Lecture 22:
Side channel attacks 🔍

CPSC 4440/5440, 2026 Spring

Instructor: Fan Zhang

Yale

Computer Science

Questions

- What are three security properties of a TEE?
  - Integrity, confidently, remote attestation
- SGX is the first _____ ?
- What's the trust model of SGX?

Today:

Side-channel Attacks!

The concern: Access patterns

- **Access pattern** = the sequence of memory locations (addresses) a program reads/writes over time, including their order and frequency.
- Access pattern may leak information if it's **secret-dependent**.
- Example: Naive implementation of AES

Access patterns can directly leak keys

```
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]];
...
```

<- Which part of RoundKey is accessed directly leaks k (key)

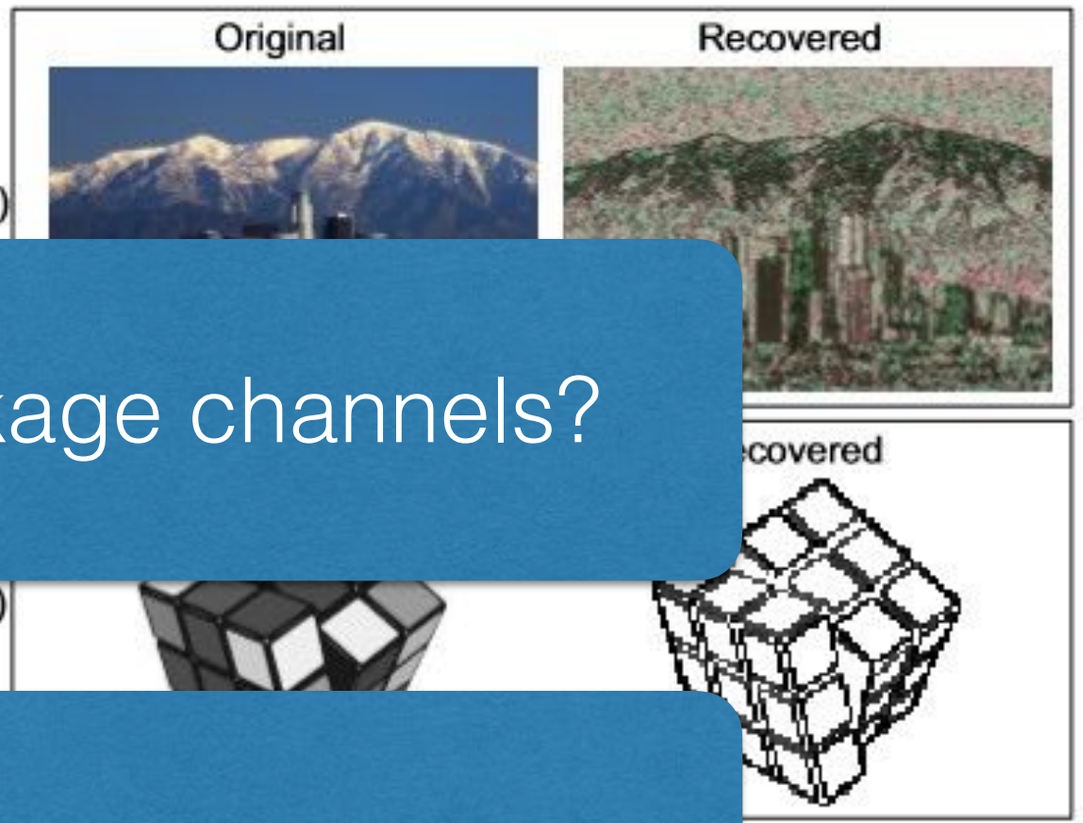
Key questions: How do attackers learn access patterns? How accurate? How fast/slow?

Threat model of SGX

- **Trust assumptions**
  - CPU pack is secure (e.g., attacker can't read secrets from registers)
  - SGX logic (part of CPU) is correctly implemented.
  - The only software TCB is "the app": System software is malicious (OS, hypervisor, firmware, etc)
- **Potential leakage channels [considered out of scoped by Intel SGX's trust assumption]**
  - SGX claims no security against physical attacks
  - **Page-level access pattern leakage** (b/c OS manages page table)
  - **Finer-grain access pattern leakage** through sides channel in "the app" (today's lecture)

Recall: Page-level access pattern leakage

- Page = 4KB memory regions
- Attacker (e.g., OS) learns **which pages are accessed**
- How to exploit the such leakage?



2015 IEEE Symposium on Security and Privacy

Controlled-Cl
Channels f

Yuanzhong Xu
The University of Texas a
yxu@cs.utexas.edu

***Abstract*—The presence of large numb**
abilities in popular feature-rich commodity operating systems has inspired a long line of work on excluding these operating systems from the trusted computing base of applications, while retaining many of their benefits. Legacy applications continue to run on the untrusted operating system, while a small hypervisor or trusted hardware prevents the operating
the applications' memory.
In this paper, we introduce controlled

code, the shielding systems has to completely insulate the application from all types of adversarial action by the untrusted operating system. This is hard, given the operating system's role in resource management and providing services to the

Are there other leakage channels?

Yes: Cache-based timing side channels.

All kinds of side-channel attacks (before SGX)

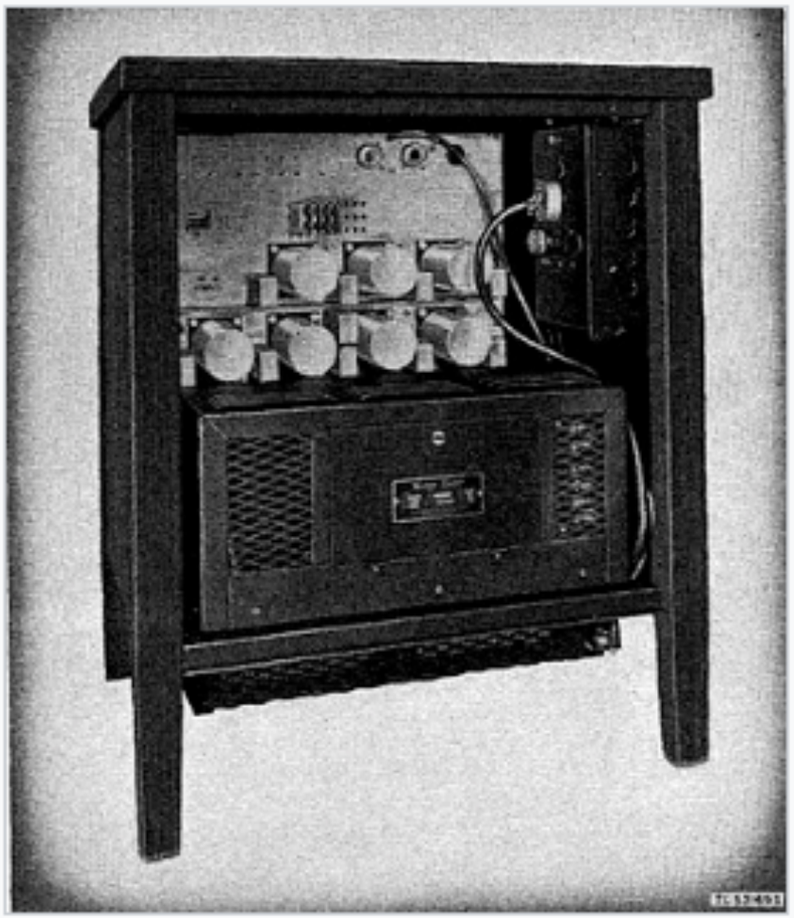
Side channels

- (The concept is not specific to SGX)
- A *side channel* is a source of information <u>beyond the output specified by an abstraction</u>.
- Problematic if they are observable by attacker and *secret-dependent* 🔑

Two classes of side channels

- **Emission**: What out-of-band signal is generated in the course of performing the operation?
  - Electromagnetic radiation
  - Sound (acoustic leakage attacks)
  - Error messages (e.g., in CCA attacks)
- **Consumption**: How much of certain resource is being used to perform an operation?
  - Timing ⏰
  - Power consumption 🔌
  - Network traffic 🛜

The first documented vulnerability



Bell 131B2 mixer, used to XOR teleprinter signals with one time tapes, was the first device from which classified plain text was extracted using radiated signals.

- During WWII, US military uses teleprinter to transmit OTP messages (Left).
- **Unbreakable, perfect secrecy, right?**
- It turns out: electrical signals carries information about the plaintext.
- Manufacturer found that plaintext can be recovered from adjacent building (100-ft radius).

U.S. government's emission security (EMSEC) standards

- Formally established in 1950s–1960s, still actively enforced (details classified)

van Eck Phreaking

- 1985: Wim van Eck demonstrates image recovery from CRT monitors with off-the-shelf equipment (“spied on through walls”)

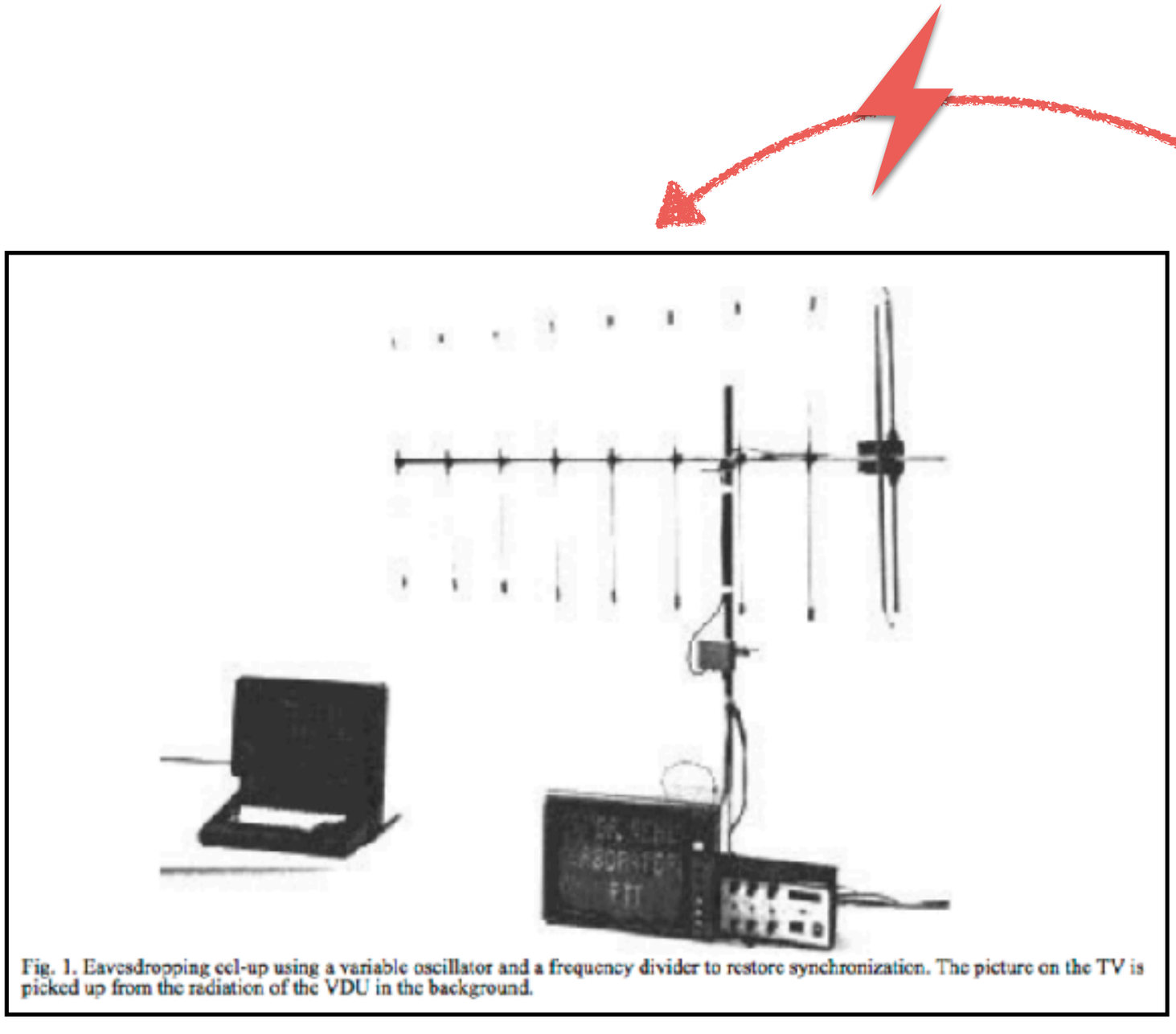
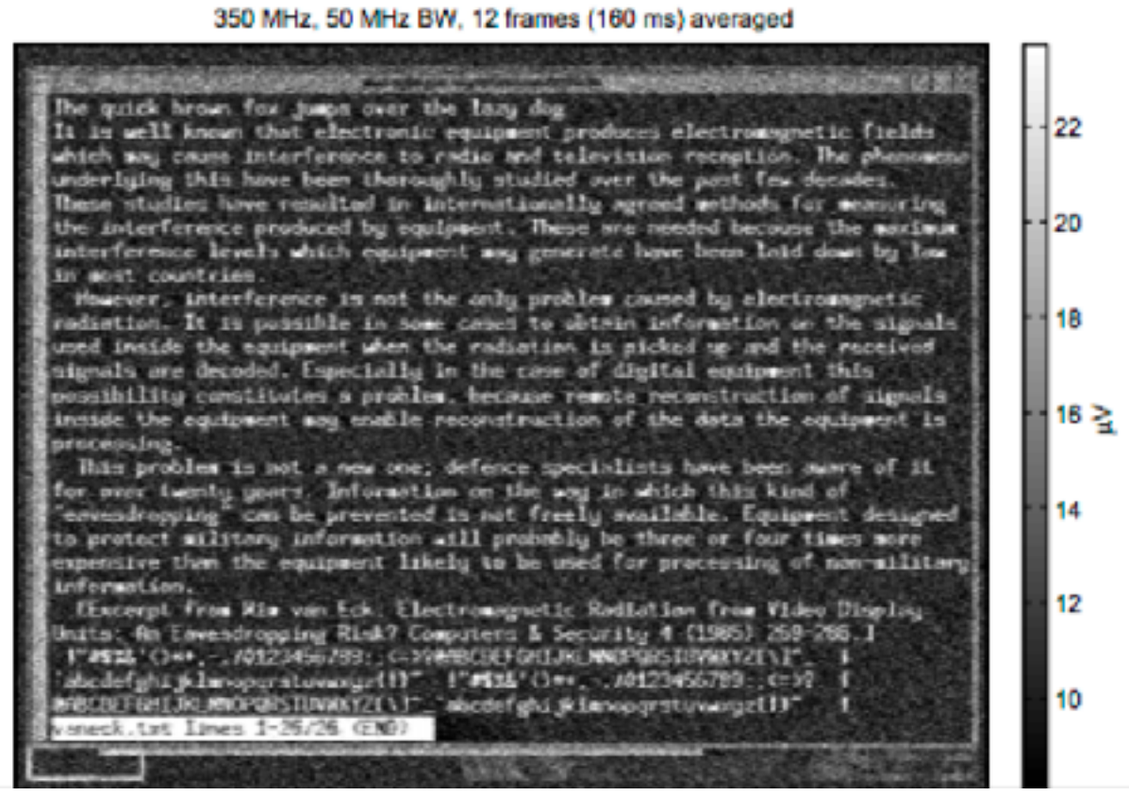


Fig. 1. Eavesdropping eel-up using a variable oscillator and a frequency divider to restore synchronization. The picture on the TV is picked up from the radiation of the VDU in the background.

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

- Pick up radiation from screen connection cable



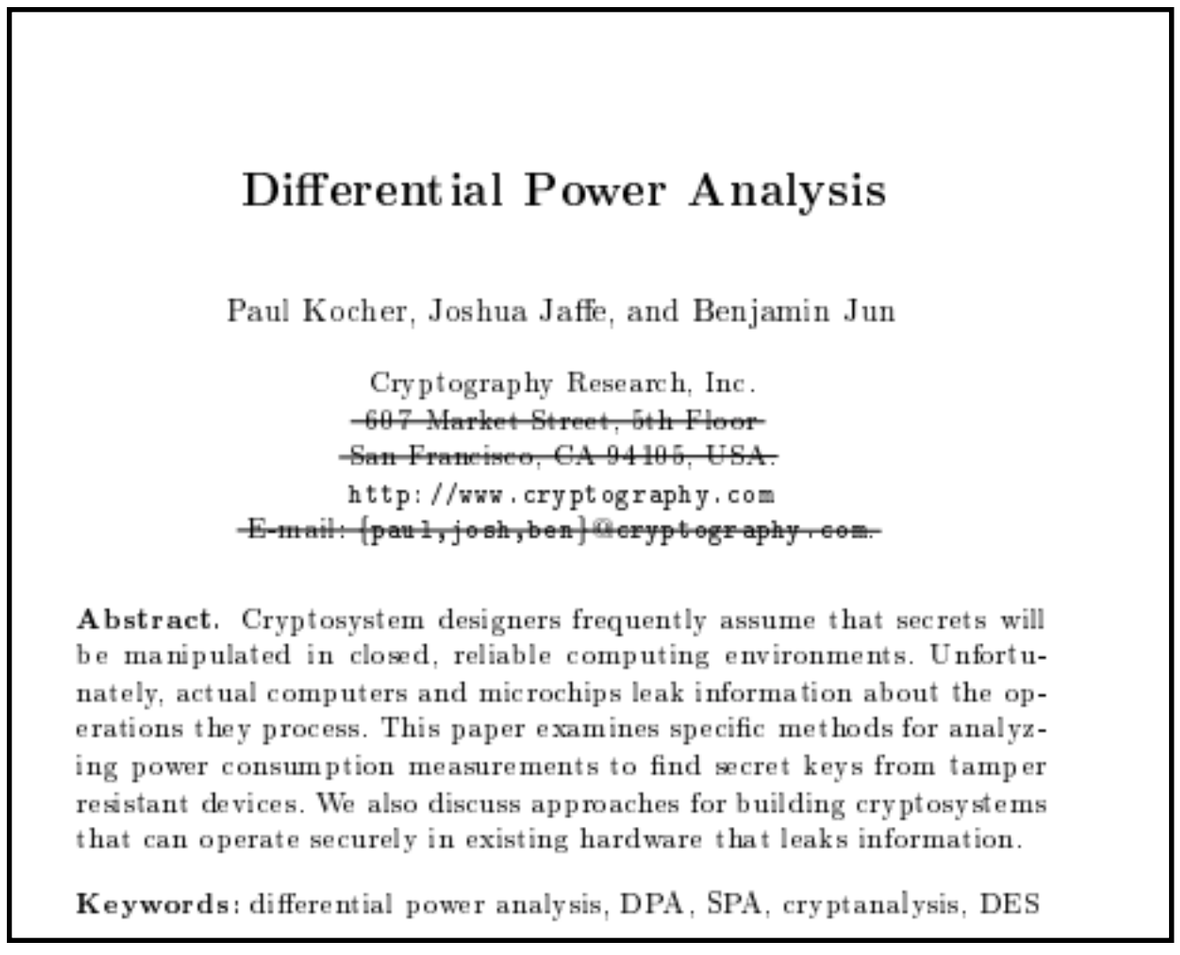
First software side channels

Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems

Paul C. Kocher

- CRYPTO ’96
- Show that keys can be recovered from timing (how long cryptographic operations take)
- **Coined term “side-channel attack”** and opened up the field of implementation-level crypto attacks.

Power-side channels



Differential Power Analysis

Paul Kocher, Joshua Jaffe, and Benjamin Jun

Cryptography Research, Inc.
~~607 Market Street, 5th Floor~~
~~San Francisco, CA 94105, USA.~~
http://www.cryptography.com
~~E-mail: {paul,josh,ben}@cryptography.com.~~

Abstract. Cryptosystem designers frequently assume that secrets will be manipulated in closed, reliable computing environments. Unfortunately, actual computers and microchips leak information about the operations they process. This paper examines specific methods for analyzing power consumption measurements to find secret keys from tamper resistant devices. We also discuss approaches for building cryptosystems that can operate securely in existing hardware that leaks information.

Keywords: differential power analysis, DPA, SPA, cryptanalysis, DES

- First paper in ’99 [Kocher Jaffe Jun]
- Simple power analysis (SPA) and differential power analysis (DPA) exploit secret-dependent power consumption
- Requires physical probing

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
  - ML can make such attacks more powerful

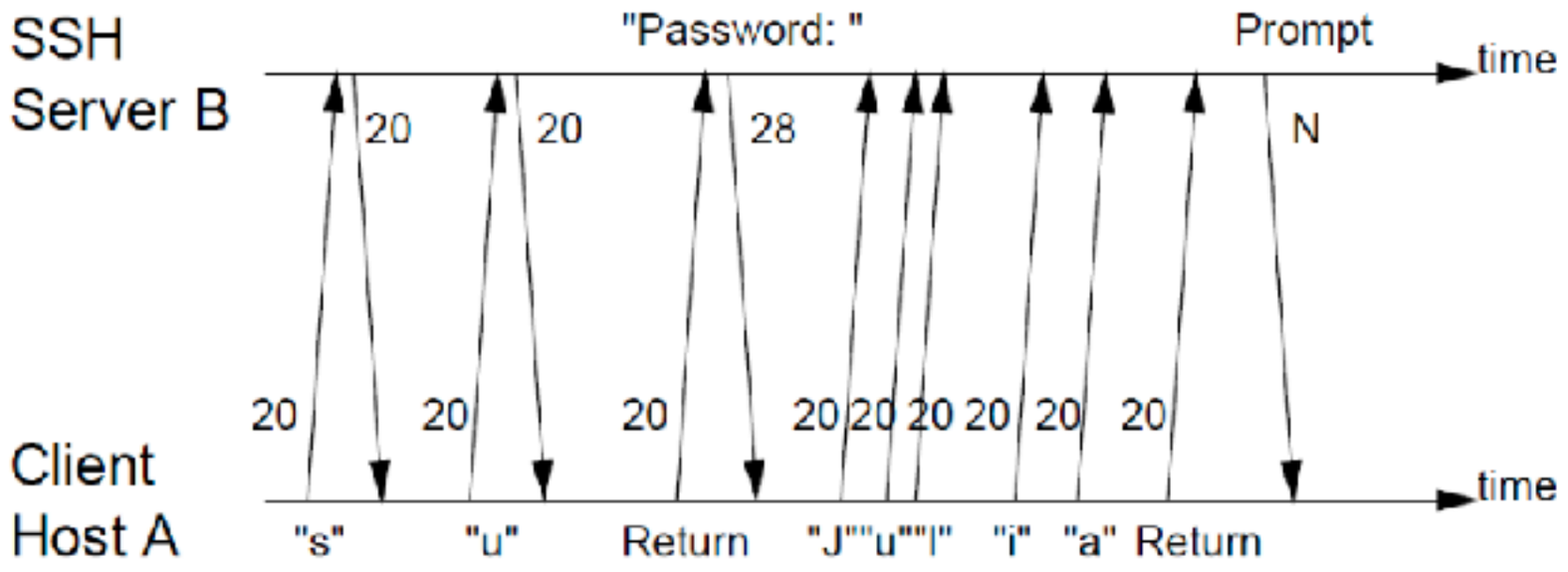


Figure 1: The traffic signature associated with running SU in a SSH session. The numbers in the figure are the size (in bytes) of the corresponding packet payloads.

Some bizarre ones

- Sound
  - Keyboard acoustics ⇝ passwords [USENIX ’05, S&P ’04]
  - “Spearphone: Speech Privacy Exploitation via Accelerometer-Sensed Reverberations from Smartphone Loudspeakers” [WiSec ‘20]
  - “Hey Alexa, What Did I Type? Acoustic Side Channel Attacks on Virtual Keyboards” [NDSS ’21]
  - “Synesthesia: Detecting Screen Content via Acoustic Side Channels” [USENIX ’22]
  - …
- Color …

Color??

- Default web browser behavior: unvisited links are blue and visited links are purple.
- Color available to javascript!
- Attack: Malicious website enumerates URLs in invisible portion of site to sniff browser history.
- Exploited in the wild. “Fixed” in 2010, new browser features re-exposed the side channel.

somemore

- Fault induced side channels
  - Injecting faults to trigger side channels (e.g., glitch power, voltage, clock, temperature, etc)
- Takeaway: There are **many** possible side channels, and they are hard to eliminate.

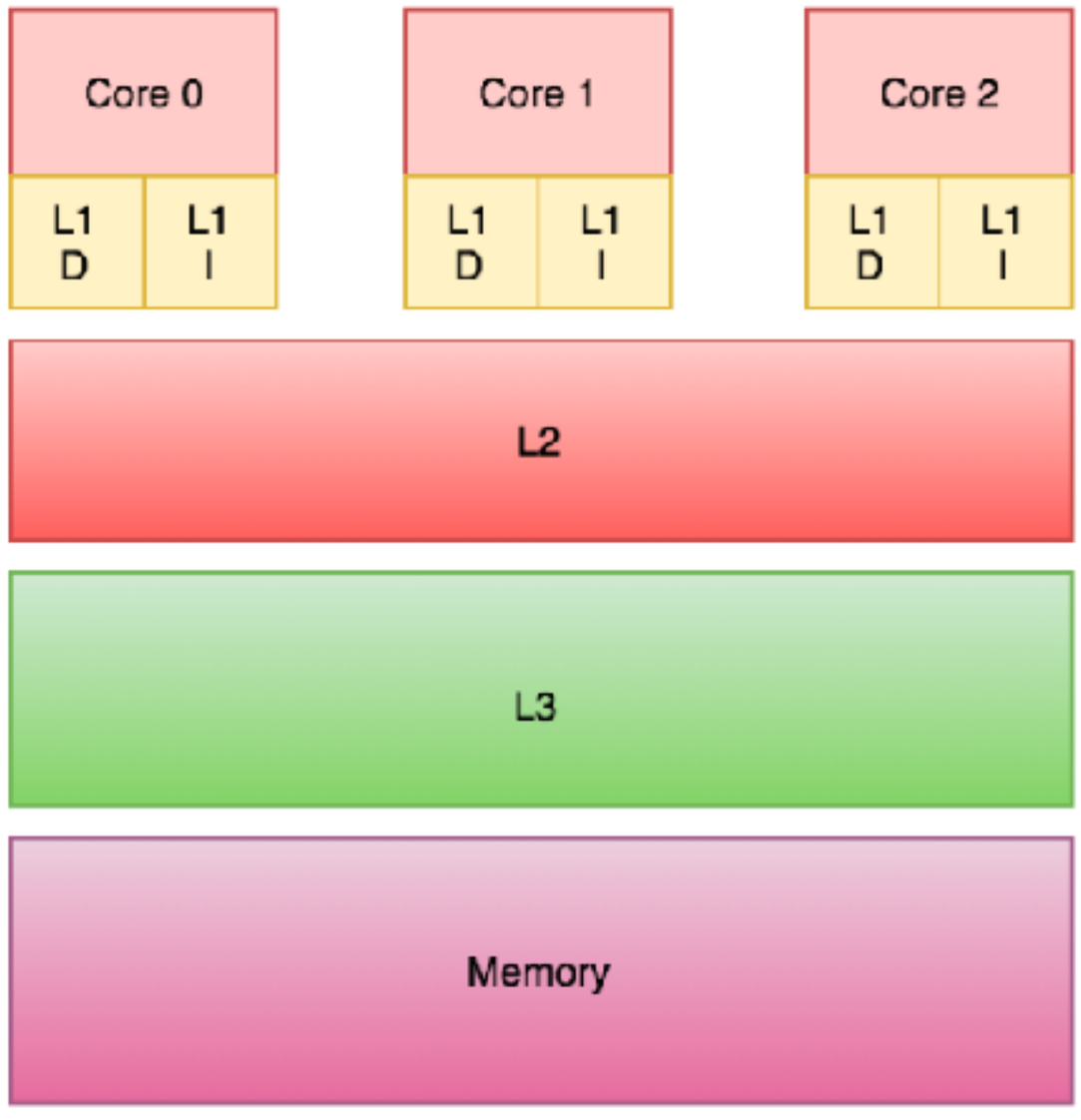
Focus: **Cache-based side channels**

Cache-based side channels

- Good for attackers:
  - Remote exploitability: can be done over SSH. No physical access required.
  - Software exploitability: no need for special hardware.
- Realistic concern, also works for SGX
  - Intel leaves it out of scope (up to developer's to implement mitigation)

Memory and cache

- Cache is **problematic for security** b/c it is shared, not manageable at software level by design.
- Caches organized in hierarchy: closer to the core are faster and smaller (more expensive)



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

CPU
Sends address,
Receives data

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

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

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

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

CPU
Sends address,
Receives data

Addr: 2A1C0700

MEMORY CACHE

Fast

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

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..

MEMORY CACHE

Fast

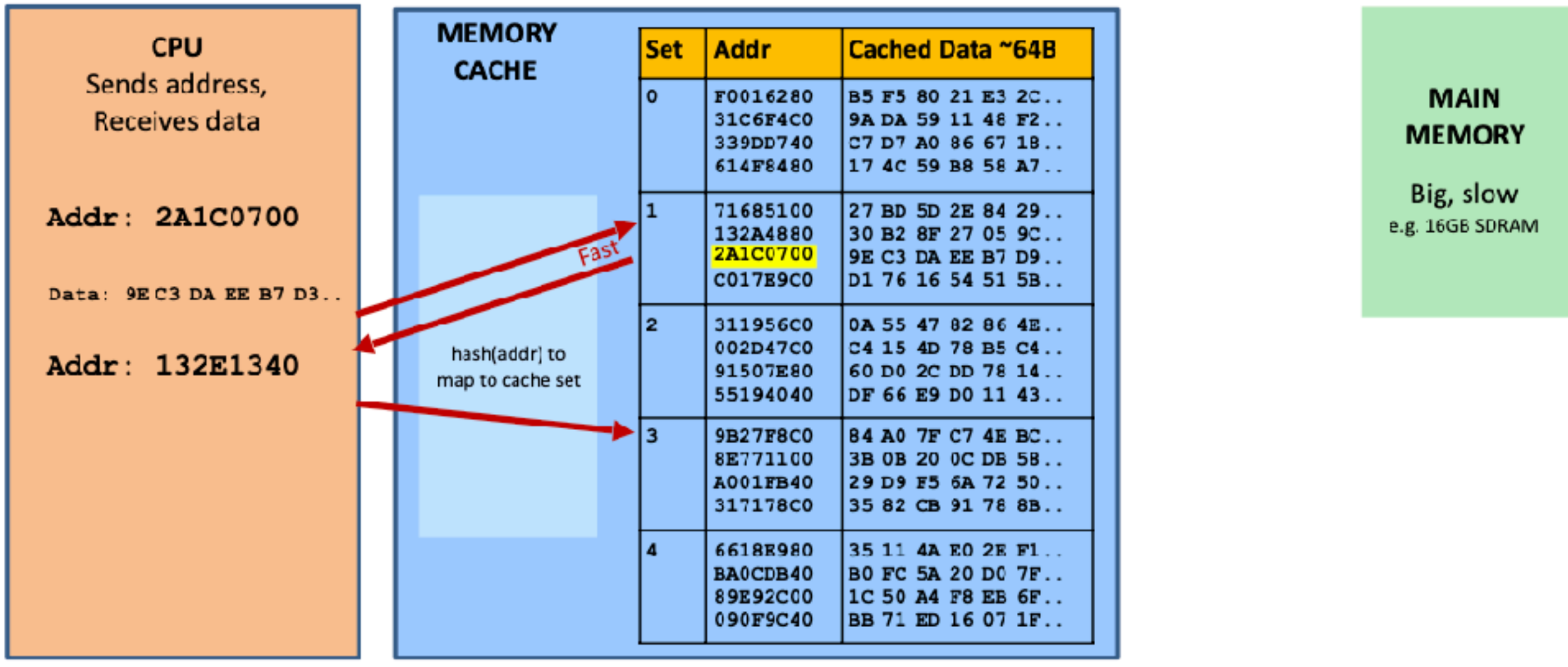
hash(addr) to map to cache set

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..

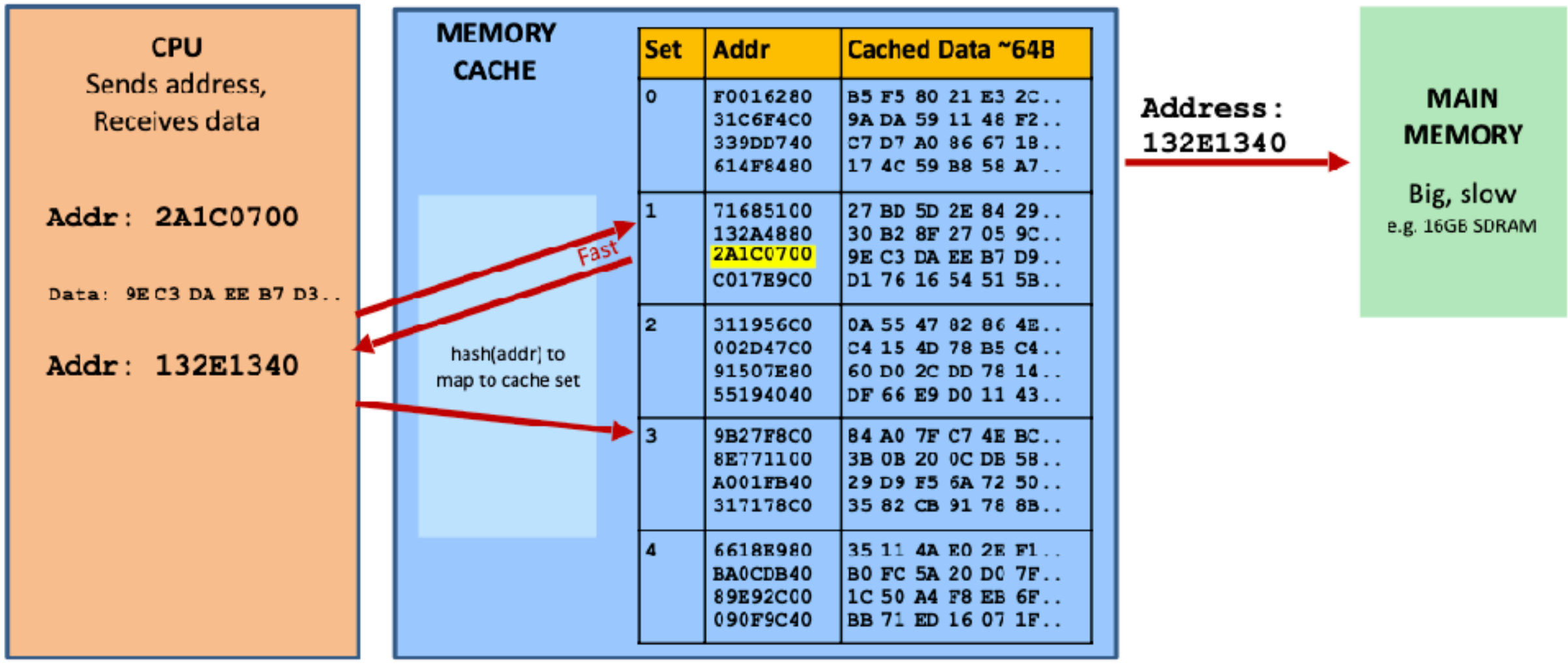
MAIN MEMORY

Big, slow
e.g. 16GB SDRAM

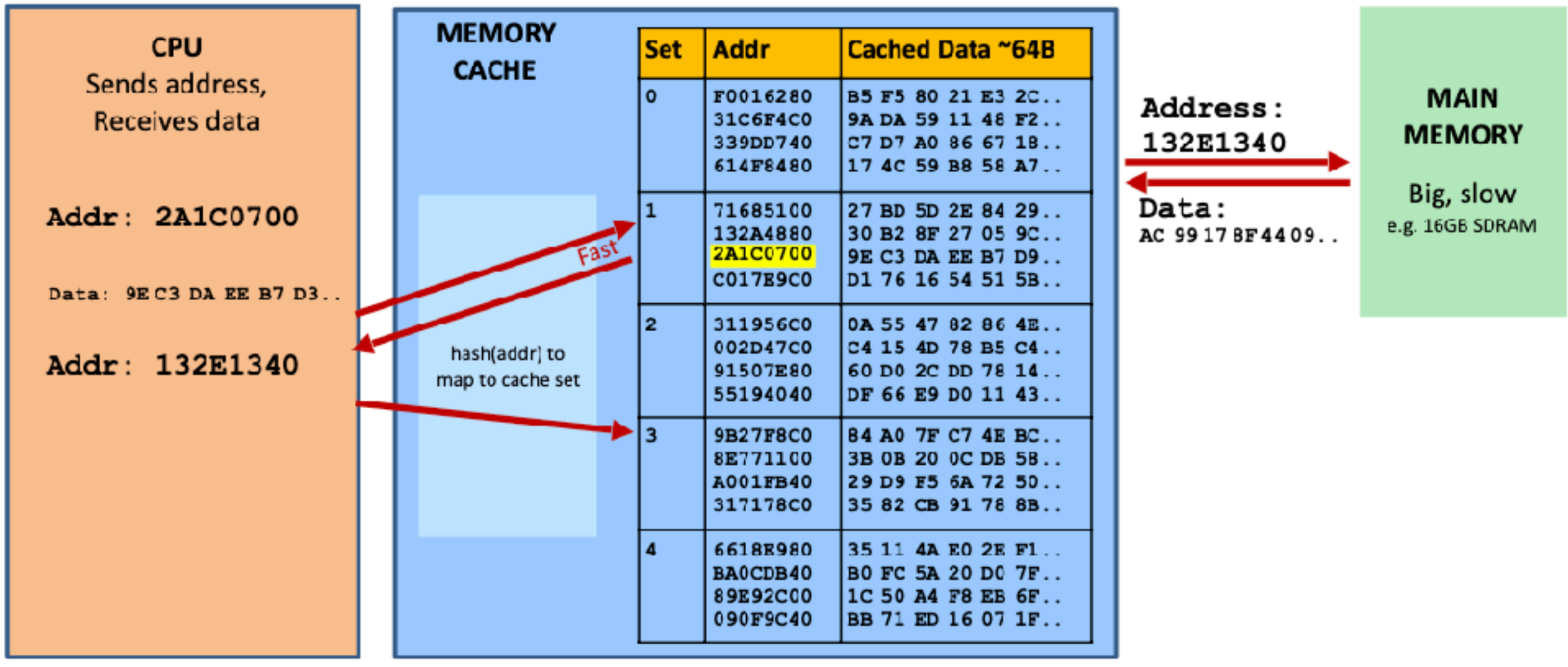
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



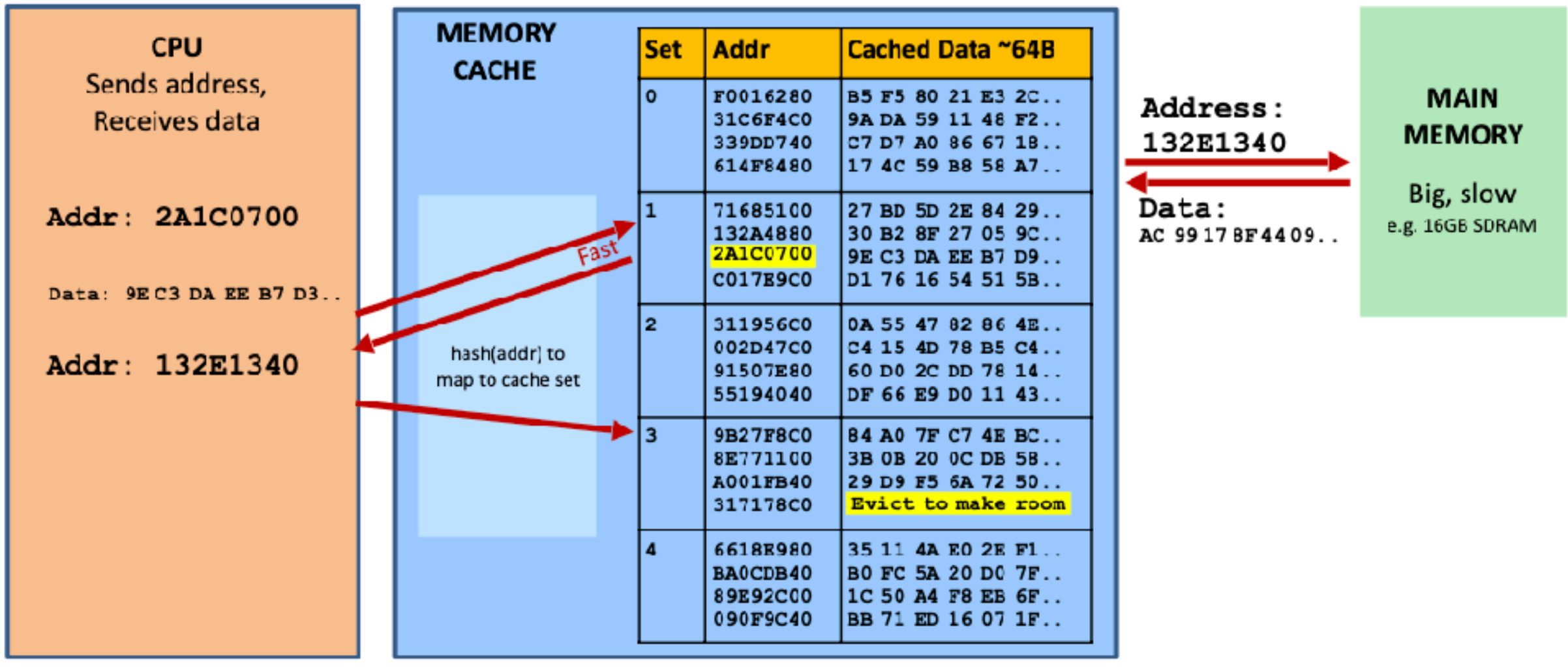
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



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

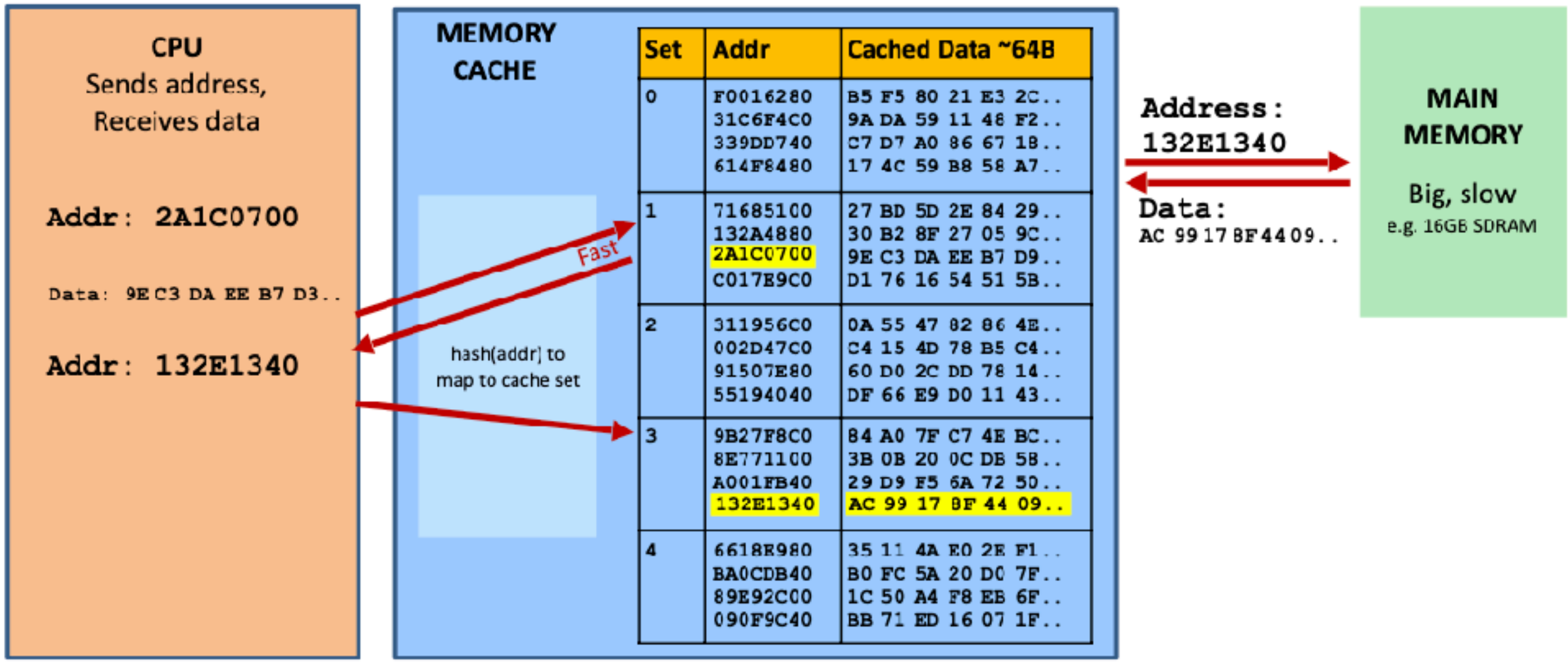


Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory

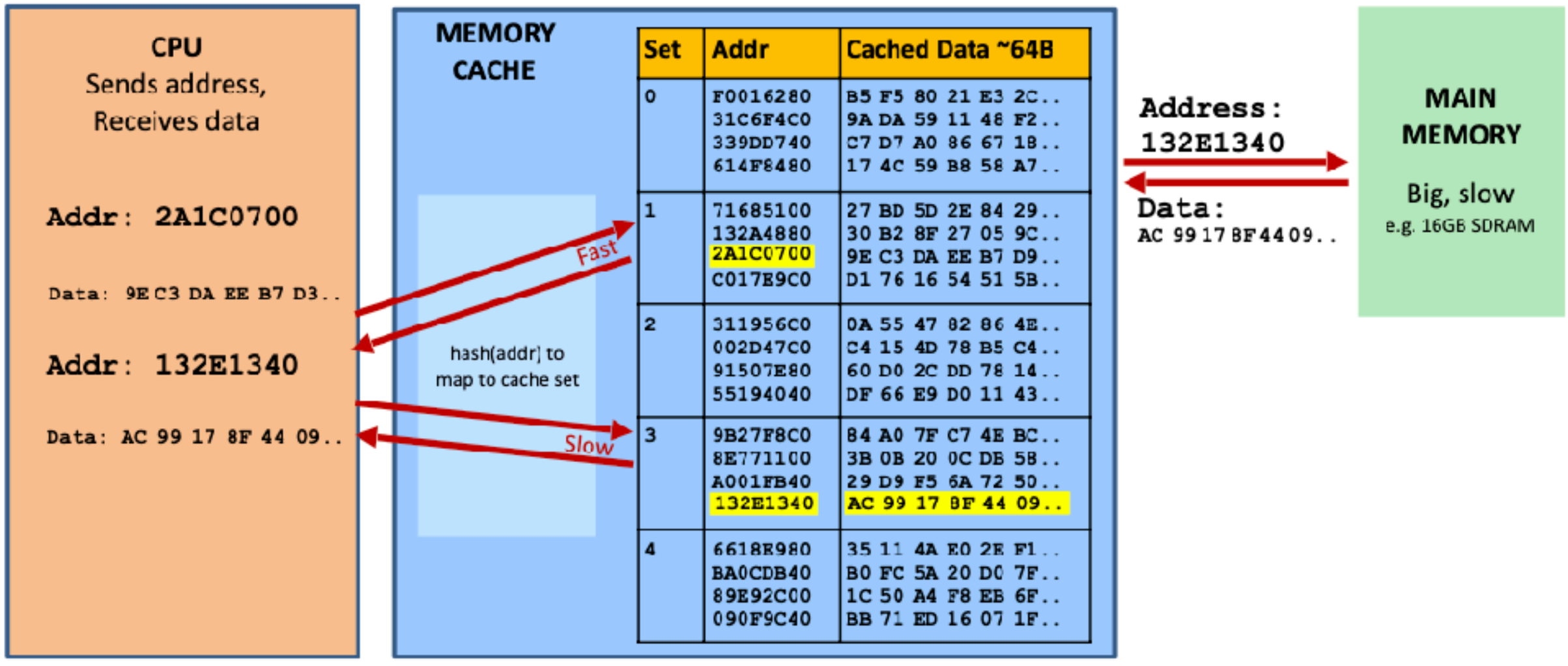


E.g., least-recently-used (LRU) replacement

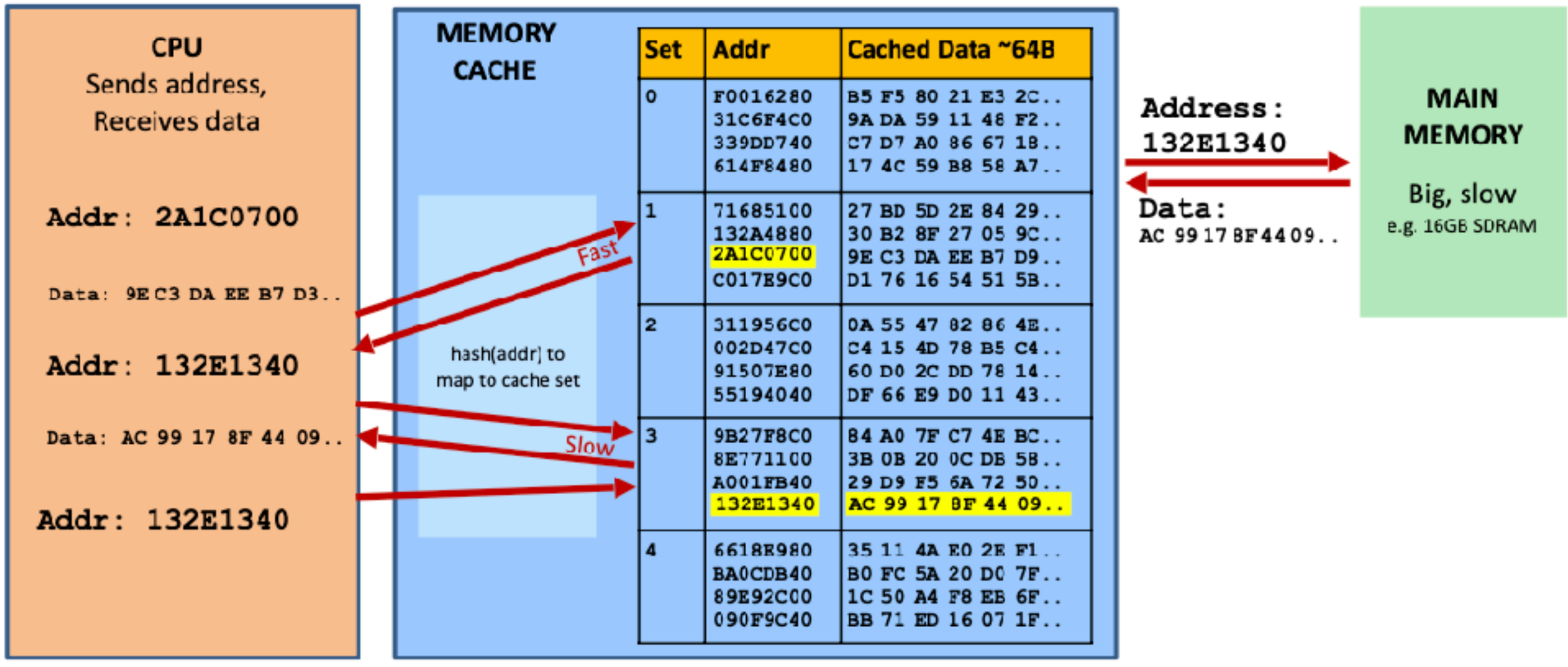
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



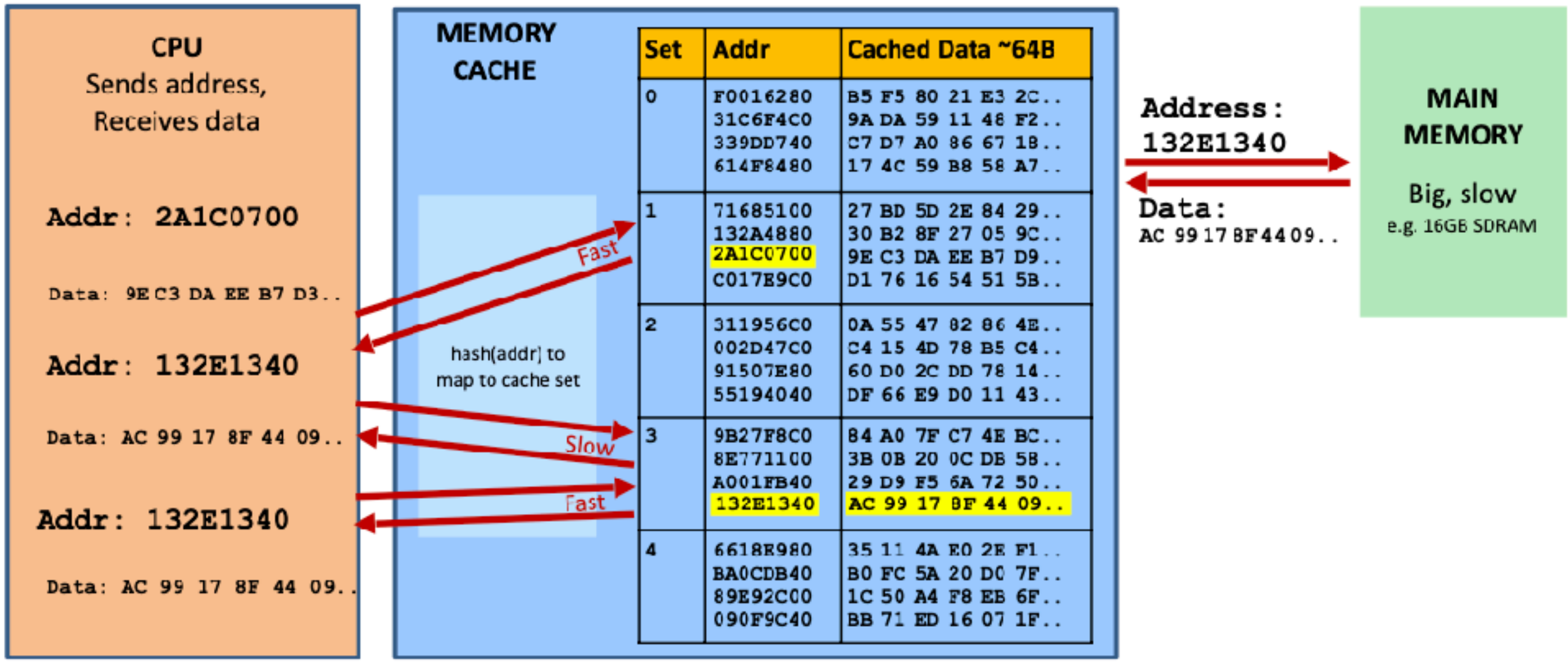
Caches hold local (fast) copy of recently-accessed 64-byte chunks of memory



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



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

Cached-based side channels

- **Threat model**: attacker *on the same physical* hardware
- **Goal**: infer *access patterns* of victim through timing
  - Basically: Address recently accessed = cache hit = fast access by attacker
- Extract secret from access patterns

General strategy

- Goal: figure out if address A is accessed by victim
- General strategy:
  - Put cache into known state
  - make victim run
  - infer what changed after run

Some important details about cache internals

Cache placement

- CPU fetches data in chunks called **cache lines** (typically 64 bytes)
  - When loading a byte, CPU loads the entire line containing that byte.
- When accessing address A, what & where to put in cache?

`0x3F=0b0000000000111111`

  - What: the line containing A (64 bytes starting at `A^~0x3F`)
  - Where: one of N slots in cached per cache placement policy.
    - 32KB L1 cache -> 512 slots

Cache placement

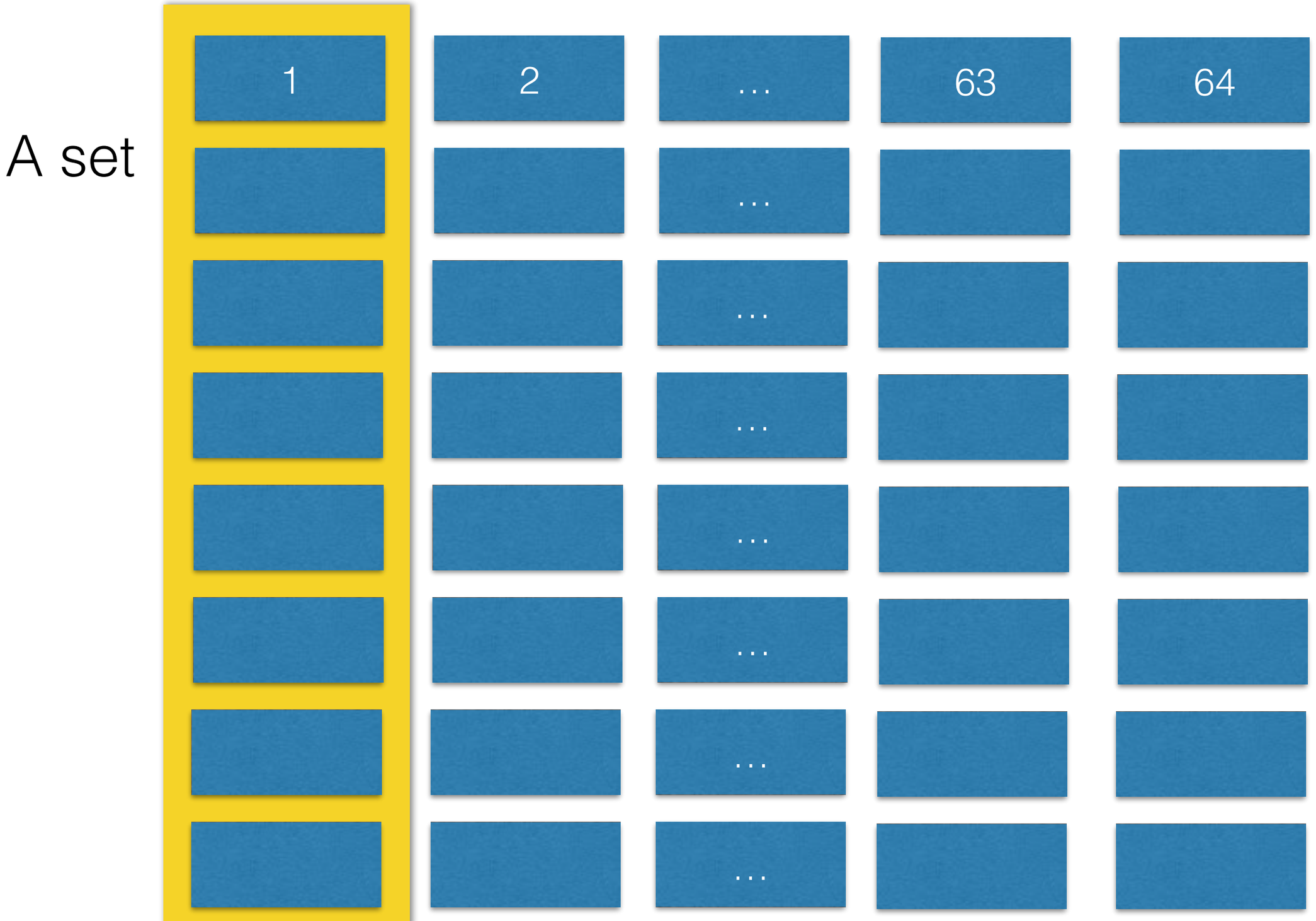
- **Direct map**: assign each line to a fixed slot
  - "Each car has a fixed spot" (E.g., m-th line is stored in slot `m % N`)
  - Problem: contention (not used in practice)
- **Full associativity**: a line can be stored anywhere
  - "Can park anywhere in the garage"
  - Problem:
    - expensive to lookup (need to go through every entry in cache)
    - Not used in practice.

W-way set associativity

- "Each car can be parked in one of W slots" (called a W-way **set**)
- L1 cache is commonly 8-way set associative
- Supposing we have 64 sets
- When address A is accessed:
  - Parse A as [52 bit tag T][6 bit set index s][6 bit offset]
  - store $(\textbf{line}||T)$ anywhere in set s
- Lookup address A: figure out which set A belongs to, goes through all W slots

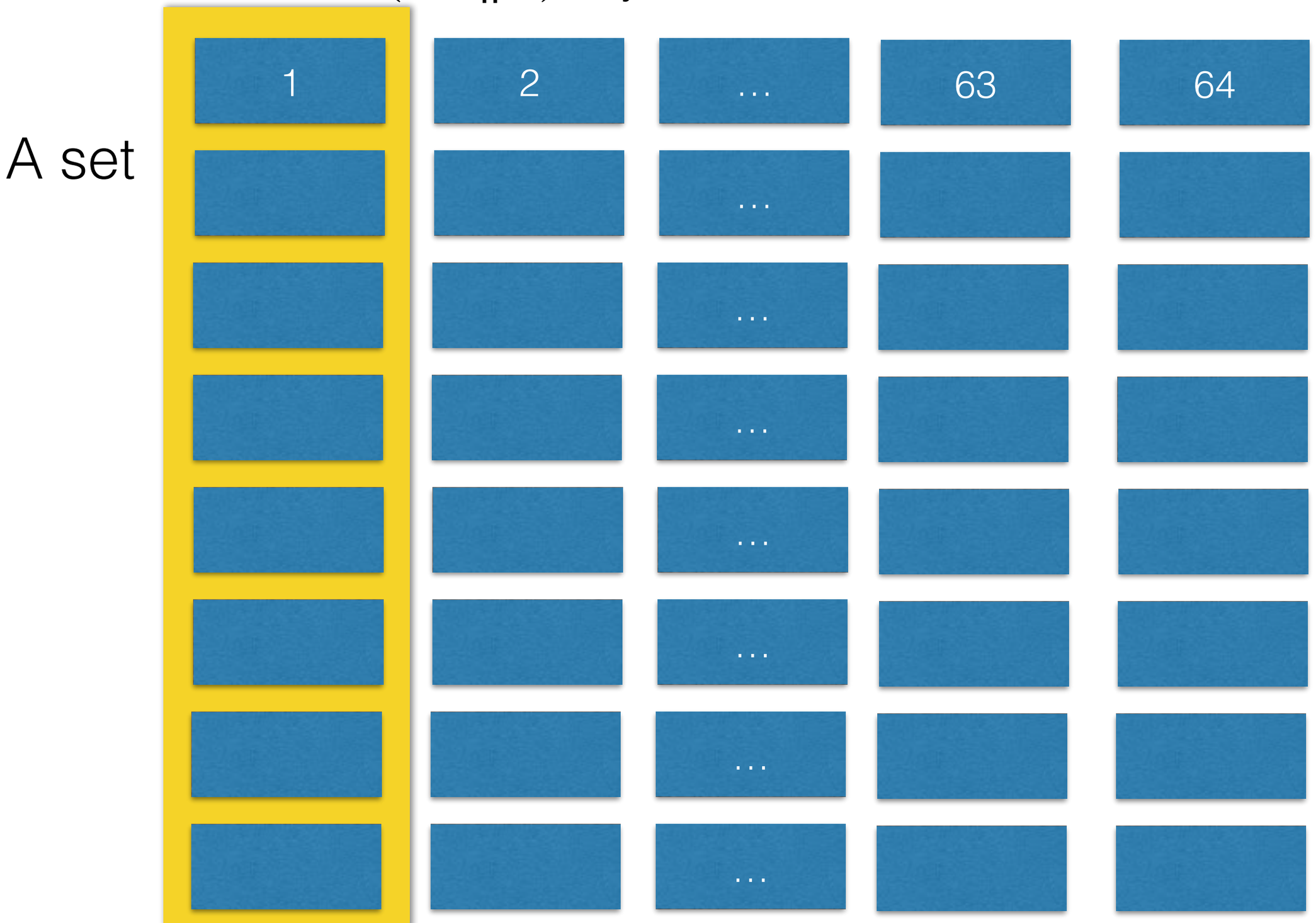
Example: 8-way set associative cache of 32 KB

- Cache line = 64B
- 32 KB / 64B = 512 slots organized into 64 sets



Example: 8-way set associative cache of 32 KB

- Parse address A as [52 bit tag T][6 bit set s][6 bit offset]
- Placement: store $(\textbf{line}||T)$ anywhere in set s



Three basic techniques

- Goal: figure out if address A is accessed by victim
  - e.g, a specific row of the lookup table
- Evict and time
  - Attacker evicts the entire set for A
  - Run victim, and see if victim slows down.
  - Learns: **some address in that set is accessed, may or may not be A.**
  - Noisy: Can only measure **total victim runtime**.

Three basic techniques

- Goal: figure out if address A is accessed by victim
  - e.g, a specific row of the lookup table
- Evict and time (low precision)
- Prime and probe
  - Attacker fills the entire set for A. Run the victim, then *attacker* accesses the set, and see if *attacker* slows down.
  - Similar learning about above.
  - **Higher precision** measurement since the attacker controls it.
- Many attacks (e.g., AES key recovery, Spectre gadgets, SGX leaks) are practical at set-level granularity.

Cache Attacks and Countermeasures: the Case of AES

2005-08-14

Dag Arne Osvik[1], Adi Shamir[2] and Eran Tromer[2]

[1] dag.arne@osvik.no
[2] Department of Computer Science and Applied Mathematics,
Weizmann Institute of Science, Rehovot 76100, Israel
{adi.shamir, eran.tromer}@weizmann.ac.il

Abstract. We describe several software side-channel attacks based on inter-process leakage through the state of the CPU's memory cache. This leakage reveals memory access patterns, which can be used for cryptanalysis of cryptographic primitives that employ data-dependent table lookups. The attacks allow an unprivileged process to attack other processes running in parallel on the same processor, despite partitioning methods such as memory protection, sandboxing and virtualization. Some of our methods require only the ability to trigger services that perform encryption or MAC using the unknown key, such as encrypted disk partitions or secure network links. Moreover, we
an extremely strong type of attack, which requires knowledge of neither the specific p
ciphertexts, and works by merely monitoring the effect of the cryptographic process
We discuss in detail several such attacks on AES, and experimentally demonstrate their
to real systems, such as OpenSSL and Linux's `dm-crypt` encrypted partitions (in the la
full key can be recovered after just 800 writes to the partition, taking 65 milliseconds)
describe several countermeasures which can be used to mitigate such attacks.

CACHE MISSING FOR FUN AND PROFIT

COLIN PERCIVAL

ABSTRACT. Simultaneous multithreading — put simply, the sharing of the execution resources of a superscalar processor between multiple execution threads — has recently become widespread via its introduction (under the name "Hyper-Threading") into Intel Pentium 4 processors. In this implementation, for reasons of efficiency and economy of processor area, the sharing of processor resources between threads extends beyond the execution units; of particular concern is that the threads share access to the memory caches.

We demonstrate that this shared access to memory caches provides not only an easily used high bandwidth covert channel between threads, but also permits a malicious thread (operating, in theory, with limited privileges) to monitor the execution of another thread, allowing in many cases for theft of cryptographic keys.

Finally, we provide some suggestions to processor designers, operating system vendors, and the authors of cryptographic software, of how this attack could be mitigated or eliminated entirely.

Three basic techniques

- Flush and reload
  - Flush/evict (`clflush`) a **specific cache line** from the cache, run the victim, attacker accessed the line, and see if the attacker slows down
    - Q: What if it does?
- Most precise (line-level granularity v.s. set level)
  - Line-level: 52 bits tag + 6 bits set index
  - Set-level: only middle 6 bits

FLUSH+RELOAD: a High Resolution, Low Noise, L3 Cache Side-Channel Attack

Yuval Yarom Katrina Falkner
The University of Adelaide

Abstract

Sharing memory pages between non-trusting processes is a common method of reducing the memory footprint of multi-tenanted systems. In this paper we demonstrate that, due to a weakness in the Intel X86 processors, page sharing exposes processes to information leaks. We present FLUSH+RELOAD, a cache side-channel attack technique that exploits this weakness to monitor access to memory lines in shared pages. Unlike previous cache side-channel attacks, FLUSH+RELOAD targets the Last-Level Cache (i.e. L3 on processors with three cache levels). Consequently, the attack program and the victim do not need to share the execution core.

We demonstrate the efficacy of the FLUSH+RELOAD attack by using it to extract the private encryption keys from a victim program running GnuPG 1.4.13. We tested the attack both between two unrelated processes in a sin-

from the shared use of the processor cache. When a process accesses a shared page in memory, the contents of the accessed memory location is cached. Gullasch et al. [29] describes a side channel attack technique that utilises this cache behaviour to extract information on access to shared memory pages. The technique uses the processor's `clflush` instruction to evict the monitored memory locations from the cache, and then tests whether the data in these locations is back in the cache after allowing the victim program to execute a small number of instructions.

We observe that the `clflush` instruction evicts the memory line from all the cache levels, including from the shared Last-Level-Cache (LLC). Based on this observation we design the FLUSH+RELOAD attack—an extension of the Gullasch et al. attack. Unlike the original attack, FLUSH+RELOAD is a cross-core attack, allowing

- USENIX '14

SGX is no exception [in principle]

CacheZoom: How SGX Amplifies The Power of Cache Attacks

Ahmad Moghimi
Worcester Polytechnic Institute
amoghimi@wpi.edu

Gorka Irazoqui
Worcester Polytechnic Institute
girazoki@wpi.edu

Thomas Eisenbarth
Worcester Polytechnic Institute
teisenbarth@wpi.edu

Abstract

In modern computing environments, hardware resources are commonly shared, and parallel computation is widely used. Parallel tasks can cause privacy and security problems if proper isolation is not enforced. Intel proposed SGX to create a trusted execution environment within the processor. SGX relies on the hardware, and claims runtime protection even if the OS and other software components are malicious. However, SGX disregards side-channel attacks. We introduce a powerful cache side-channel attack that provides system adversaries a high resolution channel. Our attack tool named *CacheZoom* is able to virtually track all memory accesses of SGX enclaves with high spatial and temporal precision. As proof of concept, we demonstrate AES key recovery attacks on commonly used implementations including those that were believed to be resistant in previous scenarios. Our results show that SGX cannot protect critical data sensitive computations, and efficient AES key recovery is possible in a practical environment. In contrast to previous works which require hundreds of measurements,

the operating system (OS) provides security and privacy services. In cloud computing, cloud providers and the hypervisor also become part of the Trusted Computing Base (TCB). Due to the high complexity and various attack surfaces in modern computing systems, keeping an entire system secure is usually unrealistic [19, 33].

One way to reduce the TCB is to outsource security-critical services to Secure Elements (SE), a separate trusted hardware which usually undergoes rigorous auditing. Trusted Platform Modules (TPM), for example, provide services such as cryptography, secure boot, sealing data and attestation beyond the authority of the OS [40]. However, SEs come with their own drawbacks: they are static components and connected to the CPU over an untrusted bus. Trusted Execution Environments (TEE) are an alternative, which provide similar services within the CPU. A TEE is an isolated environment to run software with a higher trust level than the OS. The software running inside a TEE has full access to the system resources while it is protected from other applications and the OS. Examples include ARM TrustZone [4]

- Enclaves share various cache with attacker.
- Address translation & cache placement are managed by OS!

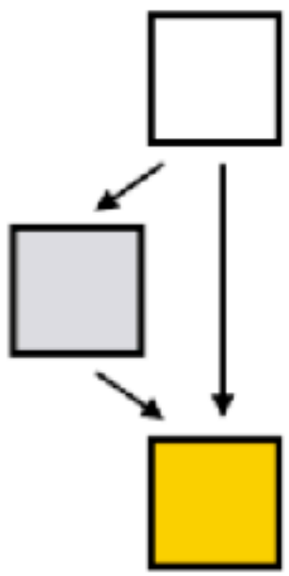
Mitigation of side channels

Mitigating side channels

- Principle: **carefully write programs to eliminate secret-dependent side effects**
  - Timing
  - Memory access patterns
- Challenge:
  - not always easy…
  - Performance

Control flow introduces time variability

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



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

Does padding the else branch work?

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

where s1 and s1' take same amount of time

Problem: Instructions access still observable

Some common tricks

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

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

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

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

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

CT_SEL

- `CT_SEL(cond, a, b)`
  - Return a if cold == true, else b.
- Common implementation

```
uint32_t CT_SEL(uint32_t a, uint32_t b, uint32_t cond) {
    // cond must be 0 or 1
    uint32_t mask = -cond; // if cond = 1 → 0xFFFFFFFF; if cond = 0 → 0x00000000
    return (a & ~mask) | (b & mask);
}
```

- Writing constant-time code is hard.

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

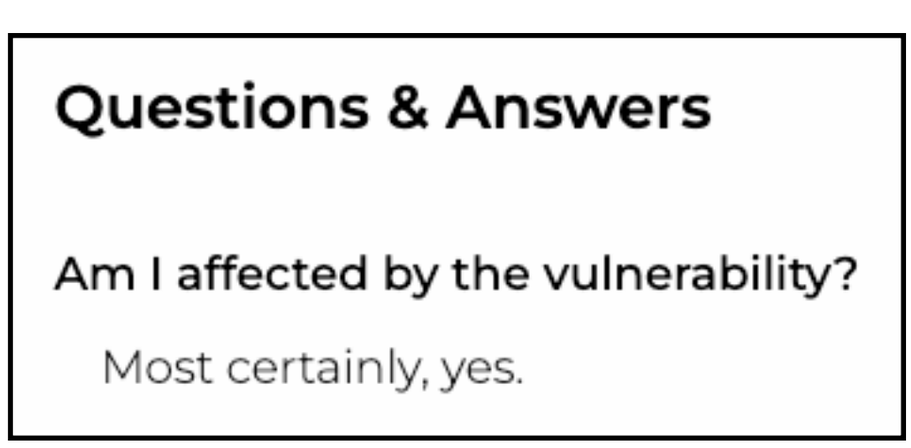
- It can be slower, larger, and more complex.

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

New opportunities for side channels: Speculative Execution

Spectre and Meltdown

- Lipp et al., Kocher et al. 2017
- **Extremely damaging attacks** affecting pretty much every CPU



https://meltdownattack.com/

Speculative Execution

- A widely used performance optimization
- Example (speculative branch execution)

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

- **Branch predictor** predicts which branch to take based on prior history
  - If “if”, starts executing compute(b) before loading finishes
- When loading does finish, check if guess was correct:
  - Correct: Save speculative work → performance gain 👍
  - Incorrect: Discard speculative work → no harm?
- Basis of the **Spectre attack**.

Spectre attack (rough idea)

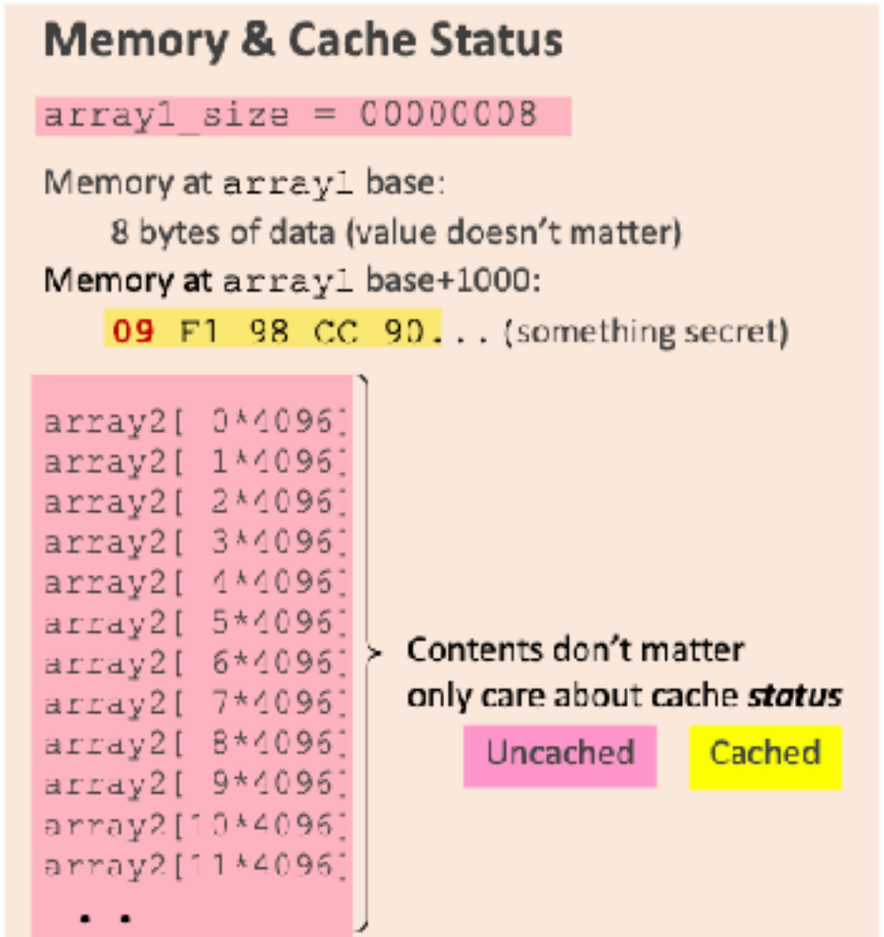
Goal: read value at `array1 + 1000`
(e.g., out of bound read)
Attacker controls input (x, here)

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

Vulnerable code pattern

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 attack (rough idea)

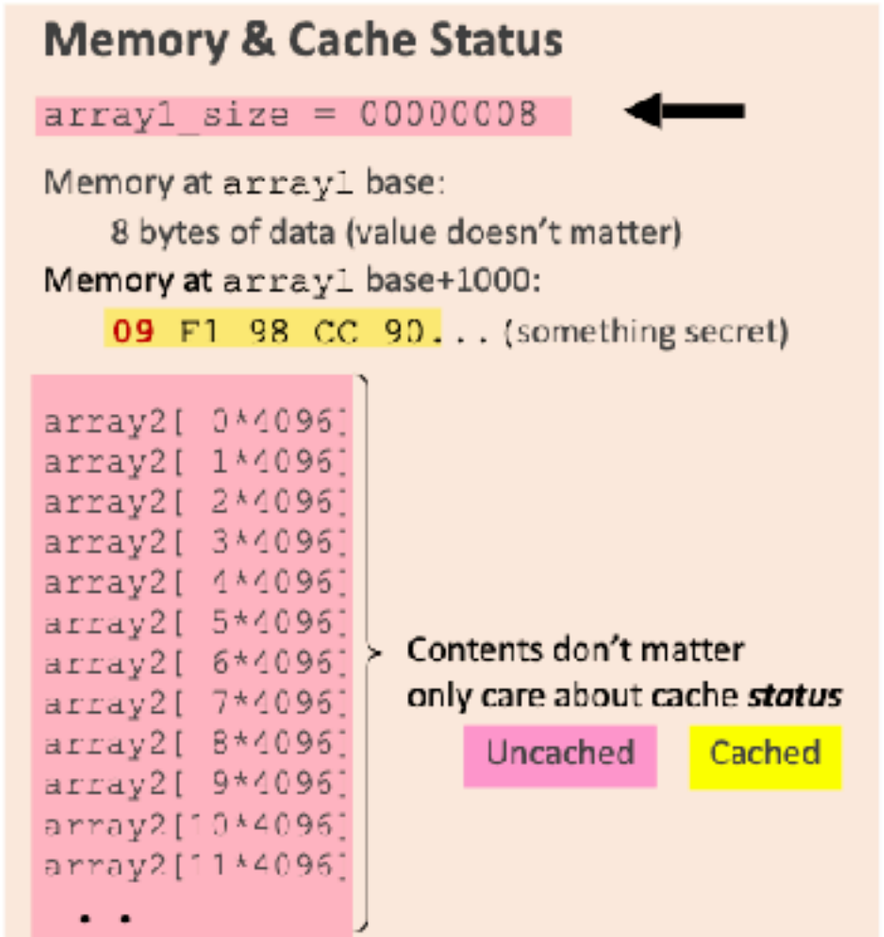
Goal: read value at `array1 + 1000`
(e.g., out of bound read)

```
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 attack (rough idea)

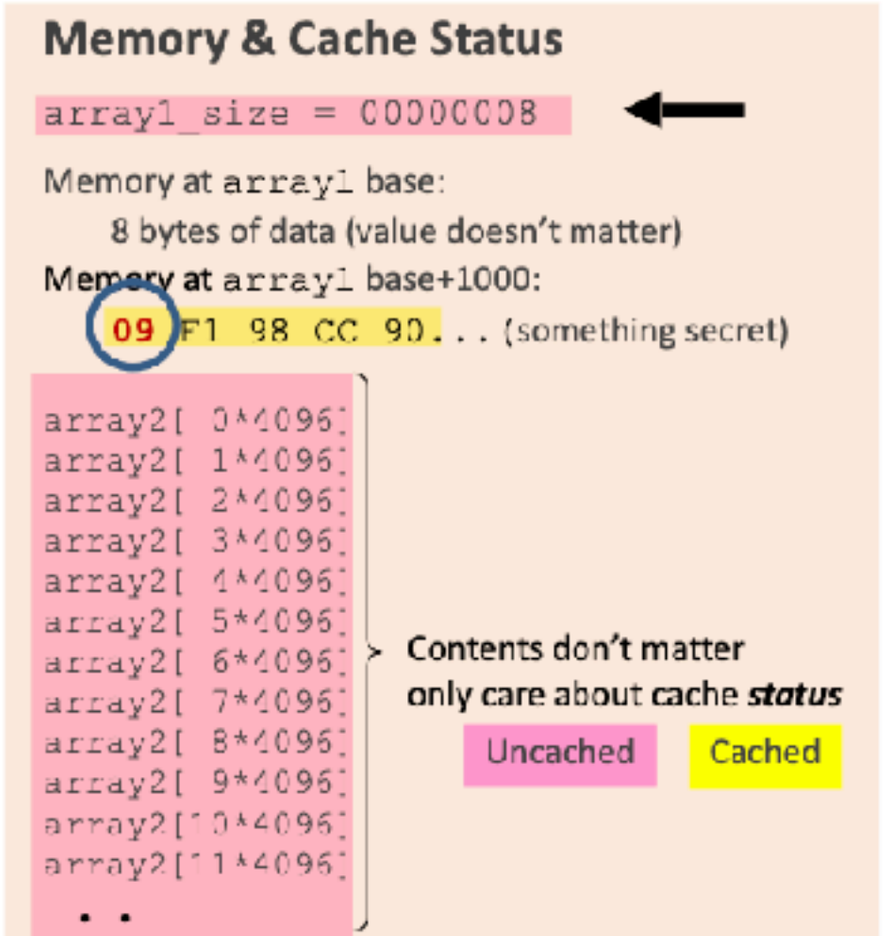
Goal: read value at `array1 + 1000`
(e.g., out of bound read)

```
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 attack (rough idea)

Goal: read value at `array1 + 1000`
(e.g., out of bound read)

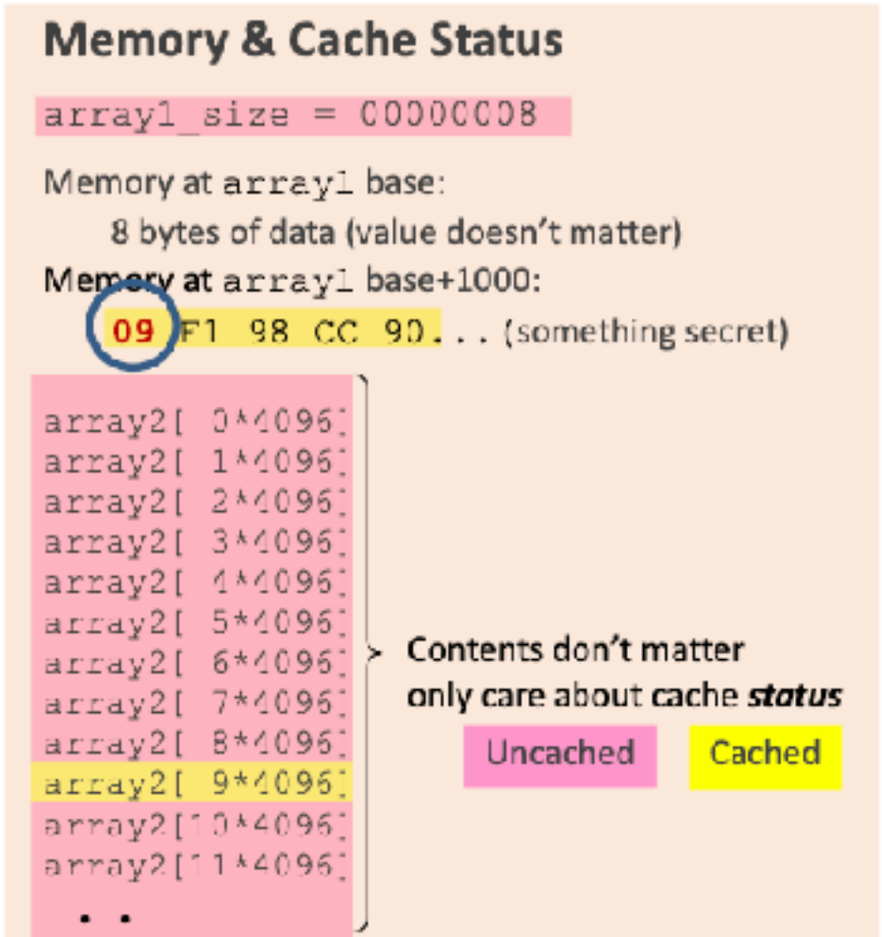
```
if (x < array1_size)
    y = array2[array1[x]*4096];
```

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 attack (rough idea)

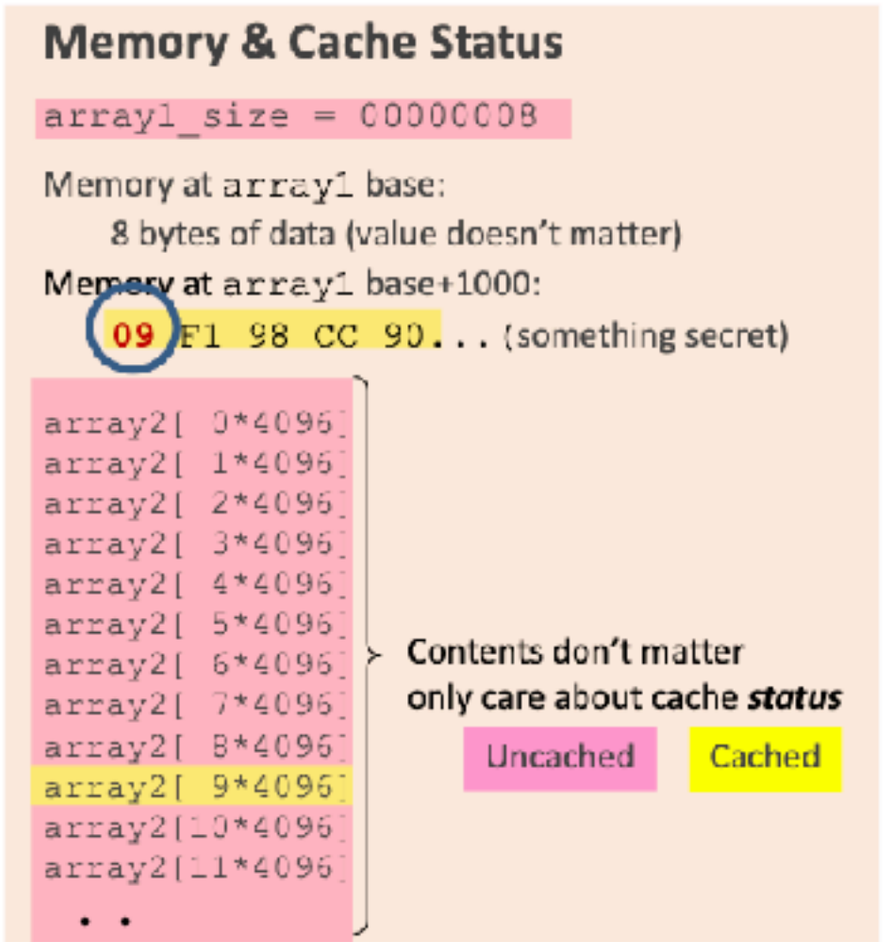
Goal: read value at `array1 + 1000`
(e.g., out of bound read)

```
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)



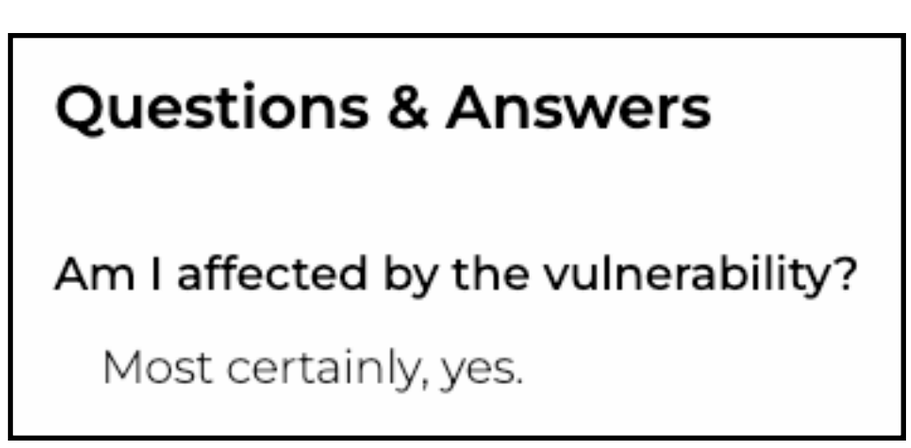
Example 2: Speculative Memory Access

```
uint8_t a = *(uint8_t*)0xFFFF0000; // some kernel address
```

- CPU will load secret from 0xFFFF0000 and store in `a`
  - Before slow security checks finish
- It continues execution (using `a` in some way)
- …until security check finishes — abort and restore `a`
- Too late - value in `a` already leaked through side channels
- Basis of **Meltdown attack**
- “a delicate race condition in the CPU’s access control logic that allows an attacker to use the results of unauthorized memory accesses in transient out-of-order instructions before they are rolled back.”

Spectre and Meltdown

- Lipp et al., Kocher et al. 2017
- **Extremely damaging attacks** affecting pretty much every CPU



https://meltdownattack.com/

How about SGX?

FORESHADOW: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution

Jo Van Bulck[1], Marina Minkin[2], Ofir Weisse[3], Daniel Genkin[3], Baris Kasikci[3], Frank Piessens[1], Mark Silberstein[2], Thomas F. Wenisch[3], Yuval Yarom[4], and Raoul Strackx[1]

[1]*imec-DistriNet, KU Leuven,* [2]*Technion,* [3]*University of Michigan,* [4]*University of Adelaide and Data61*

- Foreshadow abuses the same vulnerability as Meltdown
- Full confidentiality broken
- Also breaks integrity because it extracted the attestation key…

How about SGX?

SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution

Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, Ten H. Lai
Department of Computer Science and Engineering
The Ohio State University
{chen.4329, chen.4825, xiao.465}@osu.edu
{yinqian, zlin, lai}@cse.ohio-state.edu

- Can steal secrets from SGX enclave
- Breaks confidentiality
- Also demonstrated extraction sealing key… can break integrity too

SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution

Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, Ten H. Lai
Department of Computer Science and Engineering
The Ohio State University
{chen.4329, chen.4825, xiao.465}@osu.edu
{yinqian, zlin, lai}@cse.ohio-state.edu

- Identity vulnerable patterns (simplified)
  - Attack controls input x

```
if (x < array_len) {
    uint8_t value = secret_array[x];   // secret used here
    temp = probe_array[value * 4096];  // leaks via cache
}
```

- Turns out such patterns are very common

Intel's mitigation

- To mitigate Meltdown (Foreshadow)
  - E.g., Intel disabled support for speculative reads from SGX with microcode updates — but earlier CPUs were exploitable.
- Spectre
  - Intel suggested software mitigation

```
if (x < array_len) {
    uint8_t value = secret_array[x];
    temp = probe_array[value * 4096];
}
```

```
if (x < array_len) {
    _mm_lfence();   // serializing barrier
    uint8_t value = secret_array[x];
    temp = probe_array[value * 4096];
}
```

Summary

- SGX protects access, but not *information flow*
  - Developers are responsible for not introducing **side channels** (e.g., using constant time programming)
  - More stupid errors are possible, e.g., put secrets in logs
- **Cache-based timing side channels**
  - Evict+Time, Prime+Probe, Flush+Reload
  - Fixable with constant-time programming in principle
- Speculative execution attack
  - A hardware bug
  - Mitigation: inserting lfence() or disabling speculative exec.

Takeaways

- Exploitation in lab != real problems
  - But it definitely doesn't inspire confidence
- Secure TEE code against side channels is left for developers
- Should we deem TEE as insecure?
- Better way: Design protocols with TEE failures in mind
  - E.g., it's possible to use it only for integrity, but not for confidentiality

Summary

- “SoK: Understanding Design Choices and Pitfalls of Trusted Execution Environments” (AsiaCCS’24)
  - Vulnerable design
  - Controlled side channels
  - Micro-architectural side channels
  - Speculative execution
  - Micro-architecture flaw
  - Hardware flaw
- Most widely available TEEs have some vulnerabilities.
- All vulnerabilities break confidentiality; some also break integrity (i.e., leaked attestation keys.)

	Attack Surface	Vendor	TEE Type	Authors	Physical	Confidentiality	Integrity
Vulnerable Design	Unprotected Regs	AMD SEV	VM	Hetzelt *et al.* [32]	✗	✓	✗
	Unprotected Regs	AMD SEV	VM	Werner *et al.* [93]	✗	✓	✓
	Unprotected NPT	SEV/SEV-ES	VM	Morbitzer *et al.* [65, 66]	✗	✓	✗
	Unprotected NPT	SEV/SEV-ES	VM	Li *et al.* [53]	✗	✓	✗
	Unprotected NPT	SEV/SEV-ES	VM	Morbitzer *et al.* [67]	✗	✓	✓
	Unauthenticated Encryption	SEV/SEV-ES	VM	Li *et al.* [53]	✗	✓	✗
	TLB Misuse	SEV/SEV-ES	VM	Li *et al.* [56]	✗	✓	✓
	Debug Registers	SEV/ES/SNP	VM	AMD *et al.* [3]	✗	✓	✓
	Untrusted CPUID	SEV/ES/SNP	VM	AMD *et al.* [3]	✗	✓	✗
	Remote attestation & firmware	AMD SEV	VM	Buhren *et al.* [14]	✗	✓	✗
Controlled Side-Channel	Page Table	Intel SGX	Enclave	Xu *et al.* [98]	✗	✓	✗
	Page Table	Intel SGX	Enclave	Shinde *et al.* [79]	✗	✓	✗
	Page Table	Intel SGX	Enclave	Wang *et al.* [89]	✗	✓	✗
	Page Table	Intel SGX	Enclave	Van Bulck *et al.* [85]	✗	✓	✗
	Page Table	SEV/SEV-ES	VM	Werner *et al.* [93]	✗	✓	✗
	Timer Interrupt	Intel SGX	Enclave	Van Bulck *et al.* [83, 84]	✗	✓	✗
	Timer Interrupt	AMD SEV	VM	Li *et al.* [55]	✗	✓	✗
	Timer Interrupt	AMD SEV	VM	Van Wilke *et al.* [95]	✗	✓	✗
Micro-architectural Side-Channel	L1/ L2 Cache	Intel SGX	Enclave	Götzfried *et al.* [30]	✗	✓	✗
	L1/ L2 Cache	Intel SGX	Enclave	Moghimi *et al.* [62]	✗	✓	✗
	L1/ L2 Cache	Intel SGX	Enclave	Dall *et al.* [24]	✗	✓	✗
	L1/ L2 Cache	Intel SGX	Enclave	Brasser *et al.* [12]	✗	✓	✗
	Last-level Cache	Intel SGX	Enclave	Schwarz *et al.* [77]	✗	✓	✗
	Memory Bus	Intel SGX	Enclave	Lee *et al.* [48]	✓	✓	✗
	Branch Predictor	Intel SGX	Enclave	Lee *et al.* [50]	✗	✓	✗
	Branch Predictor	Intel SGX	Enclave	Evtyushkin *et al.* [28]	✗	✓	✗
	Branch Predictor	Intel SGX	Enclave	Huo *et al.* [34]	✗	✓	✗
	PMC	Intel SGX	Enclave	Götzfried *et al.* [30]	✗	✓	✗
	PMC	AMD SEV	VM	Li *et al.* [52]	✗	✓	✗
	Power-Consumption	Intel SGX	Enclave	Lipp *et al.* [57]	✗	✓	✗
	Power-Consumption	AMD SEV	VM	Wang *et al.* [90]	✗	✓	✗
	Ciphertext	AMD SEV	VM	Li *et al.* [52, 55]	✗	✓	✗
Speculative Execution	L1 Cache	Intel SGX	Enclave	Van Bulck *et al.* [81]	✗	✓	✗
	BTB	Intel SGX	Enclave	Chen *et al.* [16]	✗	✓	✗
	RSB	Intel SGX	Enclave	Koruyeh *et al.* [47]	✗	✓	✗
	LFB	Intel SGX	Enclave	Schwarz *et al.* [75]	✗	✓	✗
	LFB	Intel SGX	Enclave	Van Schaik *et al.* [86, 87, 88]	✗	✓	✗
	LFB	Intel SGX	Enclave	Intel [35]	✗	✓	✗
	LFB	Intel SGX	Enclave	Van Bulck *et al.* [82]	✗	✓	✗
	SIMD	Intel SGX	Enclave	Moghimi [53]	✗	✓	✗
μarchitecture Flaw	Weak Encryption	SEV/SEV-ES	VM	Du *et al.* [27]	✗	✓	✗
	Weak Encryption	SEV/SEV-ES	VM	Wilke *et al.* [94]	✗	✓	✓
	Weak Encryption & I/O	SEV/SEV-ES	VM	Li *et al.* [54]	✗	✓	✗
	INVD Instruction	SEV-ES/SNP	VM	Zhang *et al.* [99]	✗	✓	✓
	APIC	Intel SGX	Enclave	Borrello *et al.* [10]	✗	✓	✗
	MMIO stale data	Intel SGX	Enclave	Intel [37]	✗	✓	✗
Hardware Flaw	Voltage glitching	Intel SGX	Enclave	Qiu *et al.* [73]	✗	✓	✓
	Voltage glitching	Intel SGX	Enclave	Murdock *et al.* [69]	✗	✓	✓
	Voltage glitching	Intel SGX	Enclave	Kenjar *et al.* [45]	✗	✓	✓
	Voltage glitching	Intel SGX	Enclave	Chen *et al.* [17]	✓	✓	✓
	Voltage glitching	AMD SEV	VM	Buhren *et al.* [13]	✓	✓	✓
	Rowhammer	Intel SGX	Enclave	Jang *et al.* [40]	✗	✓	✗