ECE 382N-Sec (FA25):

L3: Partitioning, Randomization, and Detection

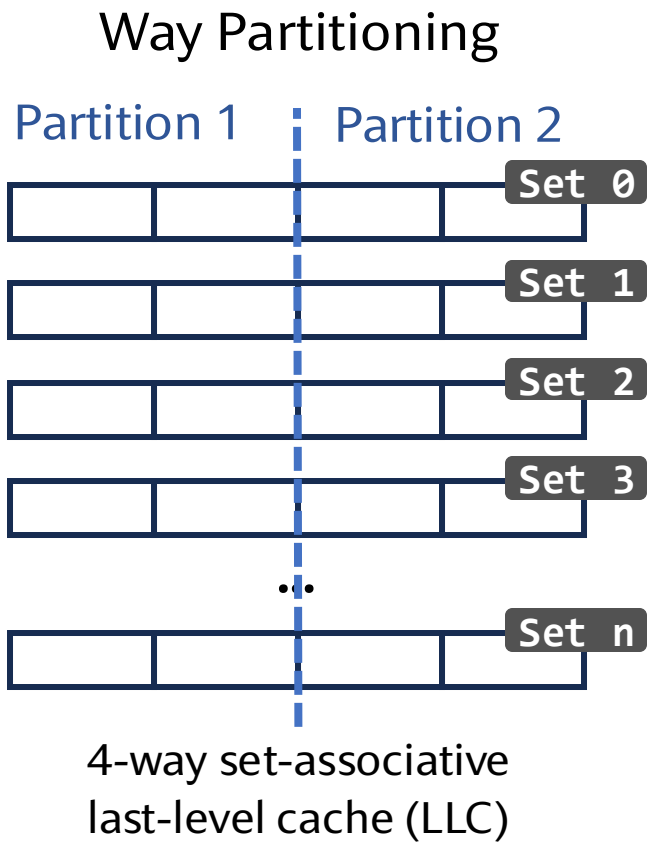
Neil Zhao

neil.zhao@utexas.edu

This Lecture

- Resource Partitioning => Limit the sharing
- Randomization => Obfuscate the resource usage
- Detection => Catch the offender

Let's Start With (Last-Level) Cache Partitioning



(Relatively) simple to implement

Not scalable

Let's Start With (Last-Level) Cache Partitioning

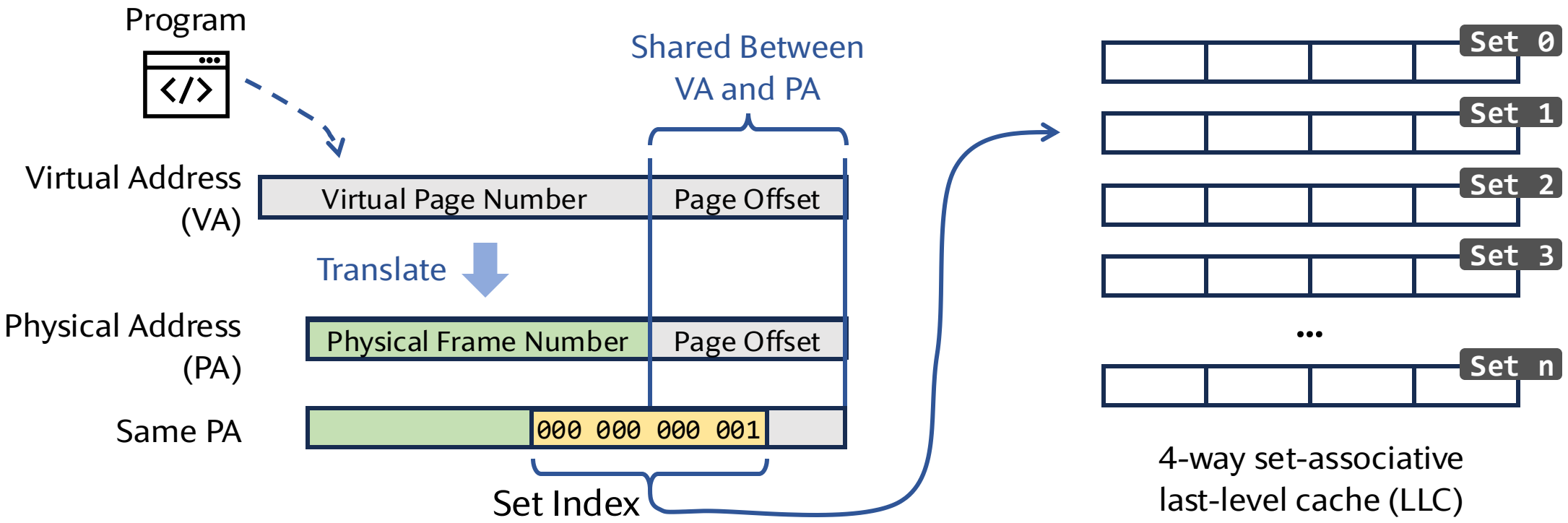
Set Partitioning

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

Scalable

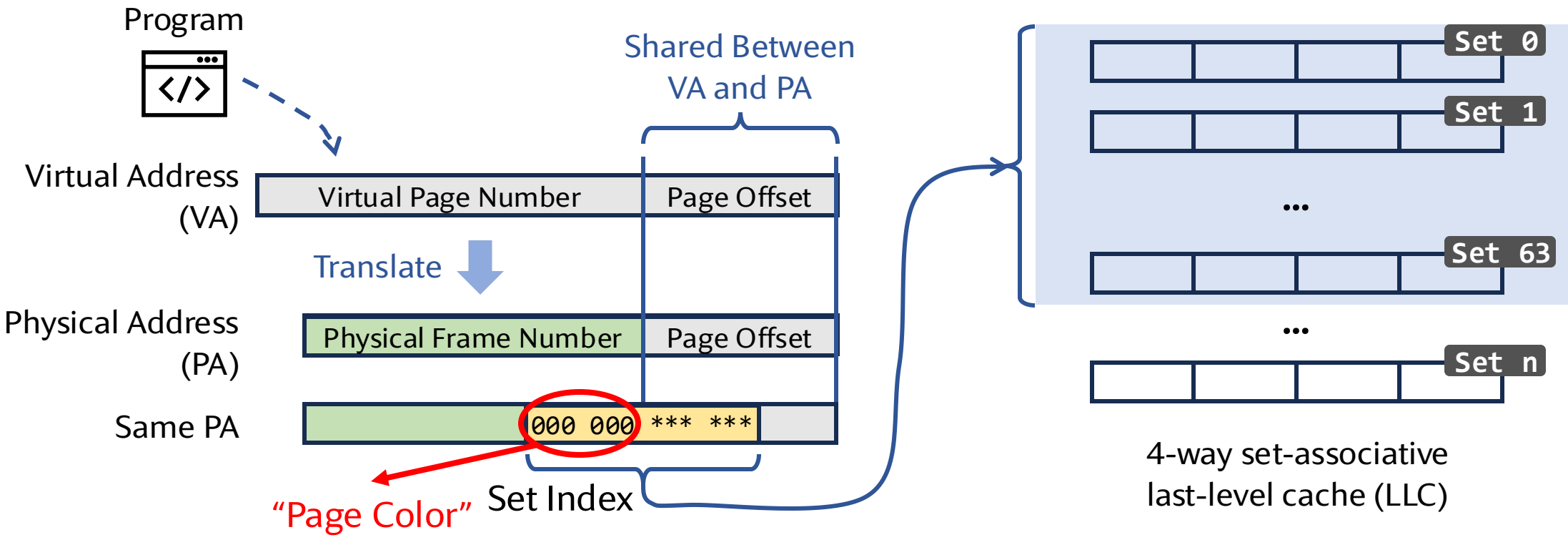
More complex design, especially if # sets a partition owns is not a power of two

Page Coloring: Software Implementation of Set Partitioning



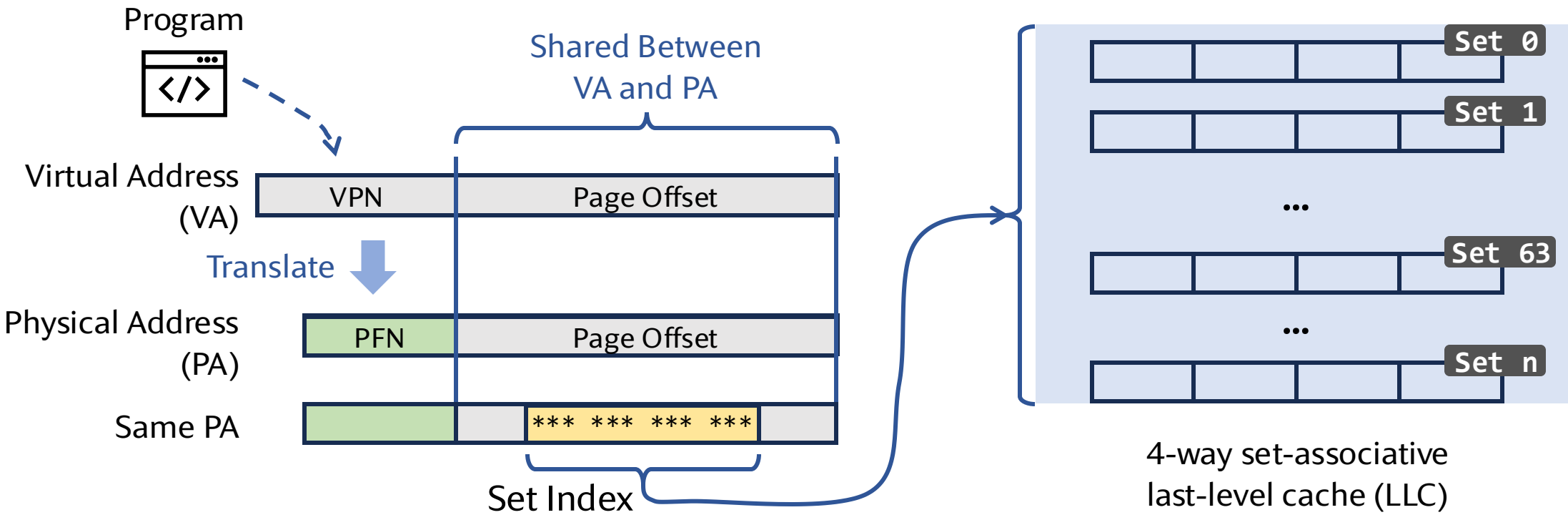
Page Coloring: Software Implementation of Set Partitioning

Cache lines from pages with different colors are mapped to different sets

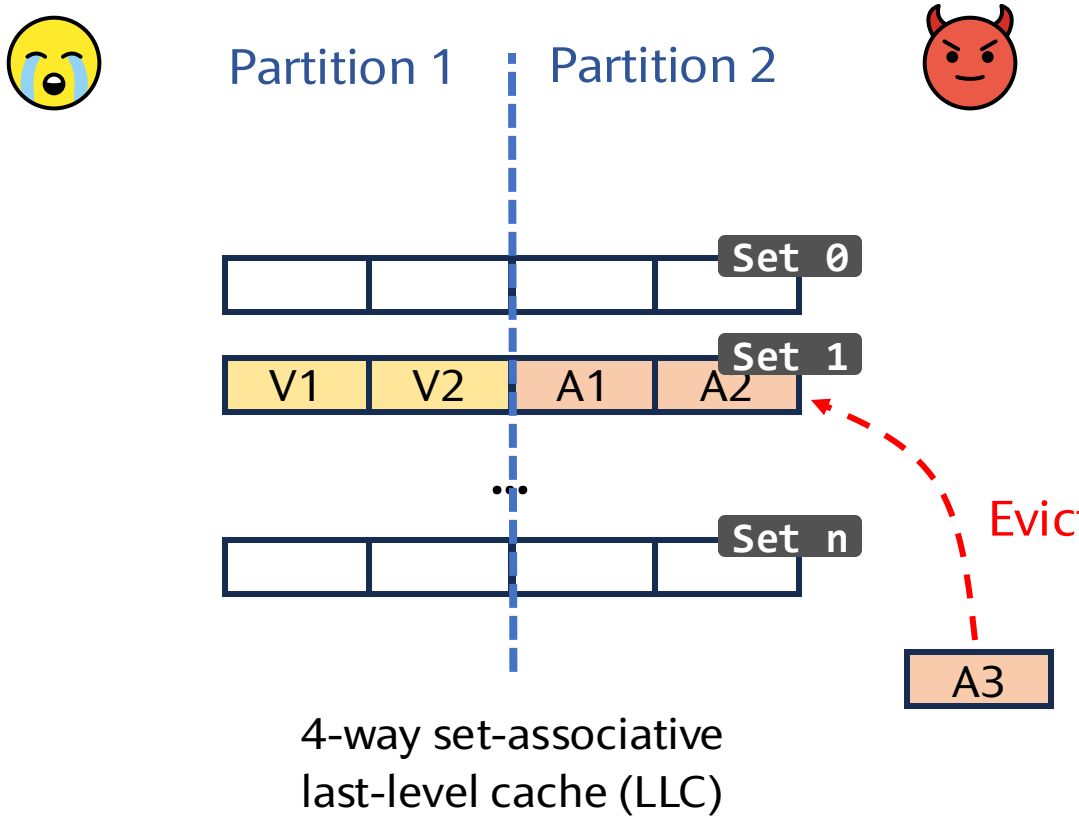


Limitation of Page Coloring

Cannot support huge pages (e.g., 2MB pages)

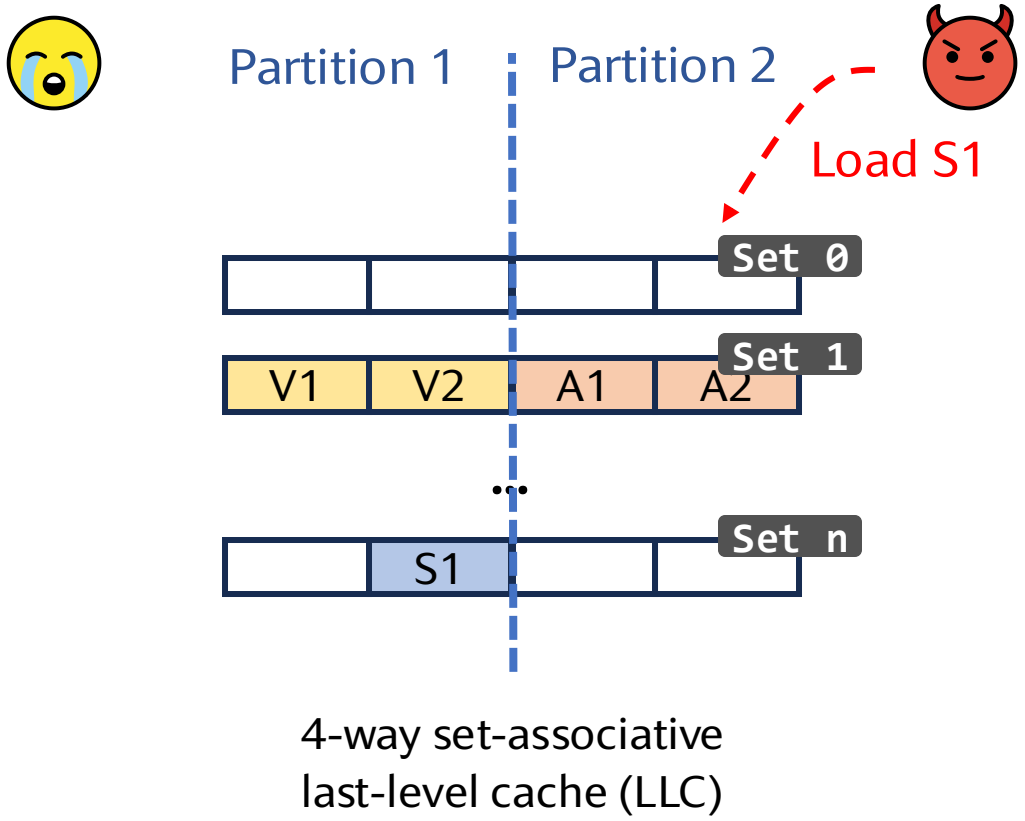


Hardware-Based Way Partitioning



Commercial implementation:
Intel Cache Allocation Technology (CAT)
Disclaimer: not built for security but for the Quality of Service (QoS)

What about Shared Memory?



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

S1 is shared between these two parties

Option 1: Duplicate the shared line in across the partitions

- 👍 Immune to Flush+Reload, Eivict+Reload, ...
- 👎 Waste cache space
- 👎 Hard to maintain cache coherence across the partitions (if the line is writable)

Option 2: Single copy, allow cross-partition hits

- 👍 Easy to implement
- 👎 Vulnerable to Flush+Reload, Evict+Reload, ...

Intel CAT went with the second option

CATalyst: A Very Clever Use of Intel CAT

CATalyst: Defeating Last-Level Cache Side Channel Attacks in Cloud Computing

LLC Prime+Probe Practical

Flush+Reload and many more

Fangfei Liu[1], Qian Ge[2,3], Yuval Yarom[2,4],
Frank Mckeen[5], Carlos Rozas[5], Gernot Heiser[2,3], Ruby B. Lee[1]

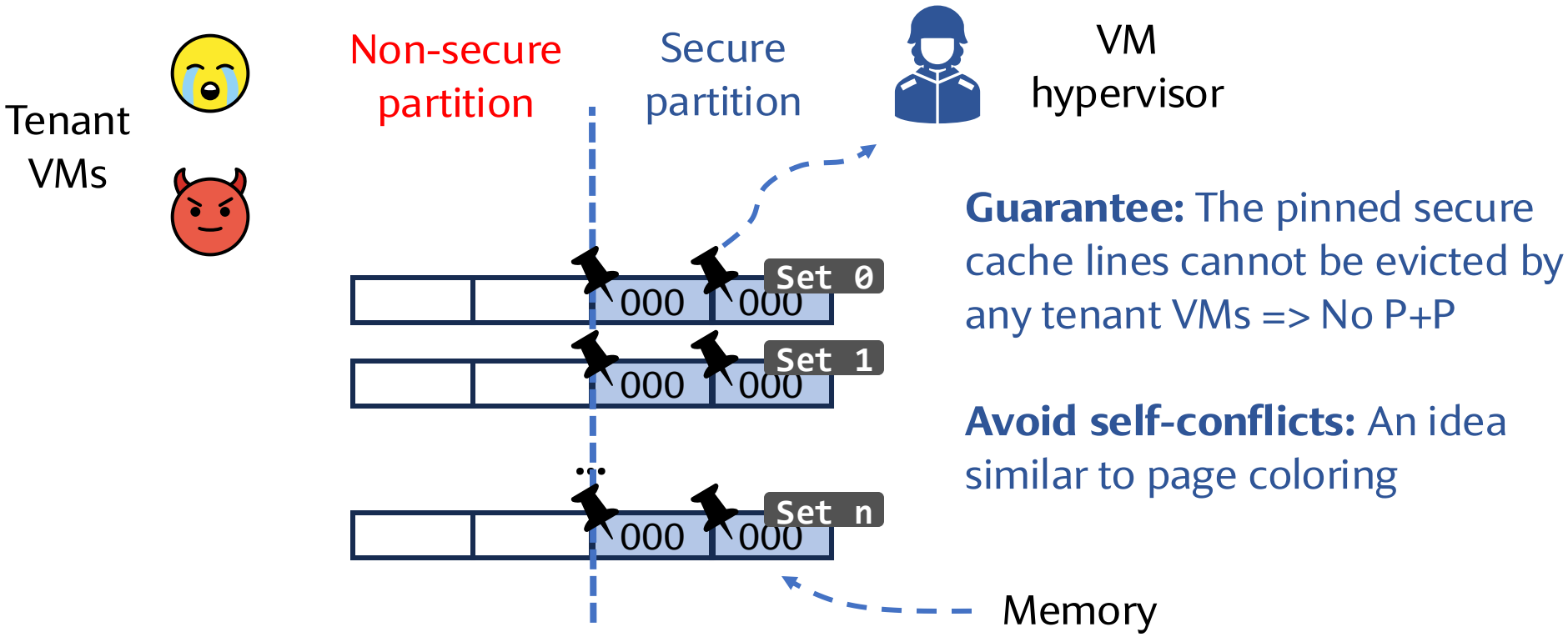
[1] Department of Electrical Engineering, Princeton University, email: {fangfeil,rblee}@princeton.edu

Motivation: Intel CAT supports only 4 partitions, not scalable!

Flush+Reload? No page sharing between VMs

CATalyst: Key Ideas

Always cache sensitive cache lines in the LLC and they cannot be evicted (i.e. pinned)
How? Leverage the Intel CAT isolation!

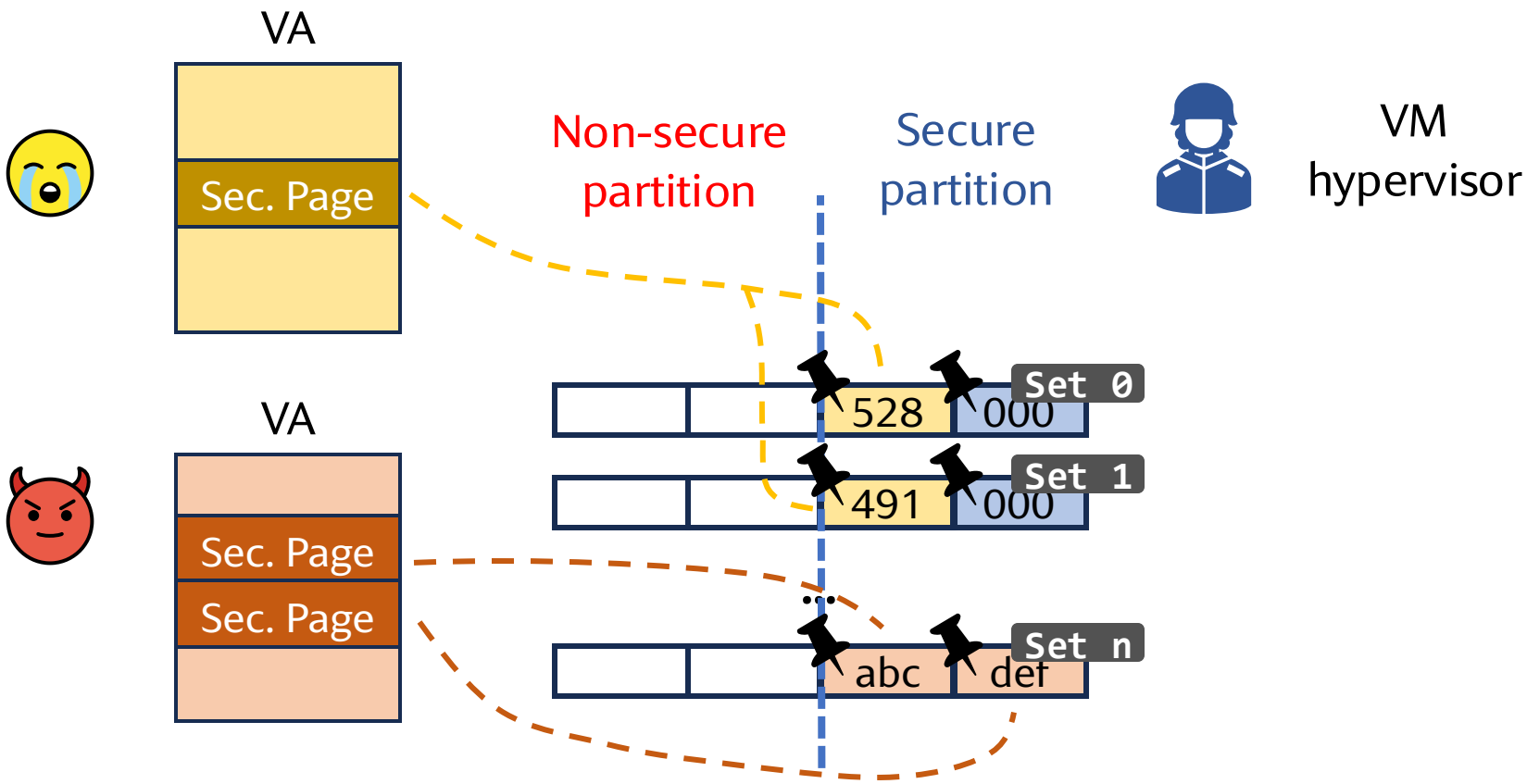


Guarantee: The pinned secure cache lines cannot be evicted by any tenant VMs => No P+P

Avoid self-conflicts: An idea similar to page coloring

CATalyst: Key Ideas

Tenant VMs can request pinned cache lines for storing “sensitive” data (page granularity)



What Should be Stored in Secure Pages/Cachelines?

Square-and-Multiply Exponentiation (Used in RSA)

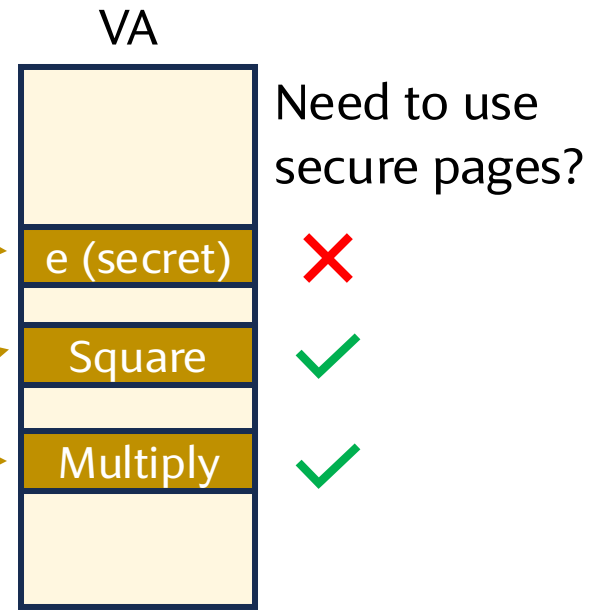
Inputs:

- 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
```

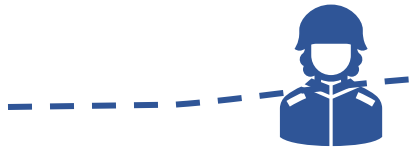


The loop body and other functions that are called from the body should also be stored in secure pages

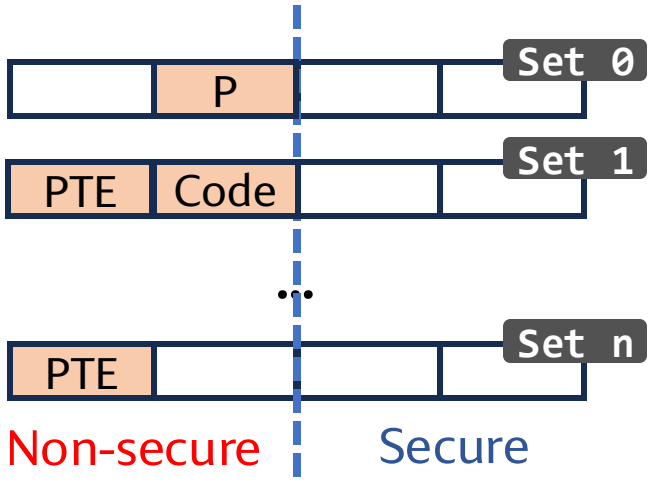
The Correct Sequence for Pinning Secure Pages

Goal: (1) Secure pages are cached in the secure partition
(2) Normal pages are only cached in the non-secure partition

```
Disable interrupts;
Preload the pinning routine;
Access the secure page P;
```



```
Flush P;
```



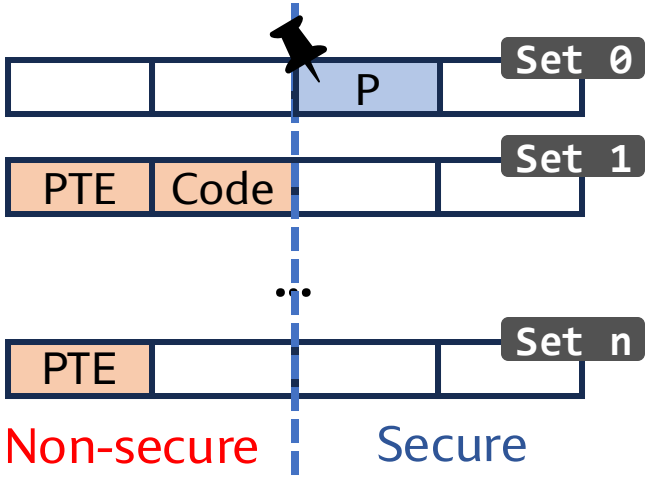
The Correct Sequence for Pinning Secure Pages

Goal: (1) Secure pages are cached in the secure partition
(2) Normal pages are only cached in the non-secure partition

```
Disable interrupts;
Preload the pinning routine;
Access the secure page P;
```

```
Flush P;
Access P;
```

```
Enable interrupts;
```

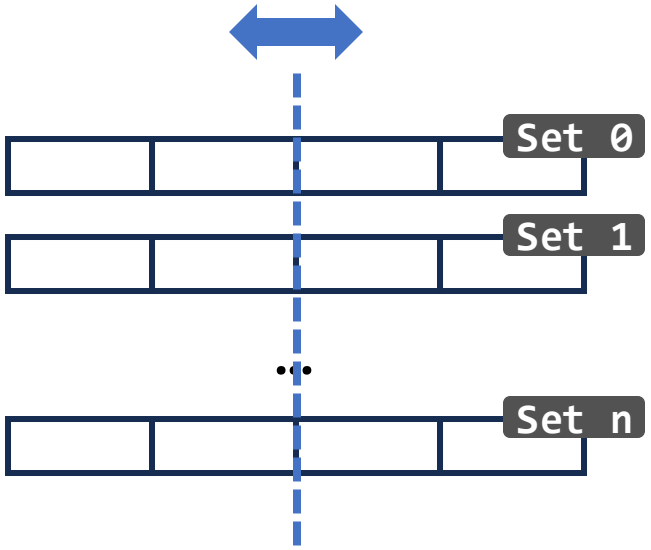


- Completed during system boot
- Need to re-pin pages after a VM terminates, as it might have flushed the page

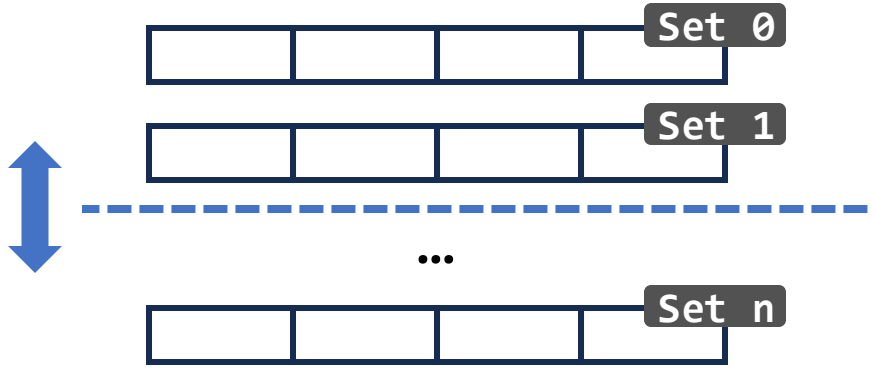
Let's Go Dynamic

Static partitioning leads to resource under-utilization

⇒ Dynamically adjust partition size according the demand of applications



Dynamic way partitioning



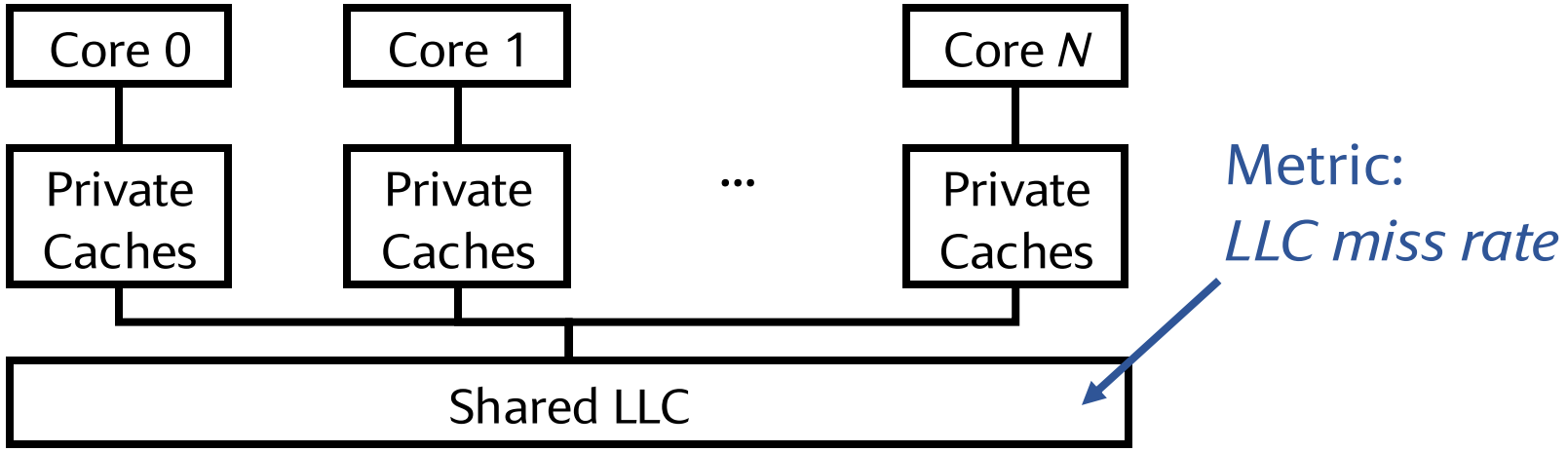
Dynamic set partitioning

A Framework for Thinking About Dynamic Partition

Component 1: Utilization Metric

Reflects a program's resource demand and guides resizing

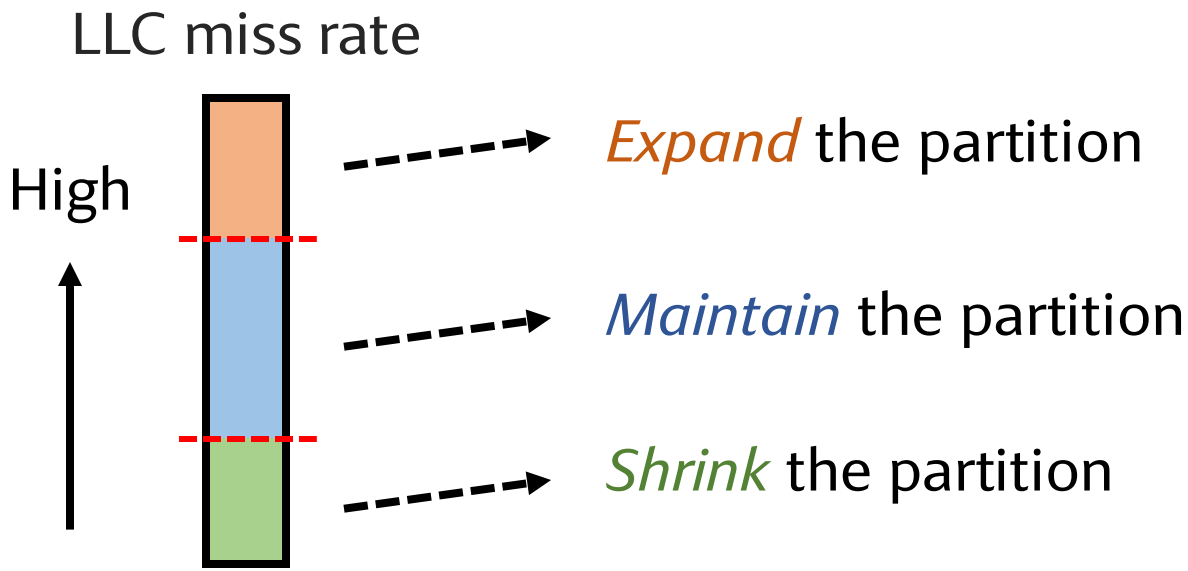
Example: Dynamic last-level cache (LLC) partitioning



A Framework for Thinking About Dynamic Partition

Component 2: Action Heuristic

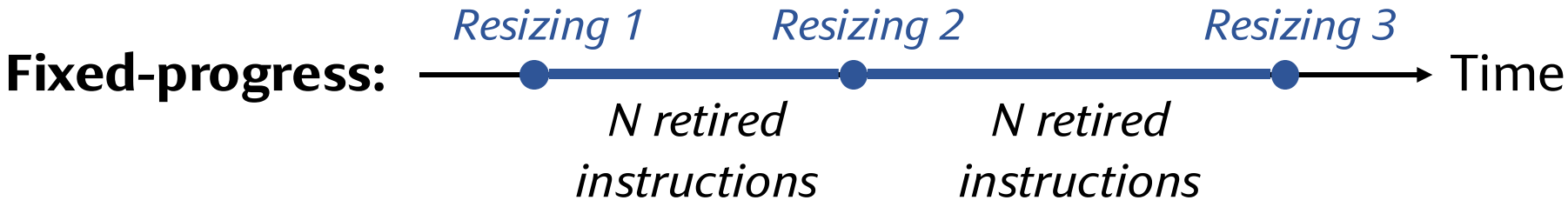
Decides *what* resizing action to perform based on the utilization



A Framework for Thinking About Dynamic Partition

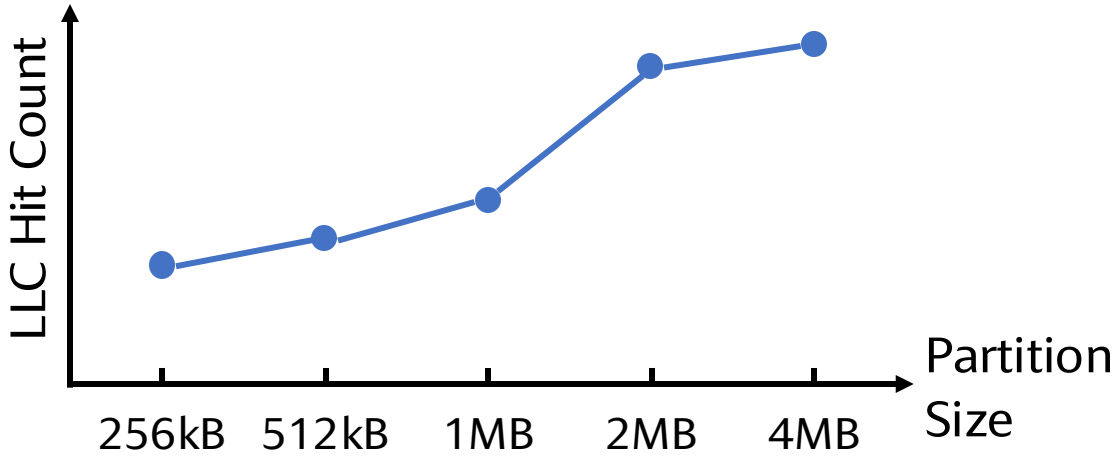
Component 3: Resizing Schedule

Determines *when* to check the utilization and perform the action

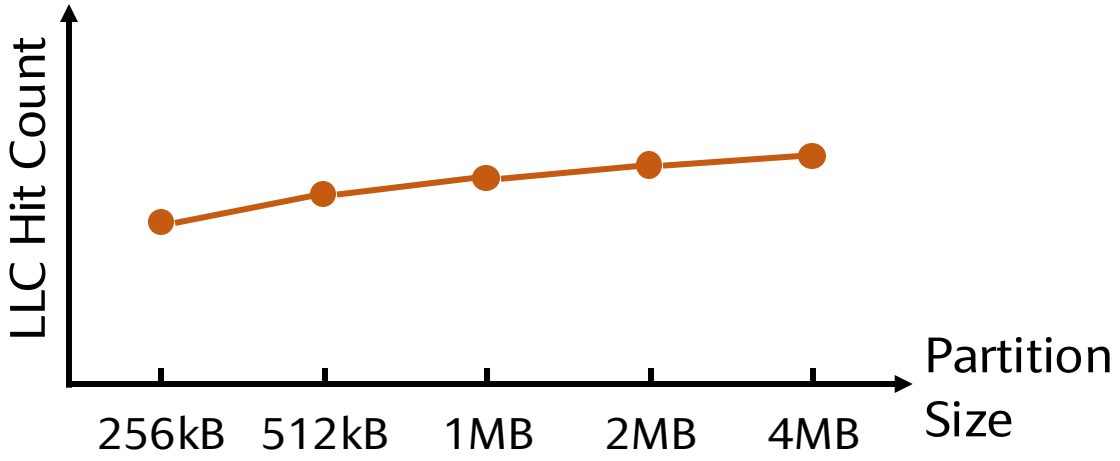


A More Realistic Dynamic Cache Partitioning Algorithm

Hit-curve of application A

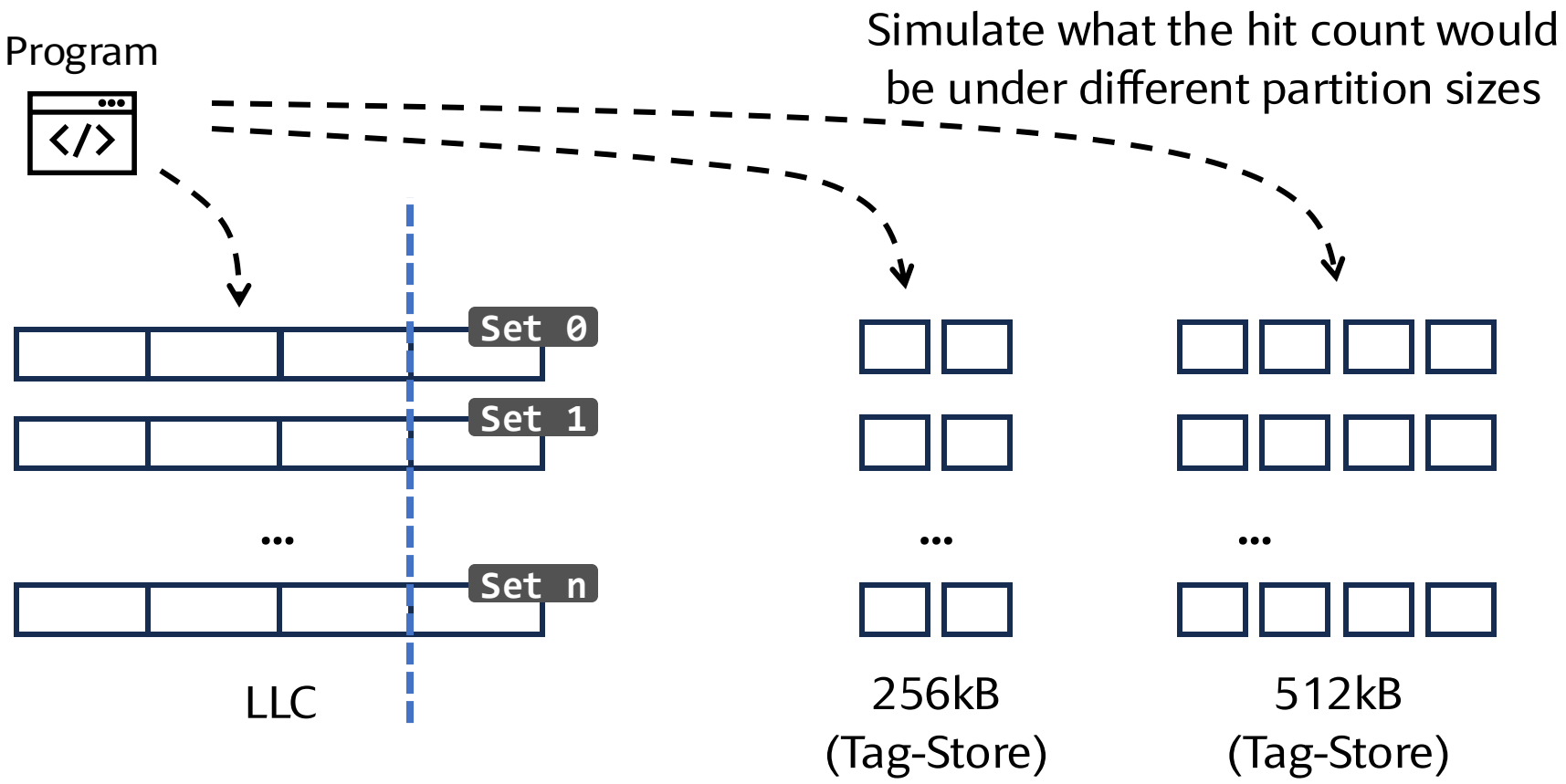


Hit-curve of application B



Action heuristic: Maximize cache hit counts across the system, subject to a total LLC size of 6MB ⇒ Knapsack problem

An Impractical Way of Measuring the Hit Curve



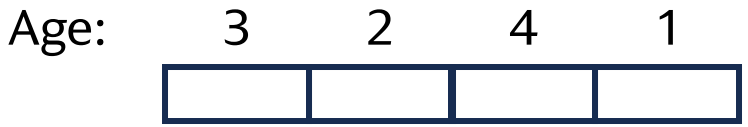
A More Practical Way of Measuring the Hit-Curve

Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches

Moinuddin K. Qureshi Yale N. Patt
Department of Electrical and Computer Engineering
The University of Texas at Austin
{*moin, patt*}*@hps.utexas.edu*

MICRO ’06 (won test-of-time award at MICRO ’24, Austin, TX)

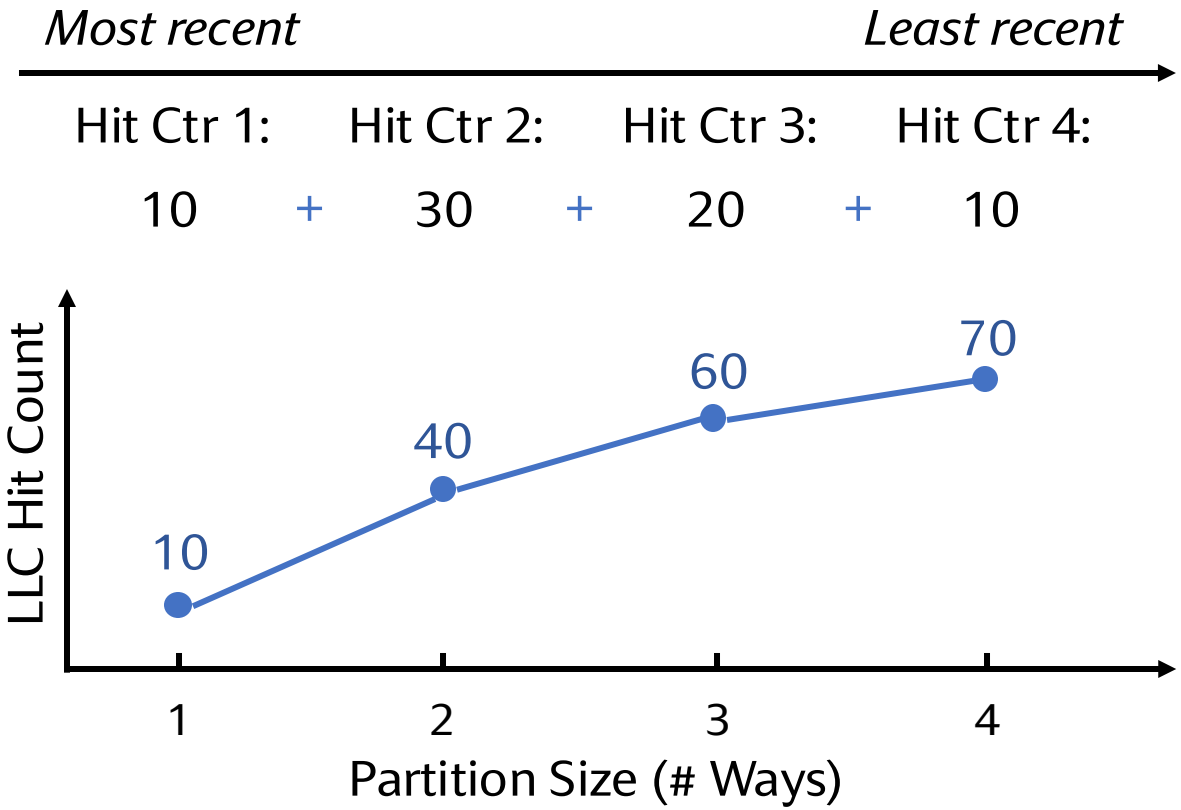
High-Level Idea: The Stack Property of LRU



1 means the most recent
4 means the least recent

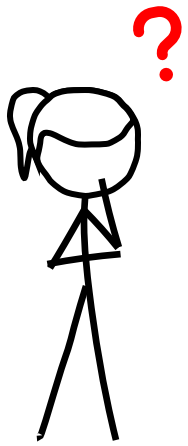
An N-way cache using LRU can always cache the top-N most recently accessed lines

⇒ Access to the i-th youngest line will hit in a W-way LRU cache, where $W \geq i$



👍 Capture the hit curve using a single extra 4-way tag store

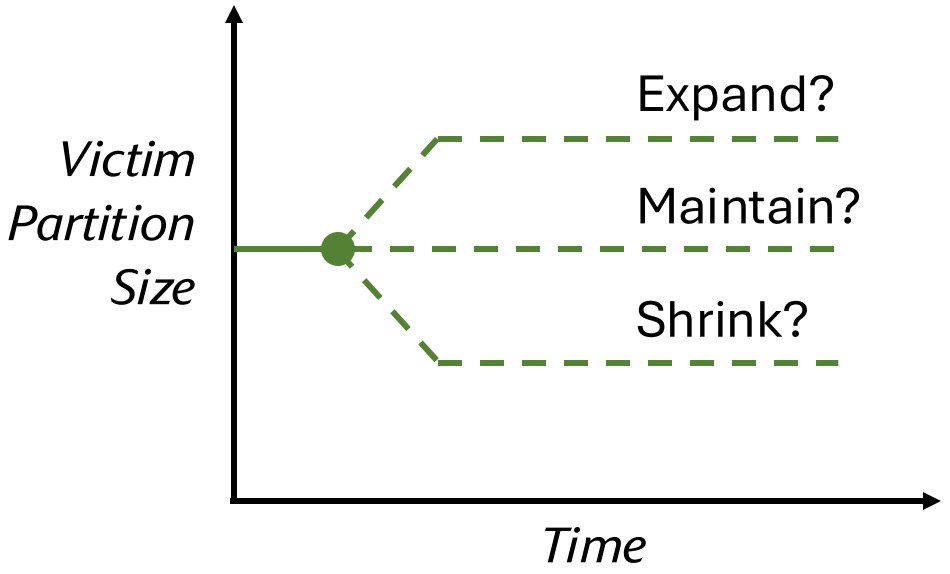
Dynamic Partitioning Leaks Information!



Secret-dependent demand

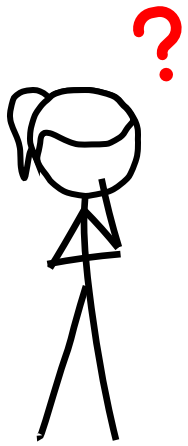
```
if (secret > 0) {
  // traverse a large array
} else if (secret < 0) {
  // traverse a small array
} else {
  // do nothing
}
```

⇒ check resizing, expand?



Action Leakage: *what* resizing action to perform

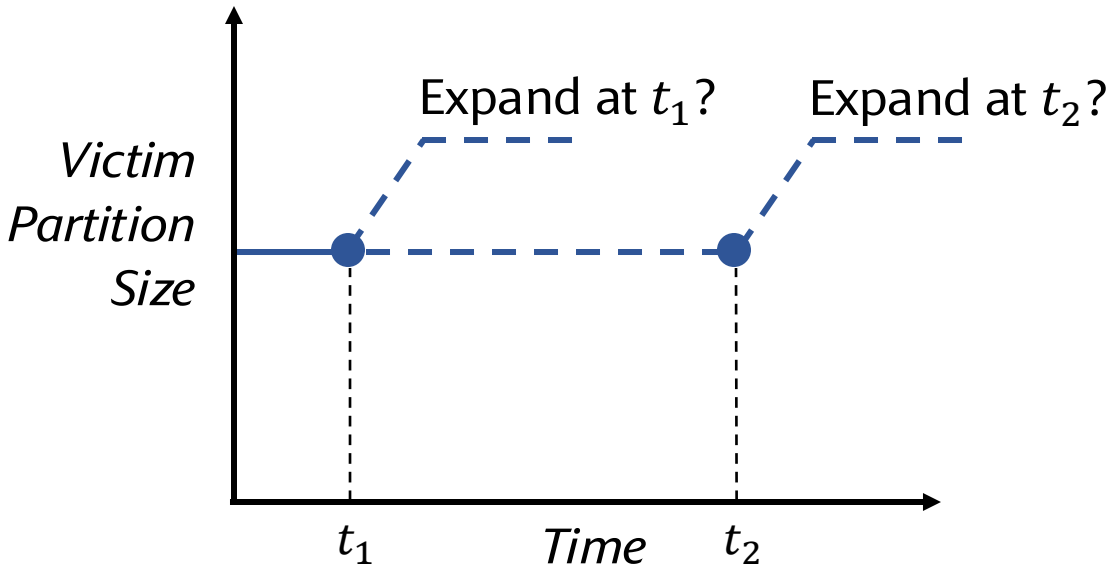
Dynamic Partitioning Leaks Information!



Secret-dependent timing

```
if (secret > 0) {
  sleep(1);
}
// traverse a large array
```

⇒ check resizing, expand!

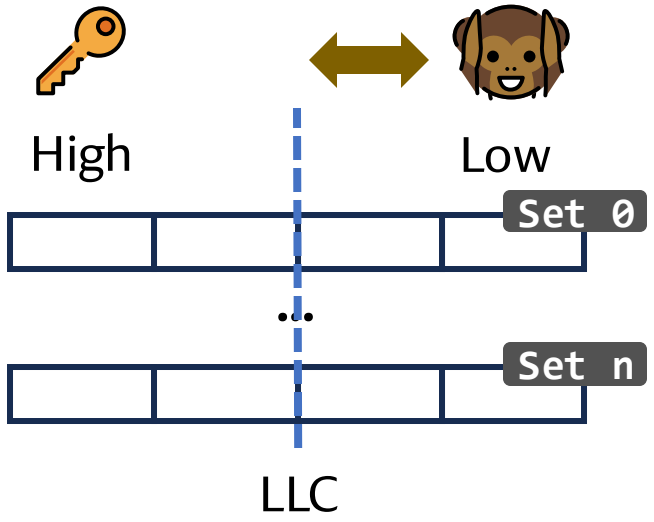


Scheduling Leakage: *when* resizing action occurs

Restricting the Direction of Information Flow

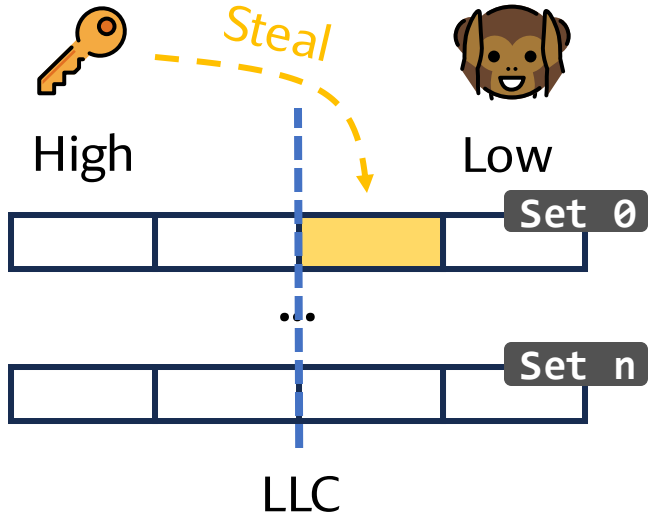
Information can flow from unprivileged domains (Low) to privileged domains (High)

SecDCP (DAC '16)



Solution: Only consider the resource demand of **Low**. Reserve one way for **High**

SecSMT (USENIX Sec '22)



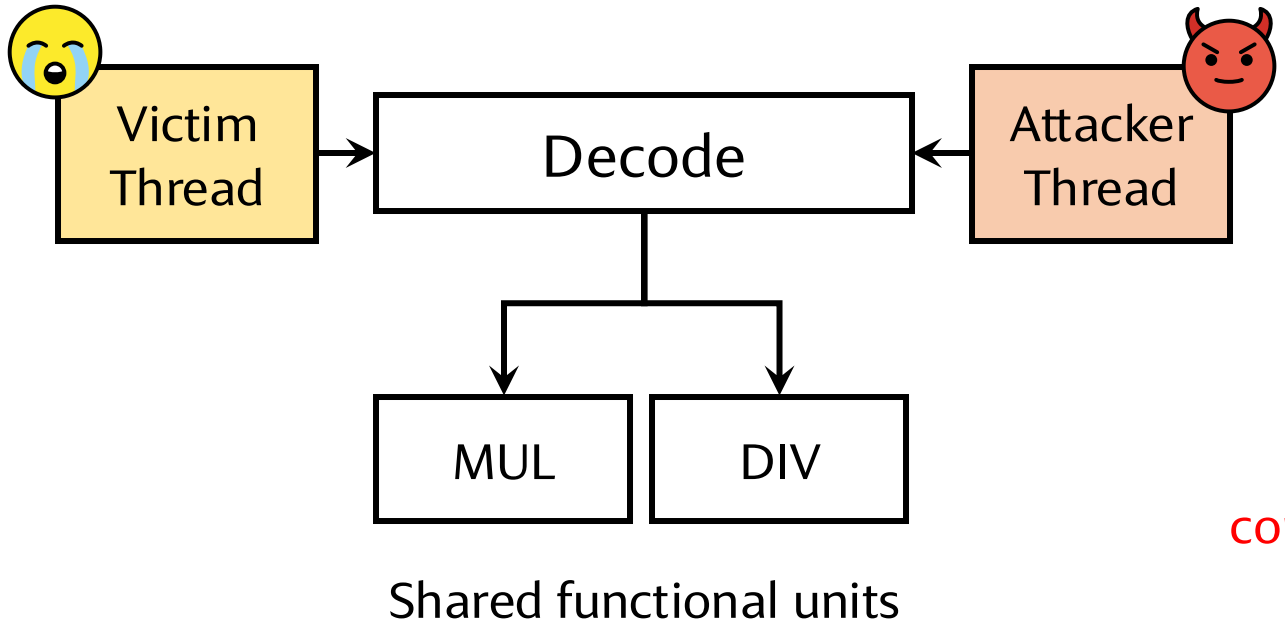
Solution: No resizing. **High** can steal unused resources of **Low**. The stolen resource is immediately released when Low tries to use it

Resource Partitioning is a General Idea

PortSmash[1] (S&P '19), cross-hyperthread attack

Victim:

```
1. if (secret)
2.   victim_div();
3. else
4.   victim_mul();
```



Attacker:

```
1. t1 = time();
2. attacker_div();
3. t2 =time();
```

Detect resource contention on the divider

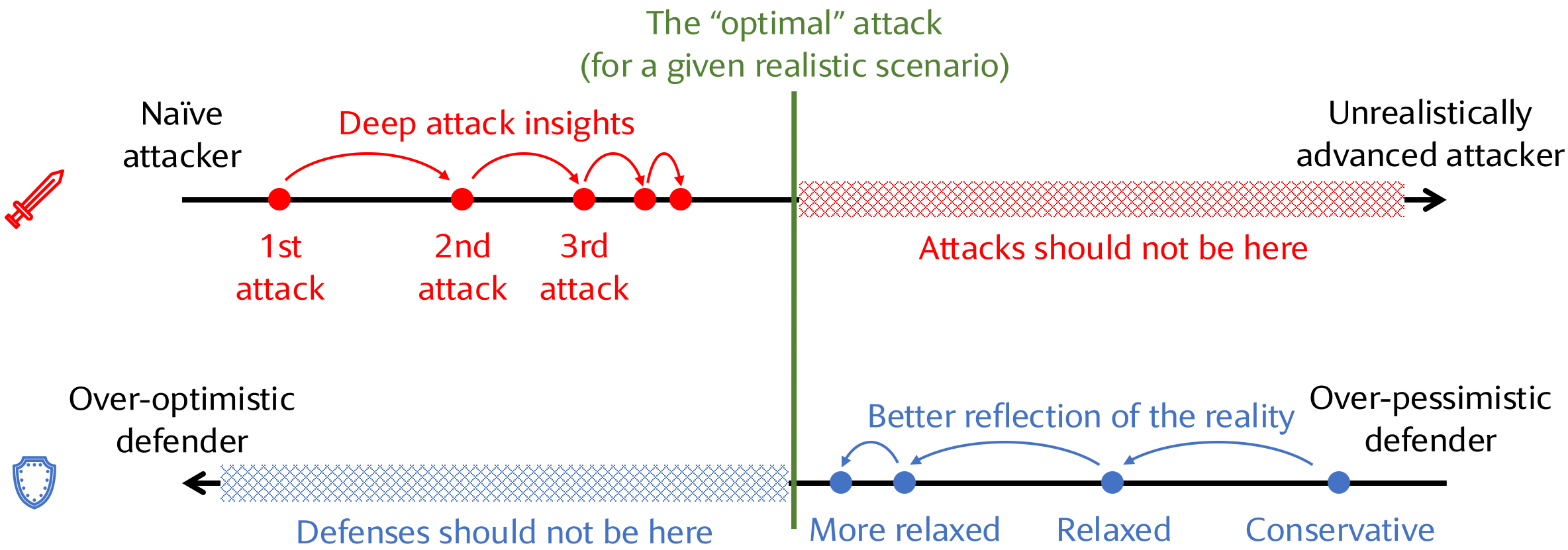
Solution: Temporally partition FUs (e.g., round robin)

[1]Aldaya et al., "Port Contention for Fun and Profit," IEEE S&P 2019

The Story of Attacking and Securing Randomized Caches

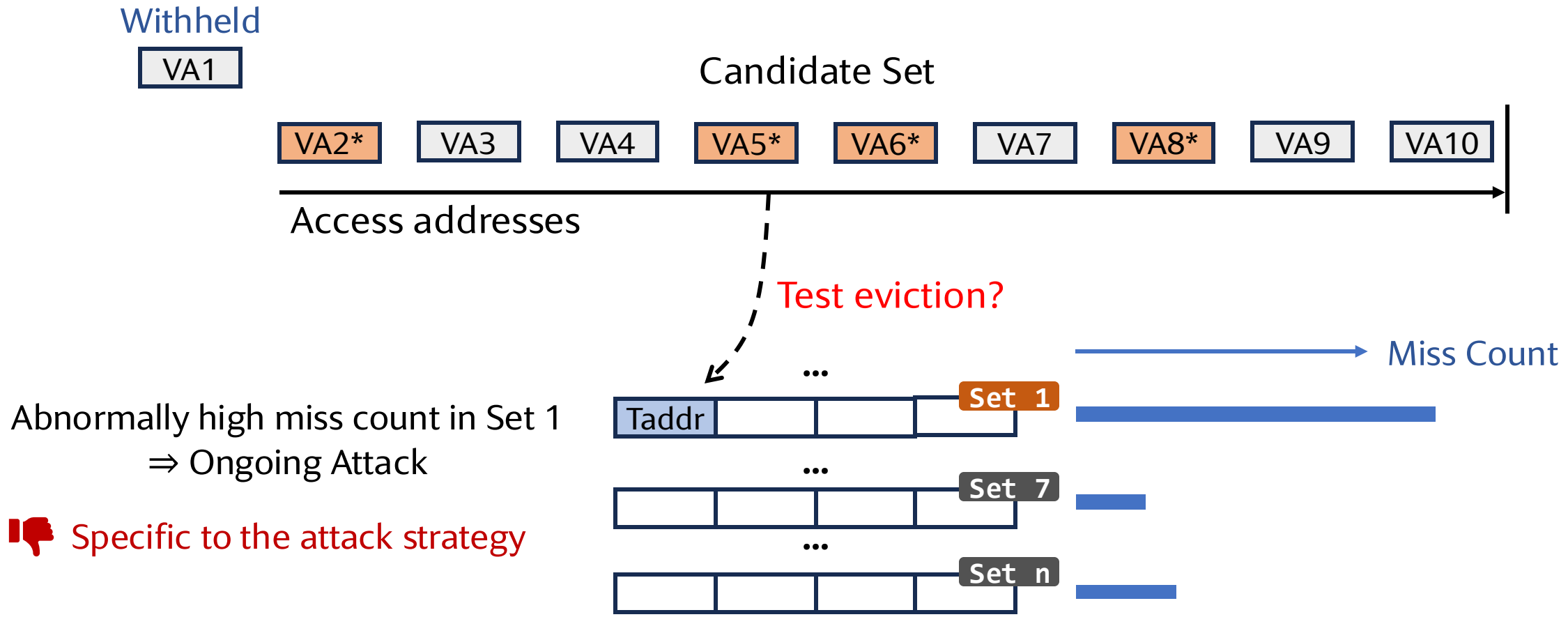
Borrowing slides from Moin Qureshi's keynote at the MAD workshop '22 (co-located with ISCA '22)

My Key Takeaway from the Story



Detecting (Cache) Side-Channel Attacks

The false-negative concern

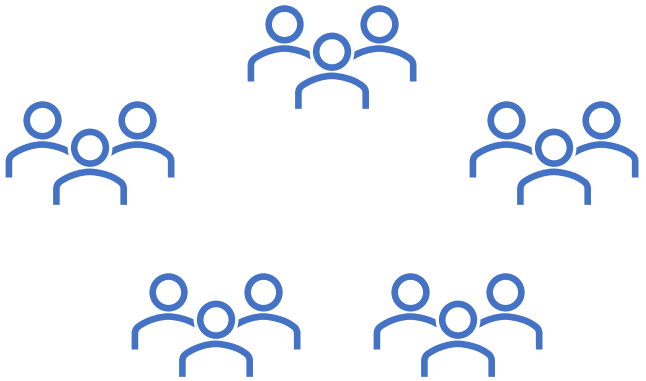


Abnormally high miss count in Set 1
⇒ Ongoing Attack

👎 Specific to the attack strategy

The False Positive Problem

Assuming the detection method has a 0.1% false positive rate and no false negatives



10,000 benign users

↓

10 false positives

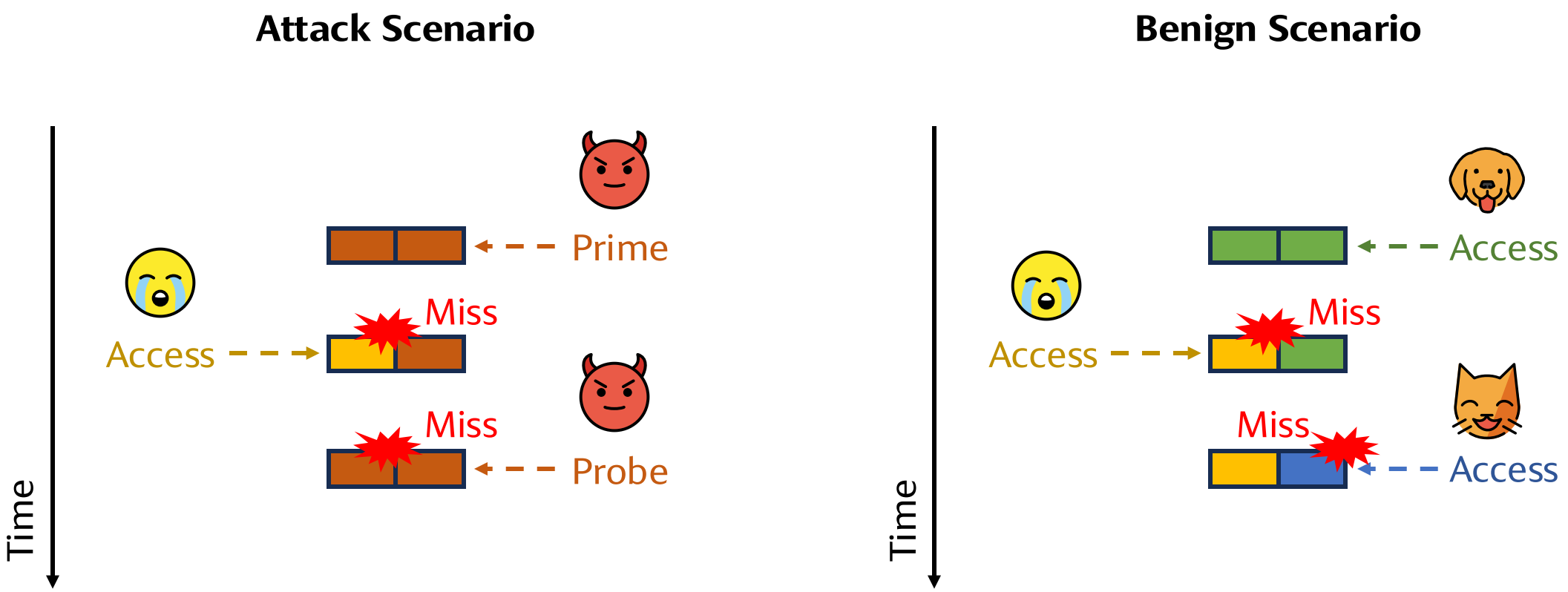
1 malicious user

↓

1 true positive

Detect Ongoing Attacks using Performance Counters

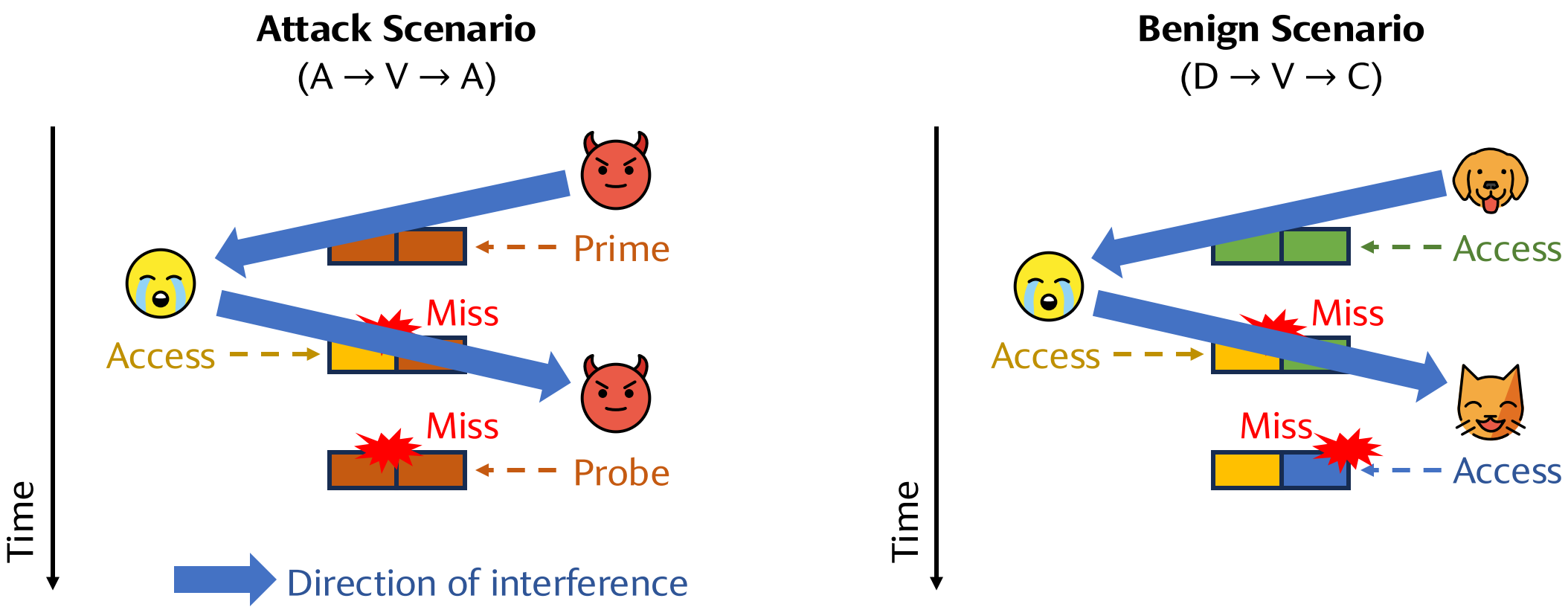
Idea: Collect cache miss count traces using performance counters
⇒ Train an ML model to detect ongoing attacks



Reduce False Positives with Cyclic Interference[1]

Observation: The interference is cyclic in an actual attack

⇒ Use cyclic interference to distinguish true attacks and false positives



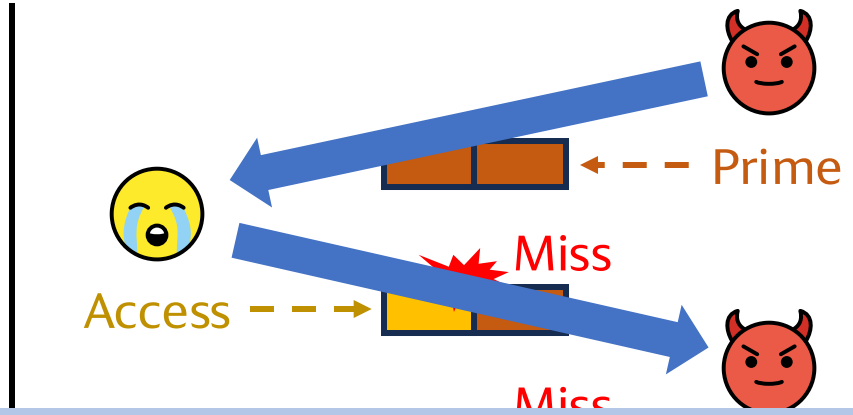
[1]Harris and Wei, et al. "Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference" (MICRO '19)

Reduce False Positives with Cyclic Interference[1]

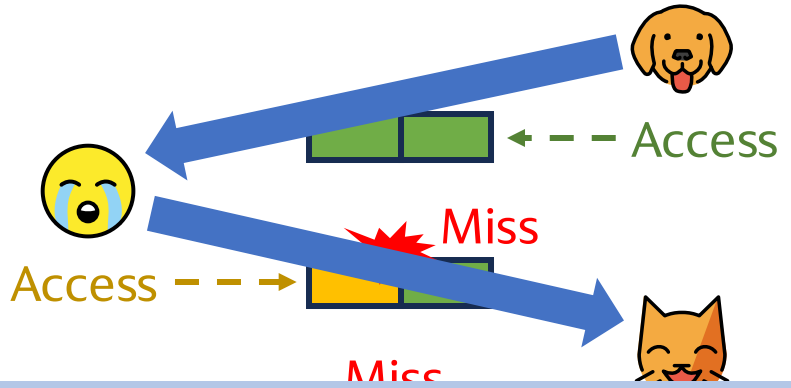
Observation: The interference is cyclic in an actual attack

⇒ Use cyclic interference to distinguish true attacks and false positives

Attack Scenario
(A → V → A)



Benign Scenario
(D → V → C)



Closing thought: Detection, even with false positives and false negatives, does make the attack harder. If the design complexity, performance overhead, and false positive rate are small/tolerable, we should adopt them even if they don't provide comprehensive guarantees

[1]Harris and Wei, et al. "Cyclone: Detecting Contention-Based Cache Information Leaks Through Cyclic Interference" (MICRO '19)

Next Lecture: Data-Oblivious Computation

We cannot fix the environment,
but can we change how we behave