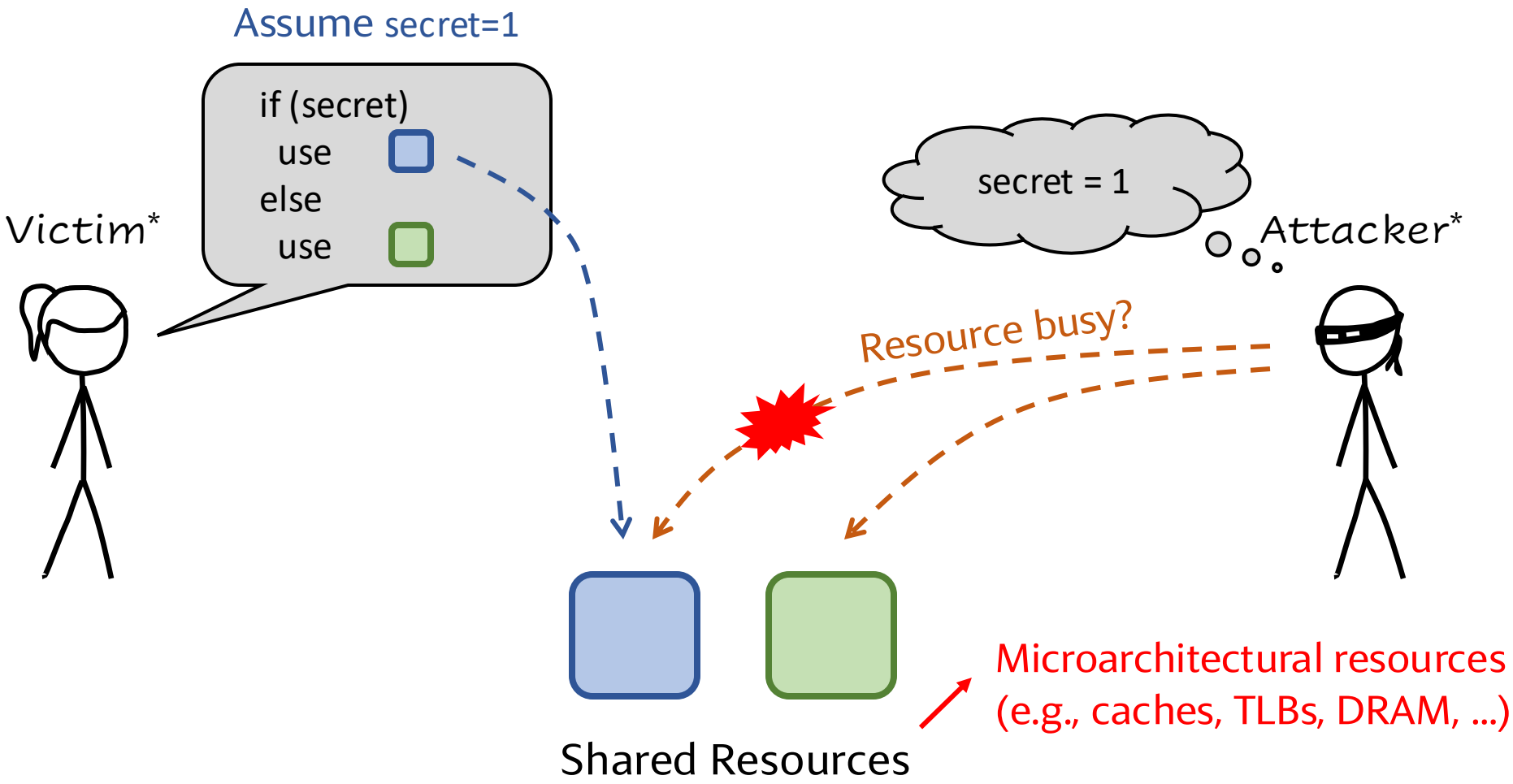
ECE 382N-Sec (FA25):

L1: Cache-Based Covert/Side Channels

Neil Zhao

neil.zhao@utexas.edu

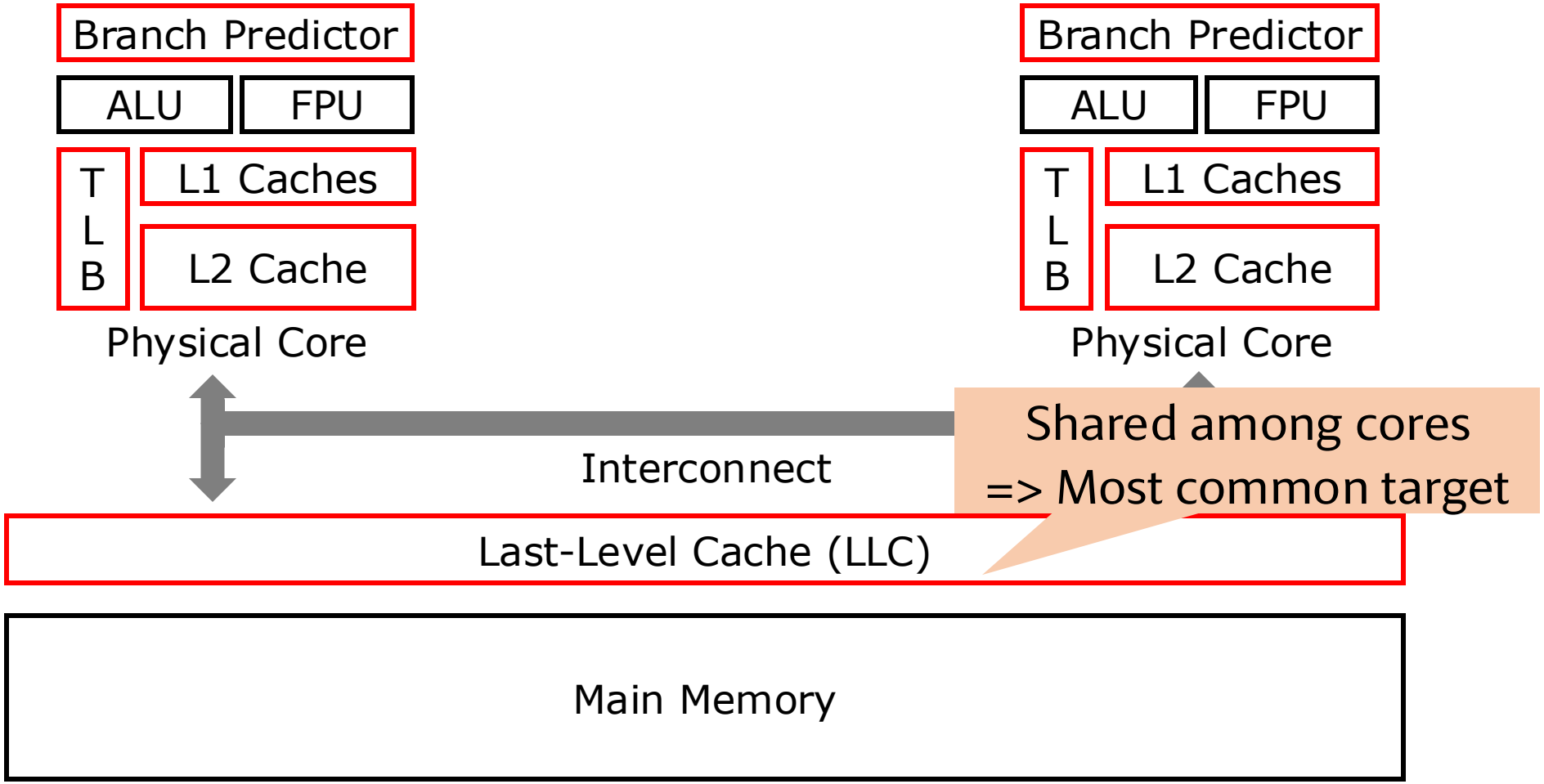
Previously on ECE 382N: Sharing Resource => Side Channel



*Characters are based on https://xkcd.com/2176 and https://xkcd.com/1808 (under a CC Attribution-NonCommercial 2.5 License)

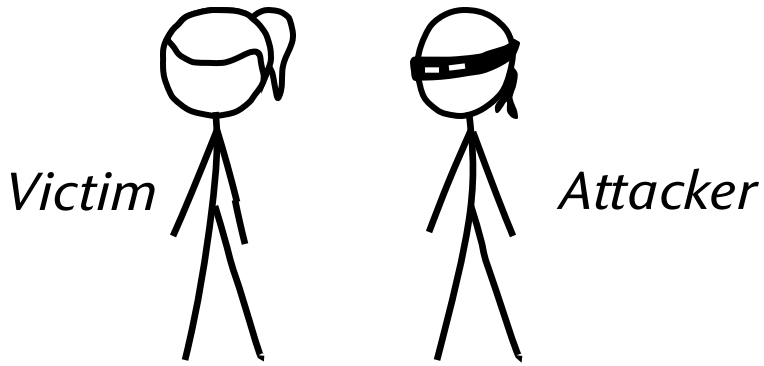
Why Cache?

Many cache and cache-like resources => large attack surface



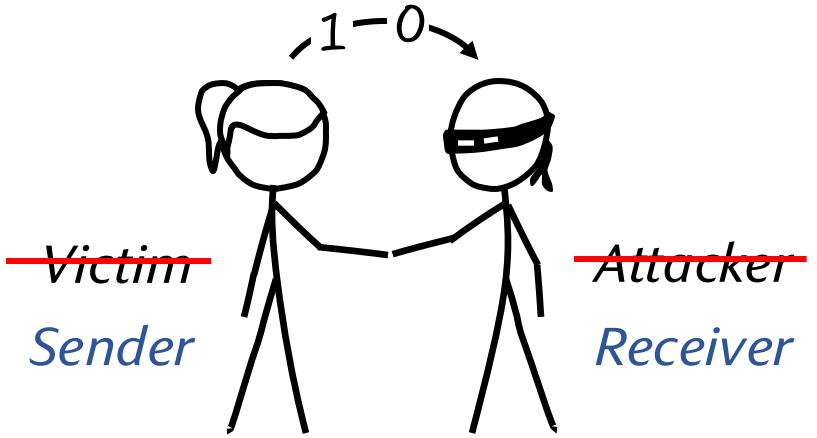
Previously on ECE 382N: Side Channels vs. Covert Channels

Side Channel



Victim is not cooperating, the leakage is incidental

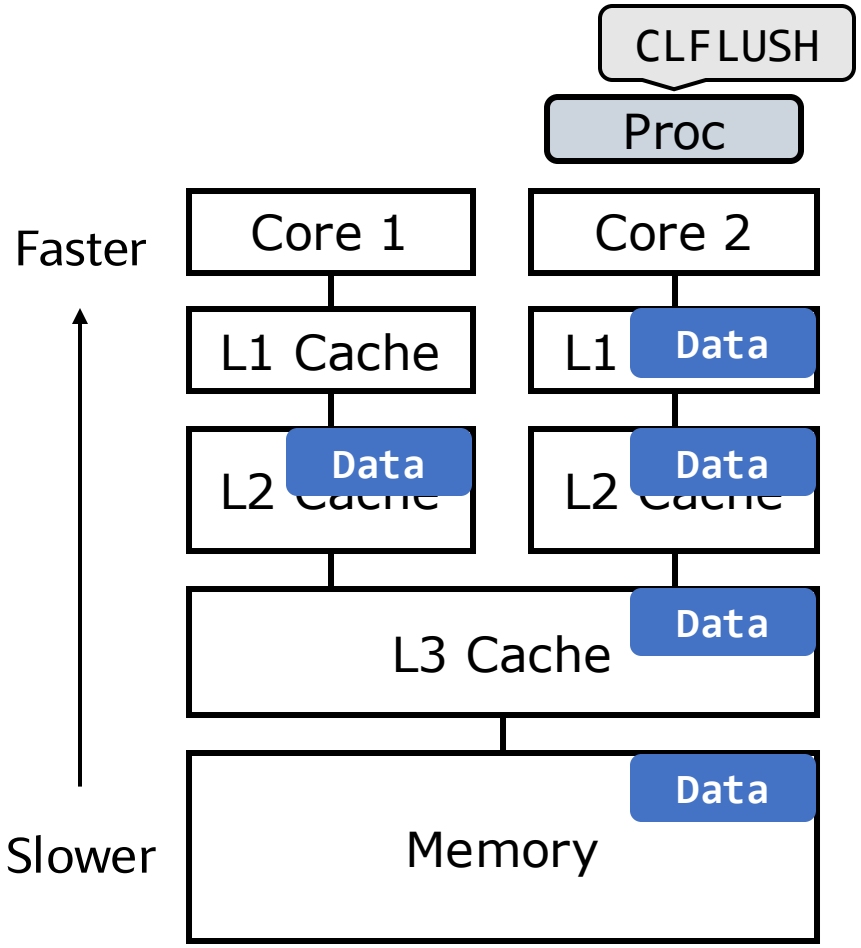
Covert Channel



Victim/sender is colluding

A covert channel is more capable than a side channel. Side-channel attacks often start with building a covert channel

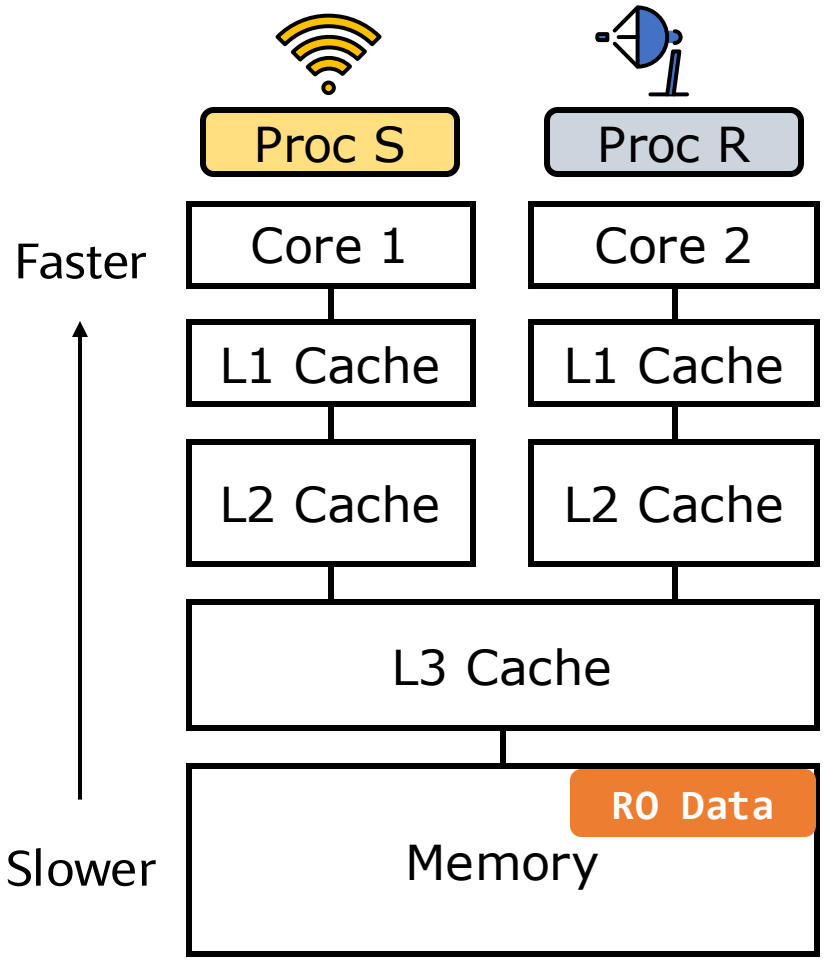
Flush+Reload: Leveraging "Cache Flush" Instructions



- CPUs often provide instructions to flush a cache line at given virtual address from **every level of the cache hierarchy**
  - Invalidate the cache line
  - Write back its data if it's modified
- Examples:
  - x86: `CLFLUSH <vaddr>`
  - ARM: `DC CIVAC <vaddr>` (Data or unified Cache line Clean and Invalidate by VA to PoC)
- Why?
  - Maintaining Coherence with Cached Memory-Mapped IO[1]
  - Push modified data to non-volatile memory for crash recovery

[1]https://sites.utexas.edu/jdm4372/2013/05/30/coherence-with-cached-memory-mapped-io/ by John D McCalpin (TACC)

Flush+Reload Covert Channel Setup



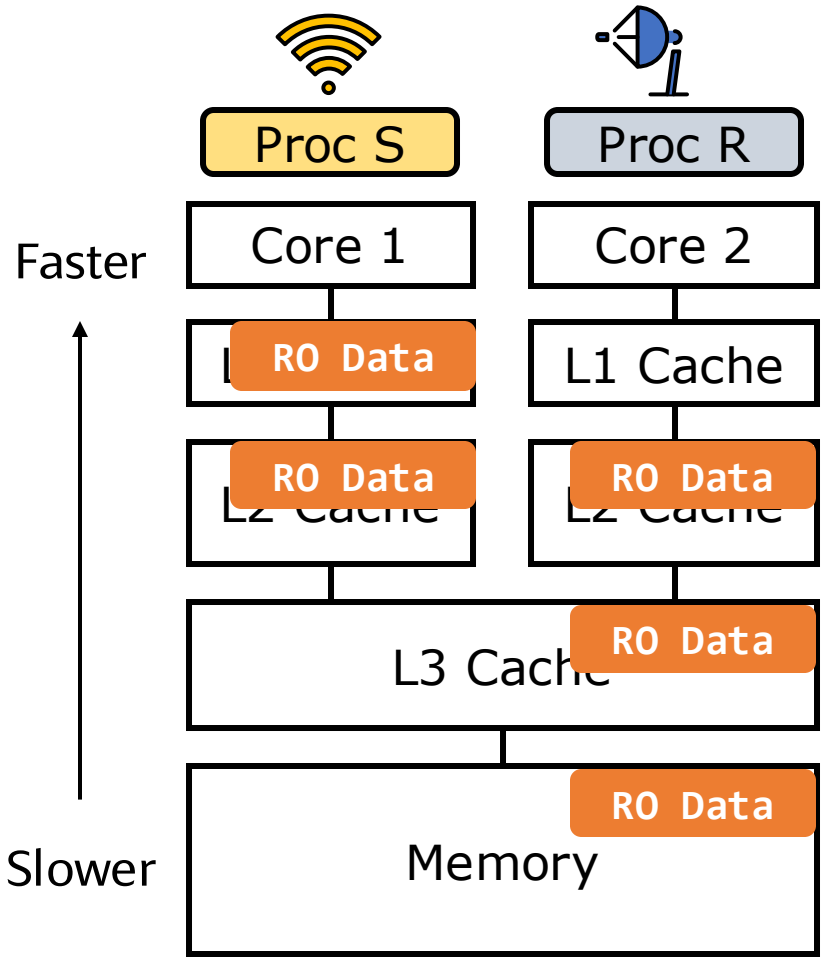
Setup:

- Sender and receiver processes run on the same host (**co-location**)
- Both processes cannot communicate with each other
- There are **read-only** memory pages shared between these two processes

Goal: Covertly transmit information from the sender to the receiver, bypassing the communication restrictions

RO Data Shared memory page (read-only)

Flush+Reload Covert Channel Timeline

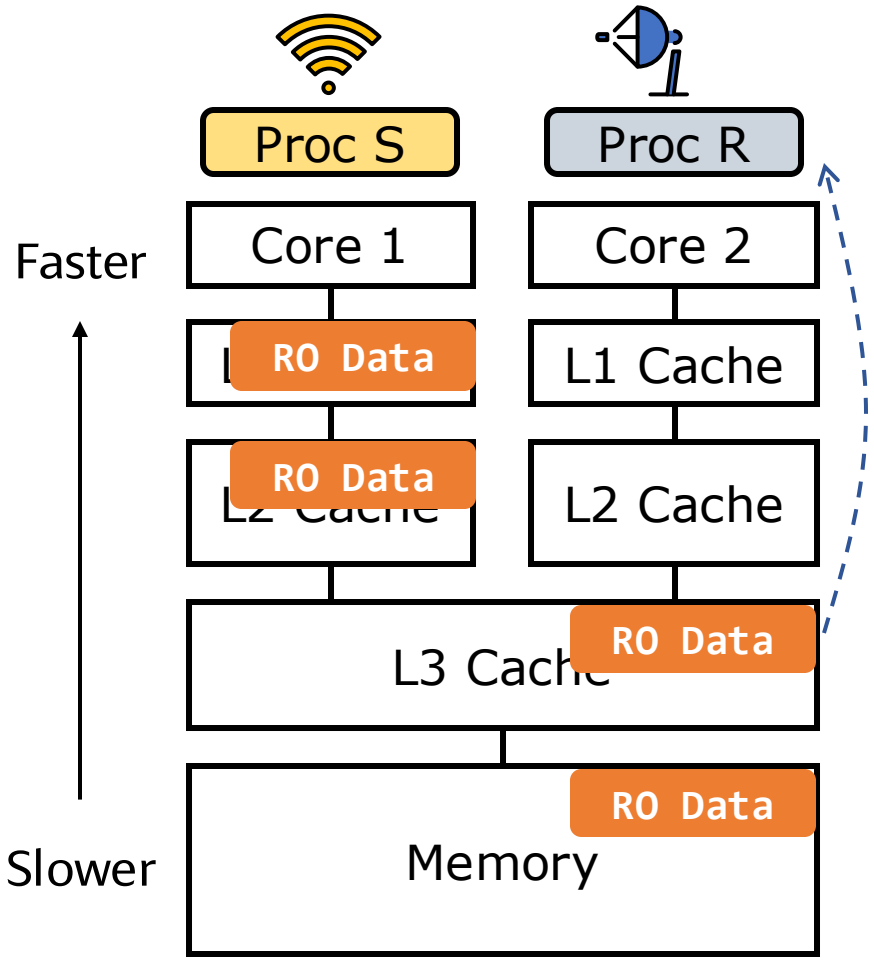


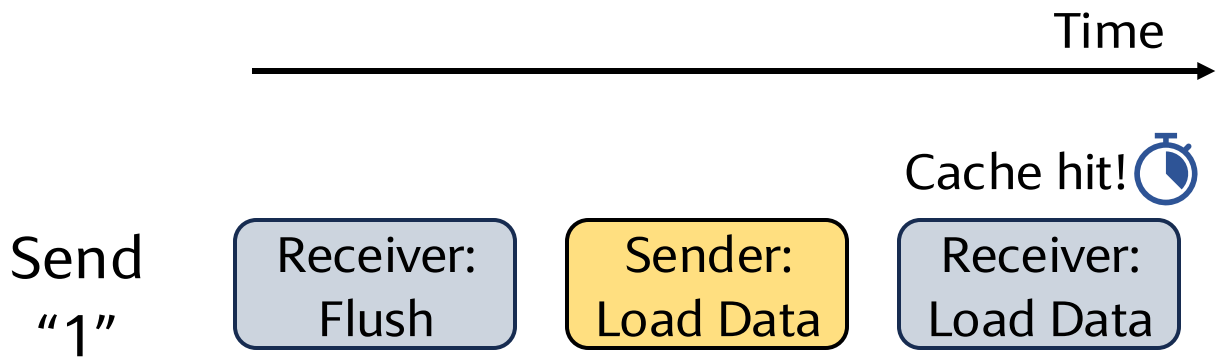
Send "1"

Receiver: Flush

RO Data Shared memory page (read-only)

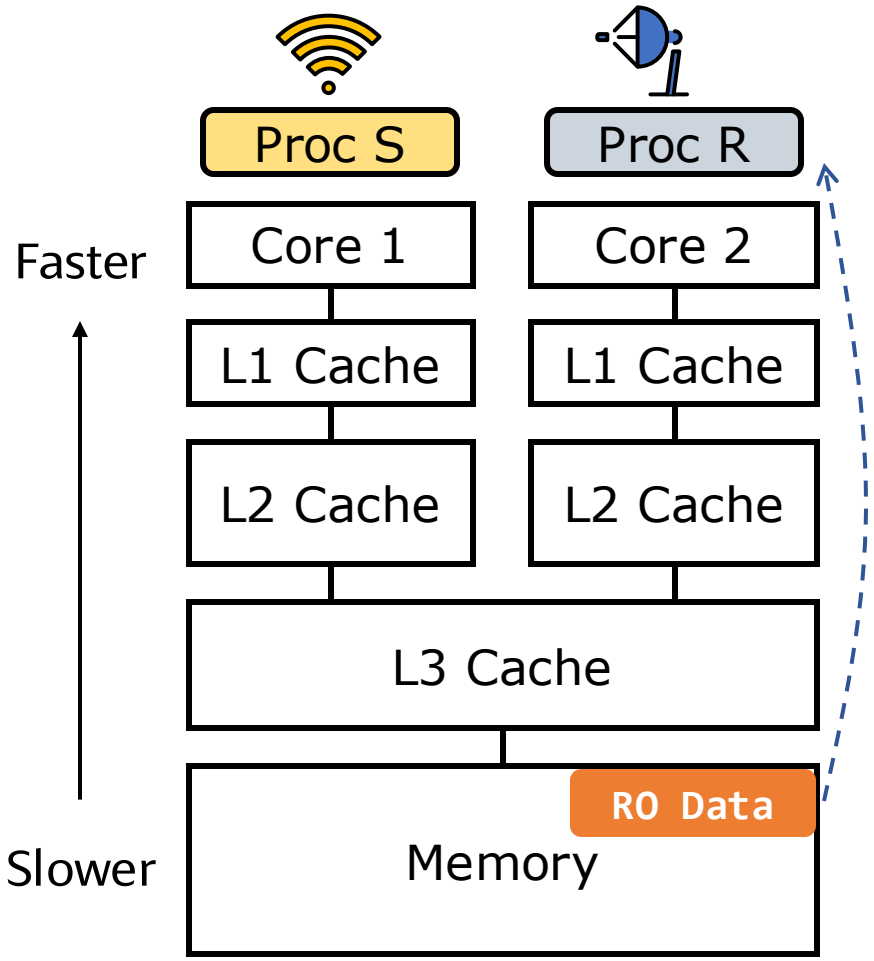
Flush+Reload Covert Channel Timeline

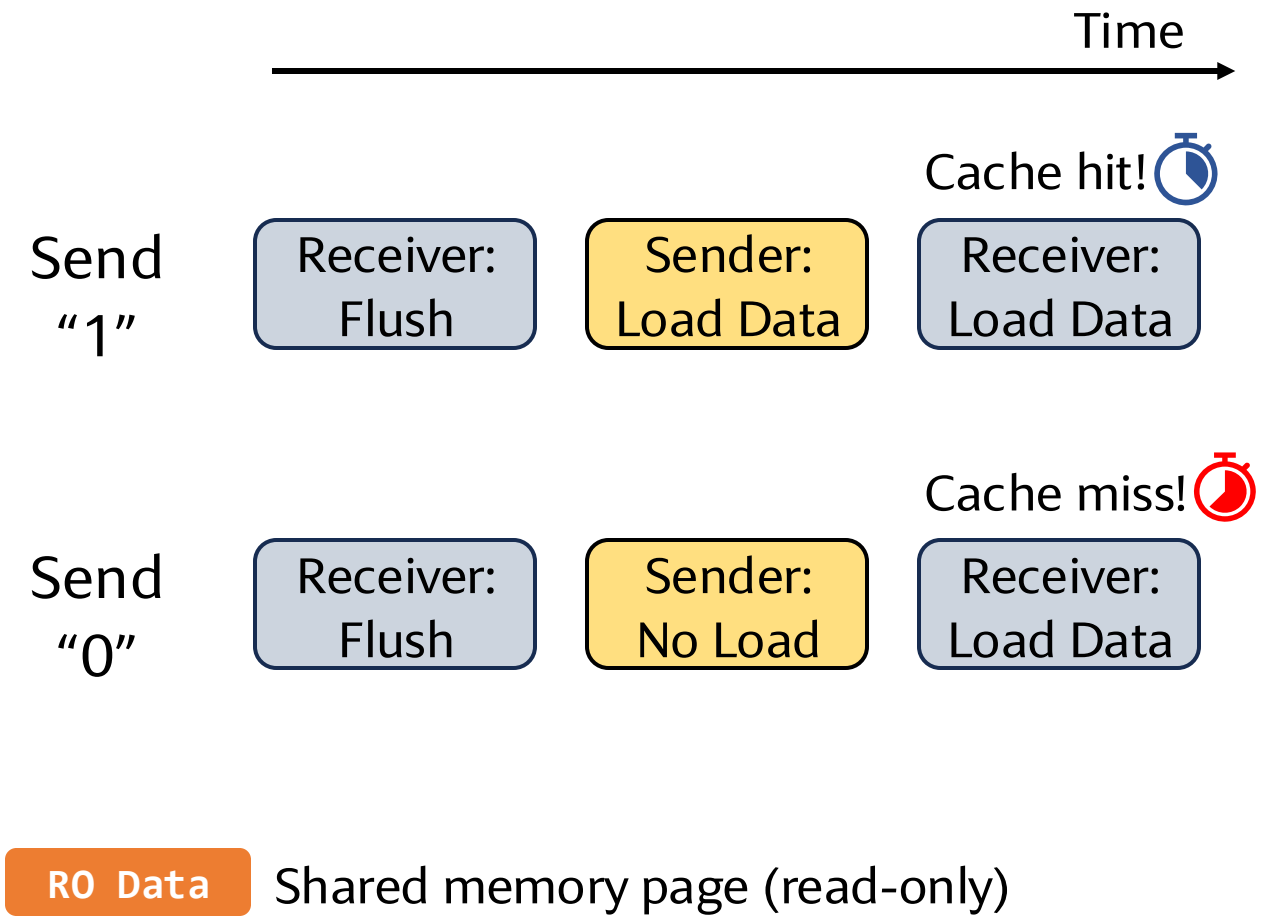




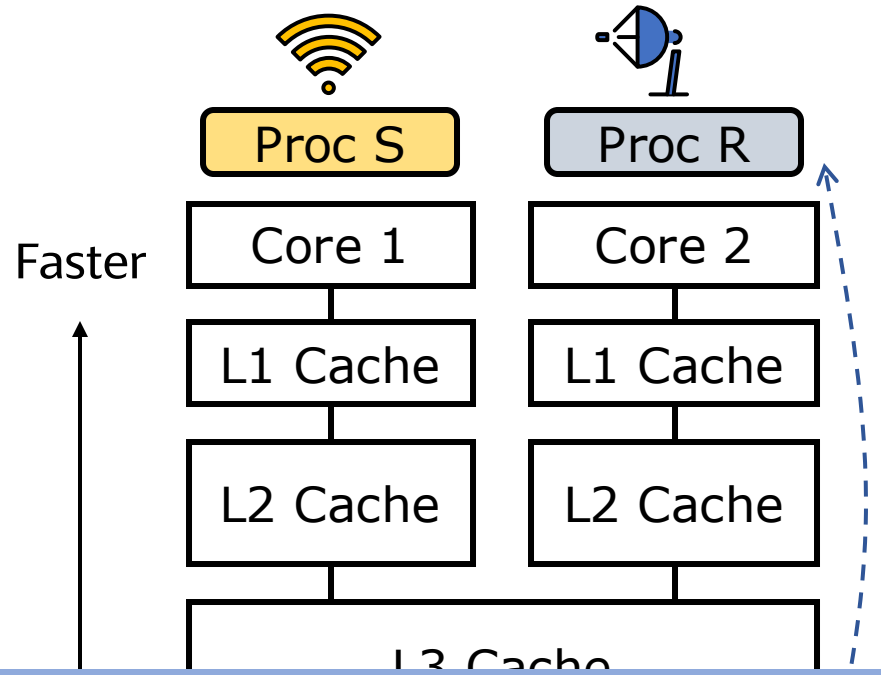
RO Data Shared memory page (read-only)

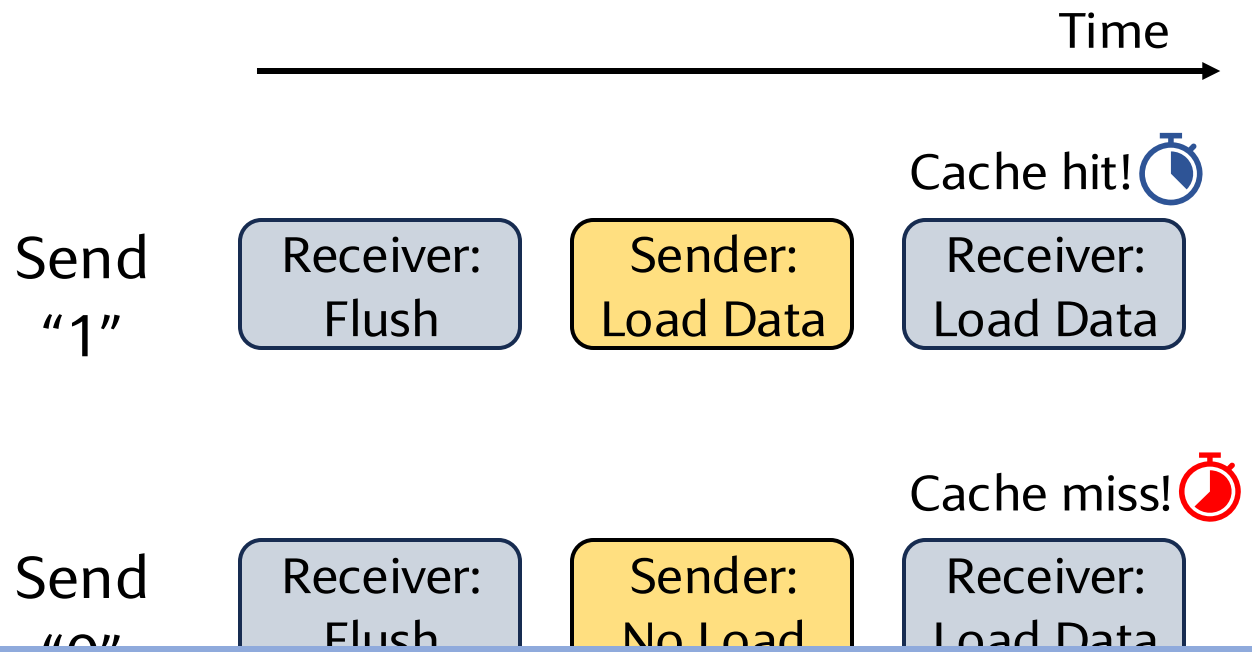
Flush+Reload Covert Channel Timeline





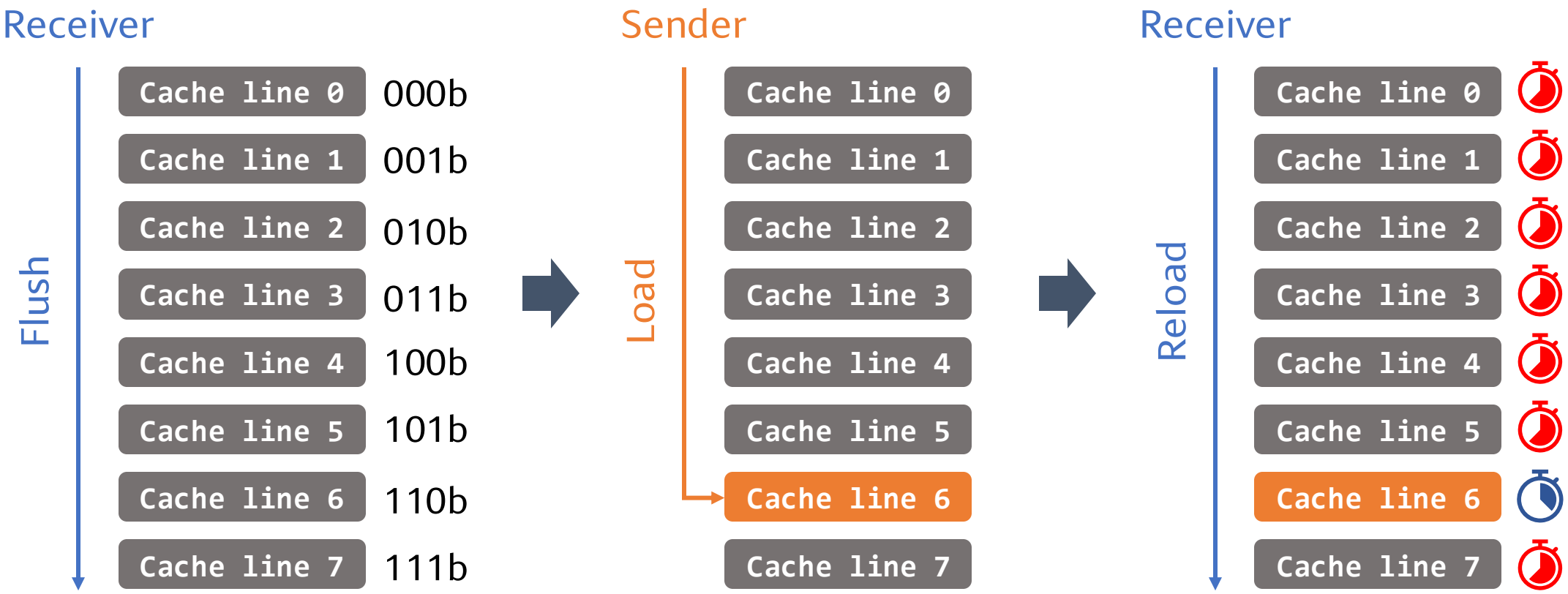
Flush+Reload Covert Channel Timeline





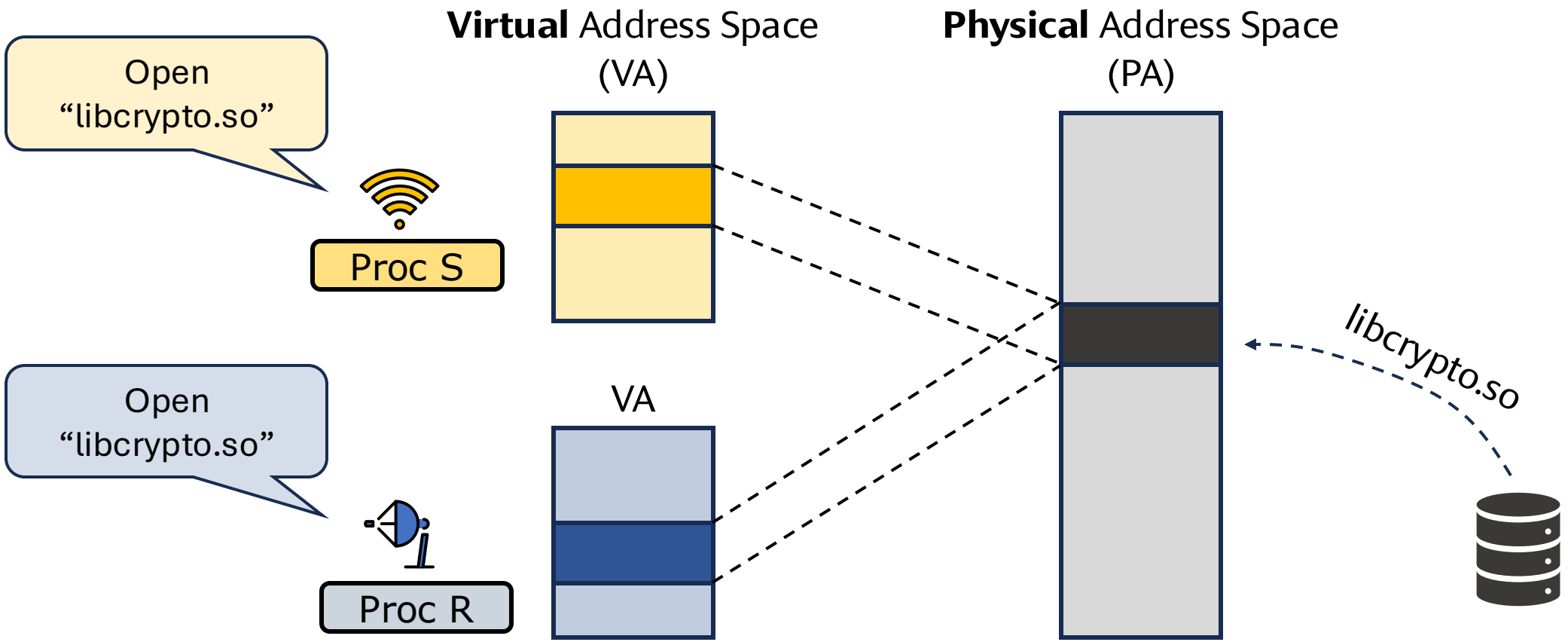
Important! The information is encoded as whether the share cache line is cached
Flush+Reload works regardless of the actual cache line data

Flush+Reload Covert Channel (Multi-Bit)



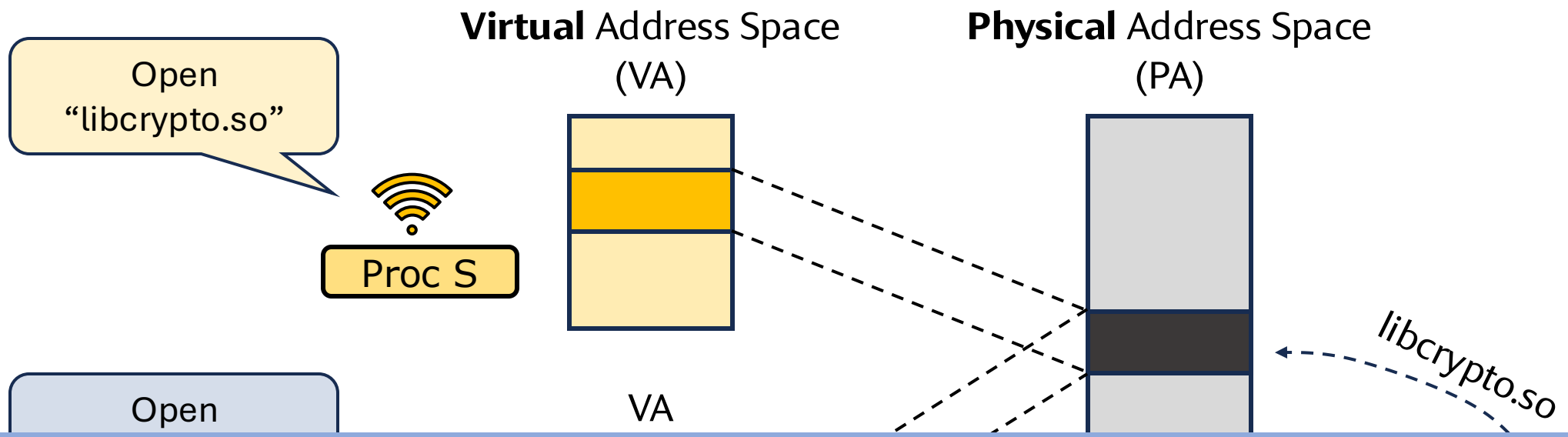
Why do Sender and Receiver Share Memory Pages?

Reason 1: Page cache



Why do Sender and Receiver Share Memory Pages?

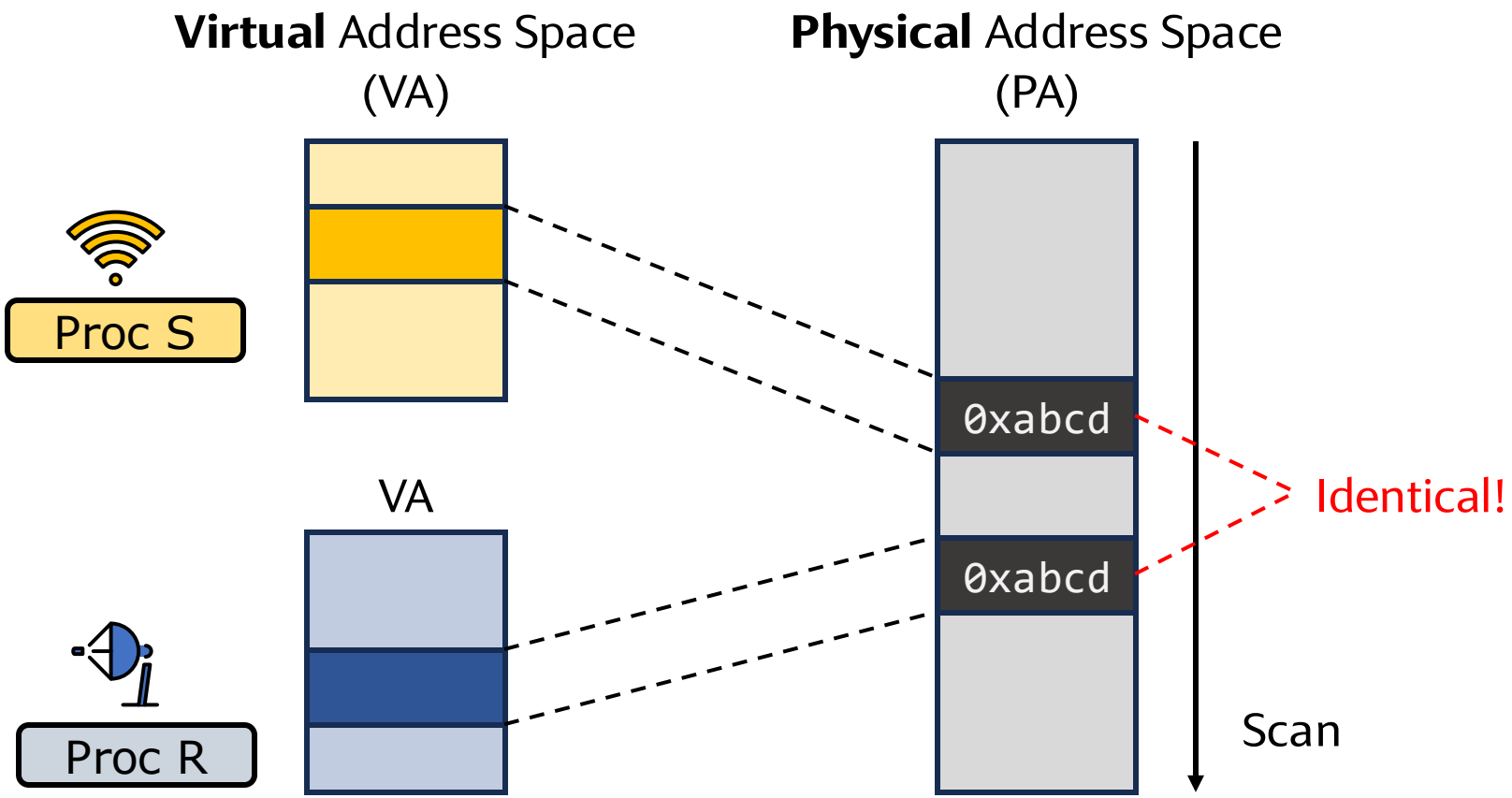
Reason 1: Page cache



Note: Page cache, by itself, creates a side channel
See “Page Cache Attacks” by Gruss et al. (CCS ’19)

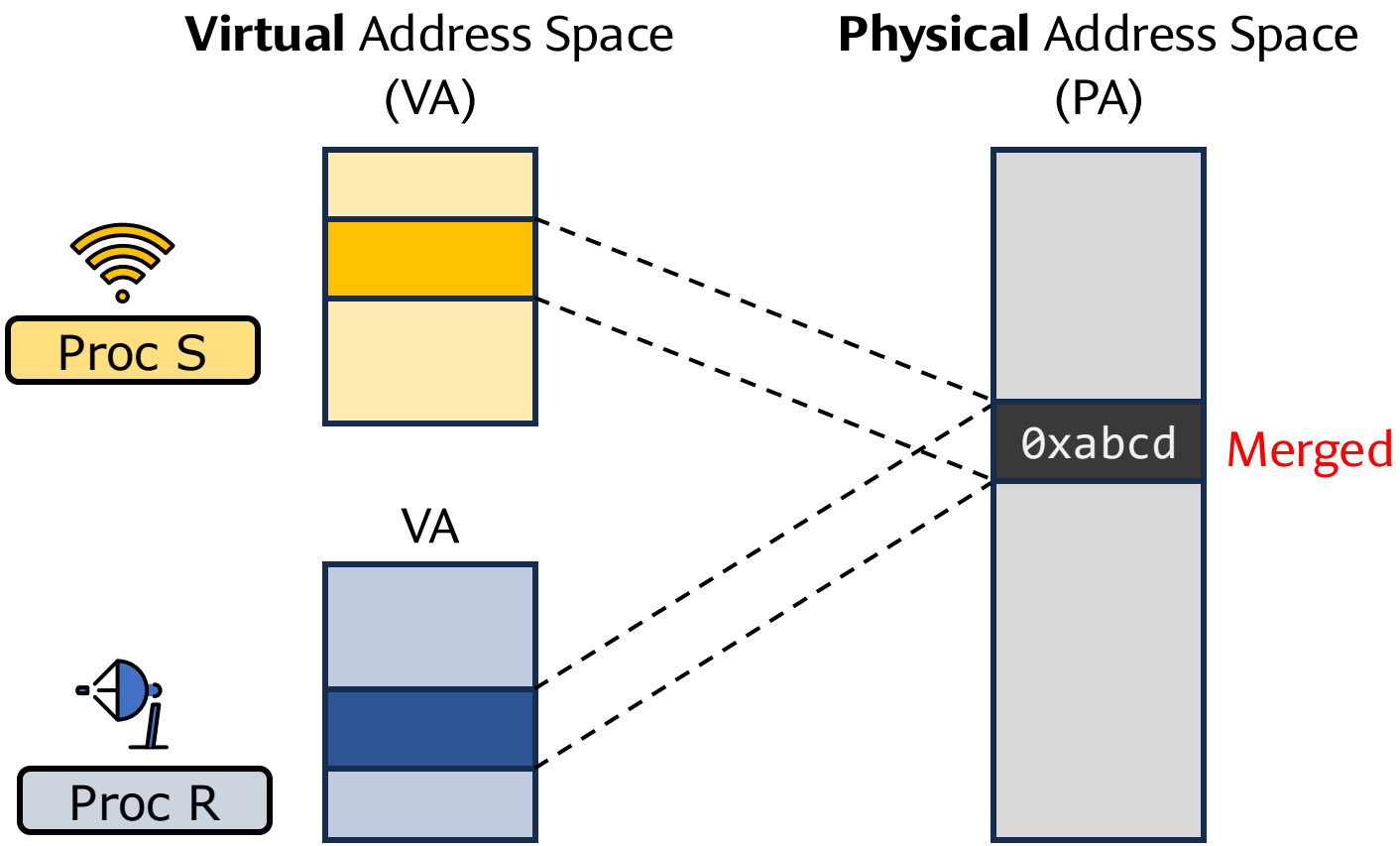
Why do Sender and Receiver Share Memory Pages?

Reason 2: Memory Deduplication



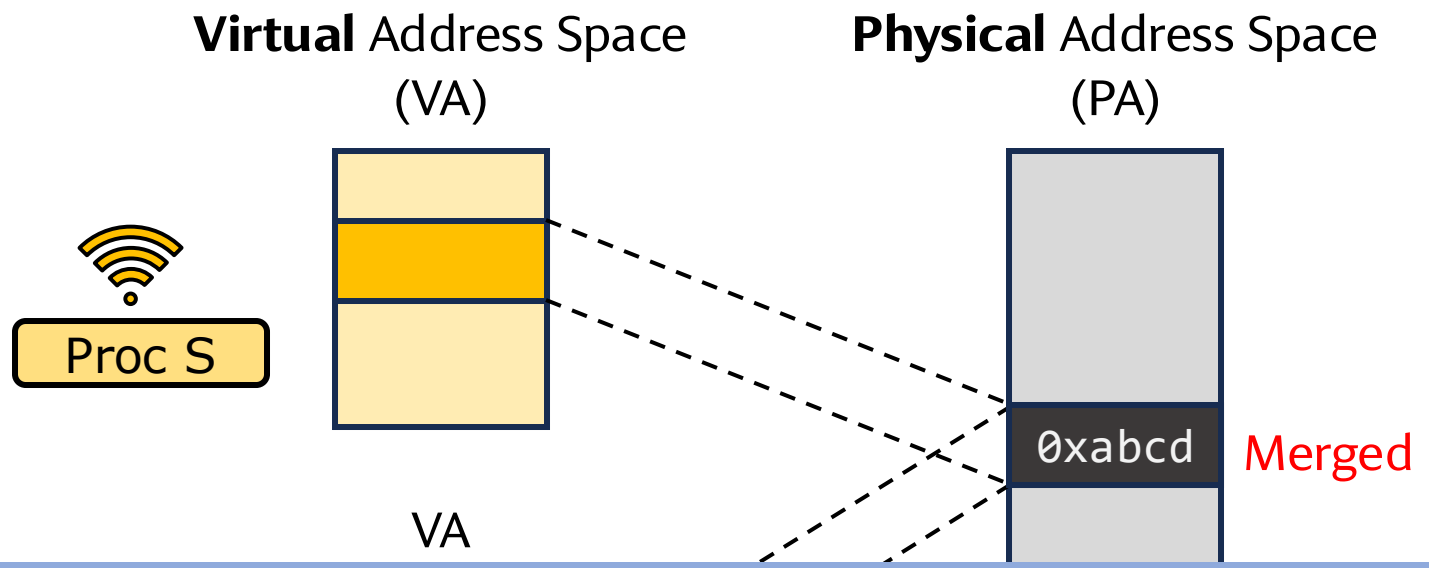
Why do Sender and Receiver Share Memory Pages?

Reason 2: Memory Deduplication



Why do Sender and Receiver Share Memory Pages?

Reason 2: Memory Deduplication



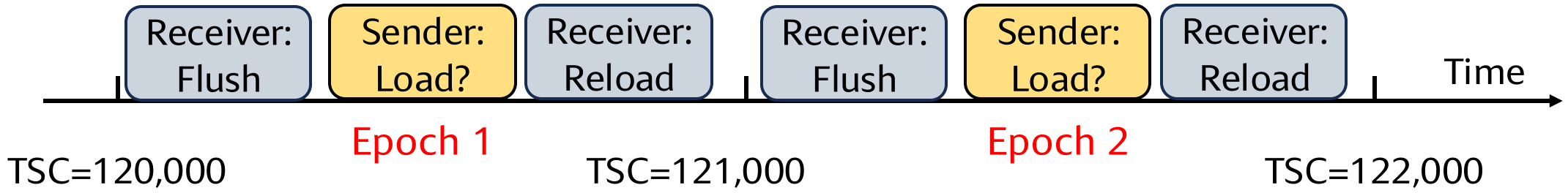
Note: Memory deduplication, by itself, creates a (remote) side channel
See “Remote Memory-Deduplication Attacks” by Schwarzl et al. (NDSS ’22)

Covert Channel Demo

Tools and shortcuts:

- Inline assembly: https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
- High-resolution timer on x86:
  - Timestamp counter (TSC)
  - Synchronized across all cores
  - Accessible via `rdtsc` and `rdtscp`, no root privileged required
  - Useful for measuring latency and synchronization

Sender-Receiver Synchronization Protocol



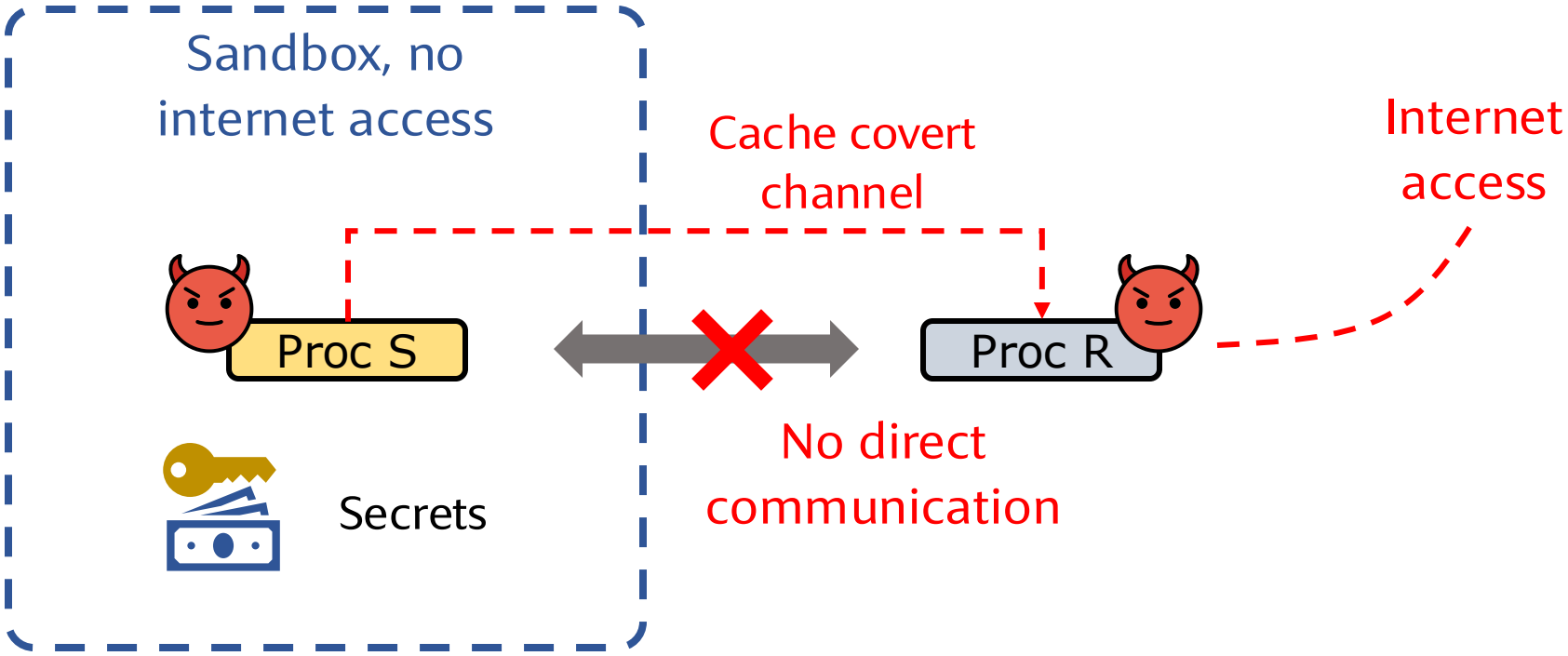
Covert Channel Demo

Try it yourself:

- Code repo: https://github.com/ece382n-sec/Example-PoCs
- ECE-LRC has the perfect machines to try it
  - Access to ECE LRC: https://cloud.wikis.utexas.edu/wiki/spaces/eceit/pages/37752870/ECE+Linux+Application+Servers

- Change existing code and see what happens (and understand why)
- Try to re-implement it based on your own understanding

Why Covert Channels?



From Covert Channel to Side Channel

Square-and-Multiply Exponentiation (Used in RSA)

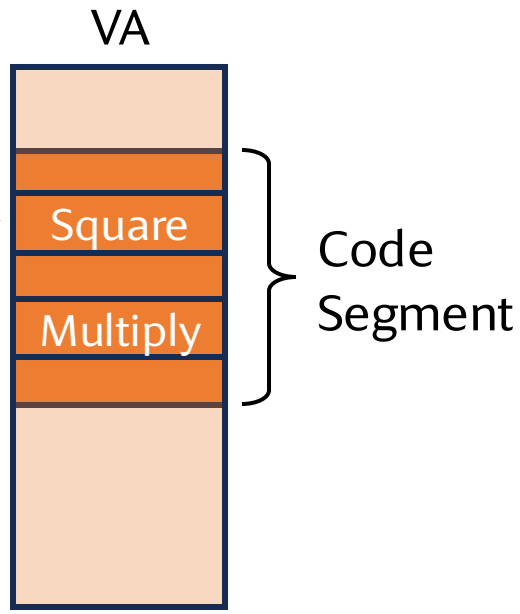
Inputs:

- b: base
- m: modulo
- e: exponent (secret!)
- n: the bit width of e

Output:

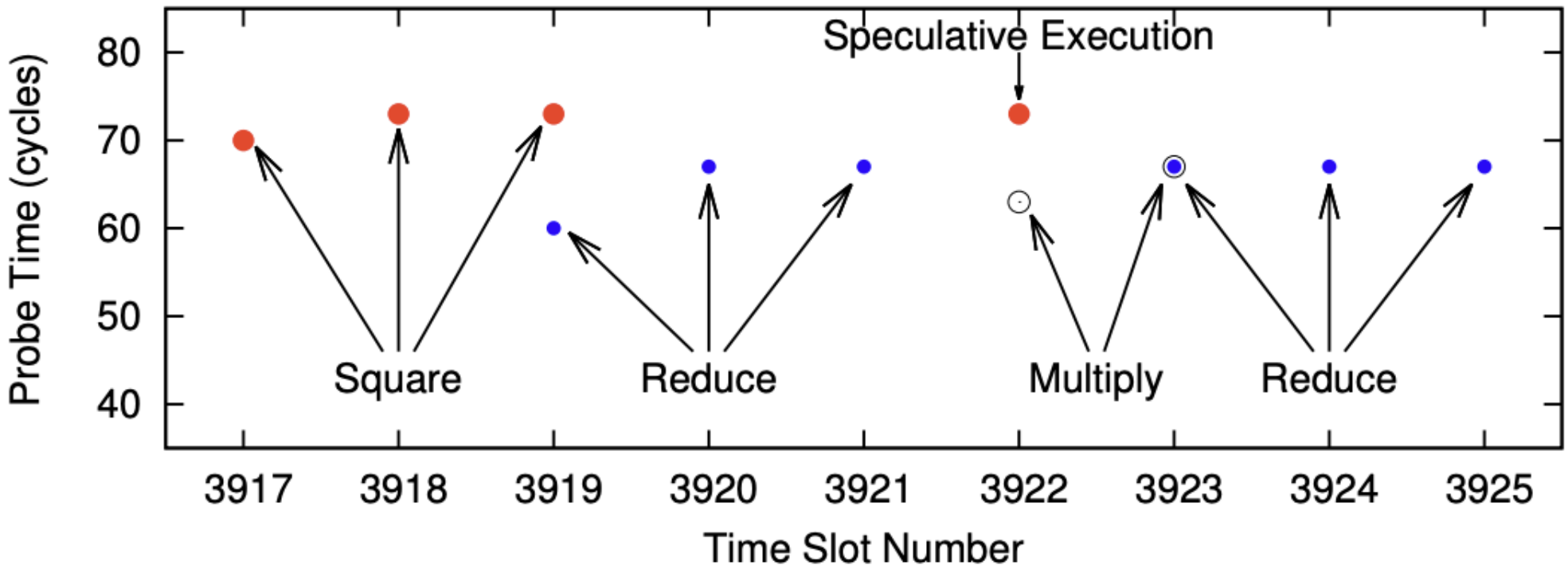
$b^e \bmod m$

```
def expMod(b, m, e, n):
  r = 1
  for i in n-1...0:
    r = square(r, r)
    r = reduce(r, m)
    if get_bit(e, i) == 1:
      r = multiply(r, b)
      r = reduce(r, m)
  return r
```



Flush+Reload monitors which function is called and when

Flush+Reload Trace for Square-and-Multiply (From Yarom et al.)



One time slot = 2500 cycles

What If We Cannot Flush... => Evict+Reload

Why?:

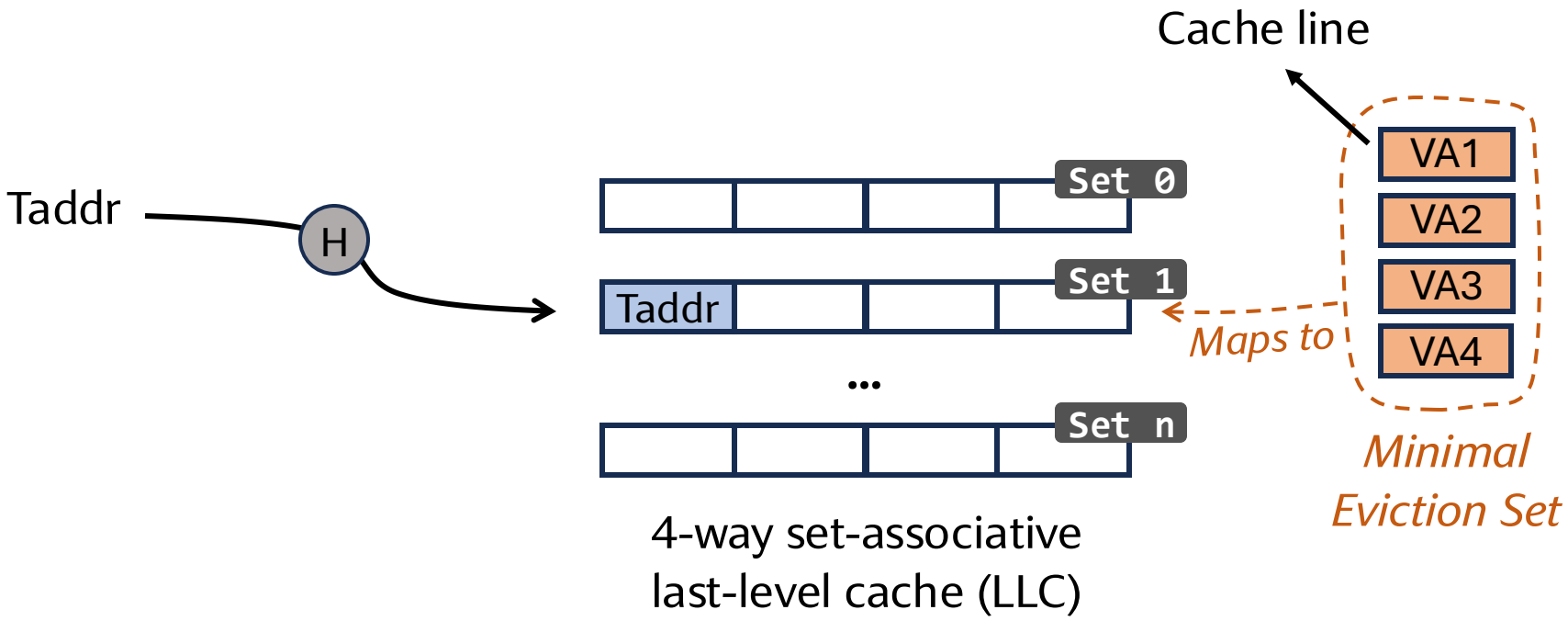
- No CLFLUSH in browser's JavaScript runtime
- Arm's "DC CIVAC <vaddr>" can be disabled in the userspace (EL0)

Workaround: Cache has a limited capacity => evict the target cache line

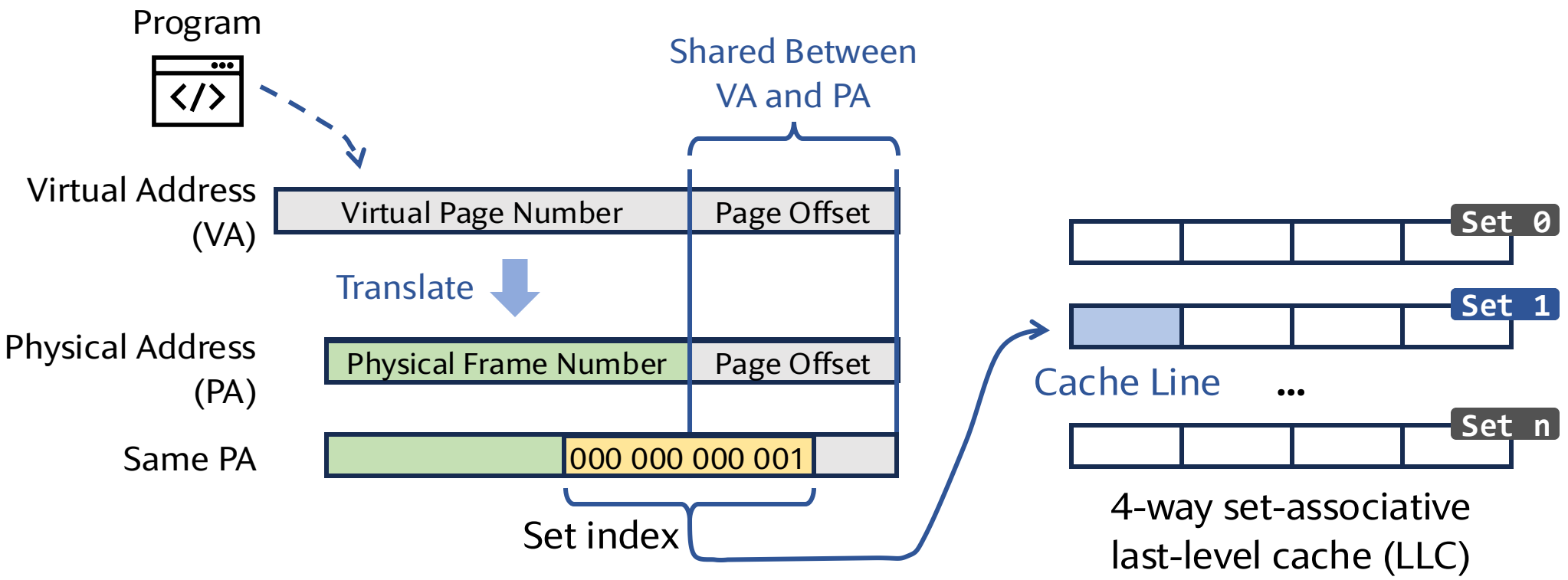
Eviction set: A set of cache lines, once accessed, evict the target cache line

A trivial eviction set: A large memory buffer that thrashes the entire cache

What We Want is a Minimal Eviction Set

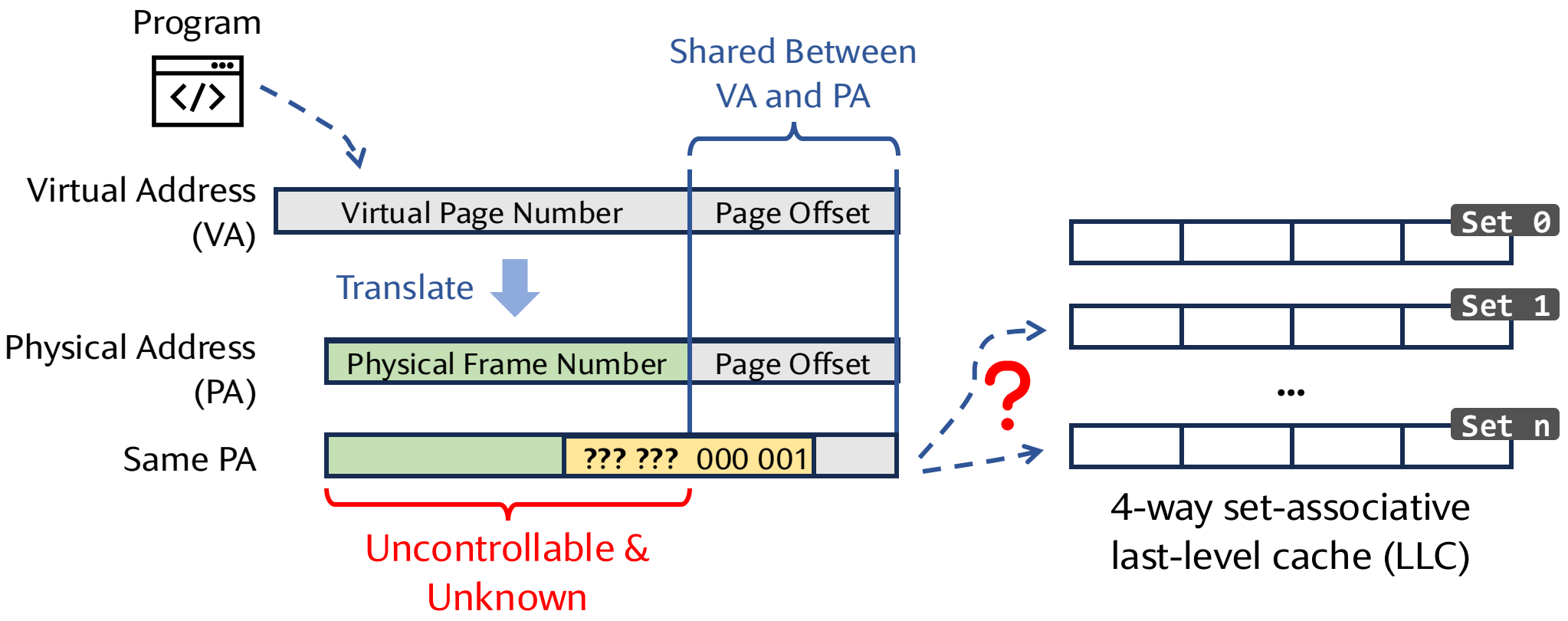


Mapping Addresses to Cache Sets

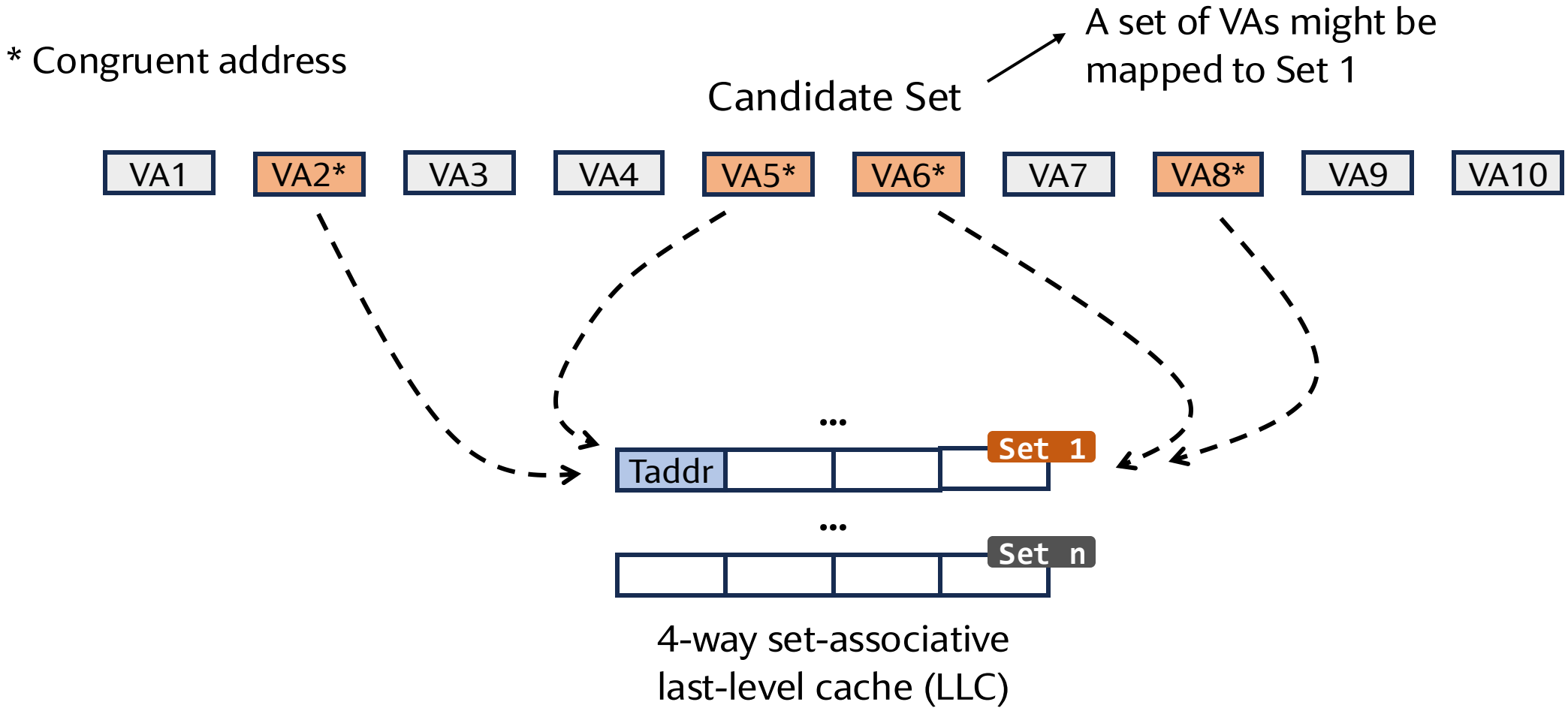


Mapping Addresses to Cache Sets

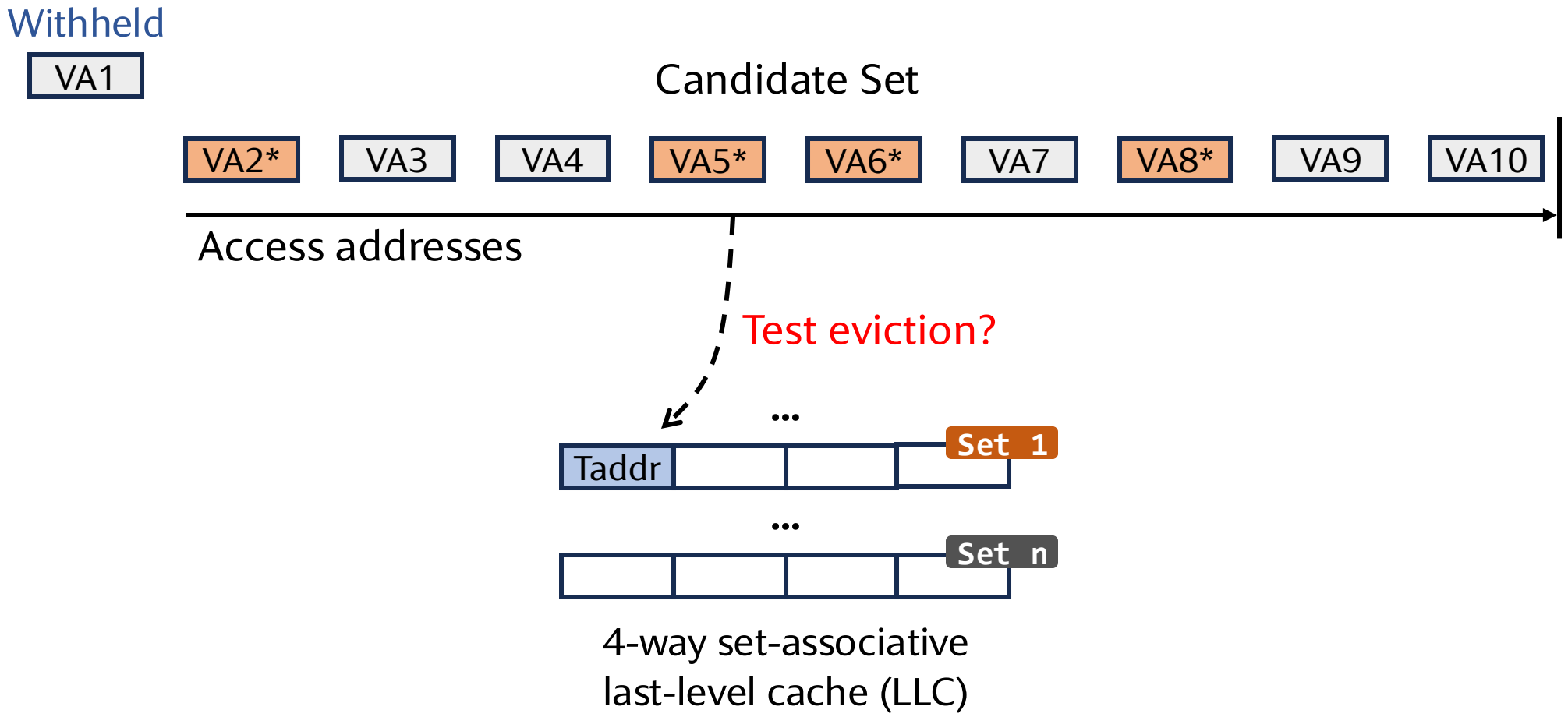
Attacker cannot force a VA to be mapped to a specific cache set



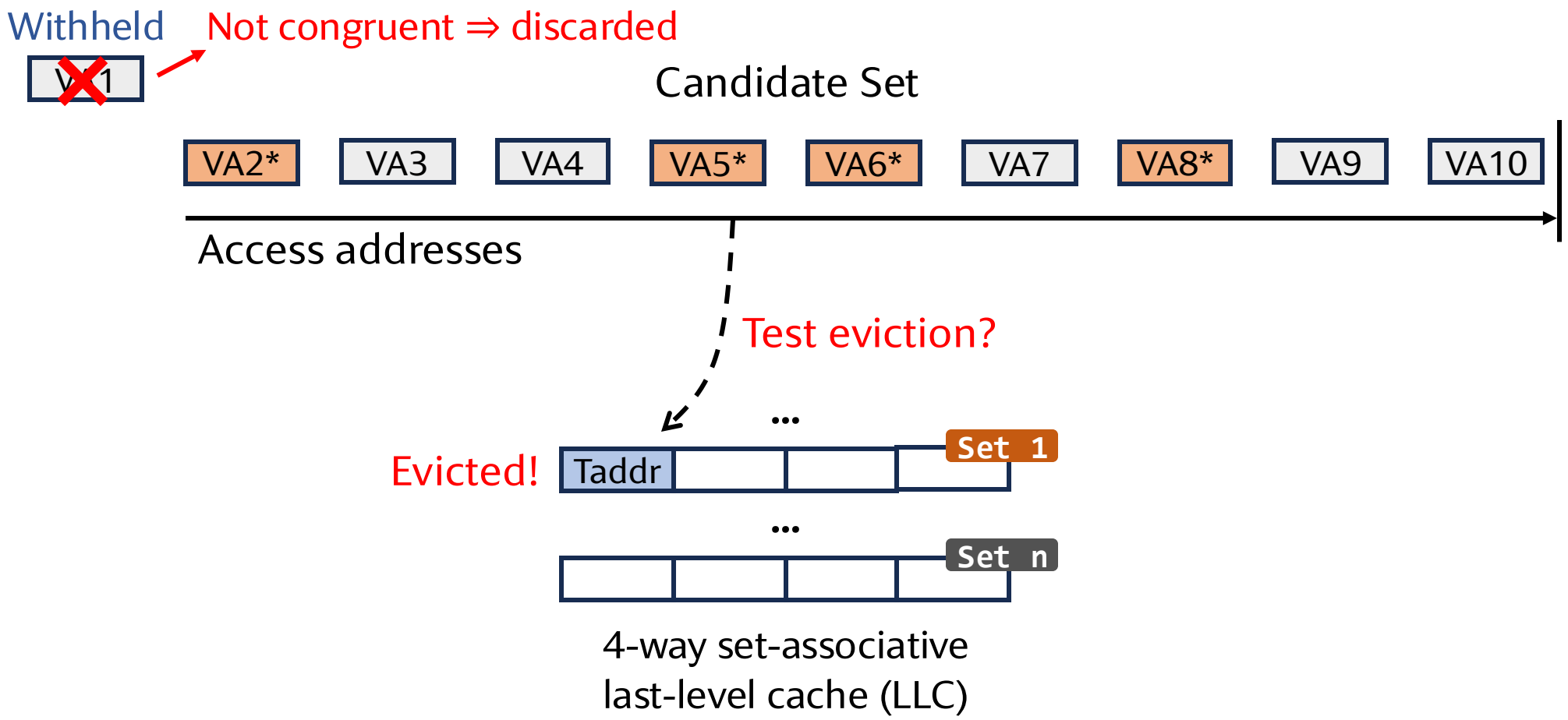
Constructing an Eviction Set



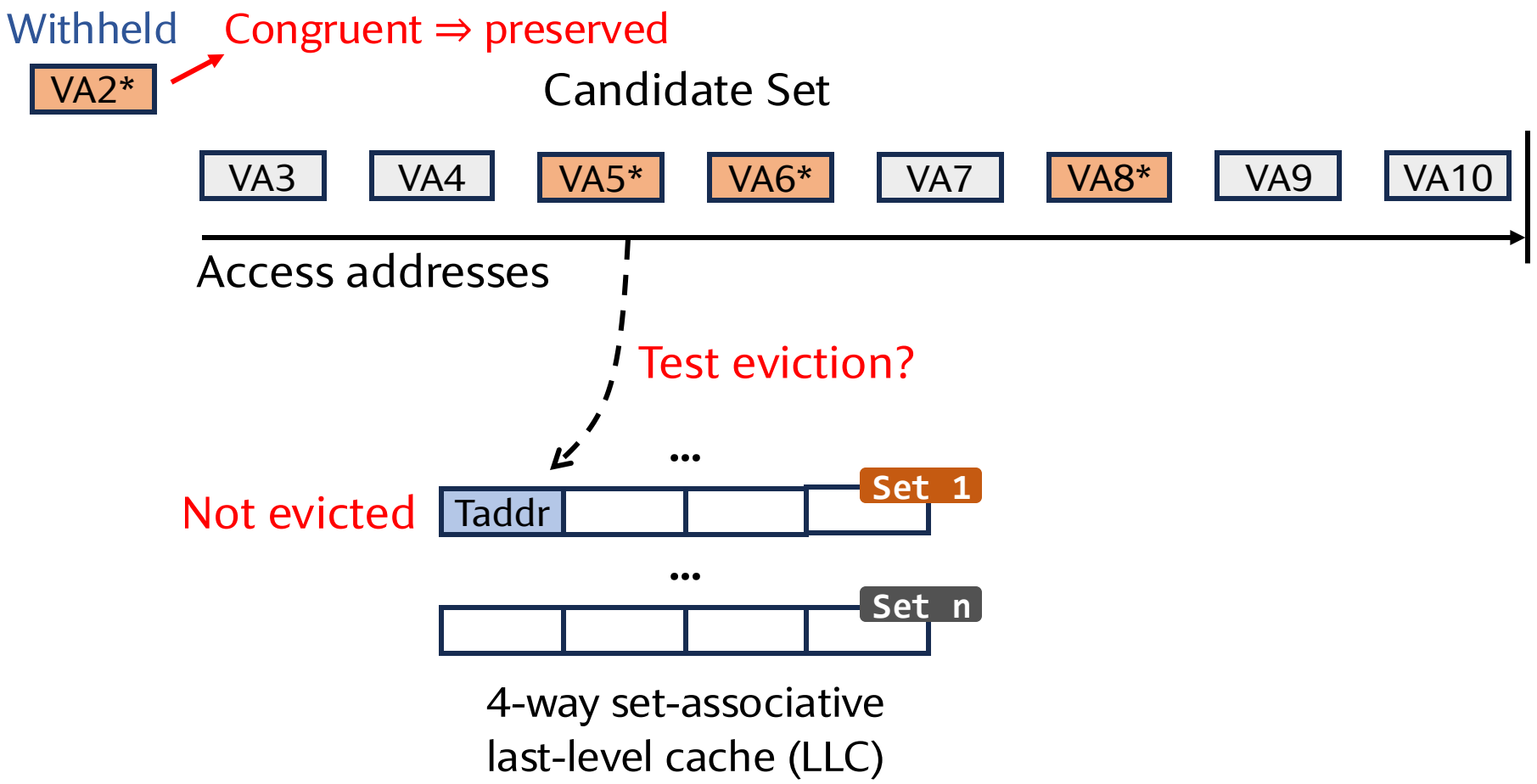
Constructing an Eviction Set



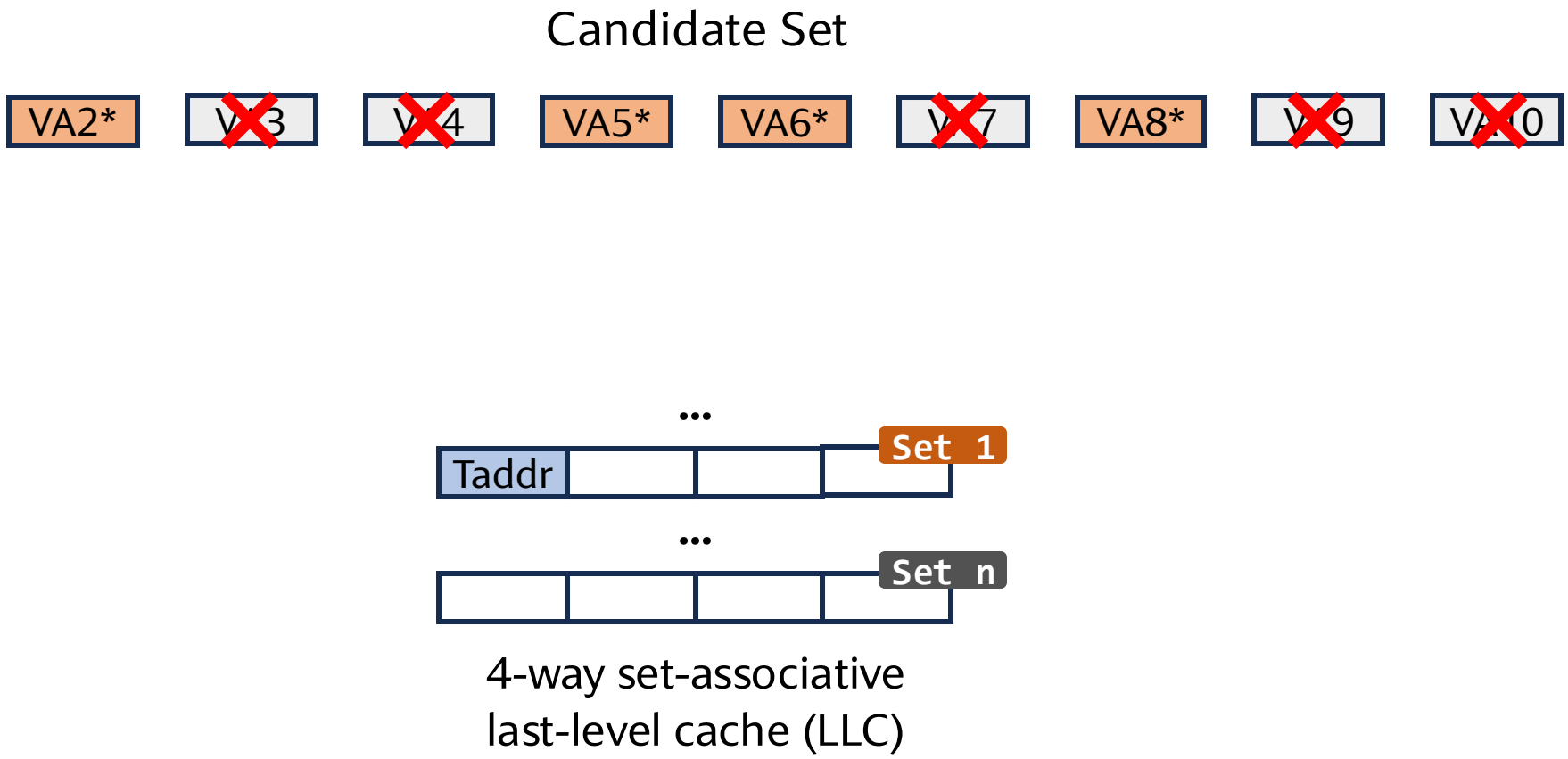
Constructing an Eviction Set



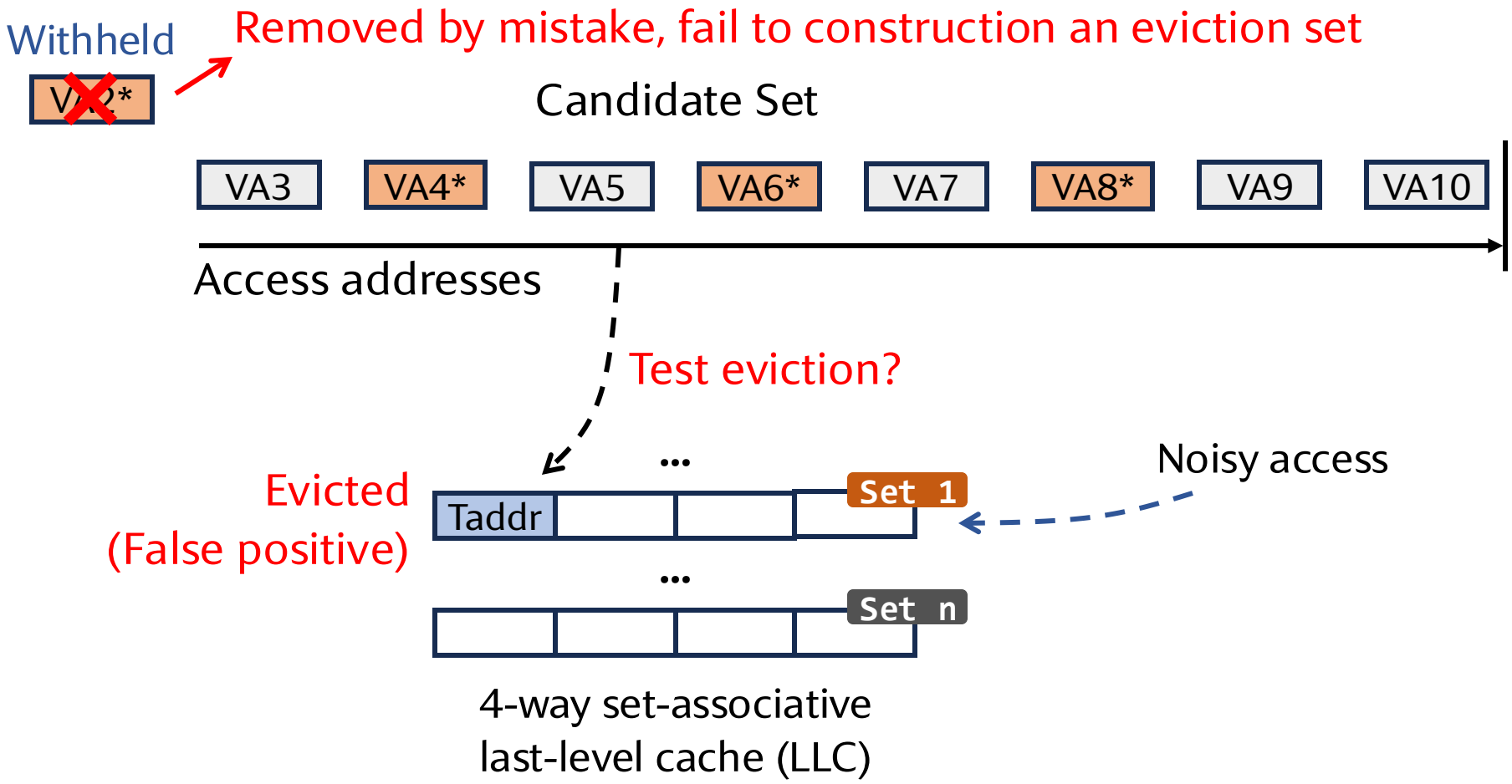
Constructing an Eviction Set



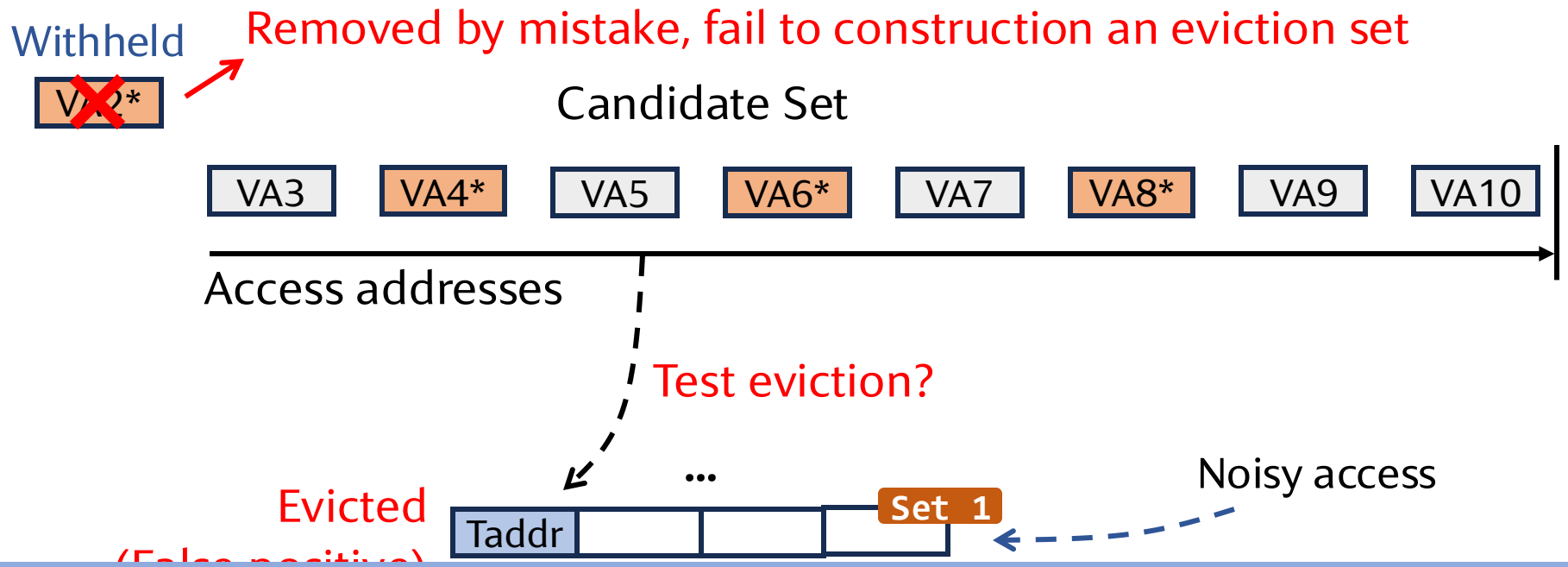
Constructing an Eviction Set



Test Eviction is Susceptible to Noise

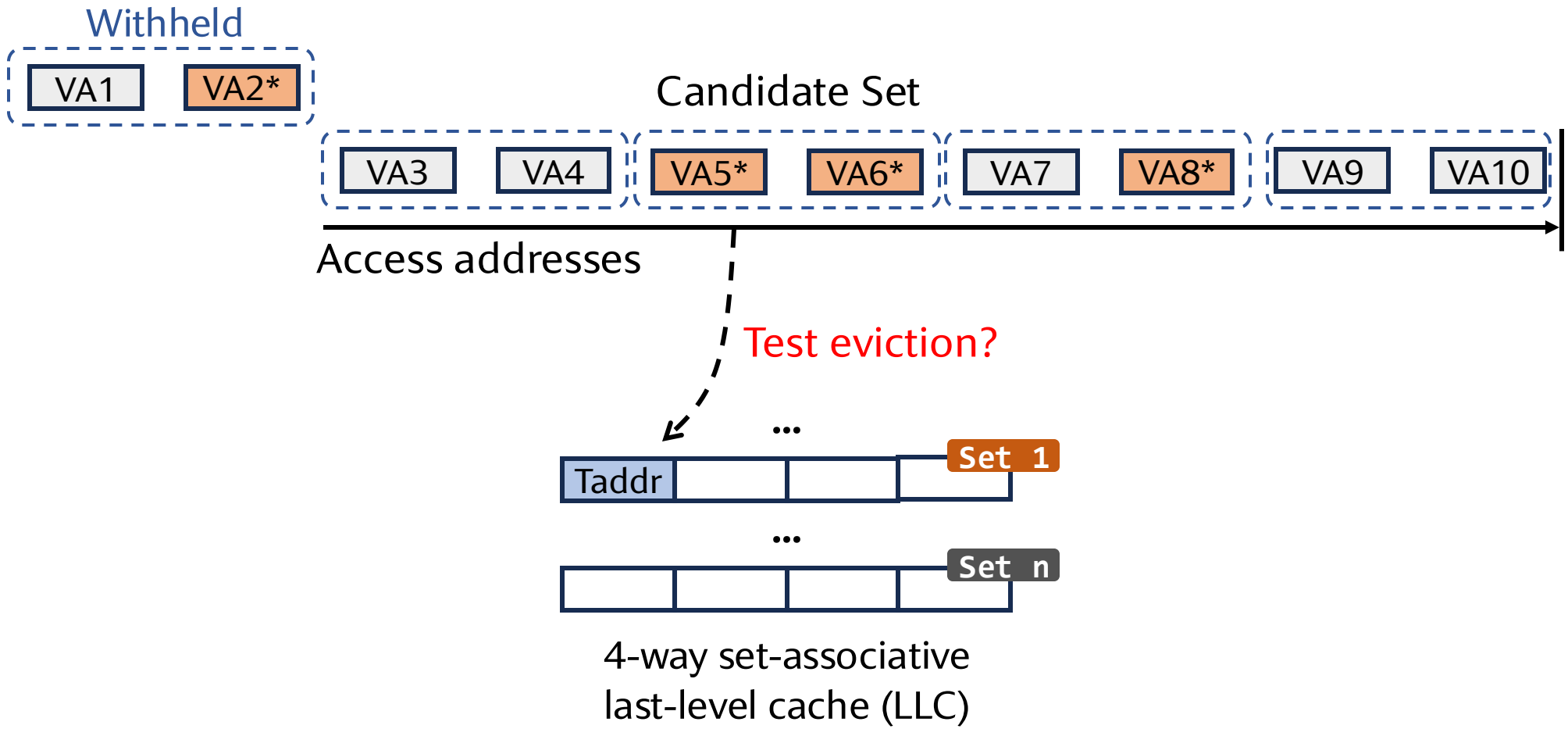


Test Eviction is Susceptible to Noise

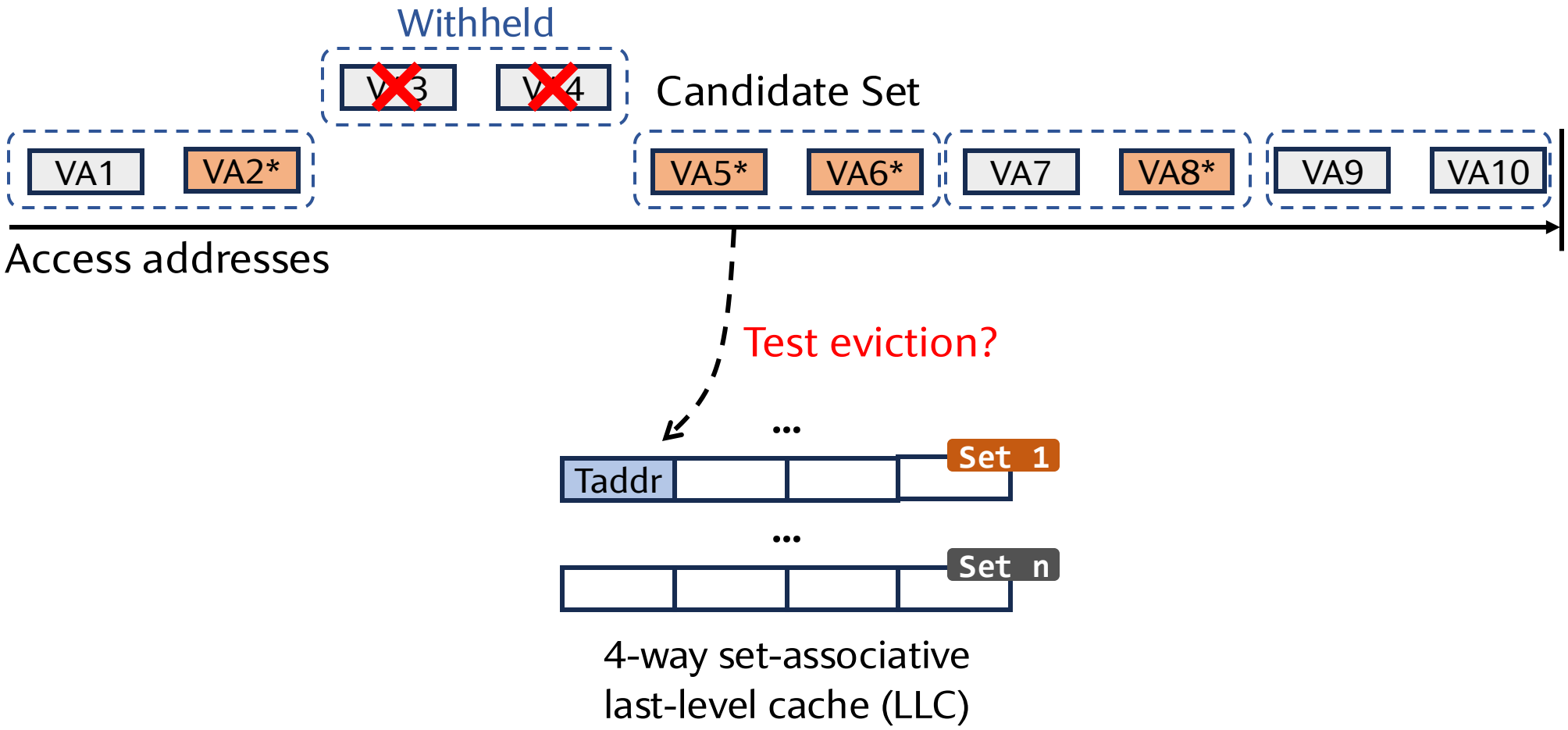


Note: Developing eviction set construction algorithms that are resilient to noise is a very good term project direction

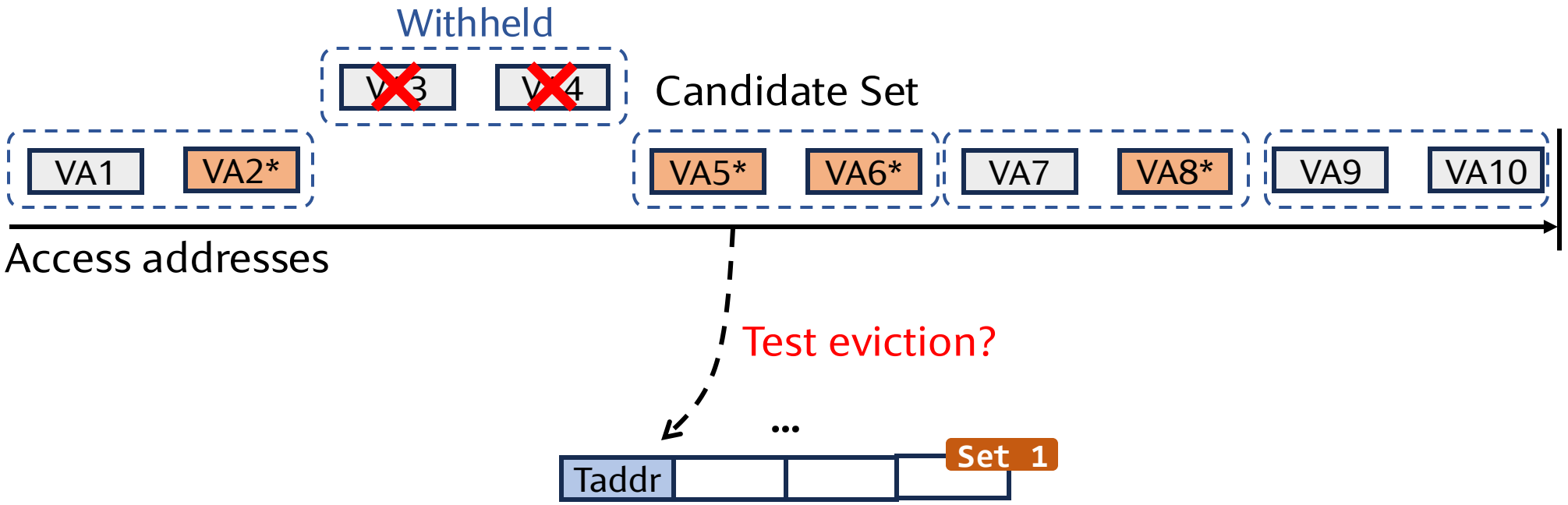
A More Efficient Algorithm of Constructing an Eviction Set



A More Efficient Algorithm of Constructing an Eviction Set

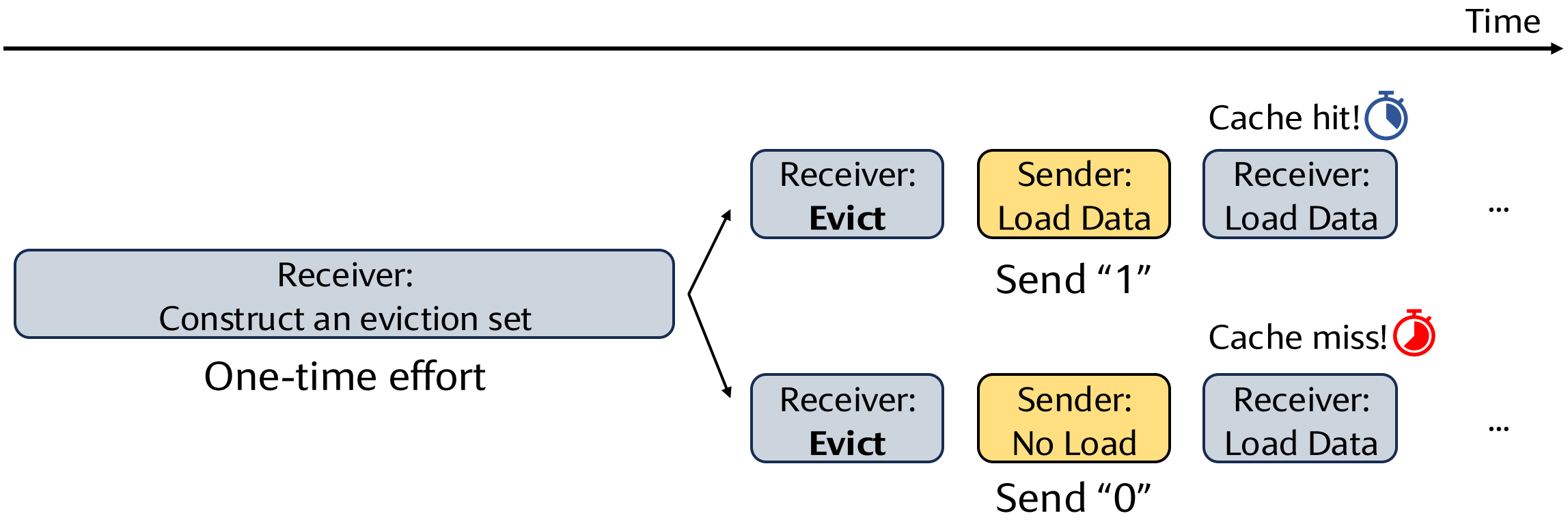


A More Efficient Algorithm of Constructing an Eviction Set



Note: Developing algorithms that can efficiently construct eviction sets, especially in a noisy environment, has been a popular research topic (and a great term project)

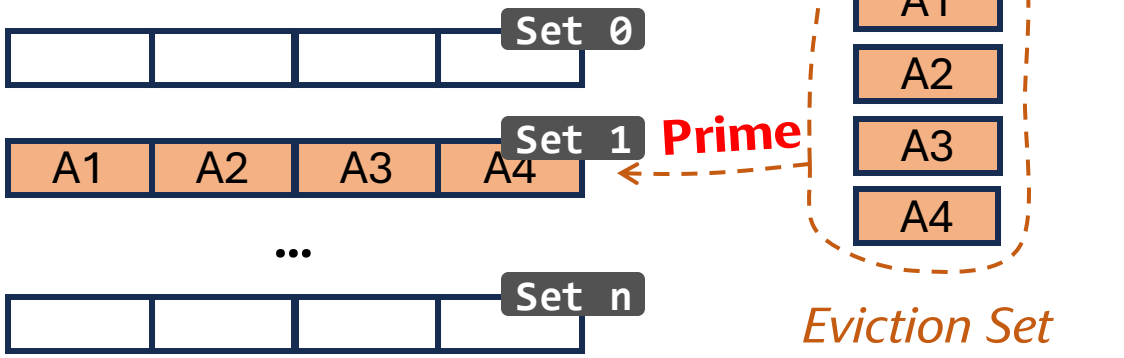
Evict+Reload Timeline



Prime+Probe: Monitor Memory Accesses to a Cache Set

Victim/Sender

Attacker/Receiver

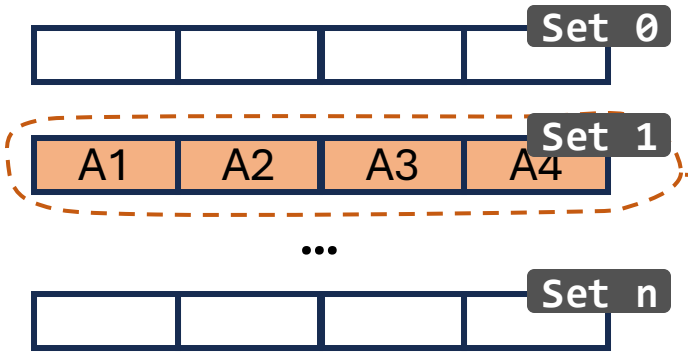


4-way set-associative
last-level cache (LLC)

Prime+Probe: Monitor Memory Accesses to a Cache Set

Victim/Sender

Attacker/Receiver

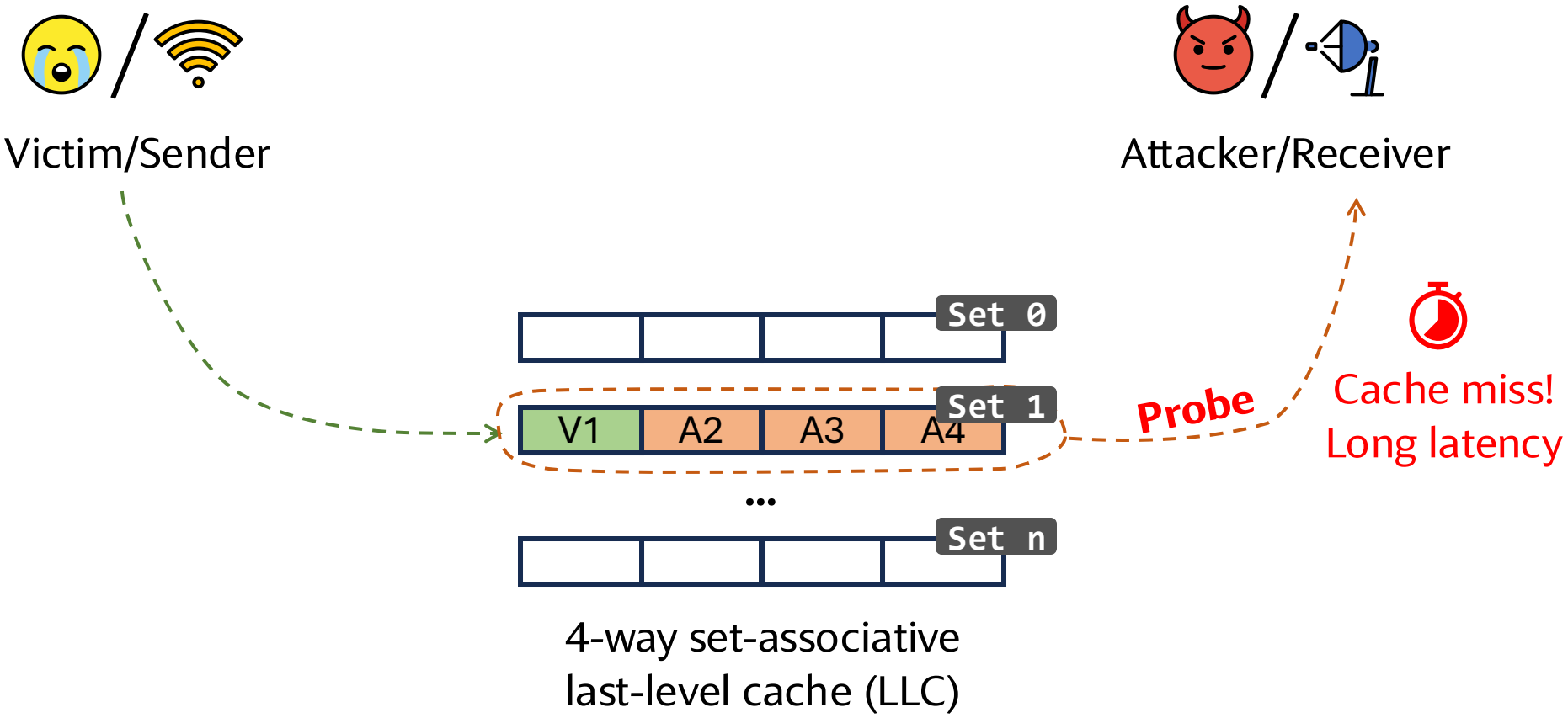


Probe

Cache hit!
Low latency

4-way set-associative
last-level cache (LLC)

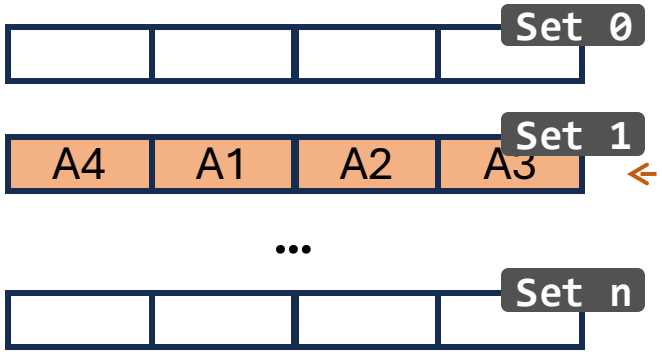
Prime+Probe: Monitor Memory Accesses to a Cache Set



Prime+Probe: Monitor Memory Accesses to a Cache Set

Victim/Sender

Attacker/Receiver



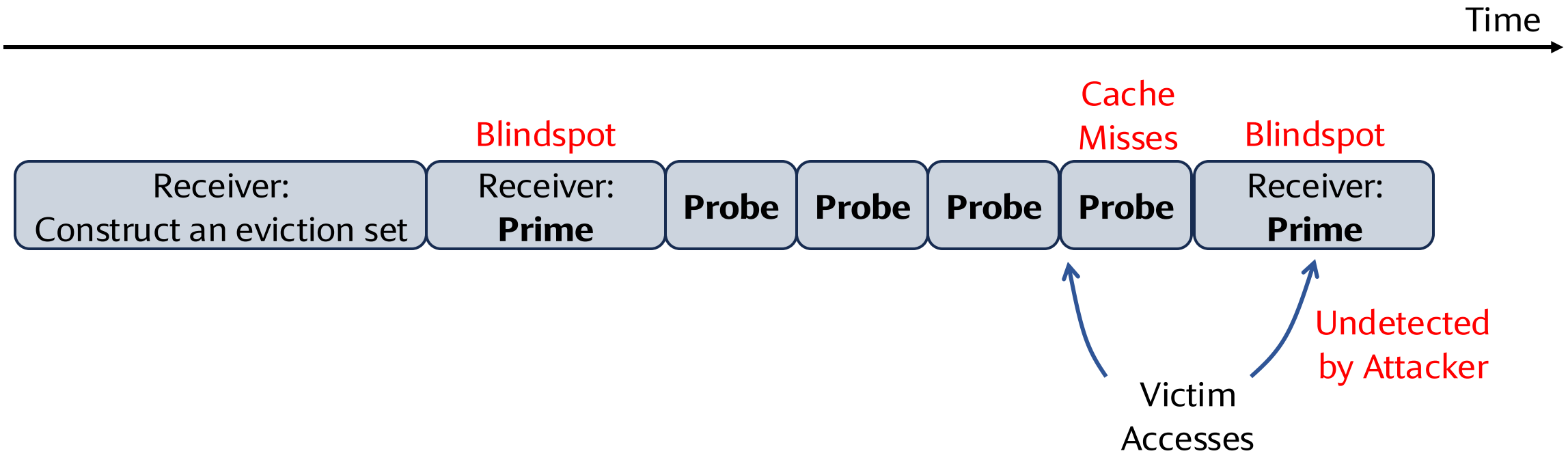
4-way set-associative
last-level cache (LLC)

Prime

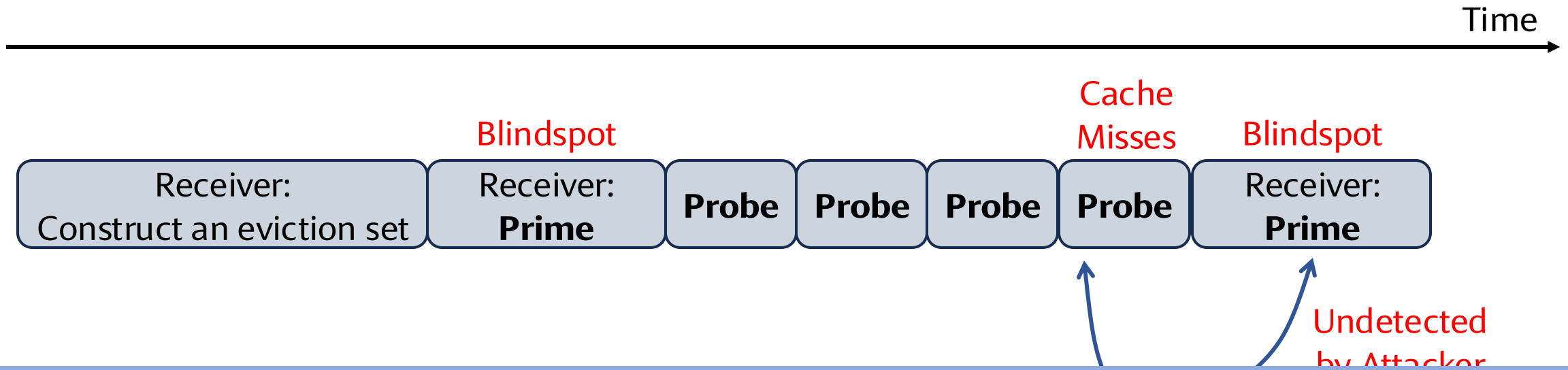
Blind spot!

Cannot detect any access occurs during priming

Prime+Probe Timeline



Prime+Probe Timeline

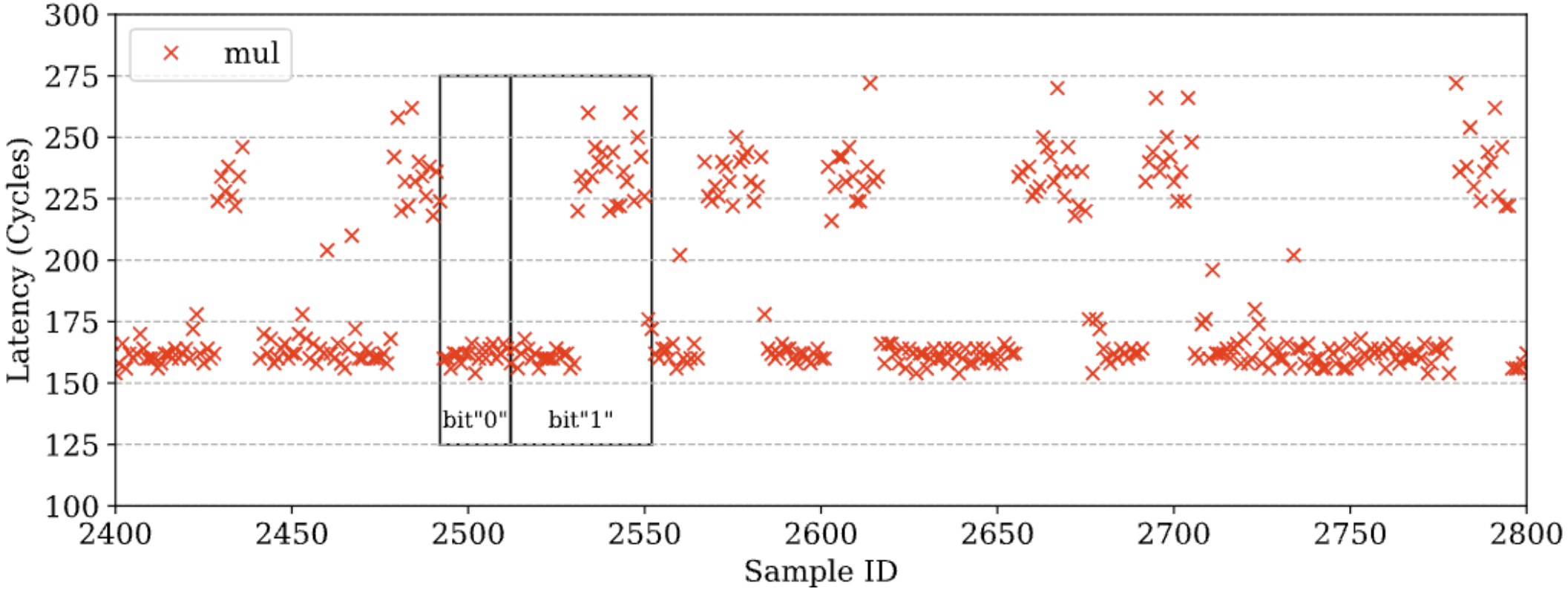


Note: Developing fast prime and probe operations is also a popular research topic (and a great term project)

Compared to Flush+Reload/Evict+Reload

- **Different leakage mechanism**
  - Flush+Reload (F+R)/Evict+Reload (E+R) => Cache reuse
  - Prime+Probe (P+P) => Cache contention
- **Pro:** P+P does not need shared pages between the attack and victim
  - This is very important because cloud vendors generally prohibit different tenants from sharing pages
  - See: "The Security Design of the AWS Nitro System" from AWS, chapter "The EC2 approach to preventing side-channels"
- **Con:** P+P is more coarse grained
  - If the victim accesses two different cache lines mapped to the same cache set
    - P+P cannot distinguish which line is accessed
    - But F+R and E+R can

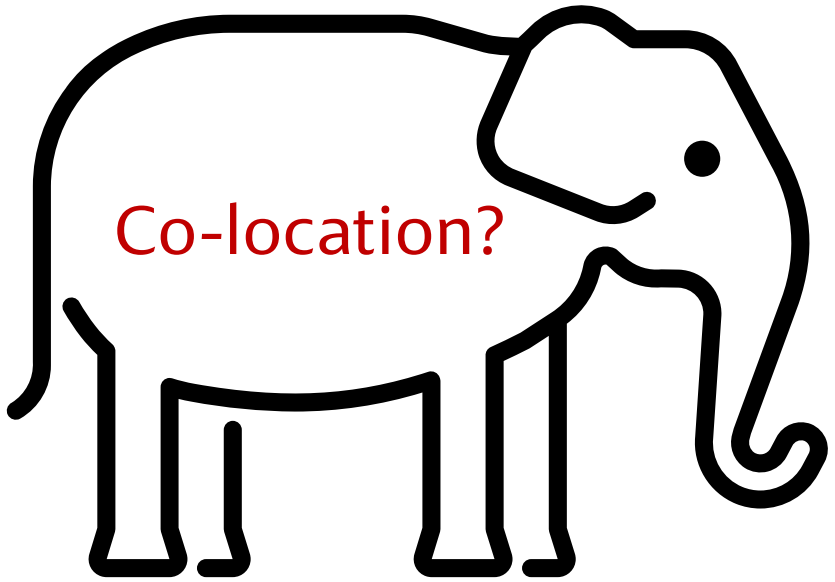
Prime+Probe Trace for Square-and-Multiply (From Yan et al.[1])



Victim executes “multiply” => Longer probe latency

[1]Yan et al. “Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World”

Next Lecture: Side Channels in Public Clouds



A Few Announcements and Reminders

- Our first paper review due this Sunday
  - Review guidelines and examples posted on Canvas
- Please form groups for the term project (due today)
- Please sign up for the presentation slots (due this Sunday)
  - **Signing up for the early-bird slots (i.e., the four Sep slots) gives you 5 bonus points**