Soviet Great Seal Bug

- 1945 Soviet gift to ambassador
- Contained passive listening device
- Would transmit when illuminated at a particular radio frequency
- Designed by Theremin
- Discovered in 1952

https://en.wikipedia.org/wiki/The_Thing_(listening_device)

TEMPEST: US/NATO side channel codename

- WWII: Bell Telephone discovers electromagnetic leakage in one-time pad teleprinters: 100-ft radius
- 1951: CIA rediscovers teleprinter leakage; 200-ft radius
- 1964: TEMPEST shielding rules established

van Eck Phreaking

“Electromagnetic Radiation from Video Display Units: An Eavesdropping Risk?” Wim van Eck 1985

- 1985: Wim van Eck demonstrates side channel image recovery from CRT monitors with off-the-shelf equipment

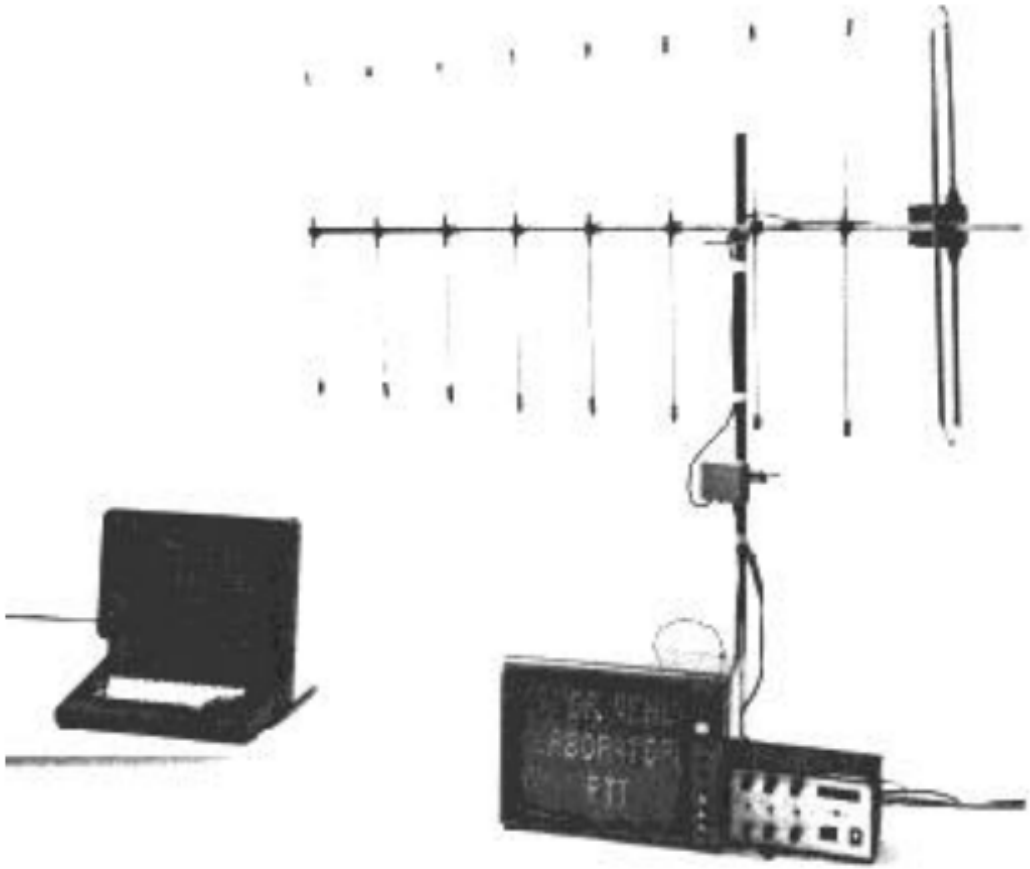
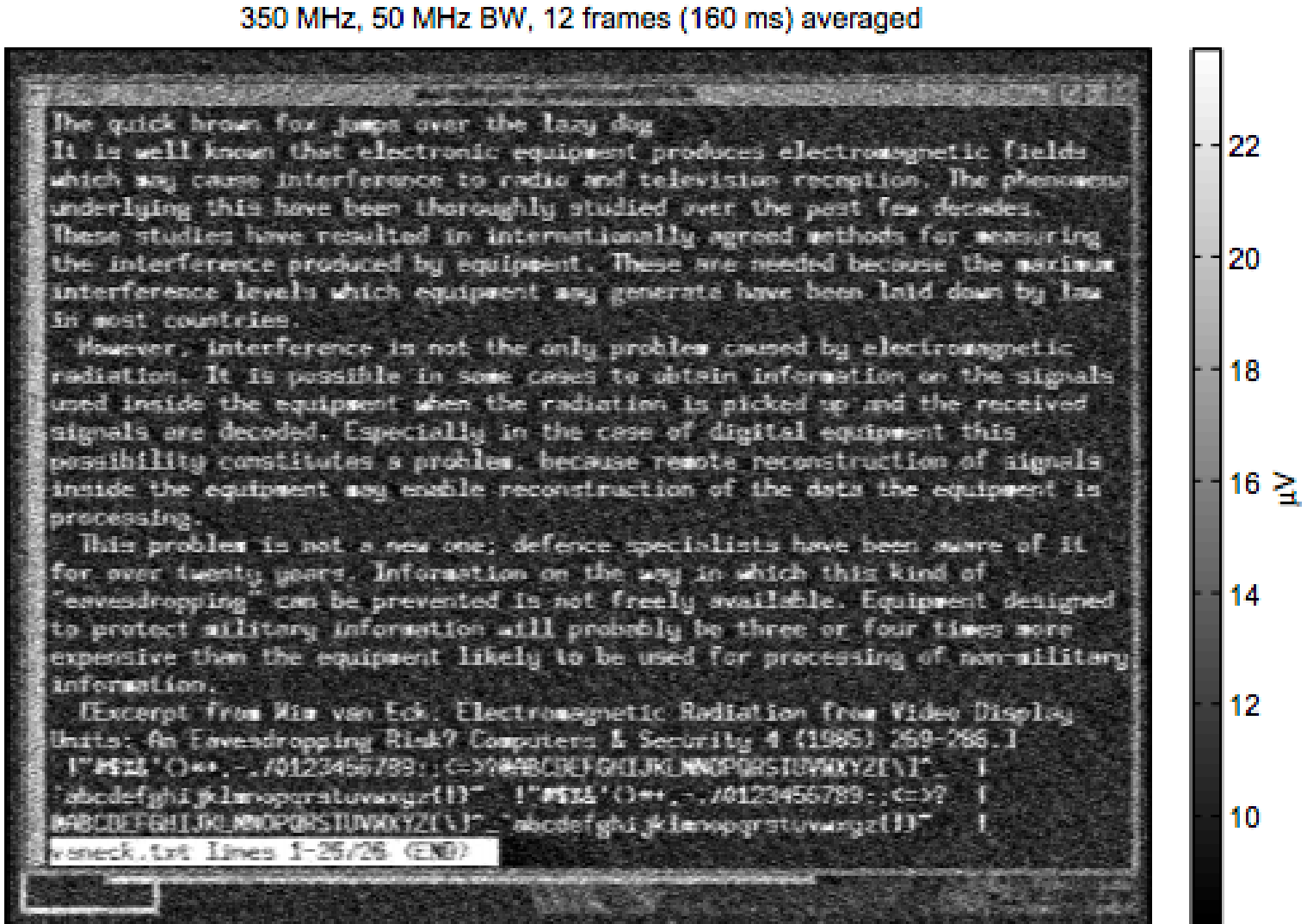


Fig. 1. Eavesdropping eel-up using a variable oscillator and a frequency divider to restore synchronization. The picture on the TV is picked up from the radiation of the VDU in the background.

“Electromagnetic Eavesdropping Risks of Flat-Panel Displays” Kuhn 2004

- Image displays simultaneously along line
- Pick up radiation from screen connection cable



Two classes of side channels

Emission: What out-of-band signal is generated in the course of performing the operation?

- Electromagnetic radiation
  - Voltage running through a wire produces a magnetic field
- Sound (acoustic attacks)
  - Capacitors discharging can make noises
- Error messages

Two classes of side channels

Emission: What out-of-band signal is generated in the course of performing the operation?

- Electromagnetic radiation
  - Voltage running through a wire produces a magnetic field
- Sound (acoustic attacks)
  - Capacitors discharging can make noises
- Error messages

Consumption: How much of a resource is being used to perform an operation?

- Timing
  - Different execution time due to program branches
  - Different execution time due to CPU cache state
- Power consumption
- Network traffic

How long does this password check take?

```
char pwd[] = ‘‘z2n34uzbnqhw4i’’;

//...

int check_password(char *buf) {
    return strcmp(buf, pwd);
}
```

Tenex password verification bug

Alan Bell 1974

- Early virtual memory implementaion in Tenex computer system.
- Character-at-a-time comparision + interrupt on memory page
- Linear-time password recovery

“Timing Analysis of Keystrokes and Timing Attacks on SSH”

Song Wagner Tian 2001

- In interactive SSH, keystrokes sent in individual packets
- Build model of inter-keystroke delays by finger, key pair
- Measure packet timing off network, do Viterbi decoding

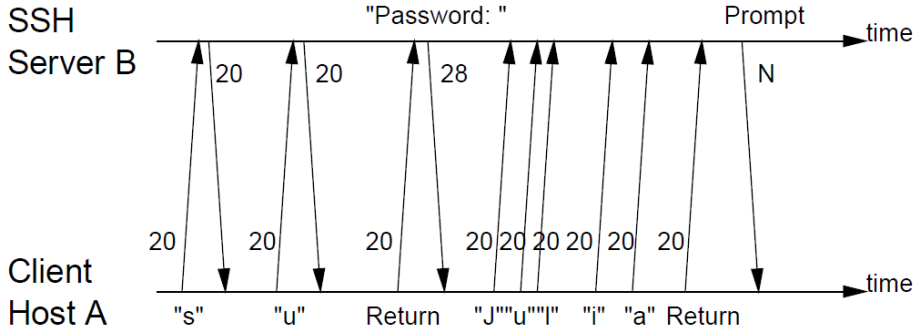


Figure 1: The traffic signature associated with running SU in a SSH session. The numbers in the figure are the size (in bytes) of the corresponding packet payloads.

Power Analysis Attacks

Kocher Jaffe Jun 98

Side-channel attacks can also leak cryptographic secrets.

Simple power analysis and differential power analysis exploit secret-dependent power consumption.

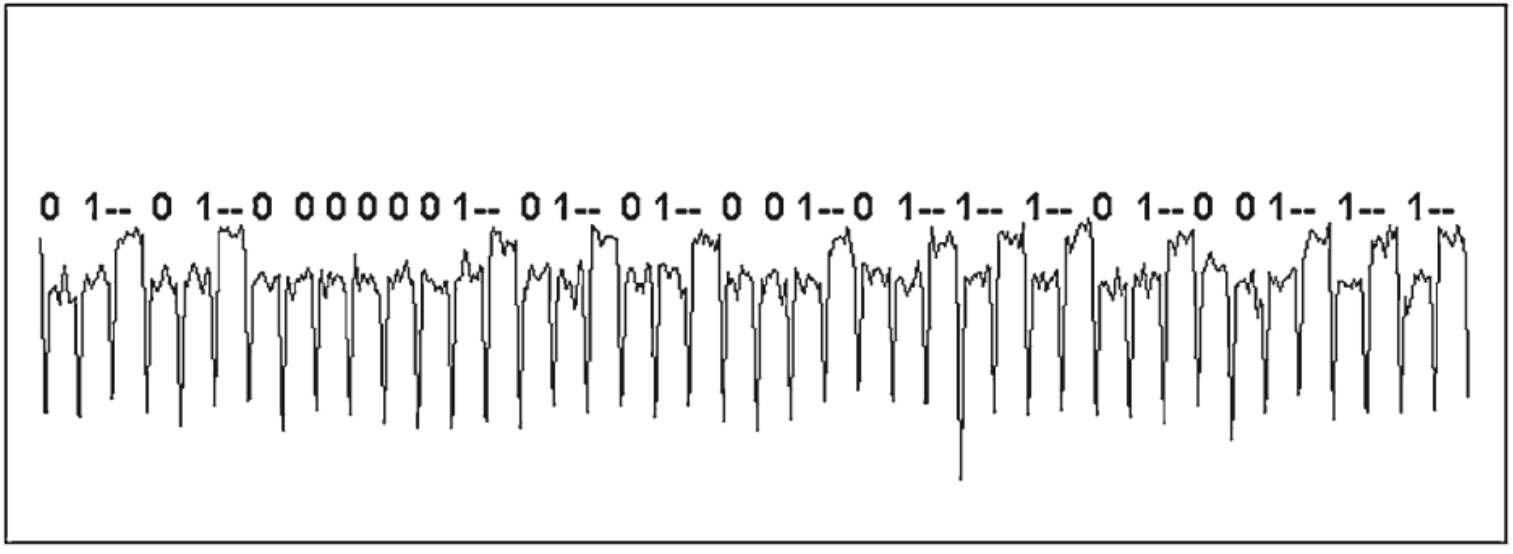
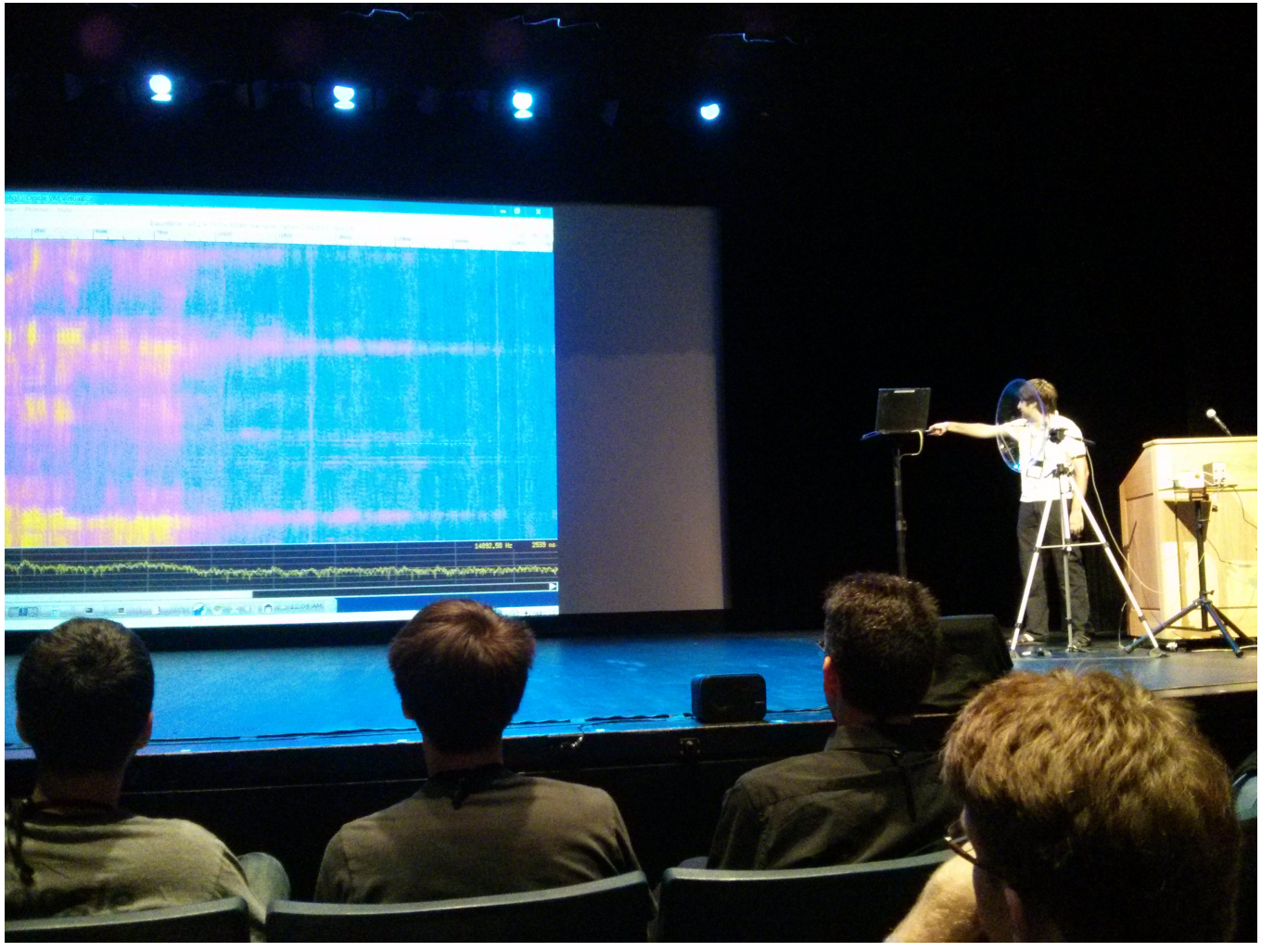


Fig. 11 SPA leaks from an RSA implementation

Acoustic Attacks

Genkin Shamir Tromer 2014

Computers emit high-pitched noise; these acoustic emanations can leak crypto keys.



Browser History Sniffing

Jang, Jhala, Lerner, Shacham 2010
revisited: Smith, Disselkoen, Narayan, Brown, Stefan 2018

Default web browser behavior: unvisited links are blue and visited links are purple.

Text display attributes available to scripts via DOM.

Victim browser visits malicious website. Malicious website enumerates URLs in invisible portion of site to sniff browser history.

Exploited in the wild. In 2012 FTC went after Epic Marketplce.

“Fixed” in 2010, new browser features re-exposed the side channel.

Active side channels

- Fault attacks induce computational errors that may leak information
- Attackers can induce faults by:
  - Glitch power, voltage, clock
  - Vary temperature
  - Subject to light, EM radiation

Using Memory Errors to Attack a Virtual Machine

Govindavajhala Appel 2003

Java heap overflow via glitched address of function pointer.

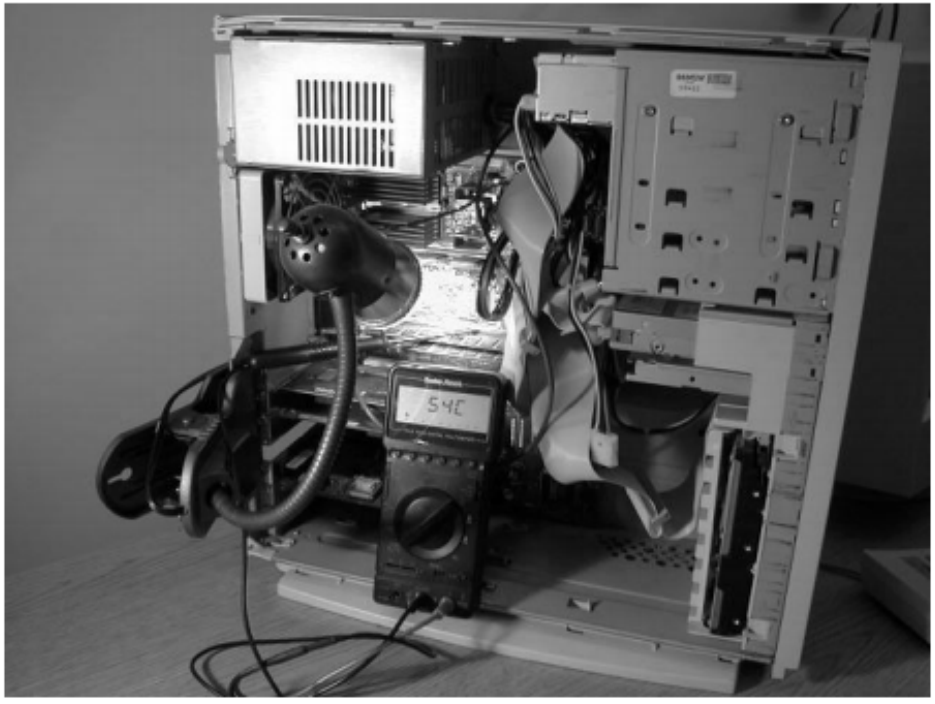


Figure 3. Experimental setup to induce memory errors, showing a PC built from surplus components, clip-on gooseneck lamp, 50-watt spotlight bulb, and digital thermometer. Not shown is the variable AC power supply for the lamp.

Types of RAM

- Volatile memory: Data retained while power is on
- Persistent memory like flash or magnetic disks retains data without power

Types of RAM

- Volatile memory: Data retained while power is on
- Persistent memory like flash or magnetic disks retains data without power

SRAM

- SRAM retains bit value while power is on without refresh
- Faster, lower density, higher cost
- Has a “burn-in” phenomenon where on startup it tends to flip bit to “remembered bit”

Types of RAM

- Volatile memory: Data retained while power is on
- Persistent memory like flash or magnetic disks retains data without power

SRAM

- SRAM retains bit value while power is on without refresh
- Faster, lower density, higher cost
- Has a “burn-in” phenomenon where on startup it tends to flip bit to “remembered bit”

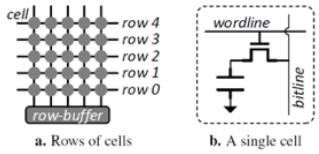
DRAM

- DRAM requires periodic refresh to retain stored value
- Capacitors charged to store data
- Higher density, lowered cost
- “Cold boot attacks” exploited capacitor discharge time to read sensitive data from physical memory

Rowhammer attacks

Seaborn and Dullien 2015

- DRAM cells are grouped into rows
- All cells in a row are refreshed together



a. Rows of cells **b.** A single cell

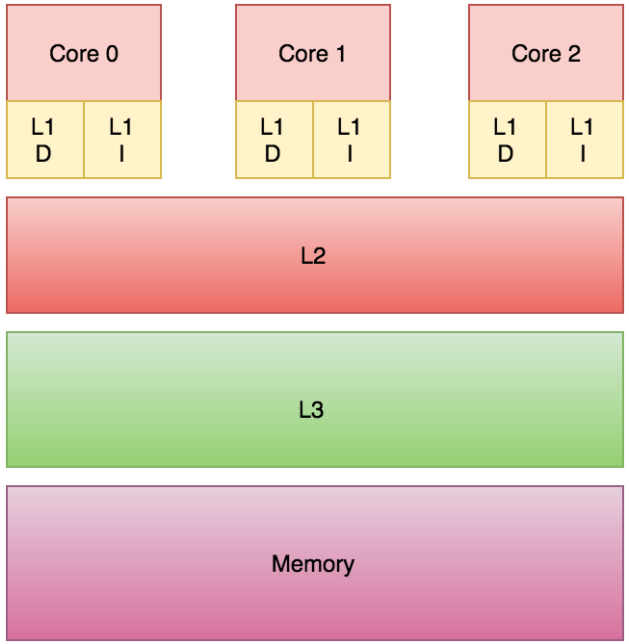
  - Repeatedly opening and closing a row within a refresh interval causes disturbance errors in adjacent rows.
  - Attacker running on the same machine as victim can cause bits to flip in victim's memory
  - Rowhammer attacks $\Rightarrow$ carry privilege escalation attacks

Today

- Overview and history of side channels
- Cache side channels and countermeasures
- Spectre attacks

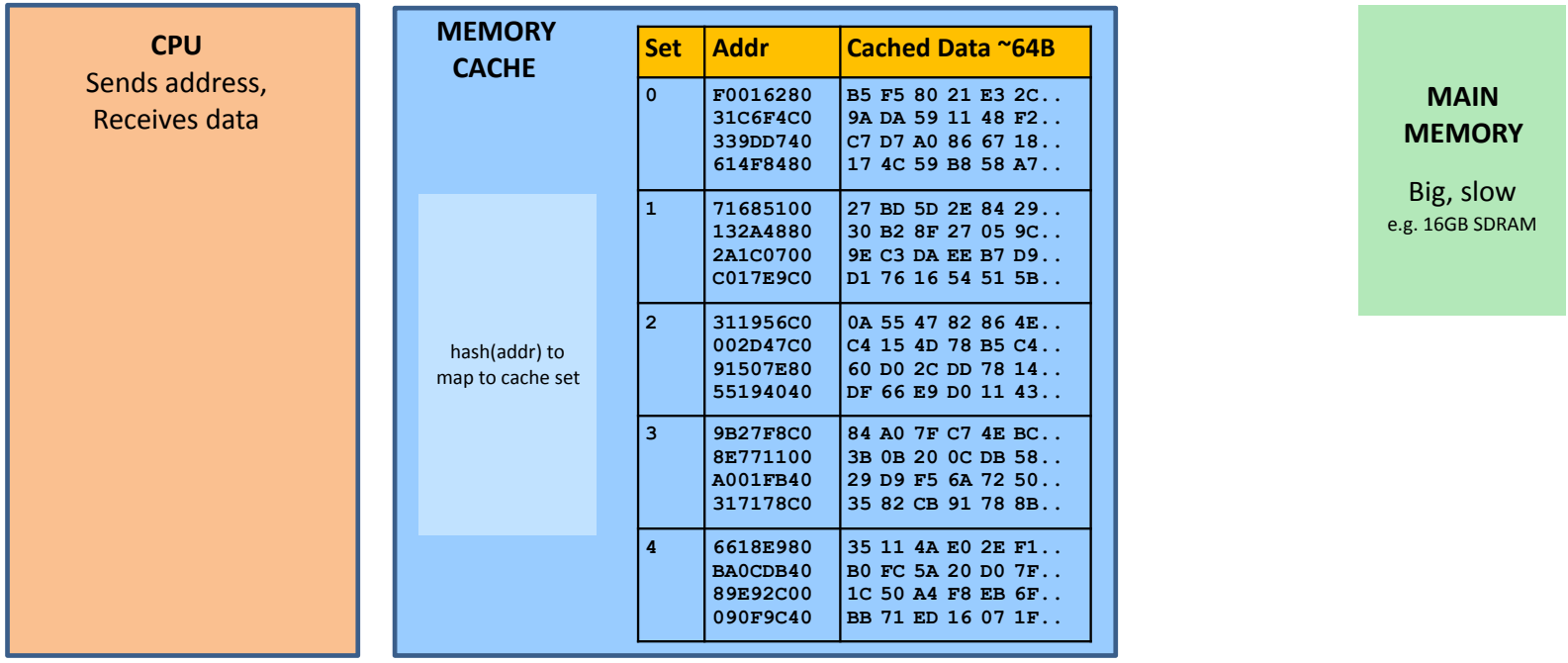
Memory and cache

- 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



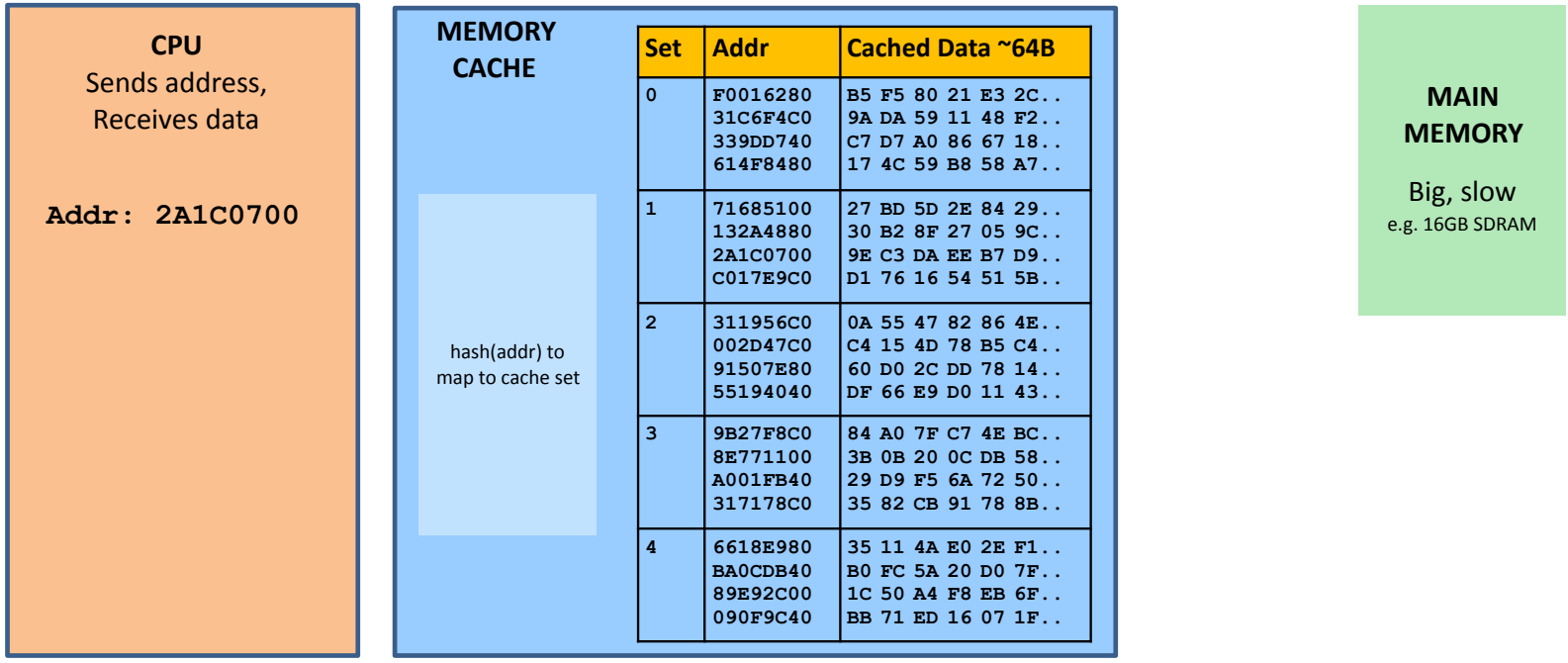
Memory and cache

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



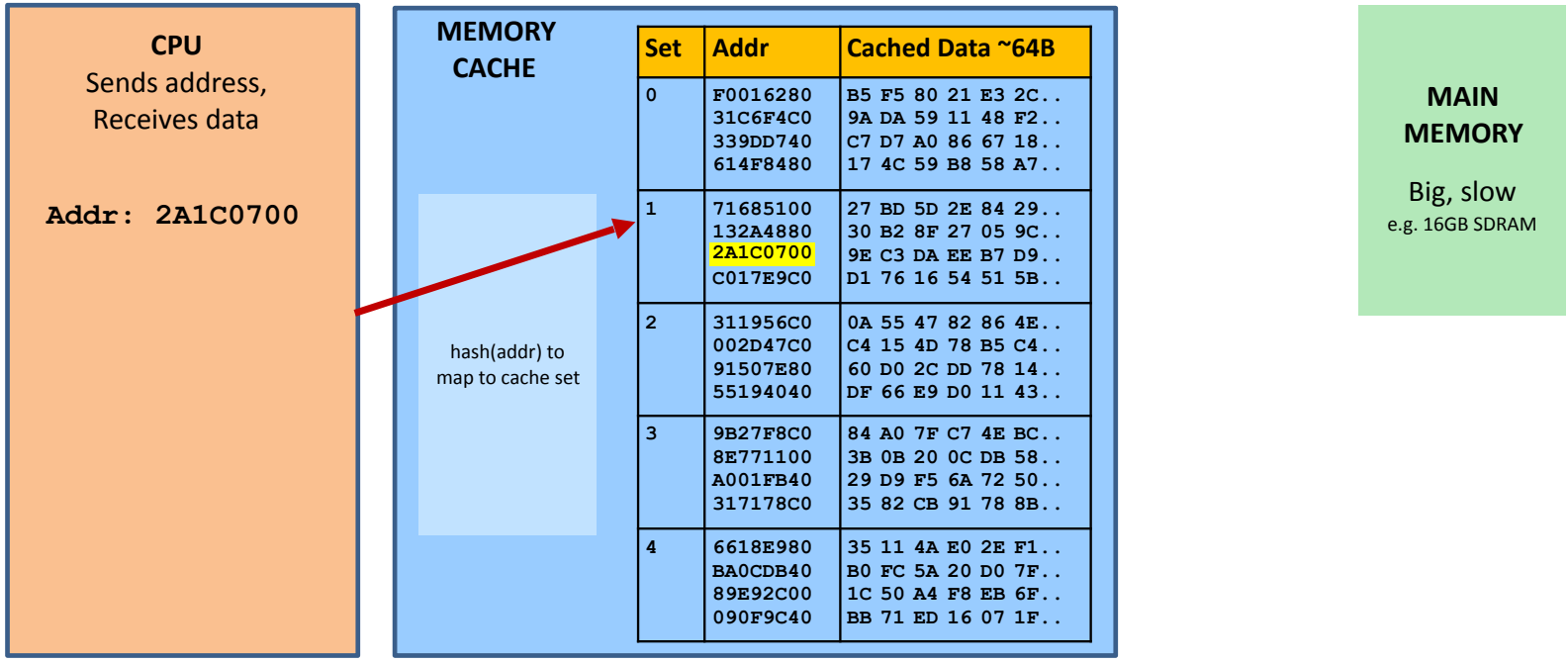
Memory and cache

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



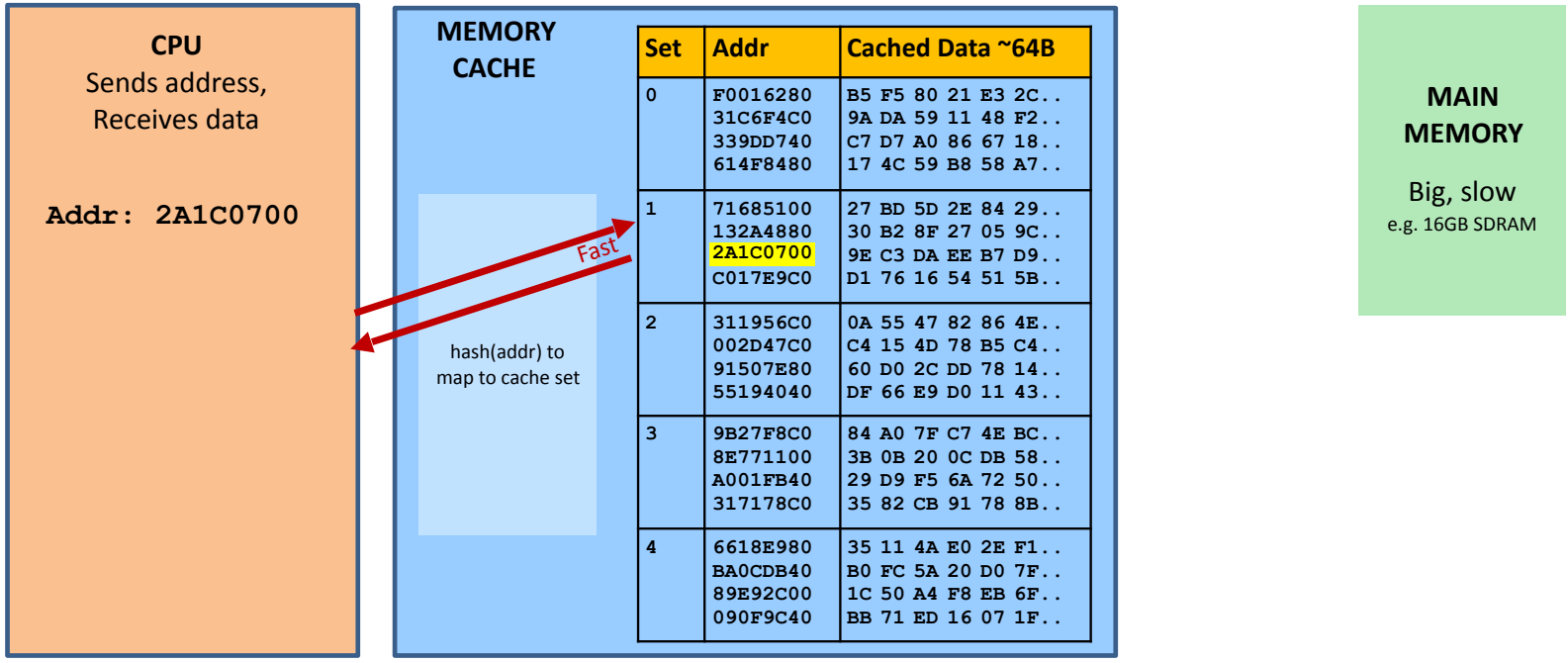
Memory and cache

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



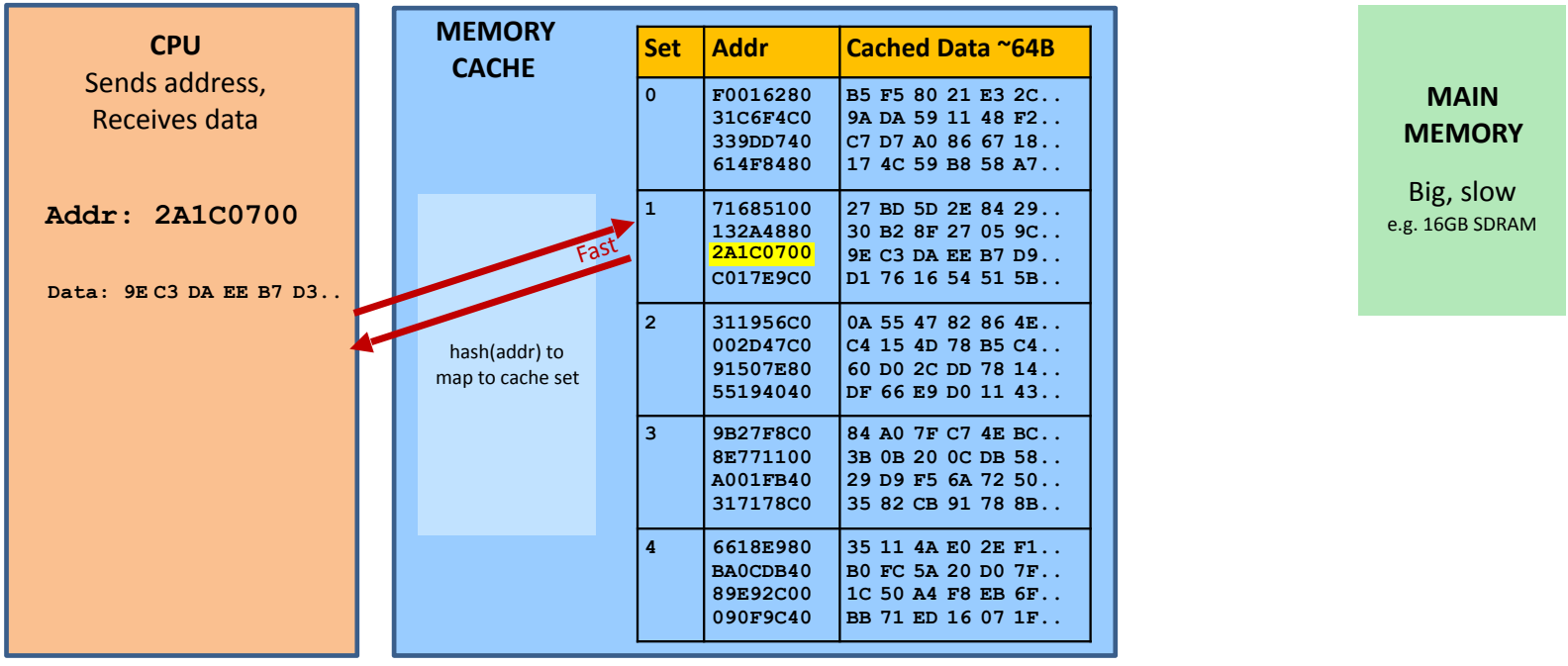
Memory and cache

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



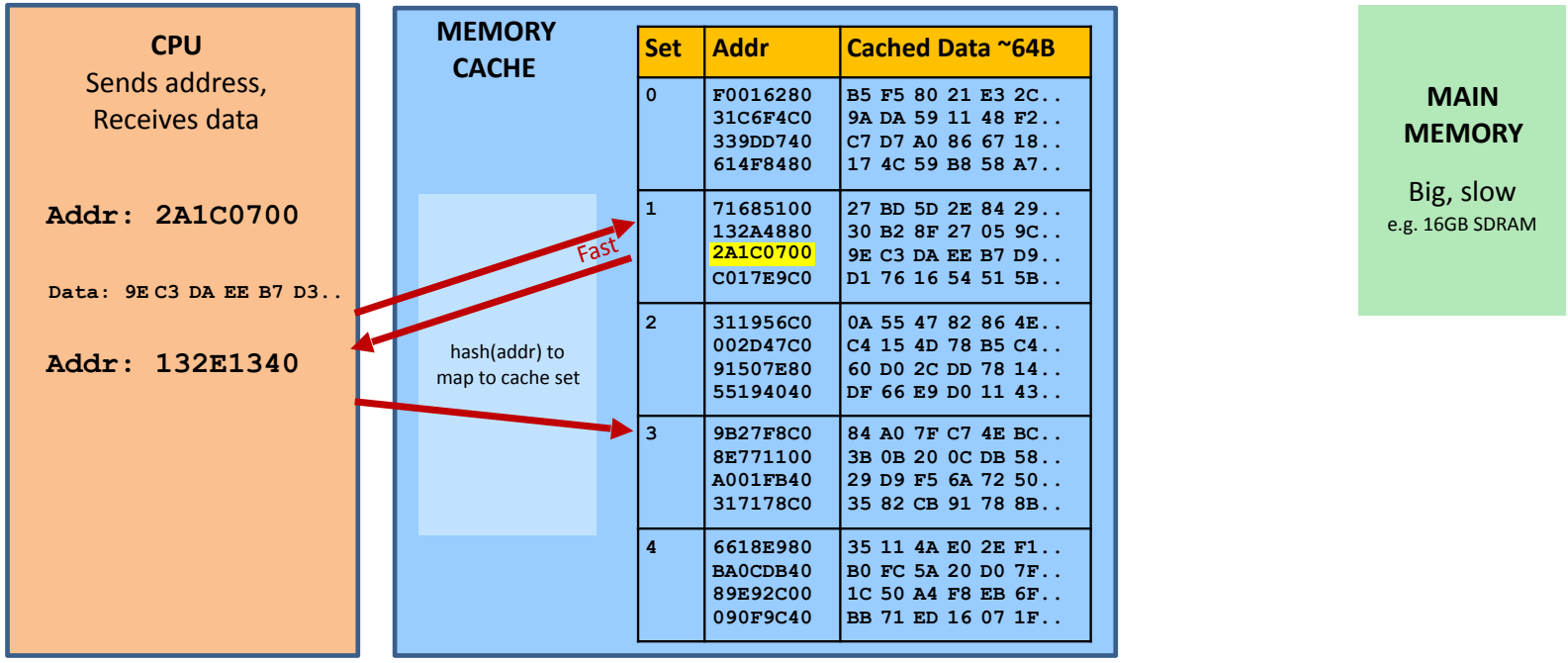
Memory and cache

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



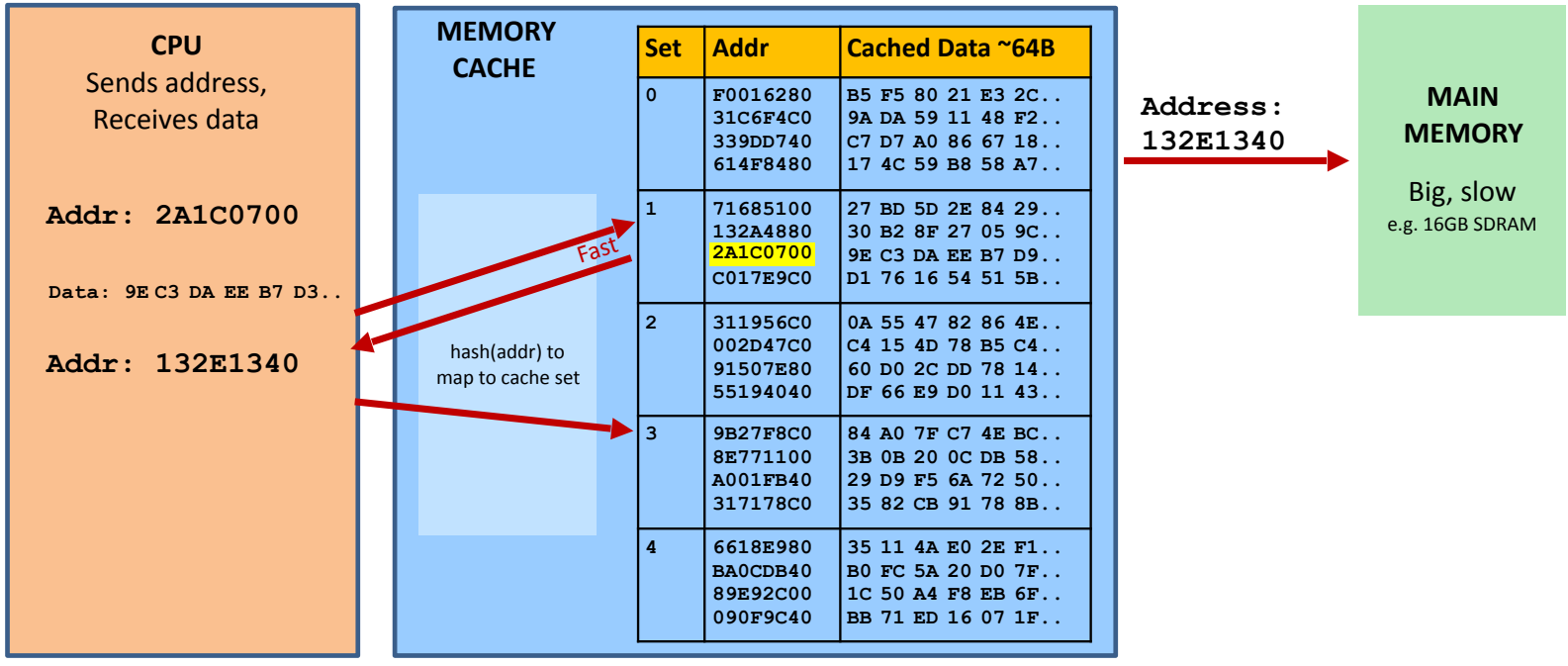
Memory and cache

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



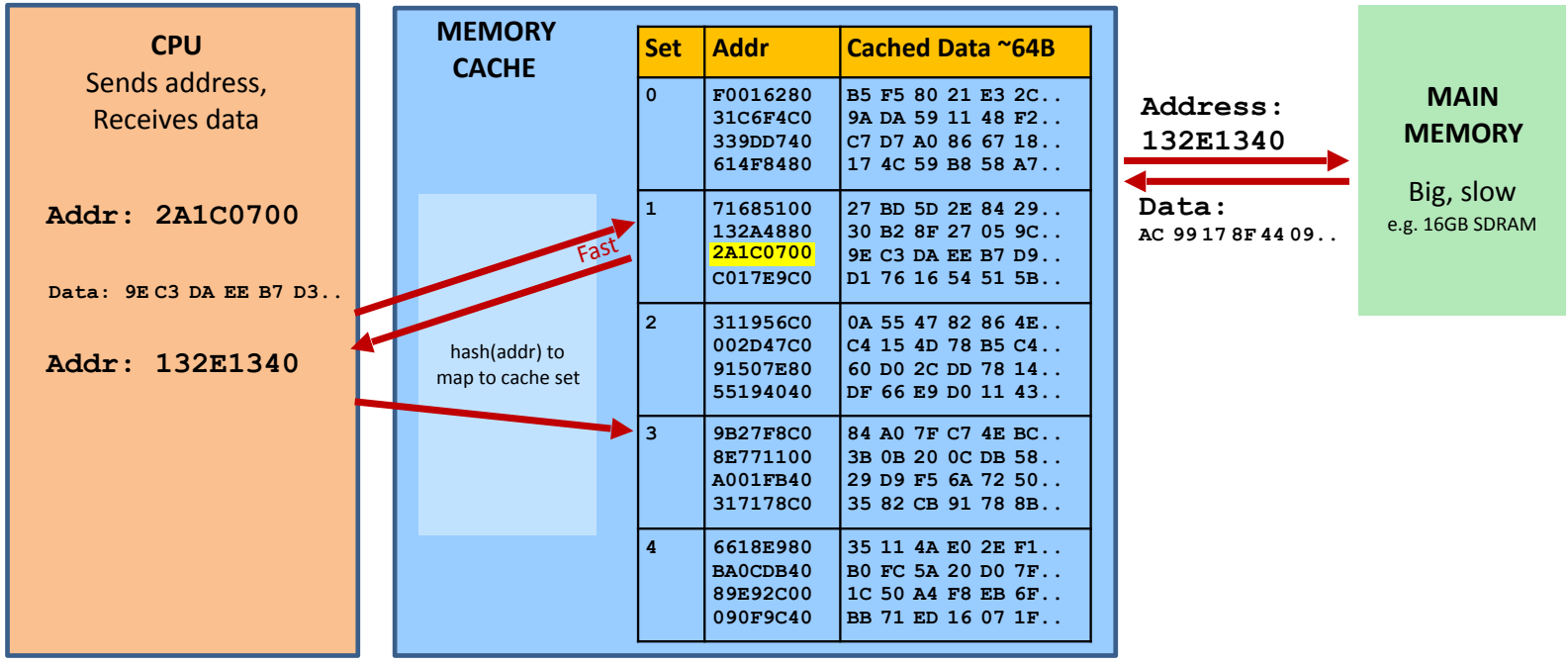
Memory and cache

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



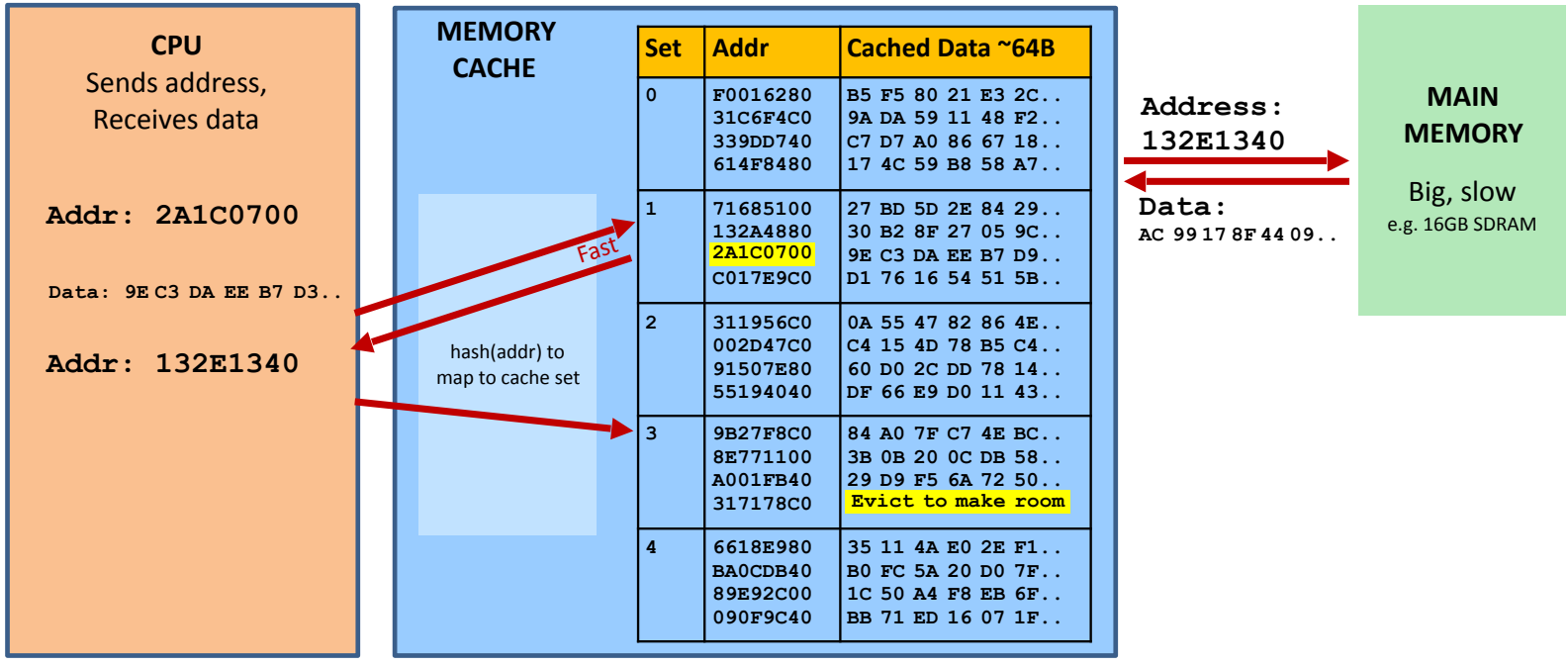
Memory and cache

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



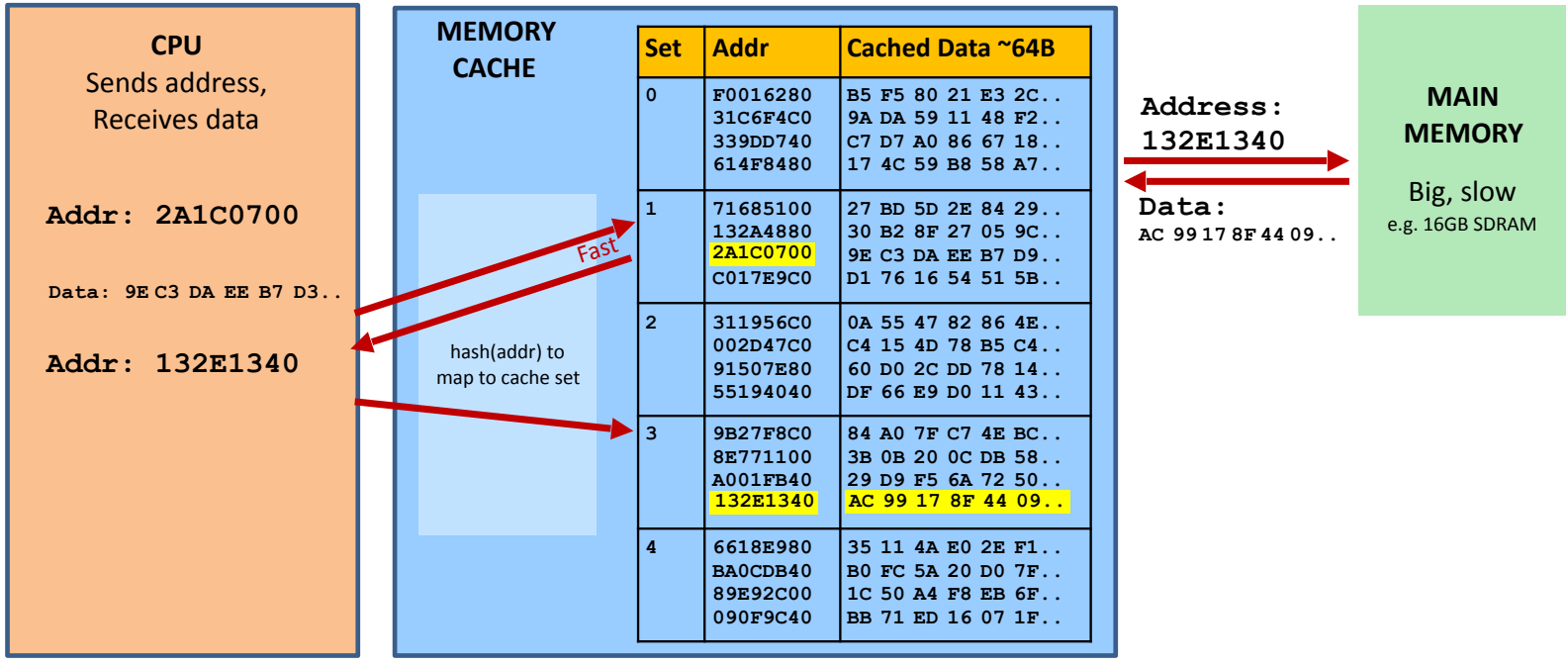
Memory and cache

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



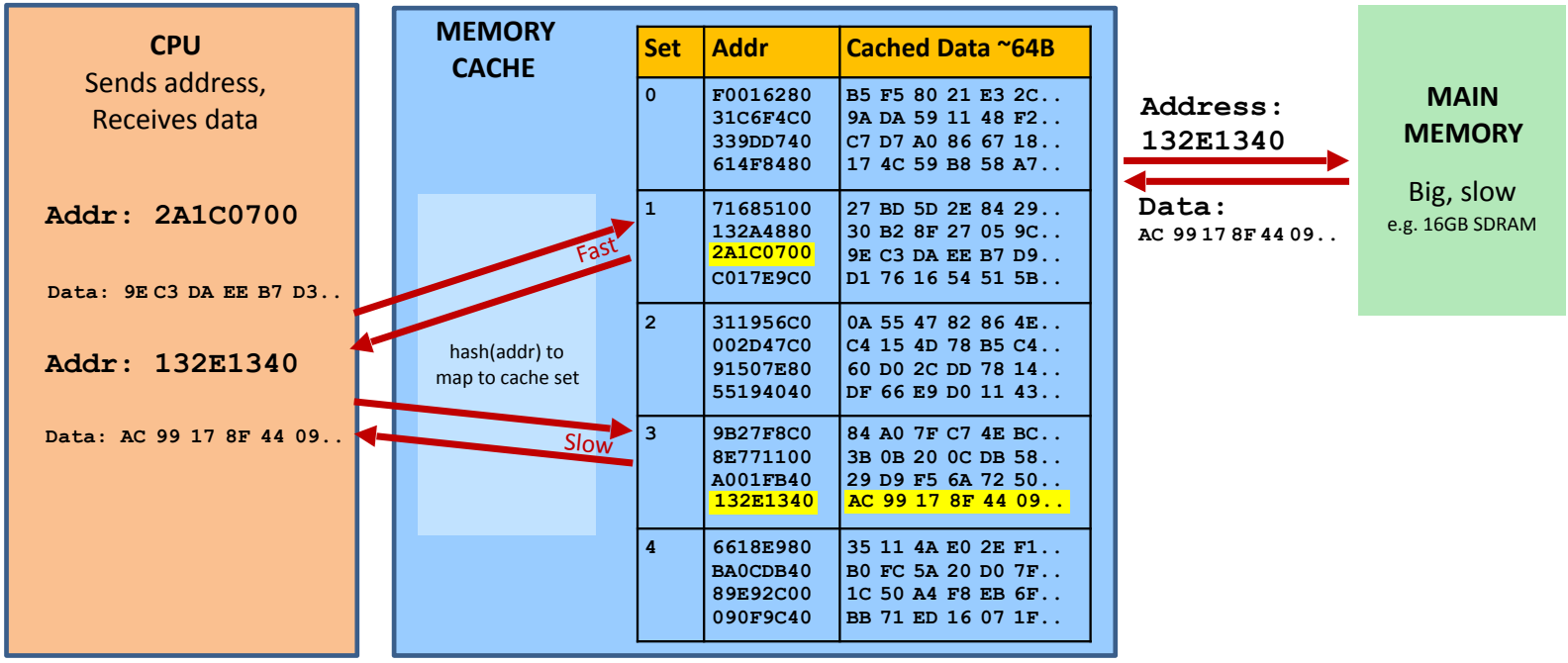
Memory and cache

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



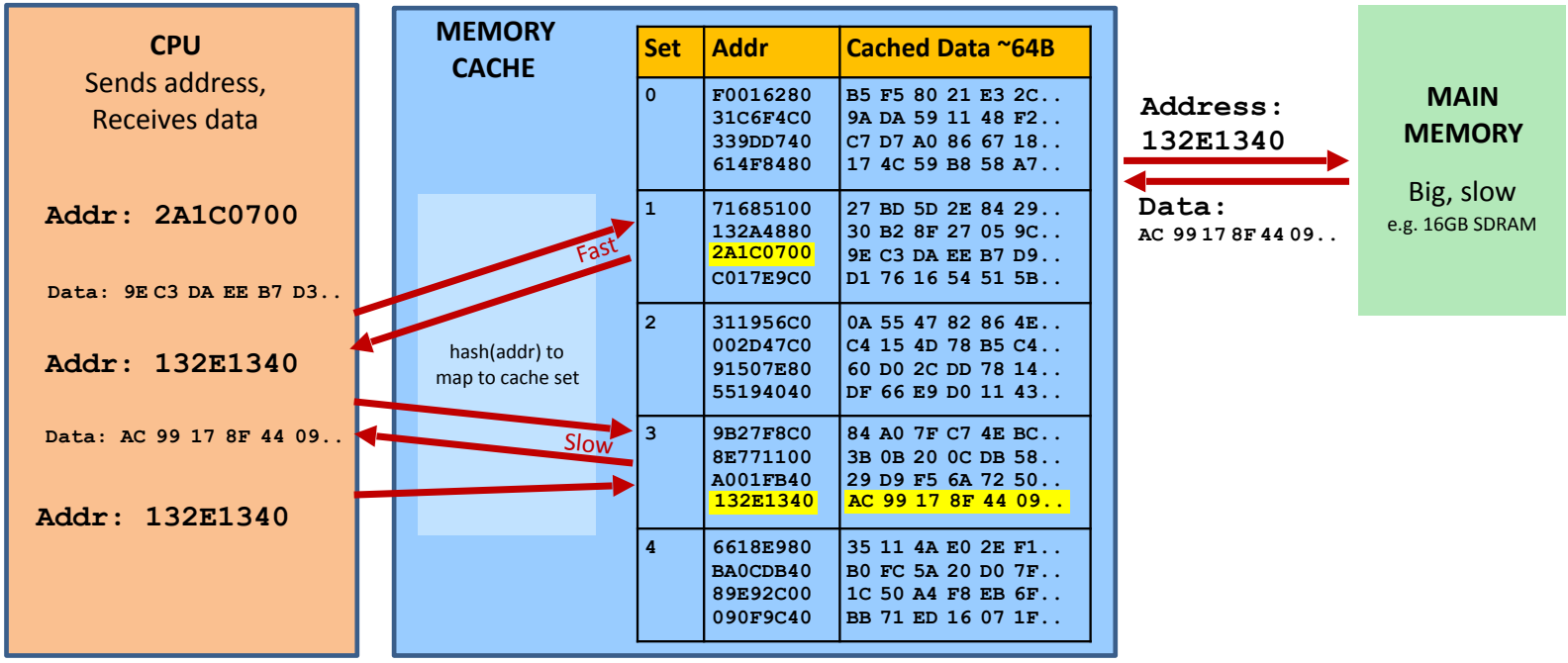
Memory and cache

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



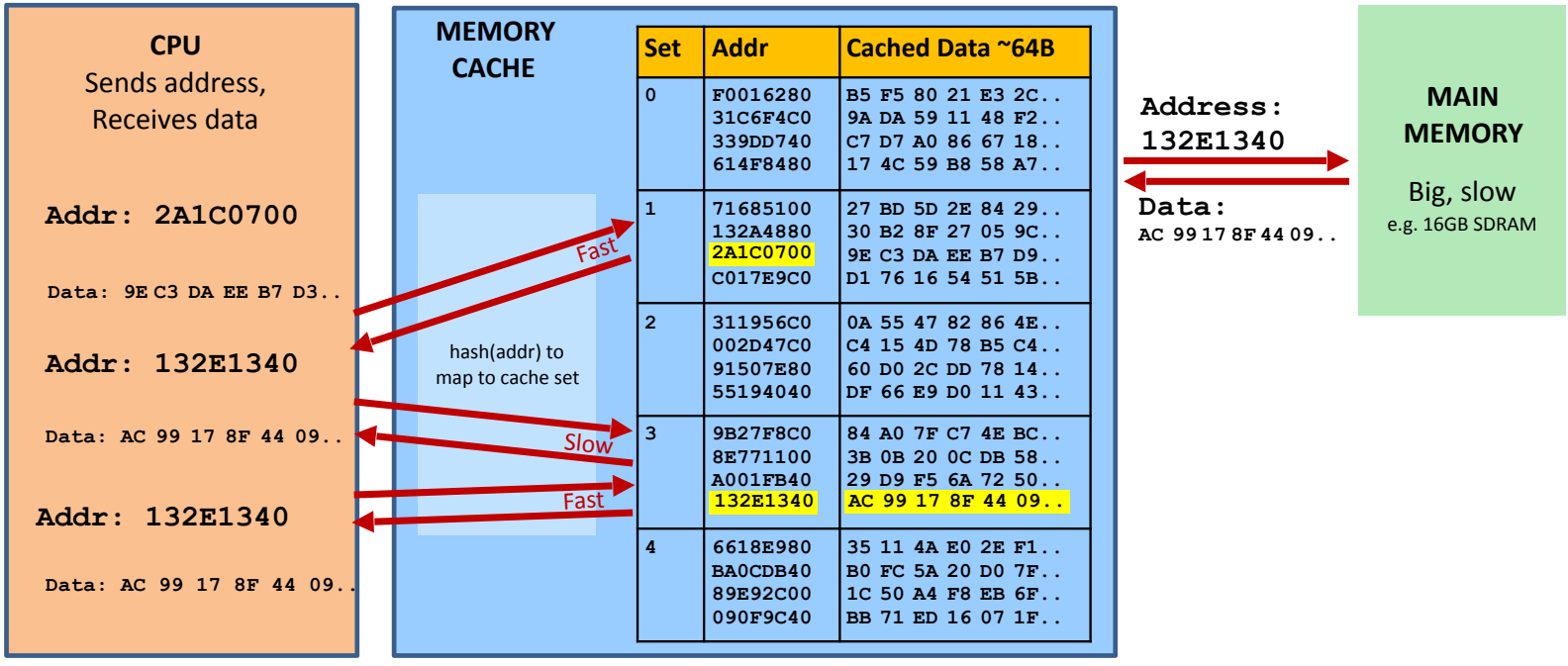
Memory and cache

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



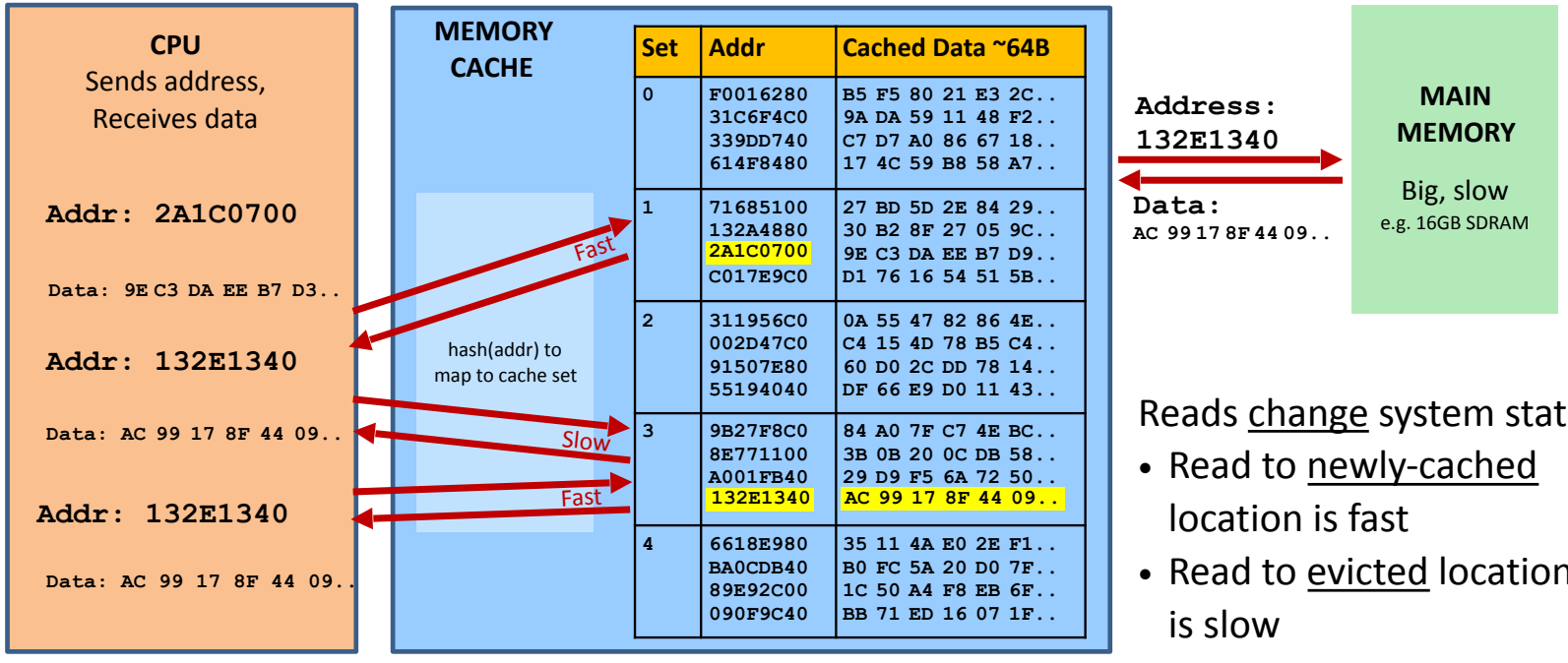
Memory and cache

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

Mitigating side channels

- Use constant-time programming techniques
- Eliminate secret-dependent execution or branches
- Hide/blind inputs

We nee to understand sources of time variability

Which is faster?

```
void foo(double x) {
  double z, y = 1.0;
  for (uint32_t i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

a. `foo(1.0);`
b. `foo(1.0e-323);`
c. They are the same

We nee to understand sources of time variability

Which is faster?

```
void foo(double x) {
  double z, y = 1.0;
  for (uint32_t i = 0; i < 100000000; i++) {
    z = y*x;
  }
}
```

a. `foo(1.0);` ←
b. `foo(1.0e-323);`
c. They are the same

Some instructions take different amounts of time depending on operands.

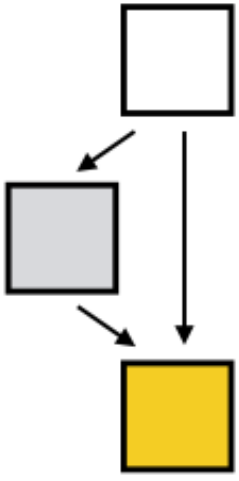
Contant-time rule: No variable-time instructions on secrets.

Control flow introduces time variability

```
m=1
for i = 0 ... len(d):
  if d[i] = 1:
    m = c * m mod N
  m = square(m) mod N
return m
```

Control flow introduces time variability

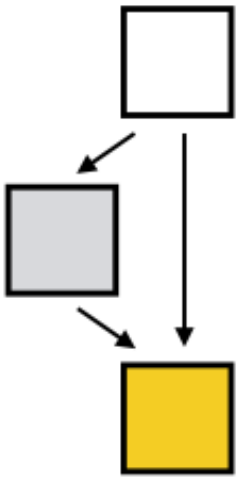
```
s0;
if (secret) {
  s1;
  s2;
}
s3;
```



secret	run	
true	s0;s1;s2;s3;	**4**
false	s0;s3;	**2**

Control flow introduces time variability

```
s0;
if (secret) {
  s1;
  s2;
}
s3;
```



secret	run	
true	s0;s1;s2;s3;	4
false	s0;s3;	2

What can we do about this?

Does padding the else branch work?

```
if (secret) {
  s1;
  s2;
} else {
  s1';
  s2';
}
```

where `s1` and `s1'` take same amount of time

Does padding the else branch work?

```
if (secret) {
  s1;
  s2;
} else {
  s1';
  s2';
}
```

where `s1` and `s1'` take same amount of time

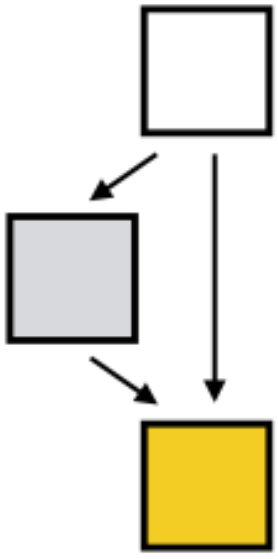
- **Problem:** Instructions are loaded from cache
  $\Longrightarrow$ observable

- **Problem:** Hardware tries to predict where branch goes
  $\Longrightarrow$ success or failure of prediction is observable

Constant-time rule: No branching on secrets

```
if (secret) {
  x = a;
}
```

```
x = secret * a
    + (1-secret) * x
```



Constant-time rule: No branching on secrets

```
if (secret) {
  x = a;              x = secret * a  + (1-secret) * x
} else {
  x = b;              x = (1-secret) * b + secret * x
}
```

Memory access patterns introduce time variability

```
void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
  ...
  for (i = Nk; i < Nb * (Nr + 1); ++i) {
  ...
    k = (i - 1) * 4;
    tempa[0]=RoundKey[k + 0];
    tempa[1]=RoundKey[k + 1];
    tempa[2]=RoundKey[k + 2];
    tempa[3]=RoundKey[k + 3];
  ...
    tempa[0] = sbox[tempa[0]];
    tempa[1] = sbox[tempa[1]];
    tempa[2] = sbox[tempa[2]];
    tempa[3] = sbox[tempa[3]];
  ...
```

Memory access patterns introduce time variability

```
void KeyExpansion(uint8_t* RoundKey, const uint8_t* Key) {
  ...
  for (i = Nk; i < Nb * (Nr + 1); ++i) {
  ...
    k = (i - 1) * 4;
    tempa[0]=RoundKey[k + 0];
    tempa[1]=RoundKey[k + 1];
    tempa[2]=RoundKey[k + 2];
    tempa[3]=RoundKey[k + 3];
  ...
    tempa[0] = sbox[tempa[0]];
    tempa[1] = sbox[tempa[1]];
    tempa[2] = sbox[tempa[2]];
    tempa[3] = sbox[tempa[3]];
  ...
```

Constant-time rule: Don’t access memory at secret index.

Writing constant-time code is hard.

There are tools to help but most code is still written by hand.

It can be slower, larger, and more complex.

(This is one of the reasons people recommend not doing your own cryptographic implementations.)

Today

- Overview and history of side channels
- Cache side channels and countermeasures
- Spectre attacks

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 $\rightarrow$ performance gain
  - Incorrect: Discard speculative work $\rightarrow$ no harm?

Spectre and Meltdown

Lipp et al., Kocher et al. 2017

Misspeculation

- Exceptions and incorrect branch prediction can cause “rollback” of transient instructions
- Old register states are preserved, can be restored
- Memory writes are buffered, can be discarded
- Cache modifications are not restored

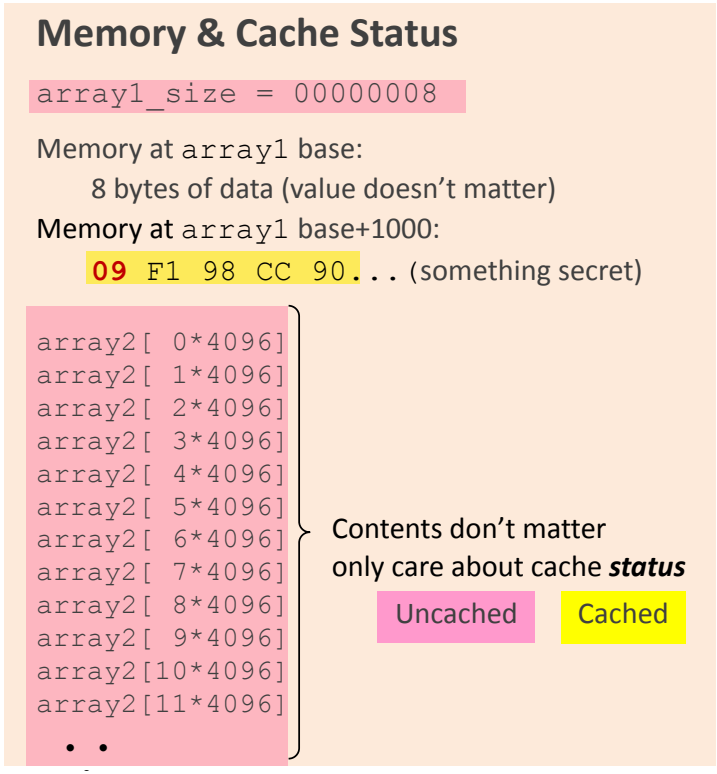
- Spectre and Meltdown carry out cache attacks against speculatively loaded data so that an unprivileged attacker process can read kernel memory, break ASLR, etc.

Spectre 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



Spectre variant 1 attack

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

Attacker calls victim with `x`=1000

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]
  . .
```

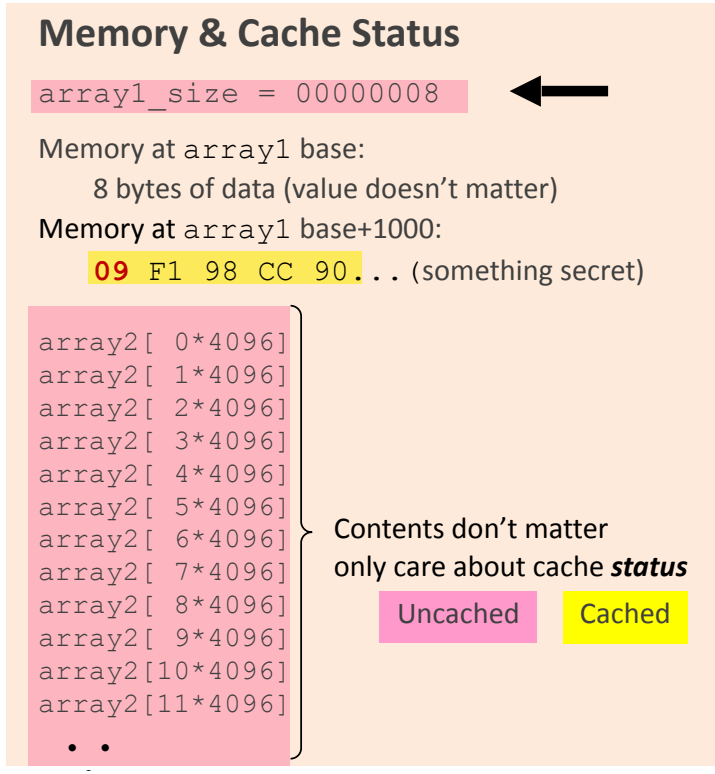
Contents don't matter
only care about cache ***status***

Uncached Cached

Spectre variant 1 attack

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

Attacker calls victim with `x`=1000

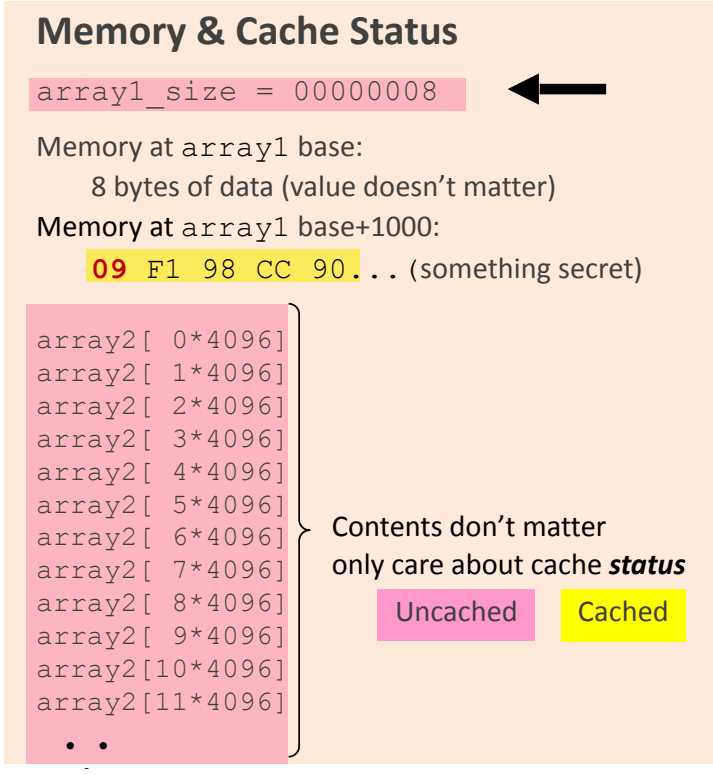


Spectre variant 1 attack

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

Attacker calls victim with `x`=1000

Speculative exec while waiting for `array1_size`:



Spectre 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

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

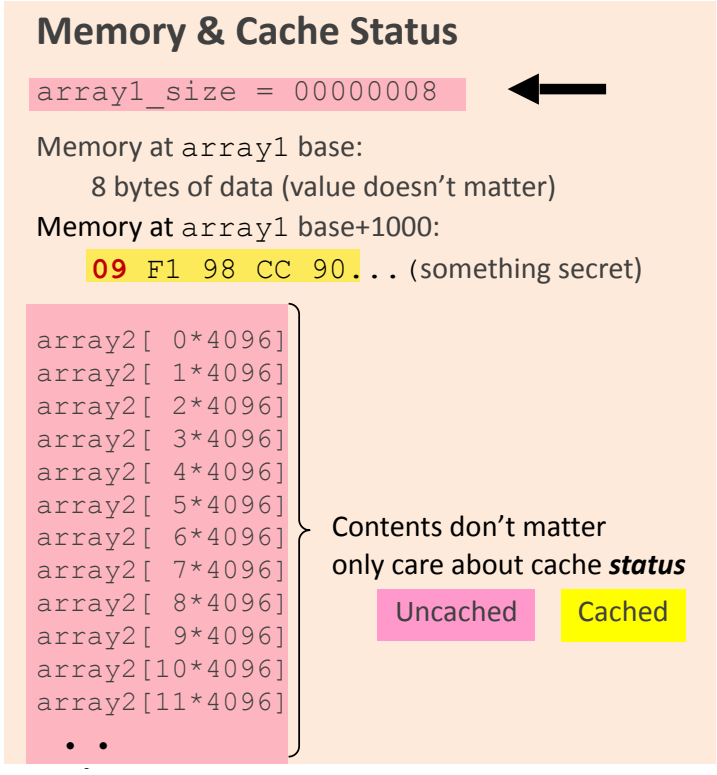
Spectre variant 1 attack

```
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)



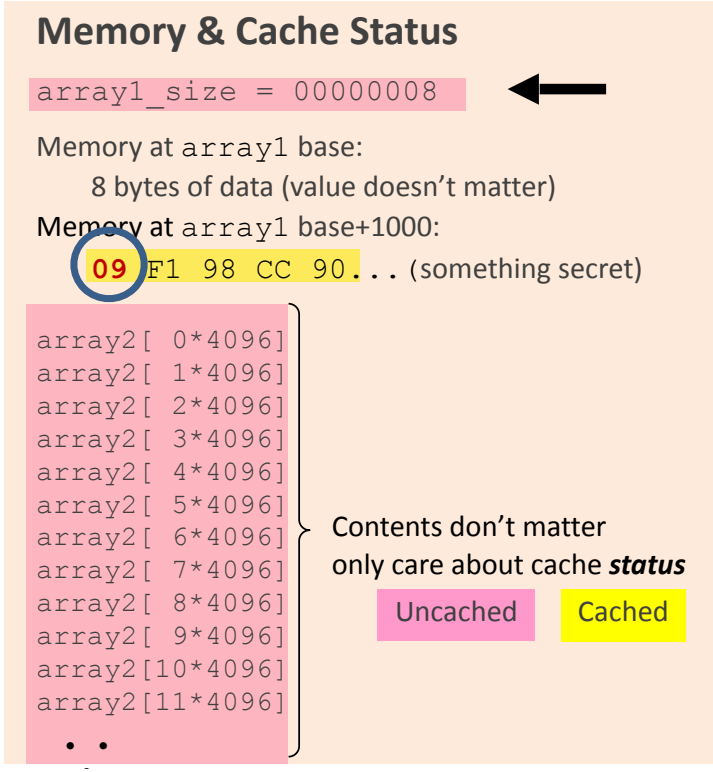
Spectre variant 1 attack

```
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)



Spectre variant 1 attack

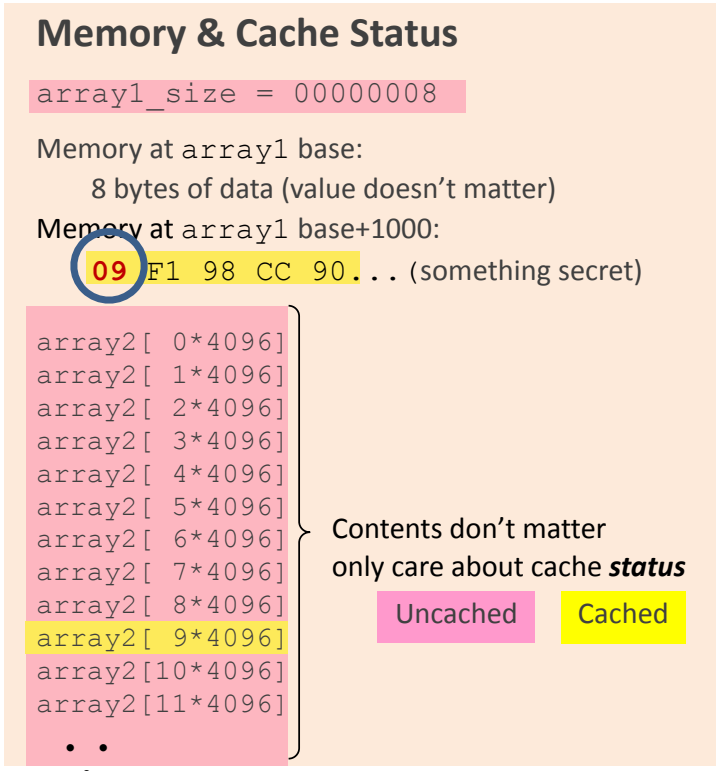
```
if (x < array1_size)
   y = array2[array1[x]*4096];
```

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



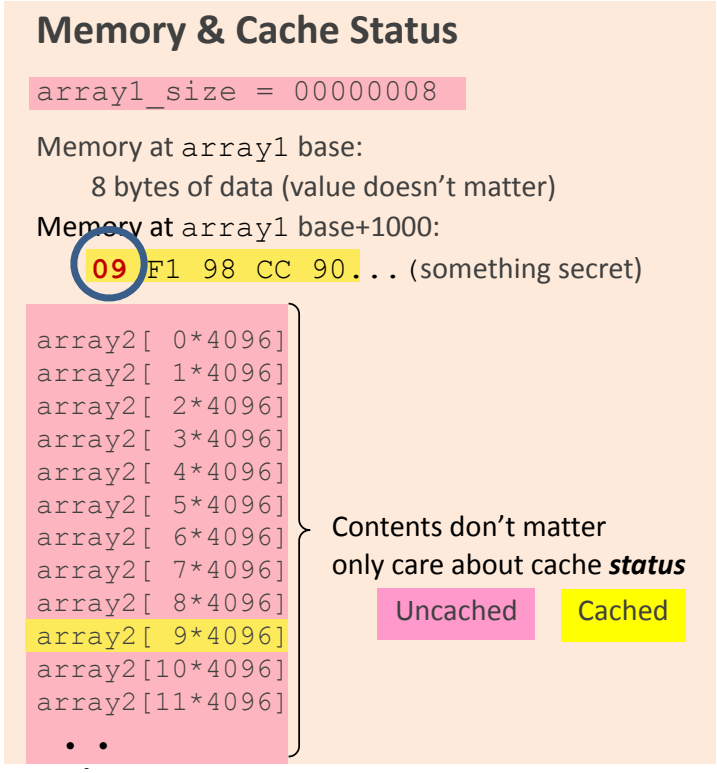
Spectre variant 1 attack

```
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)



Open question: Mitigations for Spectre and other microarchitectural attacks

Turning off speculative execution and other microarchitectural optimizations has real performance impacts.

Open question: How to prioritize performance vs. security

Today

- Overview and history of side channels
- Cache side channels and countermeasures
- Spectre attacks