Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes

Zirui Neil Zhao*, Adam Morrison, Christopher W. Fletcher, Josep Torrellas

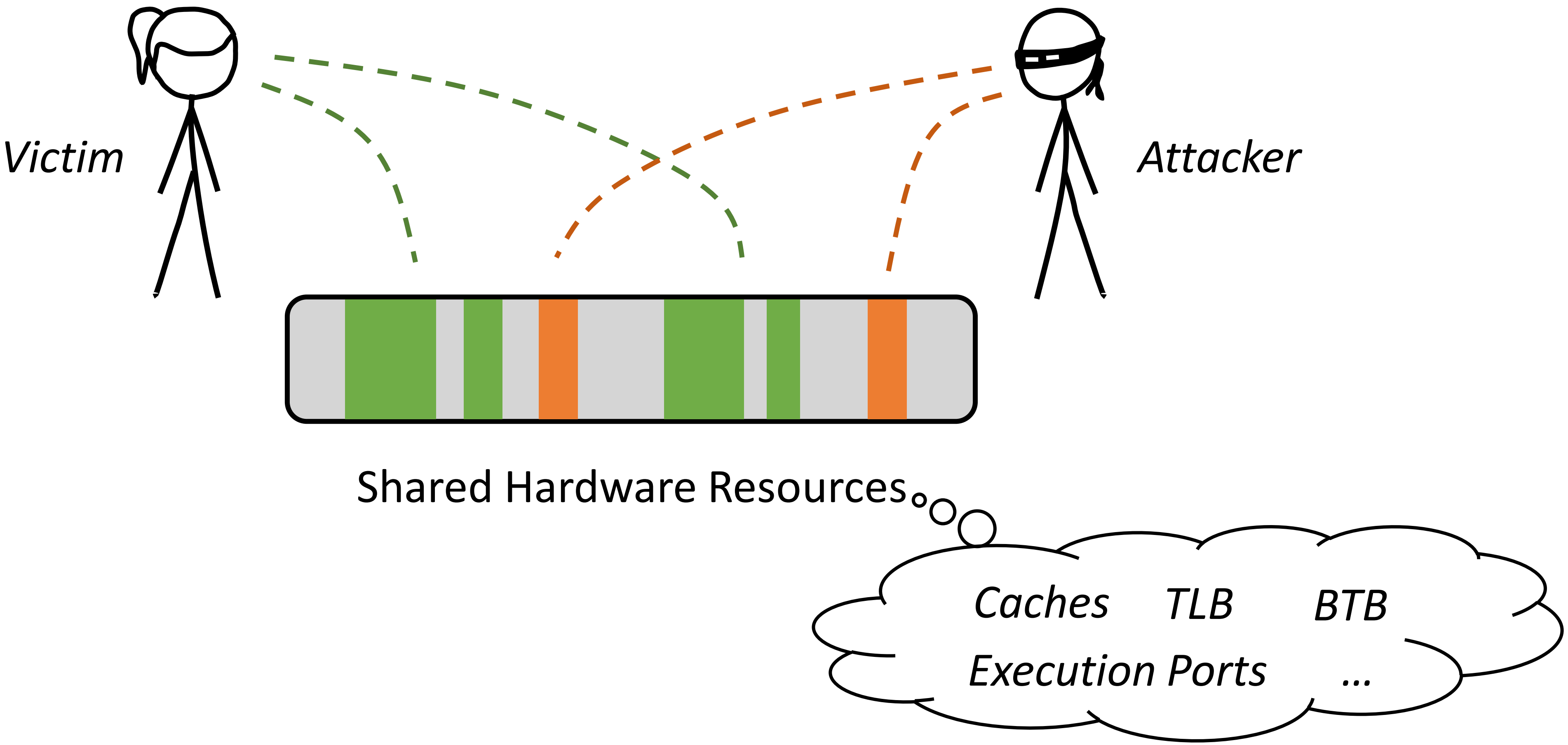
University of Illinois Tel Aviv University

ASPLOS'23 – Session 9C: Hardware Security

ILLINOIS TEL AVIV UNIVERSITY

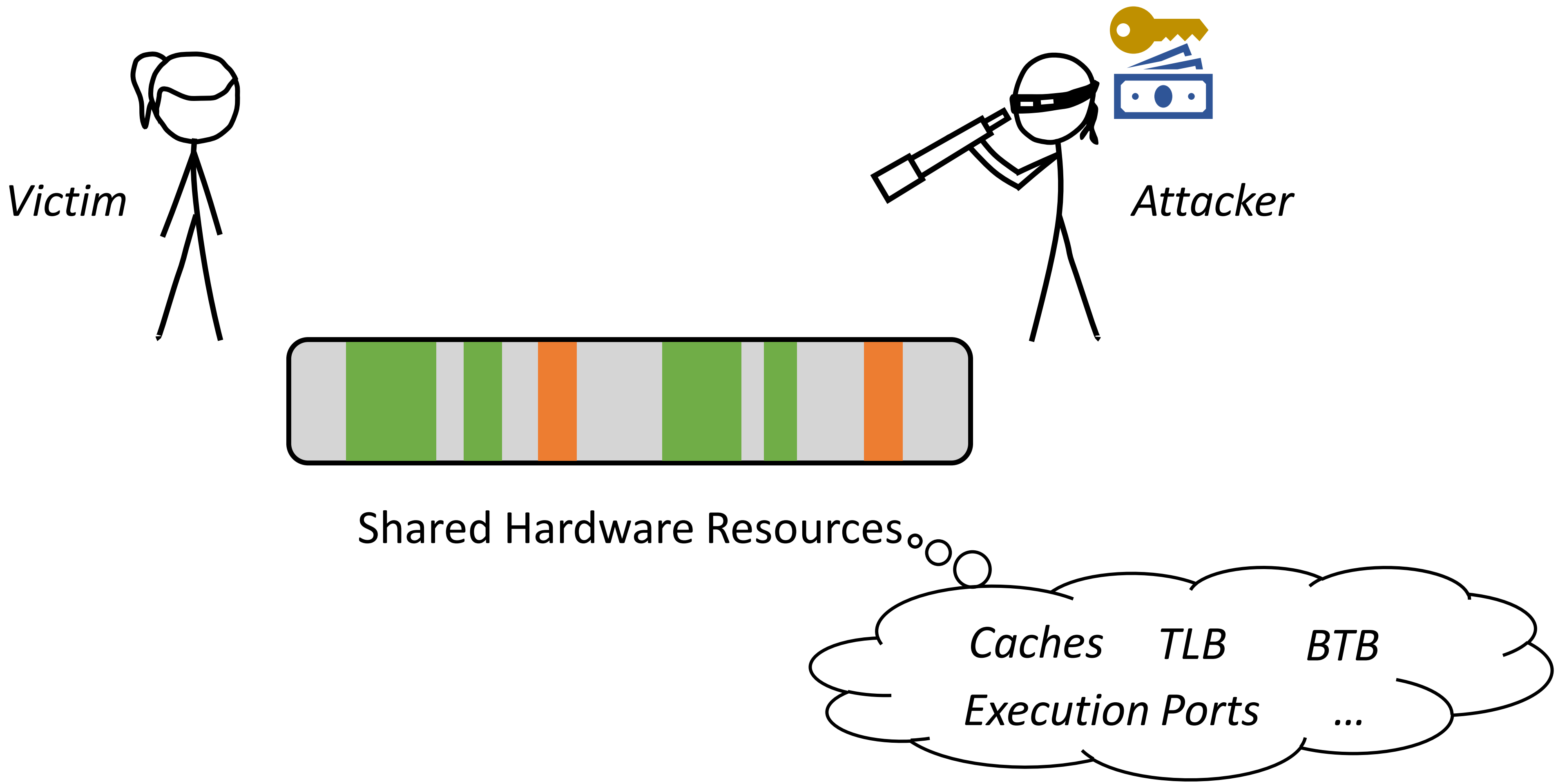
Ad *Will be on the job market this fall, seeking a faculty position*

Microarchitectural Side-Channel Attacks



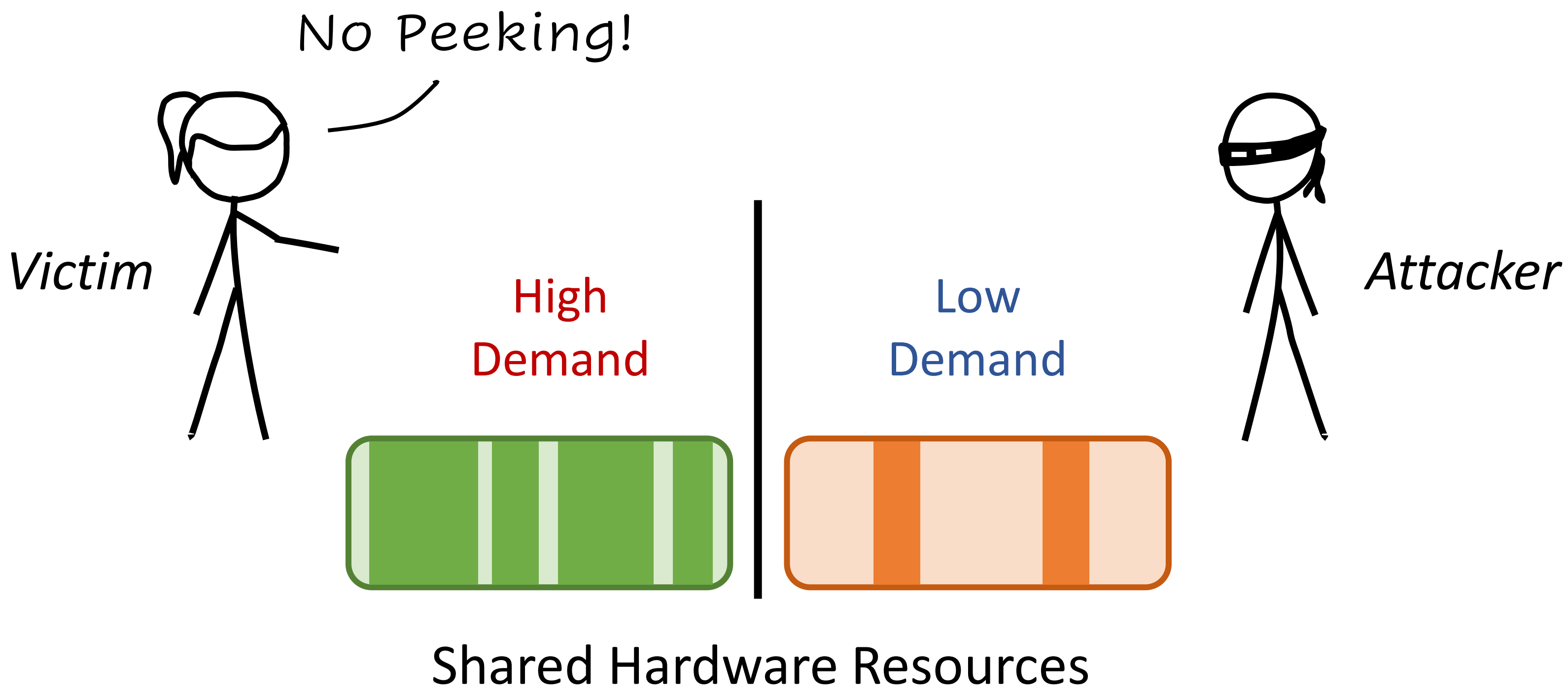
*Characters are based on https://xkcd.com/2176 and https://xkcd.com/1808/

Microarchitectural Side-Channel Attacks



*Characters are based on https://xkcd.com/2176 and https://xkcd.com/1808/

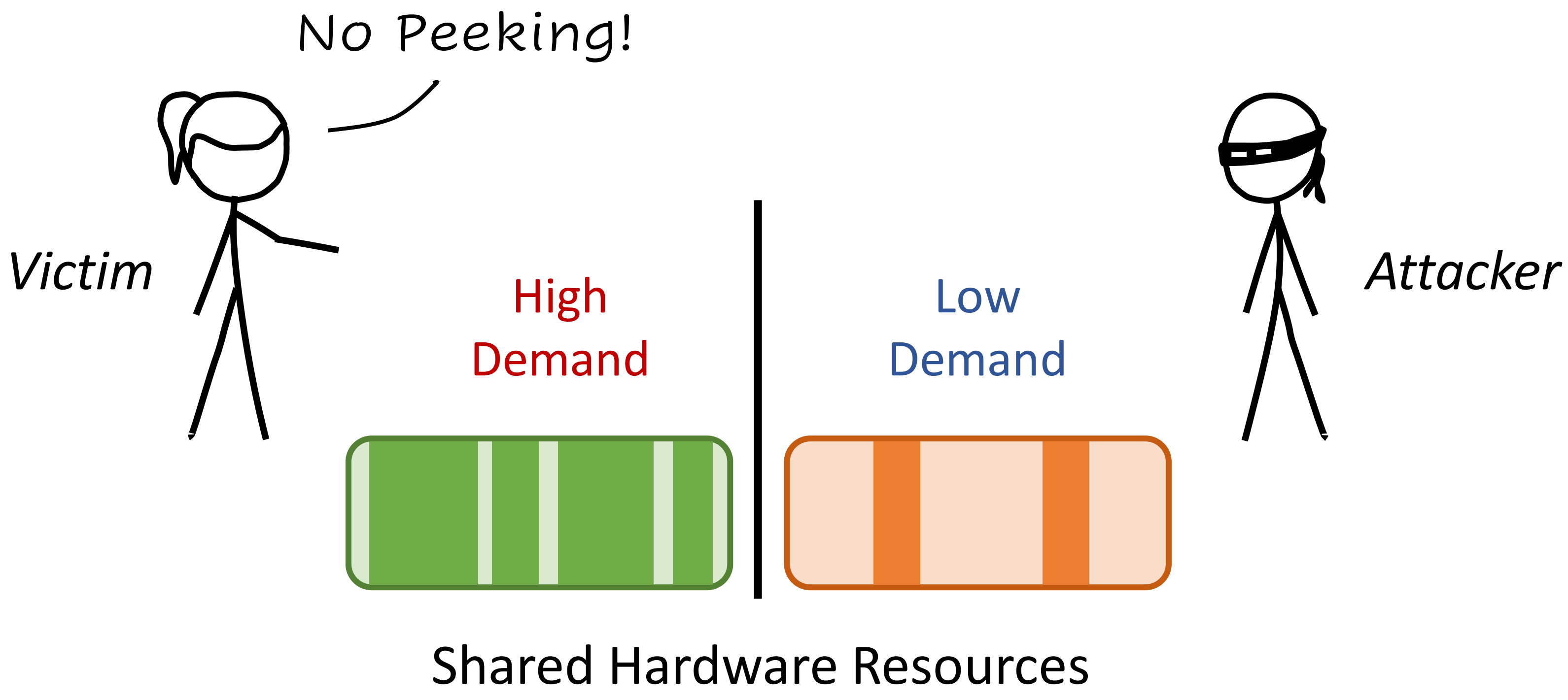
Static Resource Partitioning as a Defense



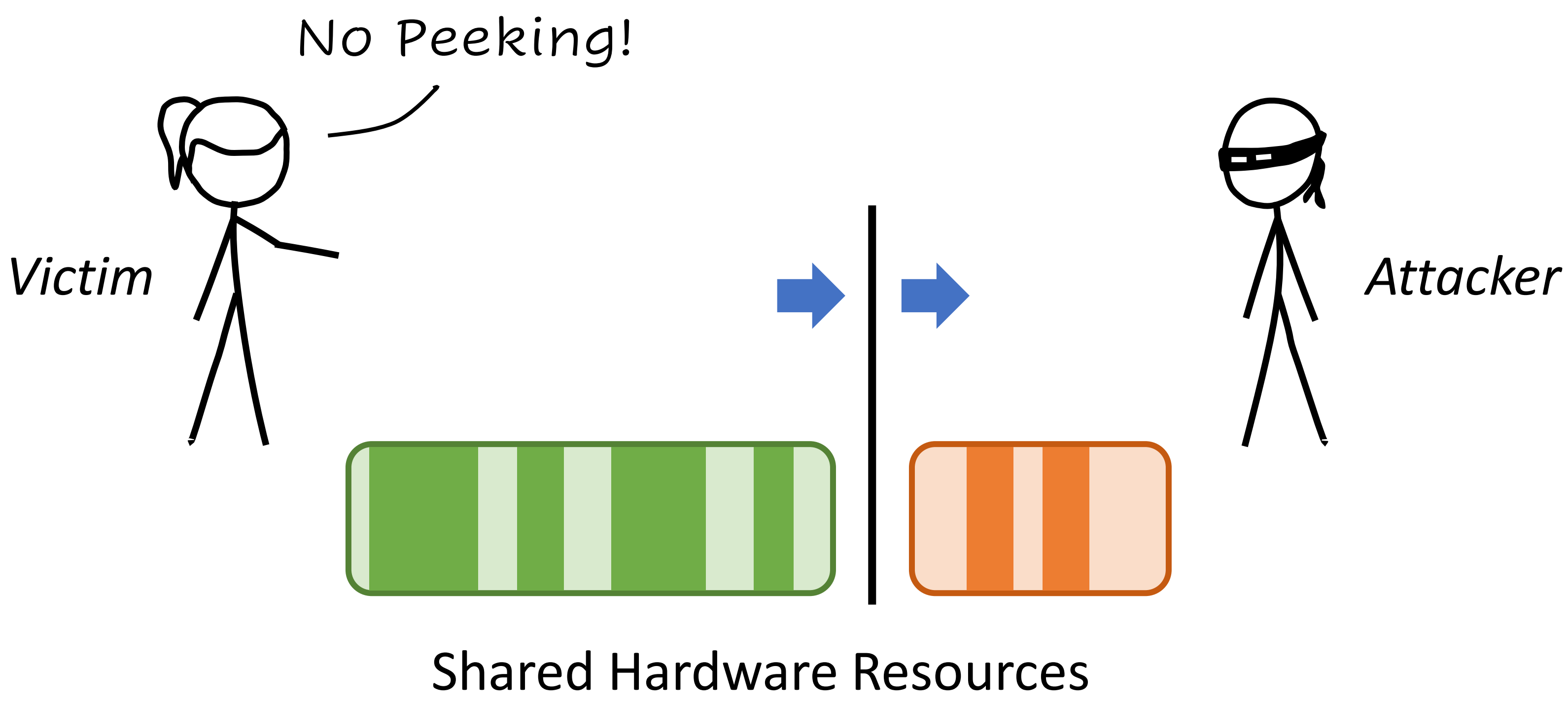
☹ Resource Starvation

☹ Resource Wastage

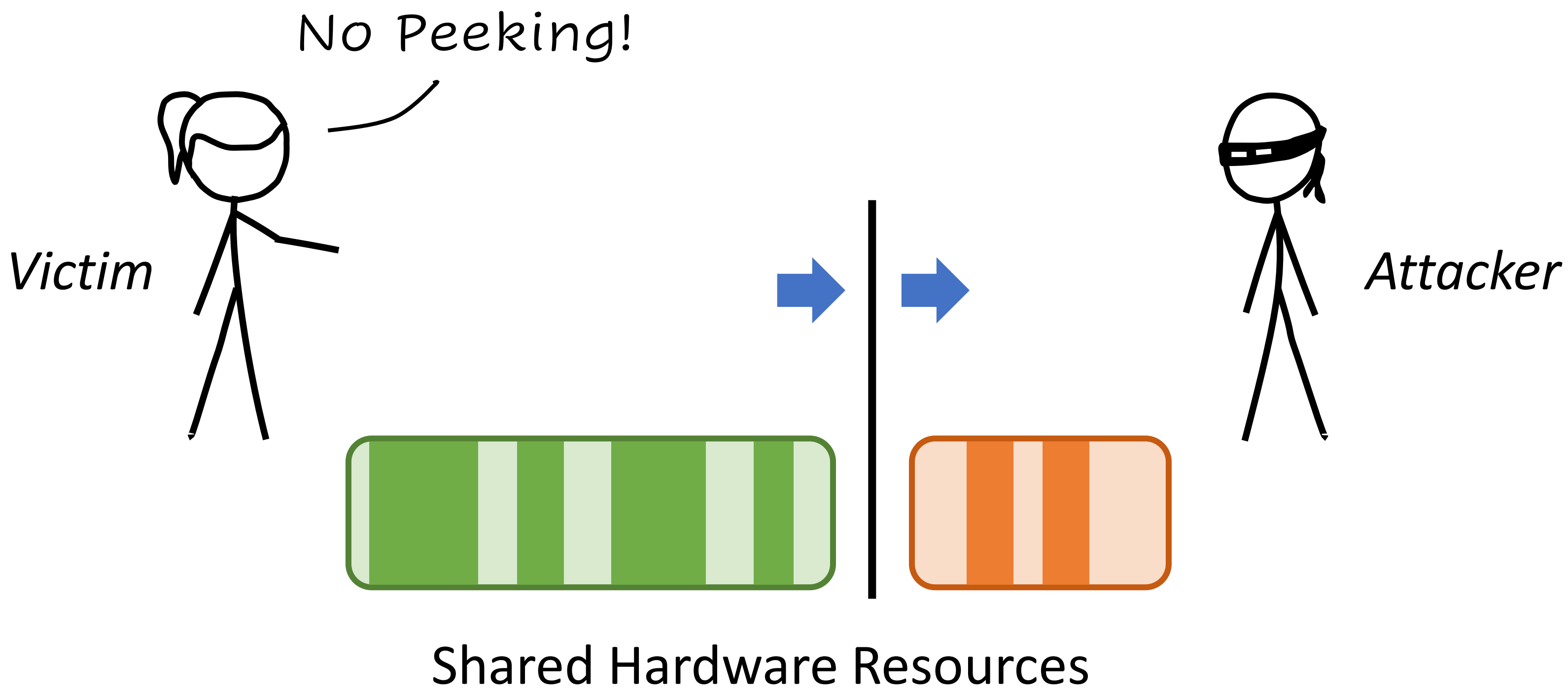
Dynamic Partitioning and Its Leakage



Dynamic Partitioning and Its Leakage

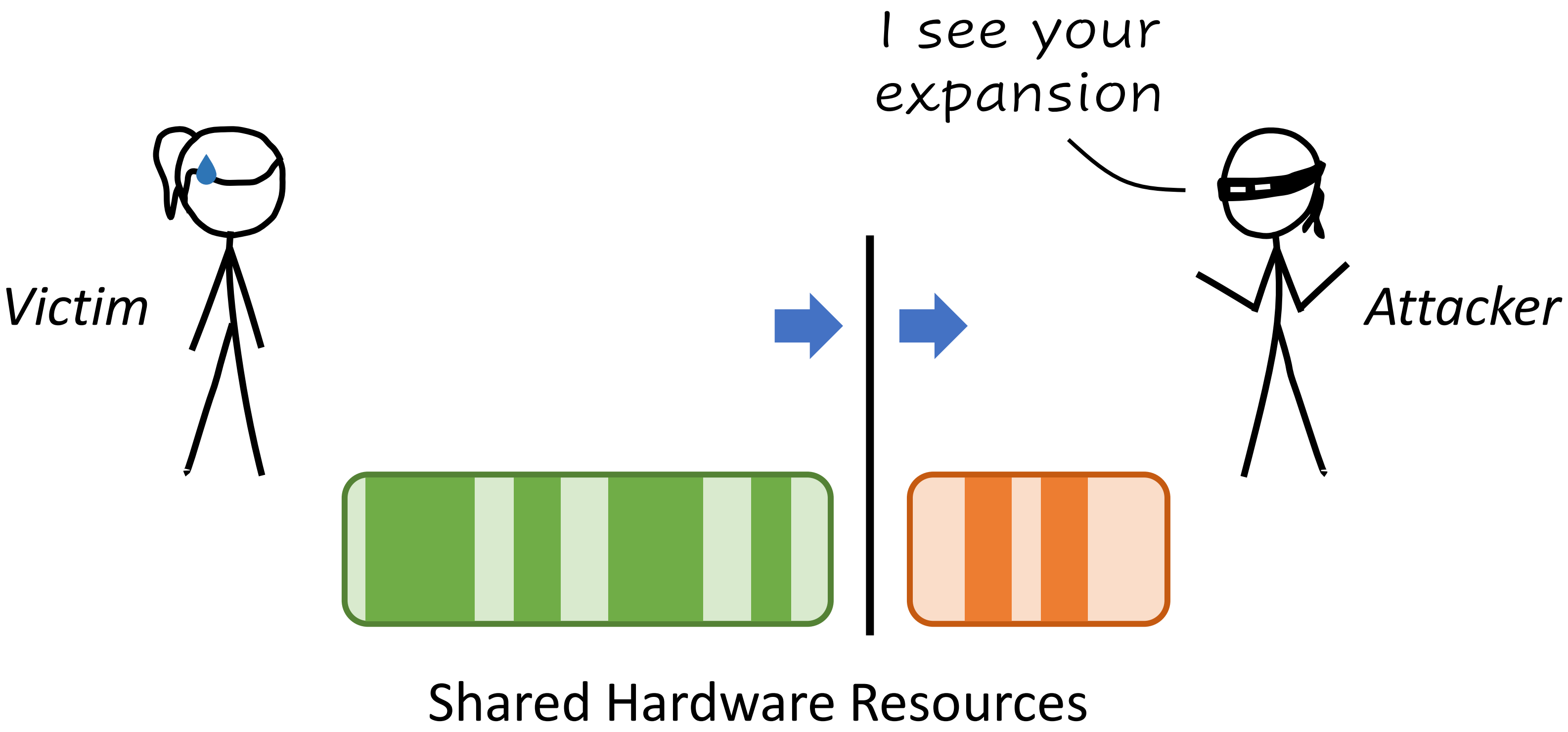


Dynamic Partitioning and Its Leakage



☺ High Performance

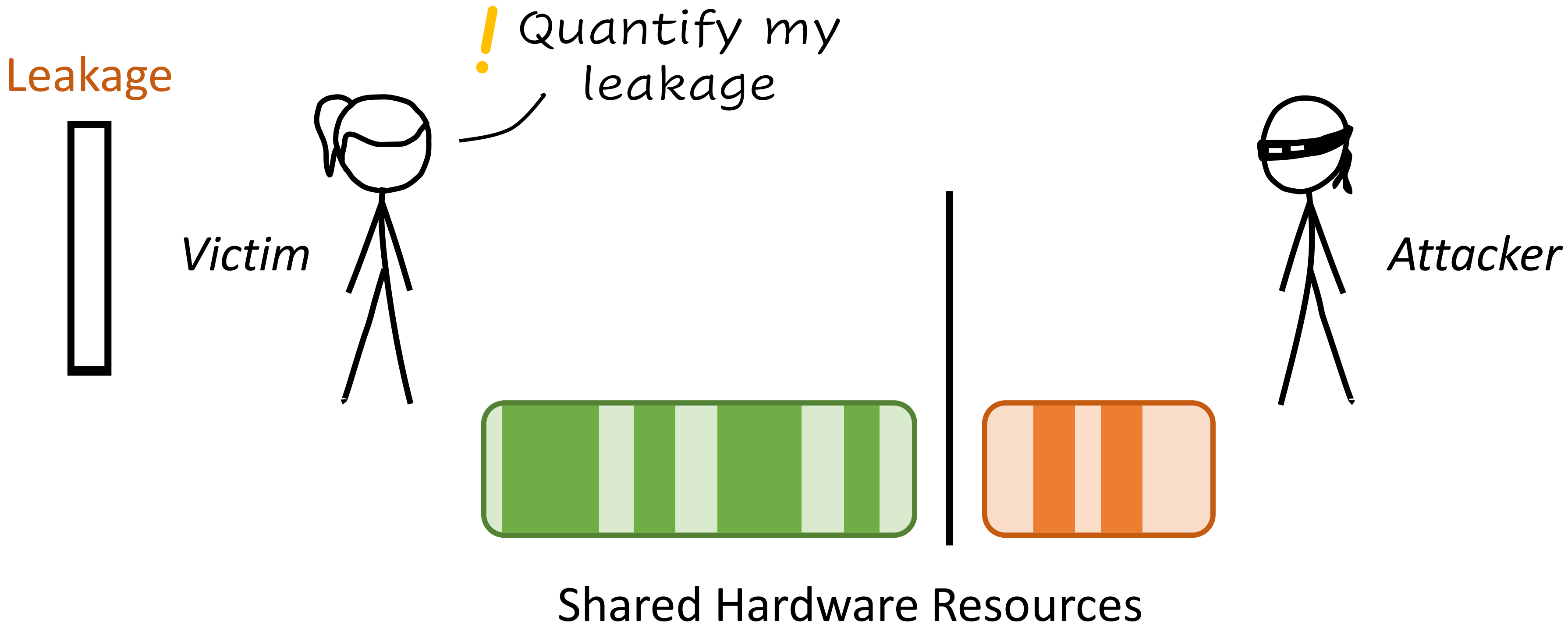
Dynamic Partitioning and Its Leakage



☺ High Performance

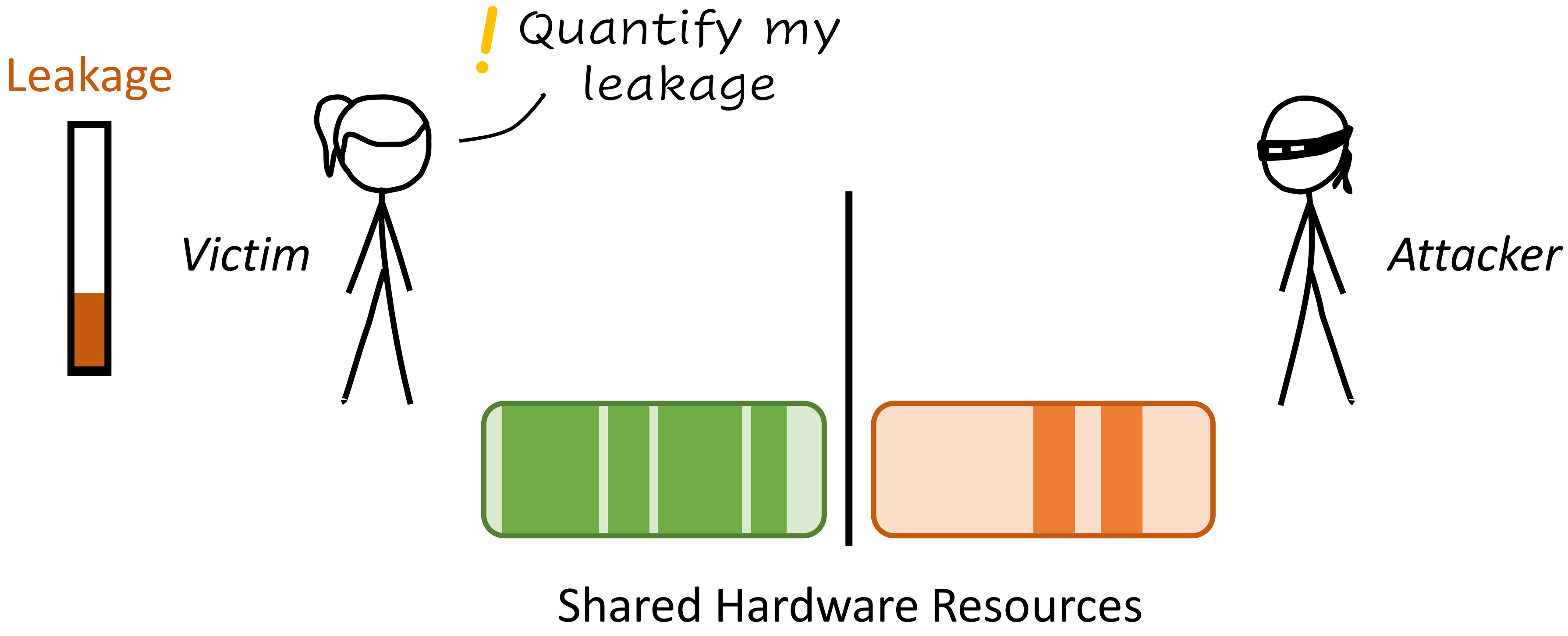
☹ Some Information Leakage

Quantify the Leakage



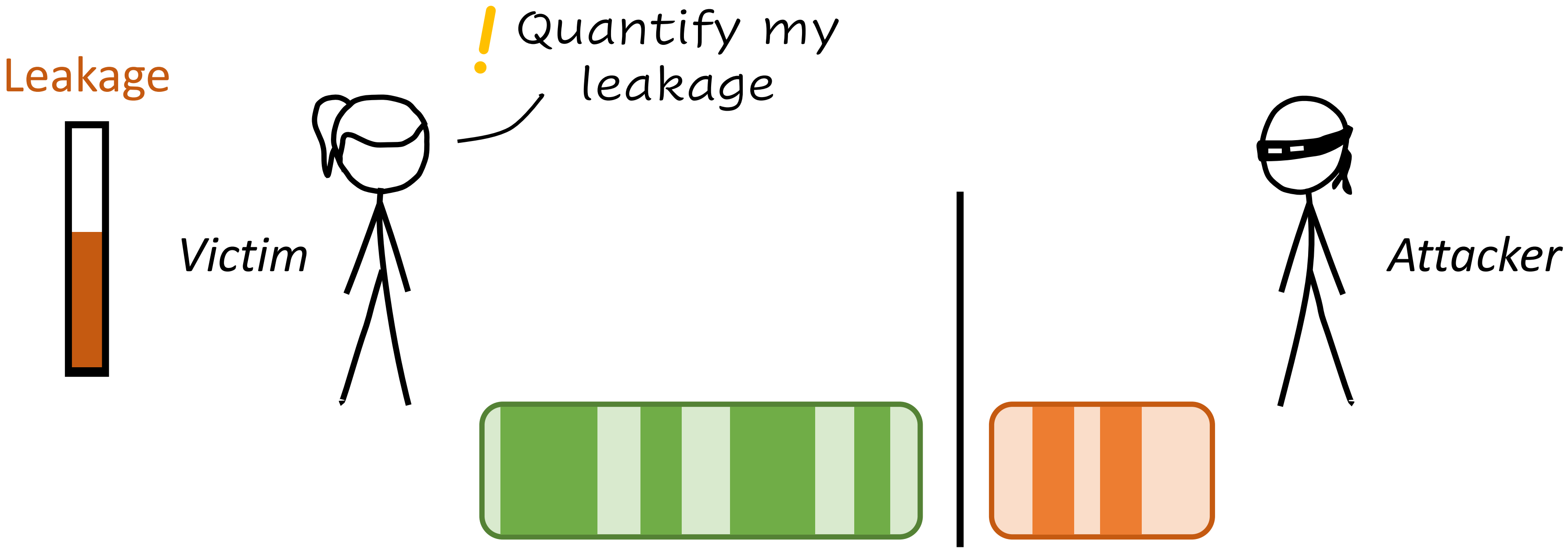
1. Measure information leakage

Quantify the Leakage



1. Measure information leakage

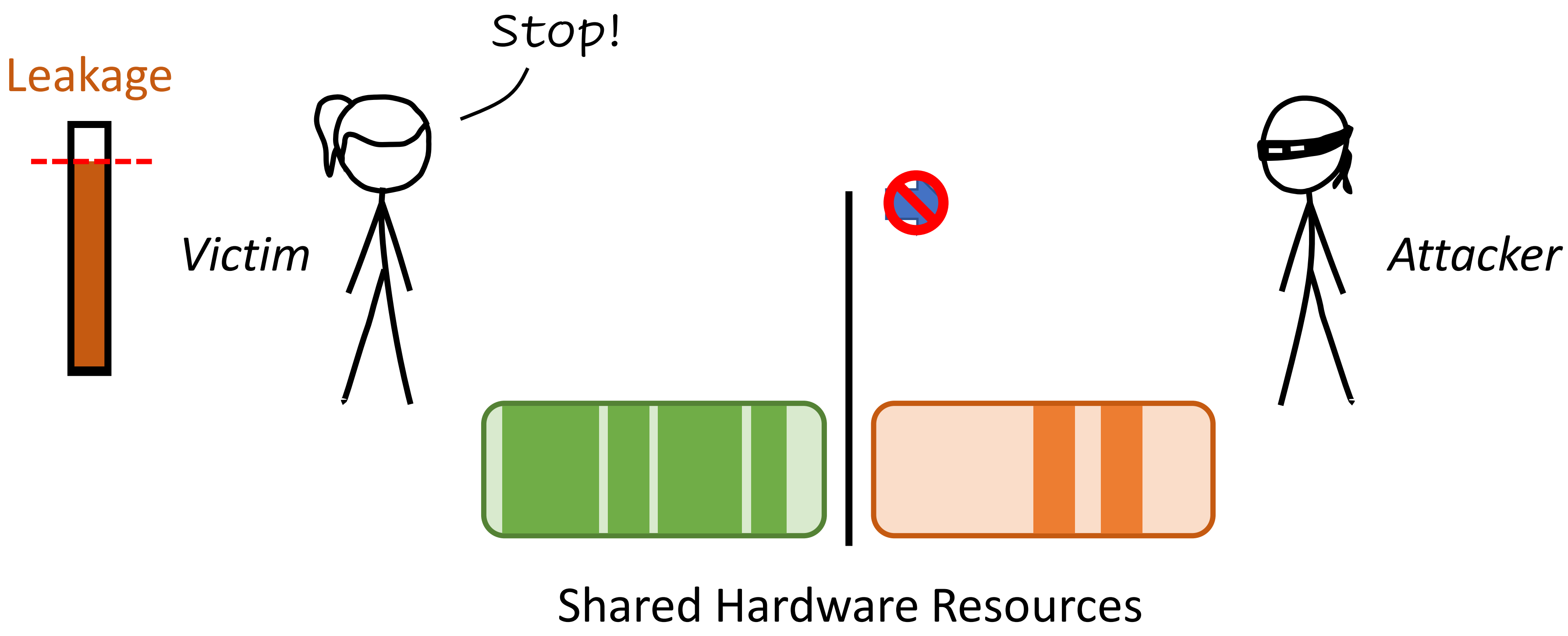
Quantify the Leakage



Shared Hardware Resources

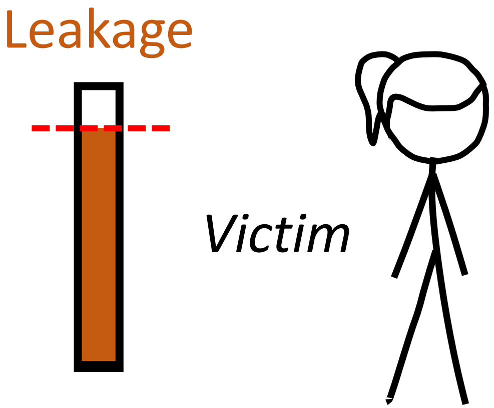
1. Measure information leakage

Quantify the Leakage

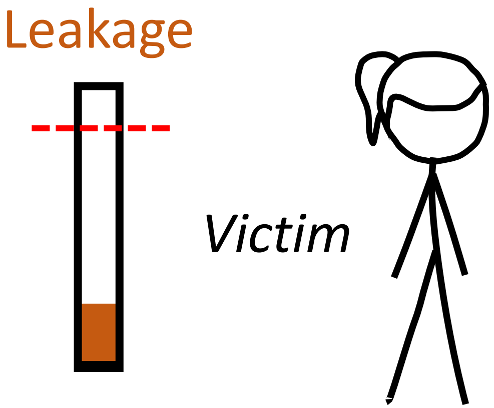


2. Stop resizing once the leakage budget is reached

Less Leakage, More Performance

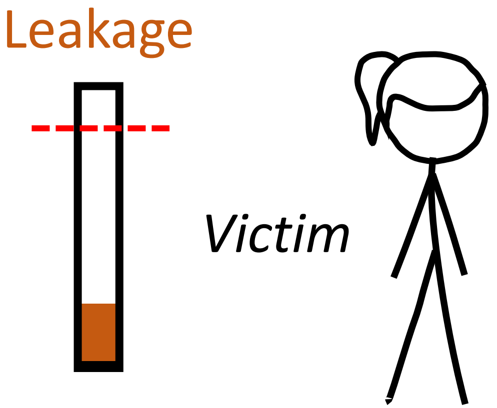


Less Leakage, More Performance



Lower leakage rate ⇒ More resizings under the budget ⇒ Better performance

Untangle: Contributions



Lower leakage rate ⇒ More resizings under the budget ⇒ Better performance

Our Main Contributions:

- A general framework to tightly quantify the leakage
  - ☺ Start fresh with leakage quantification in mind
- Designs that reduce the leakage

Threat Model

- A leakage budget
- No resizing after reaching the budget

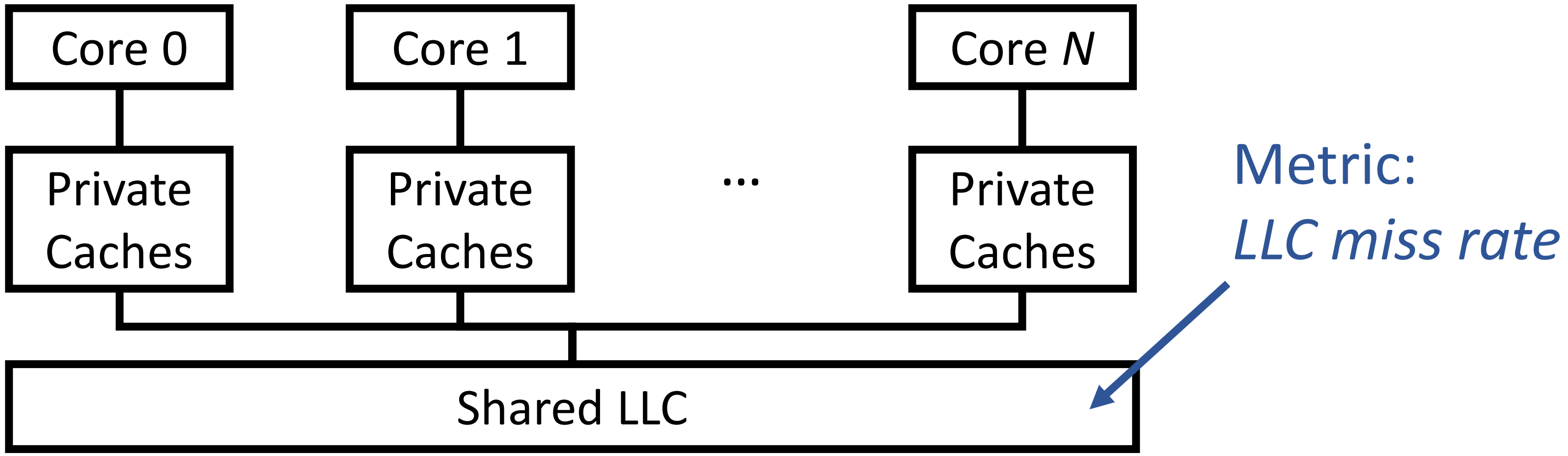
- Directly observe the victim’s resizing
- Observations are instantaneous and accurate

Generalized Dynamic Partitioning

Component 1: Utilization Metric

Reflects a program's resource demand and guides resizing

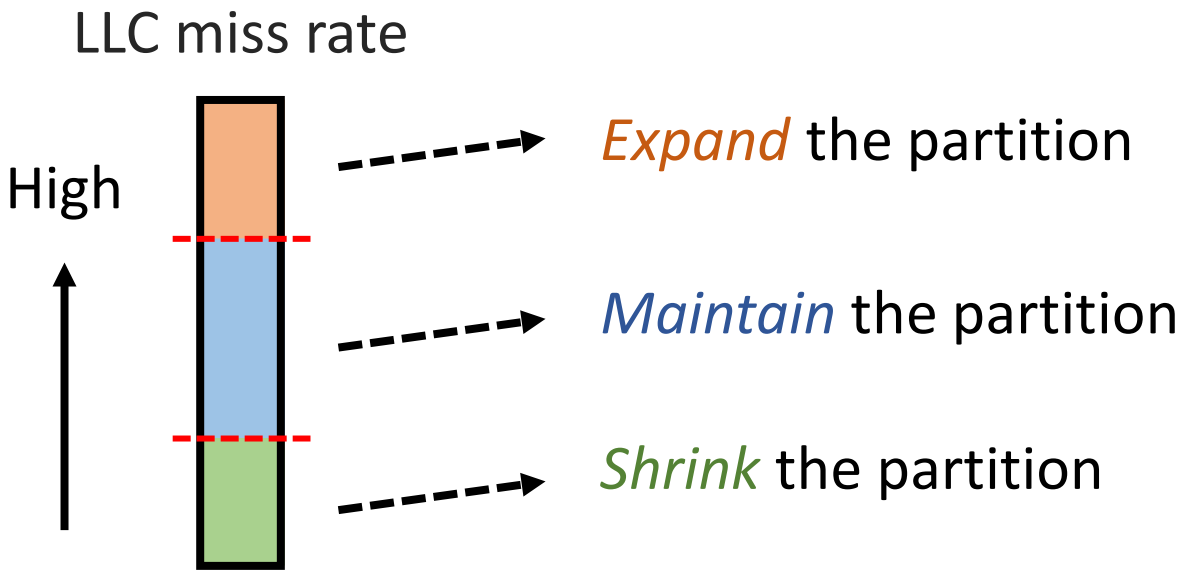
Example: Dynamic last-level cache (LLC) partitioning



Generalized Dynamic Partitioning

Component 2: Action Heuristic

Decides *<u>what</u>* resizing action to perform based on the utilization



Generalized Dynamic Partitioning

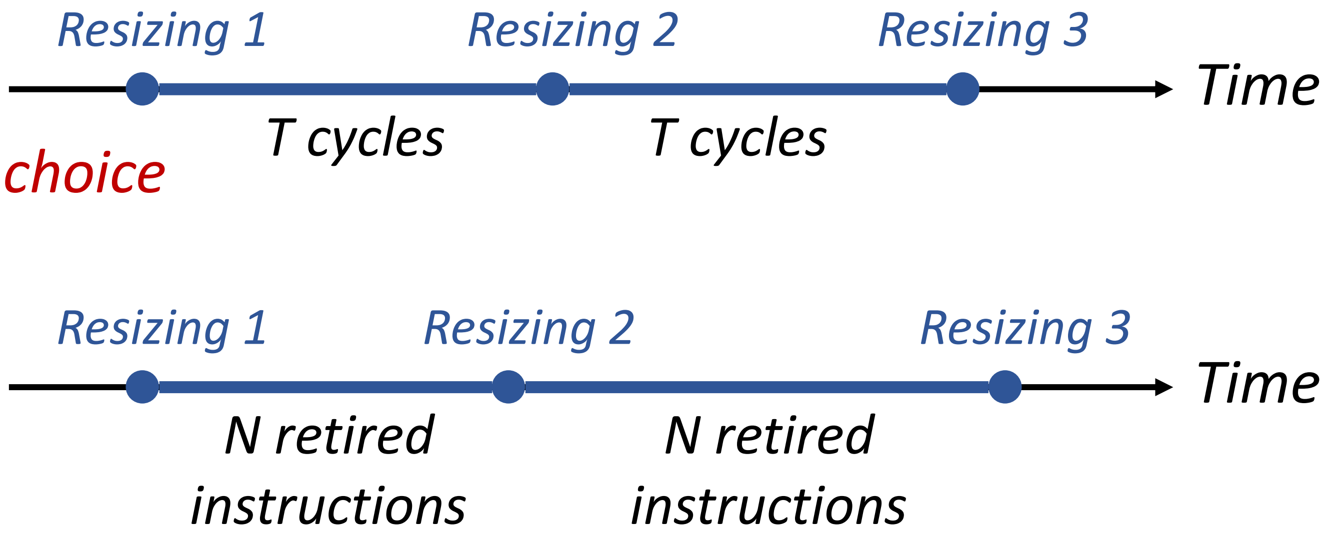
Component 3: Resizing Schedule

Determines *<u>when</u>* to check the utilization and perform the action

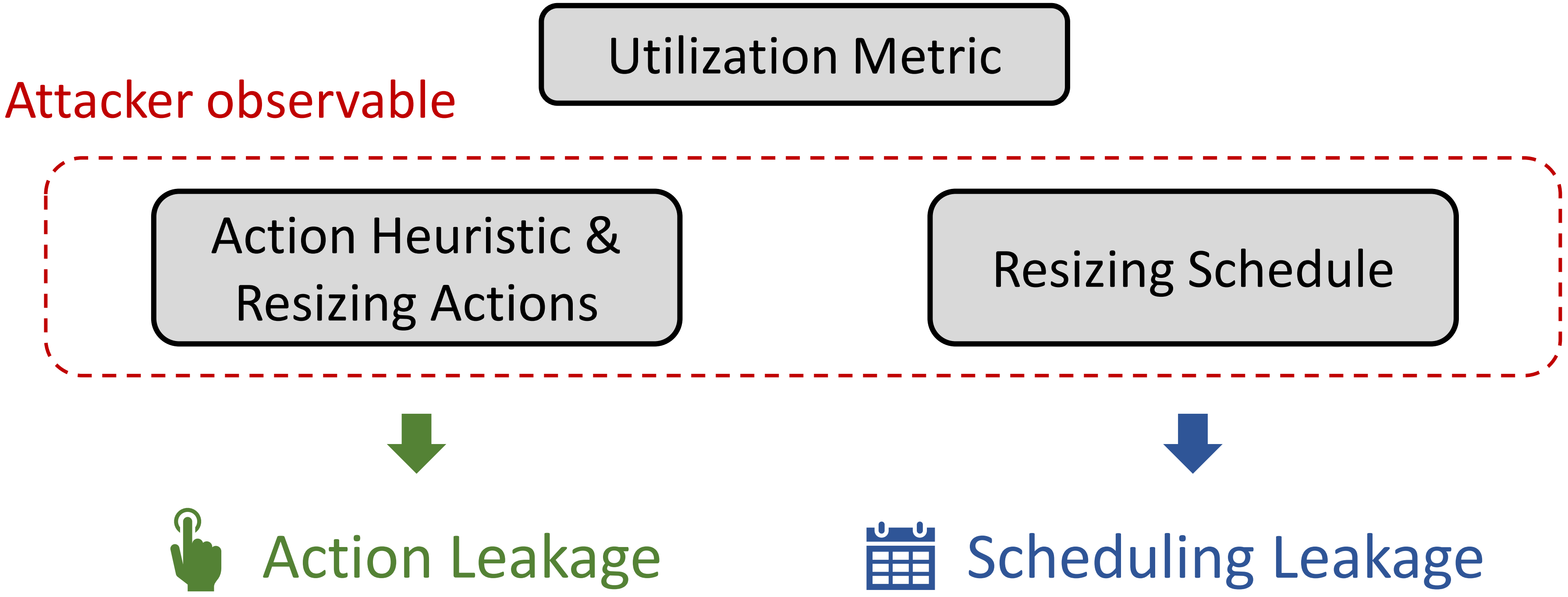
Fixed-time:

↪ *Common choice*

Fixed-progress:

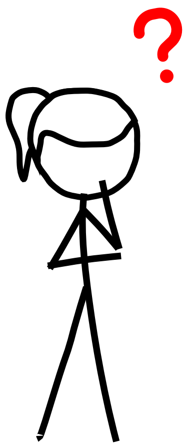


Split the Leakage



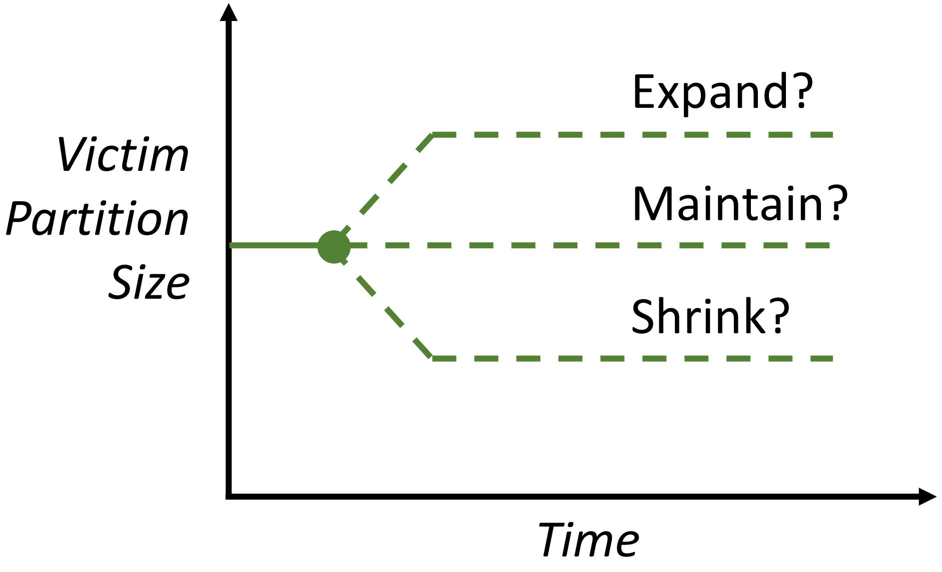
Action Leakage

Secret-dependent demand



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

⇒ check resizing, expand?



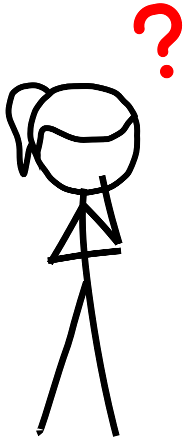
Action Leakage: *what* resizing action to perform

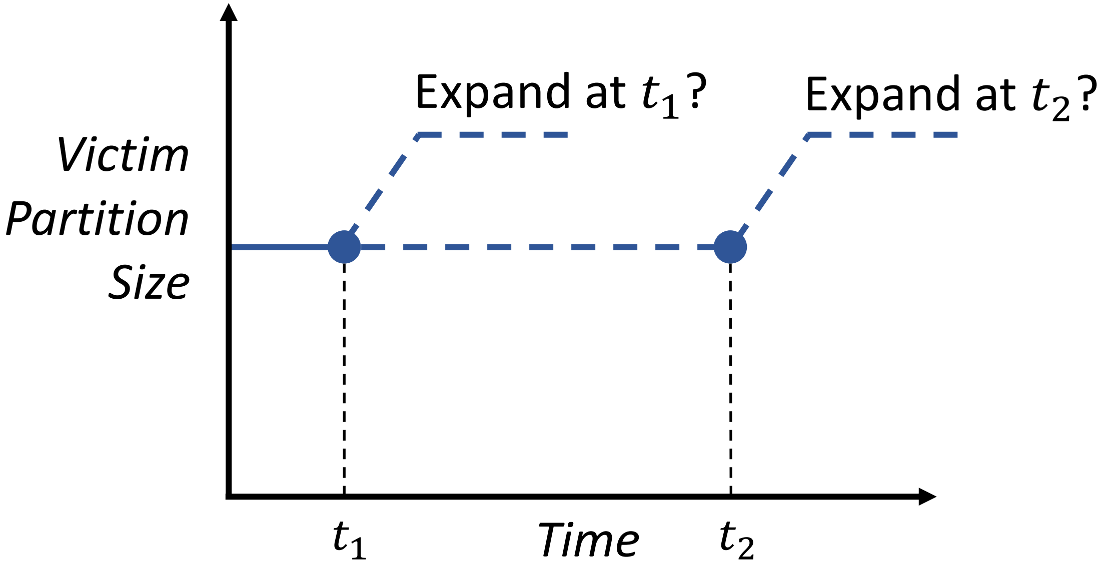
Scheduling Leakage

Secret-dependent timing

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

⇒ check resizing, expand!
```

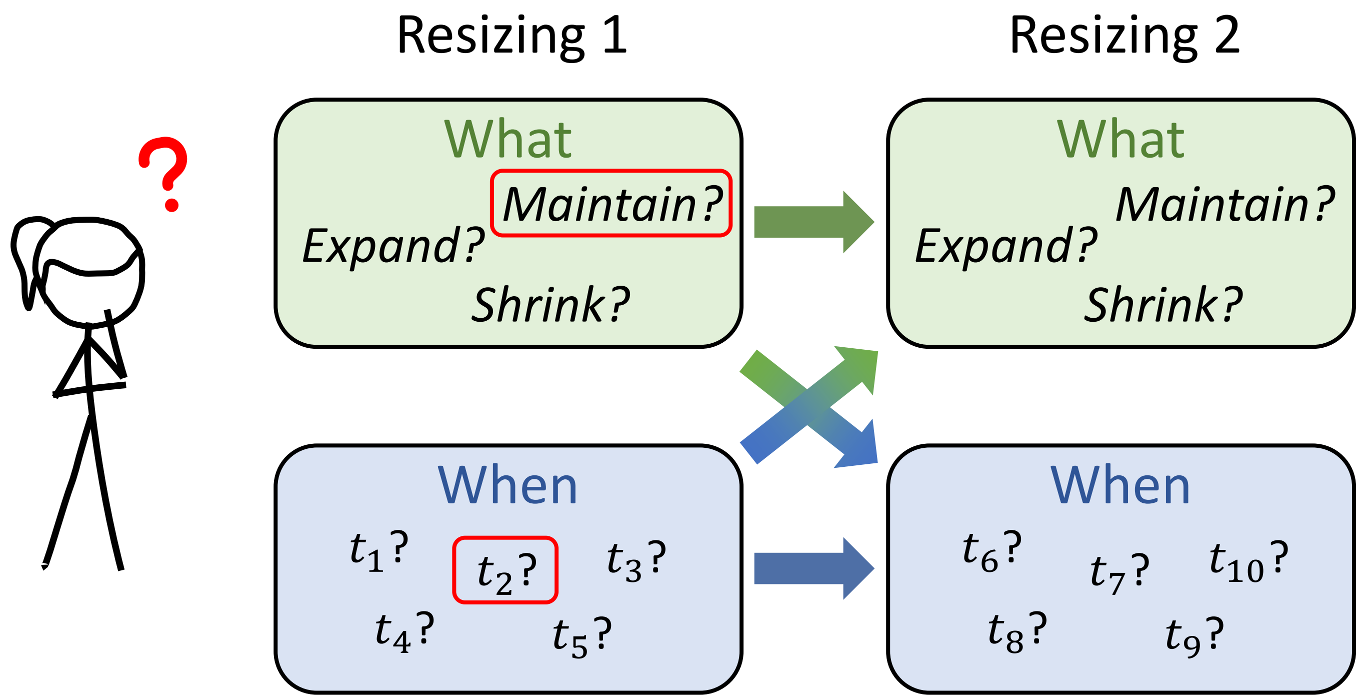




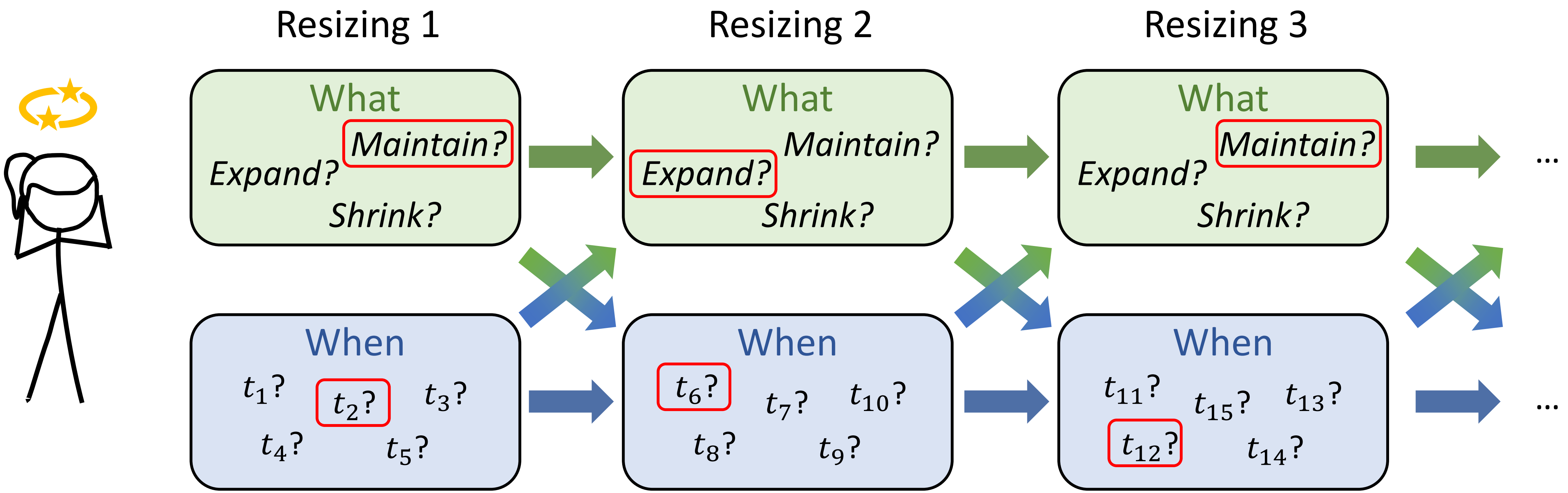
Scheduling Leakage: *when* resizing action occurs

Check out our paper for more details on how we *formally* split the leakage

“What” and “When” are Entangled

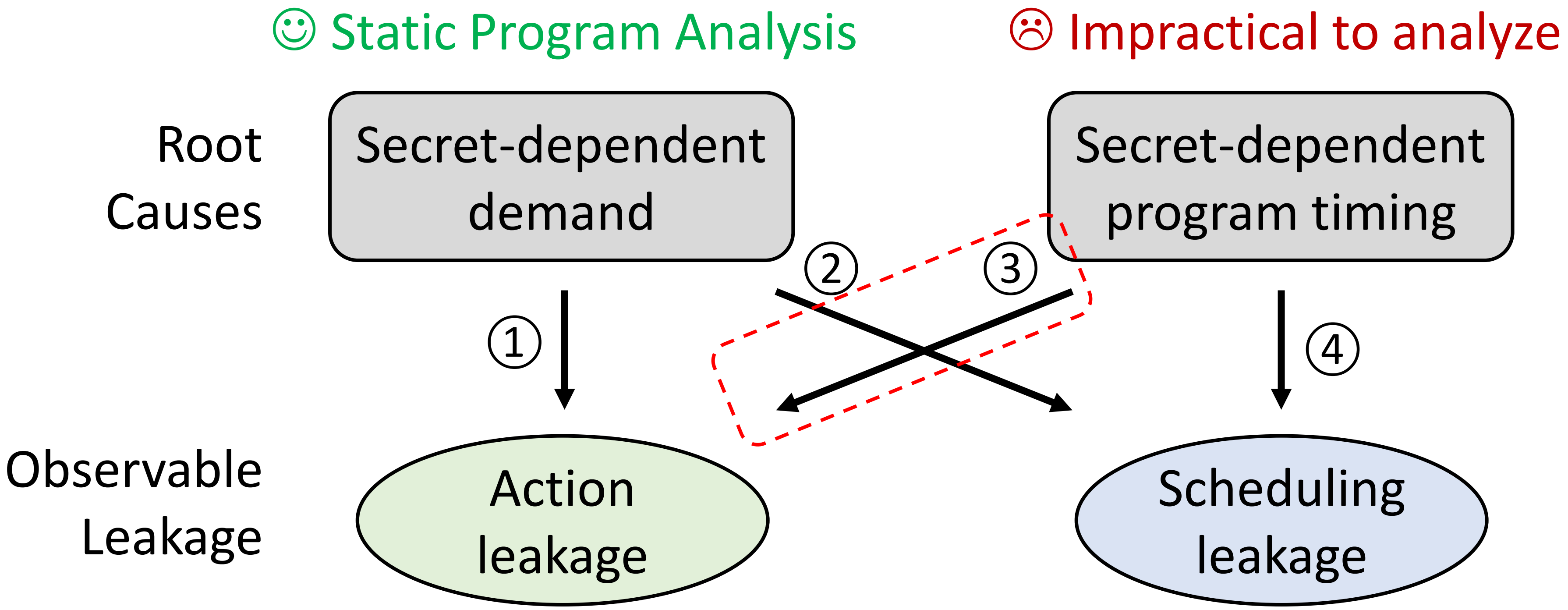


“What” and “When” are Entangled



Hard to analyze!

Untangle It!



Eliminate action leakage

Measure and reduce it without analyzing program timing

Principle 1: Timing-Independent Metric

The value of the metric cannot depend on the actual instruction timing

Example of what is *not* a timing-independent metric for cache:

Number of cache hits in the past *T* cycles

Cache hits are timing-dependent on out-of-order processors

Profiling window is timing-dependent

Principle 1: Timing-Independent Metric

The value of the metric cannot depend on the actual instruction timing

Turning it to a timing-independent metric:

Memory footprint of the past *N retired instructions*

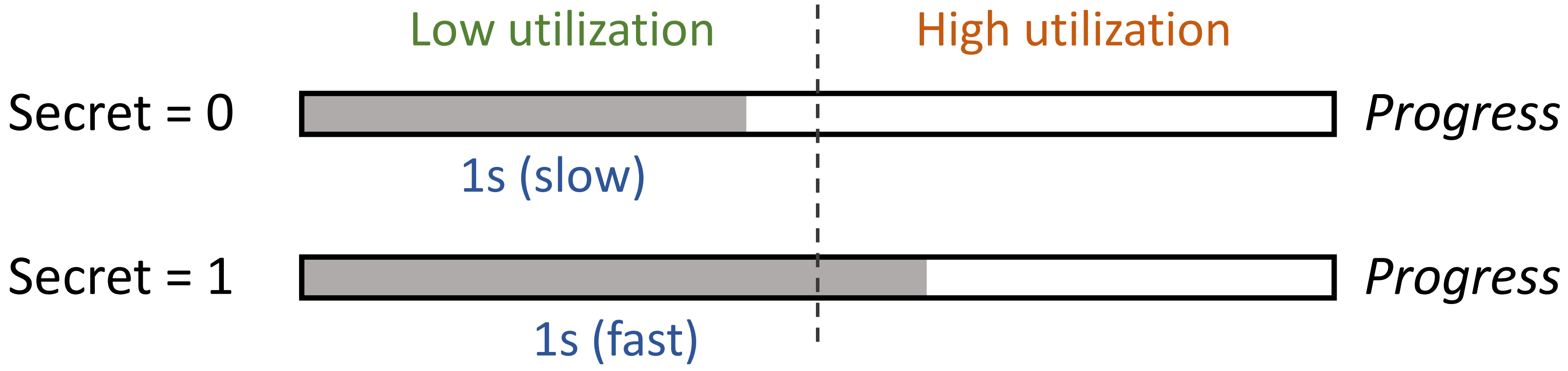
Same value regardless of cache hits or not

Same profiling window regardless how fast the program runs

Principle 2: Progress-Based Schedule

Tie resizing points to when the program has made a certain progress (e.g., every *1B* retired instructions)

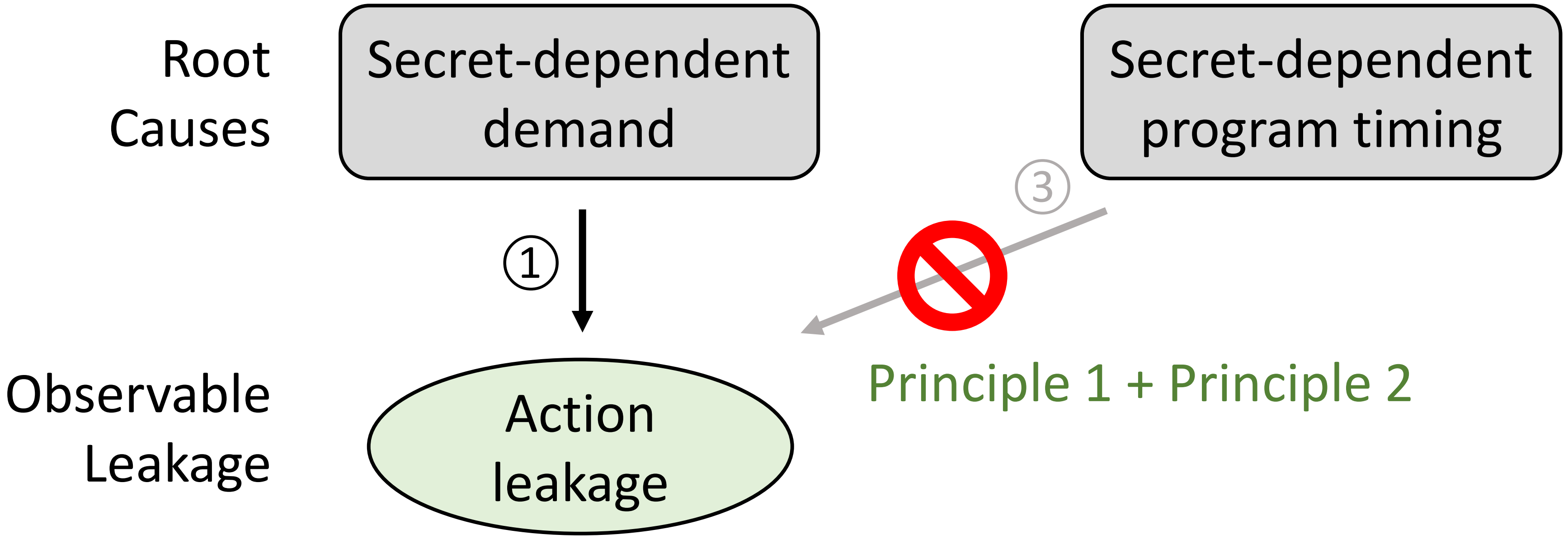
Example of why a time-based schedule fails (e.g., resize after 1s)



☺ Progress-based schedule avoids this problem

Eliminating Action Leakage

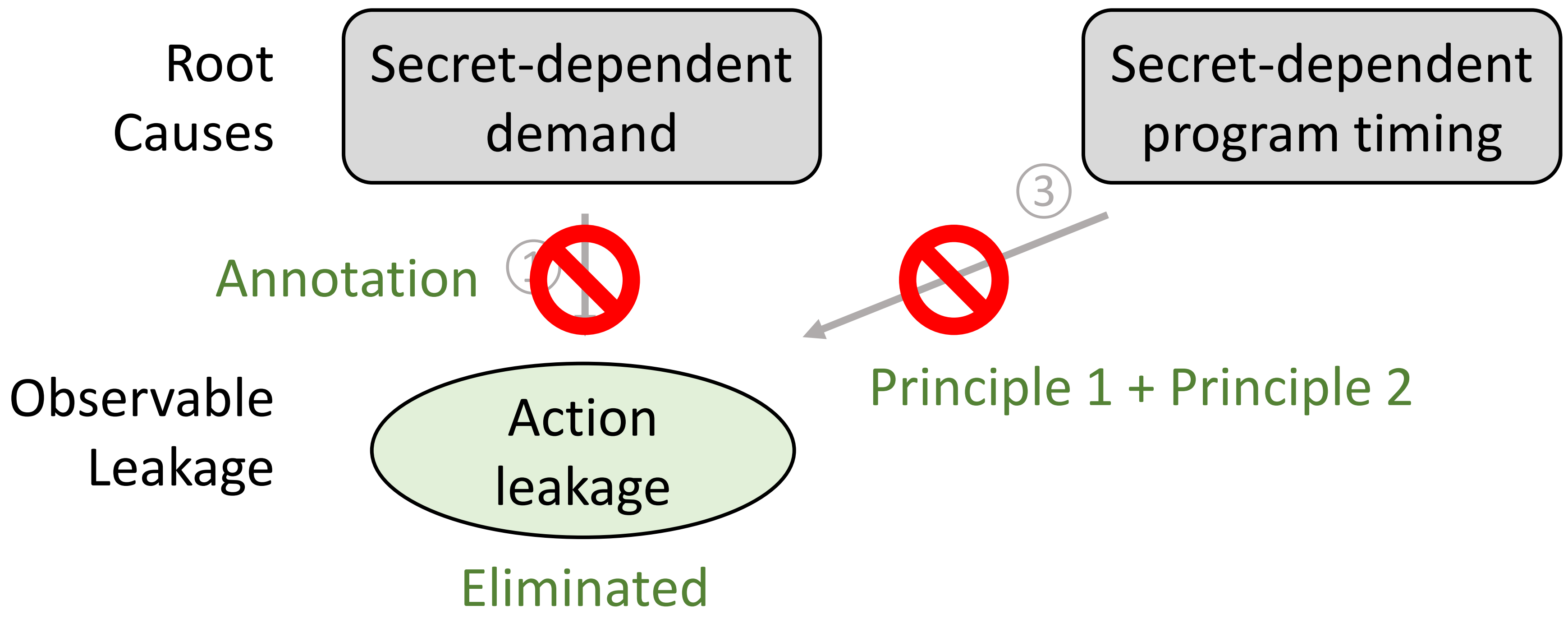
Existing Static Analyses: CacheAudit[1], CaSym[2], etc



[1]Doychev et al., “CacheAudit: A Tool for the Static Analysis of Cache Side Channels” (USENIX Security’13)
[2]Brotzman et al., “CaSym: Cache aware symbolic execution for side channel detection and mitigation” (SP’19)

Eliminating Action Leakage

Existing Static Analyses: CacheAudit[1], CaSym[2], etc

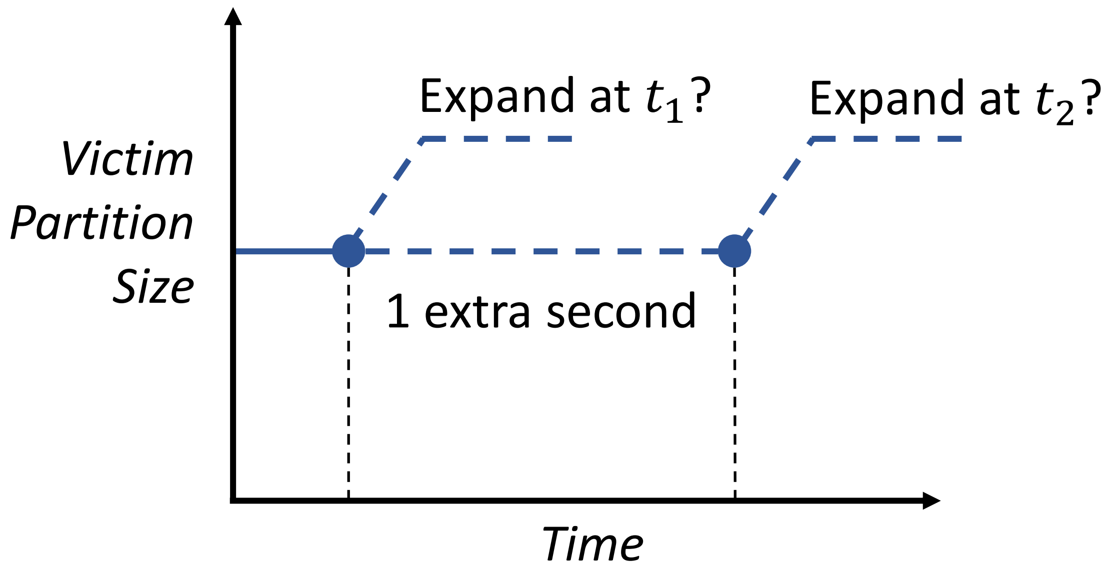


Annotation only helps if the action leakage is timing-independent

[1]Doychev et al., “CacheAudit: A Tool for the Static Analysis of Cache Side Channels” (USENIX Security’13)
[2]Brotzman et al., “CaSym: Cache aware symbolic execution for side channel detection and mitigation” (SP’19)

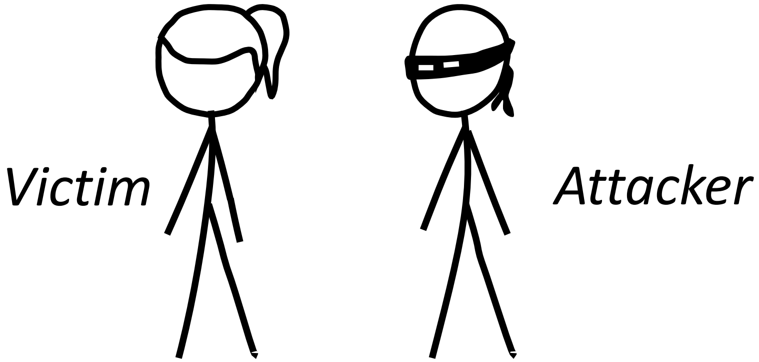
Bound Scheduling Leakage

```
if (secret > 0) {
  sleep(1);
}
// access a large array
⇒ check resizing, expand!
```

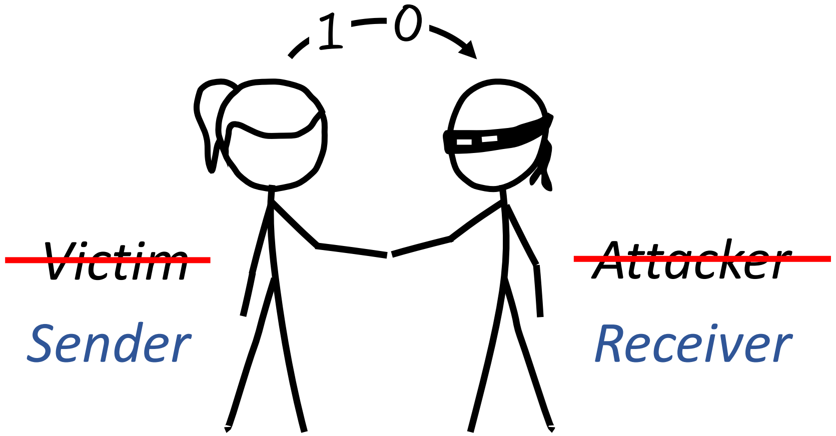


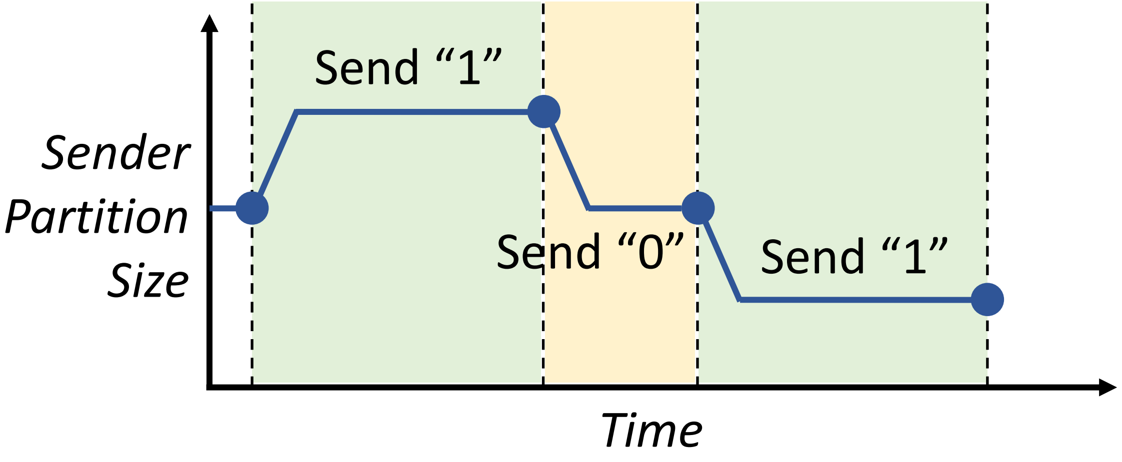
Key Insight: information is encoded as the **duration** of remaining in a certain partition size

Covert Channel



Covert Channel





Victim cooperatively sends message to attacker using the scheduling "leakage"

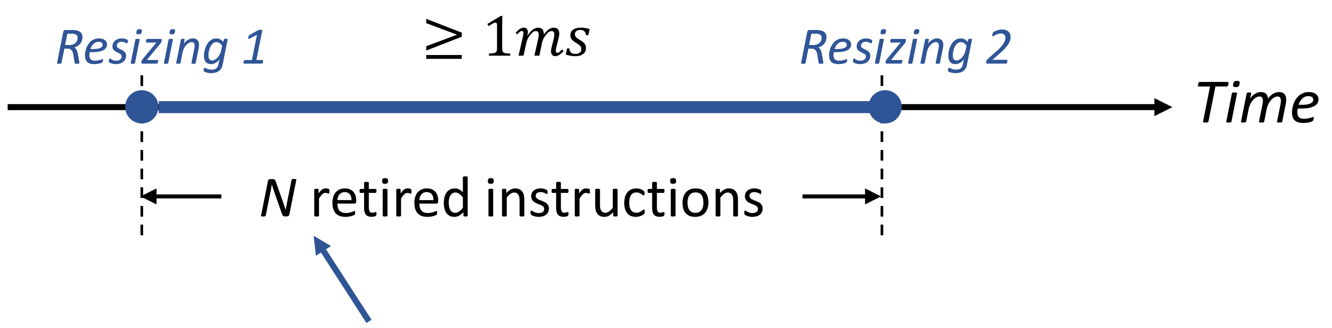
Goal: find the *maximum data rate* between the sender and receiver

↪ *A conservative upper bound of scheduling leakage rate*

☺ Measure and reduce scheduling leakage without analyzing program timing

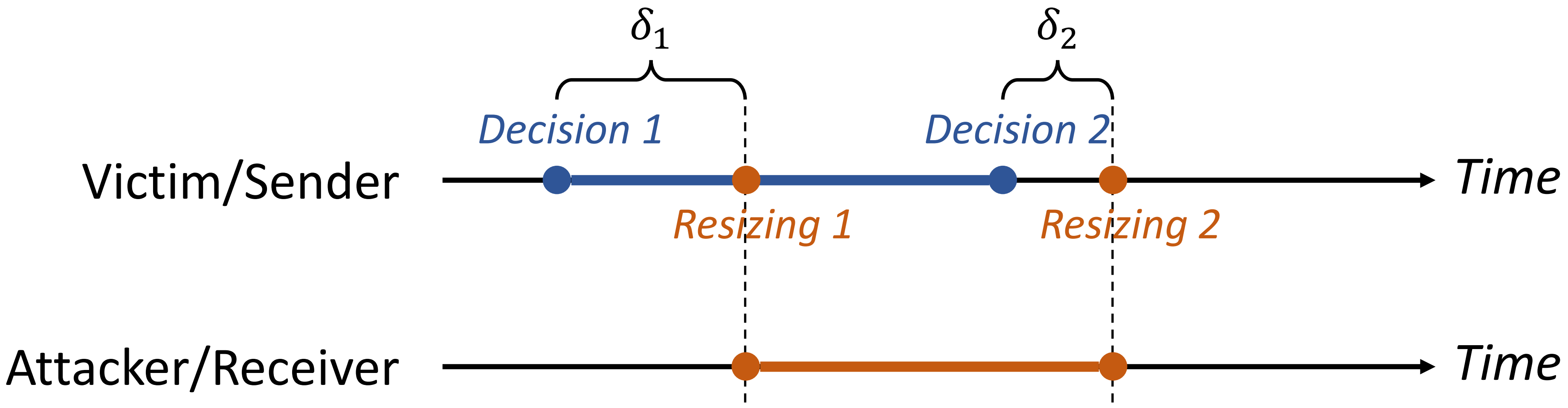
Mechanism 1: Enforce a Cooldown Time

Intuition: set a minimum wait time T_c (e.g., 1ms) between resizes to limit how often the sender can resize



Mechanism 2: Add Random Noise

Intuition: delay each action by a random time δ to disrupt the communication

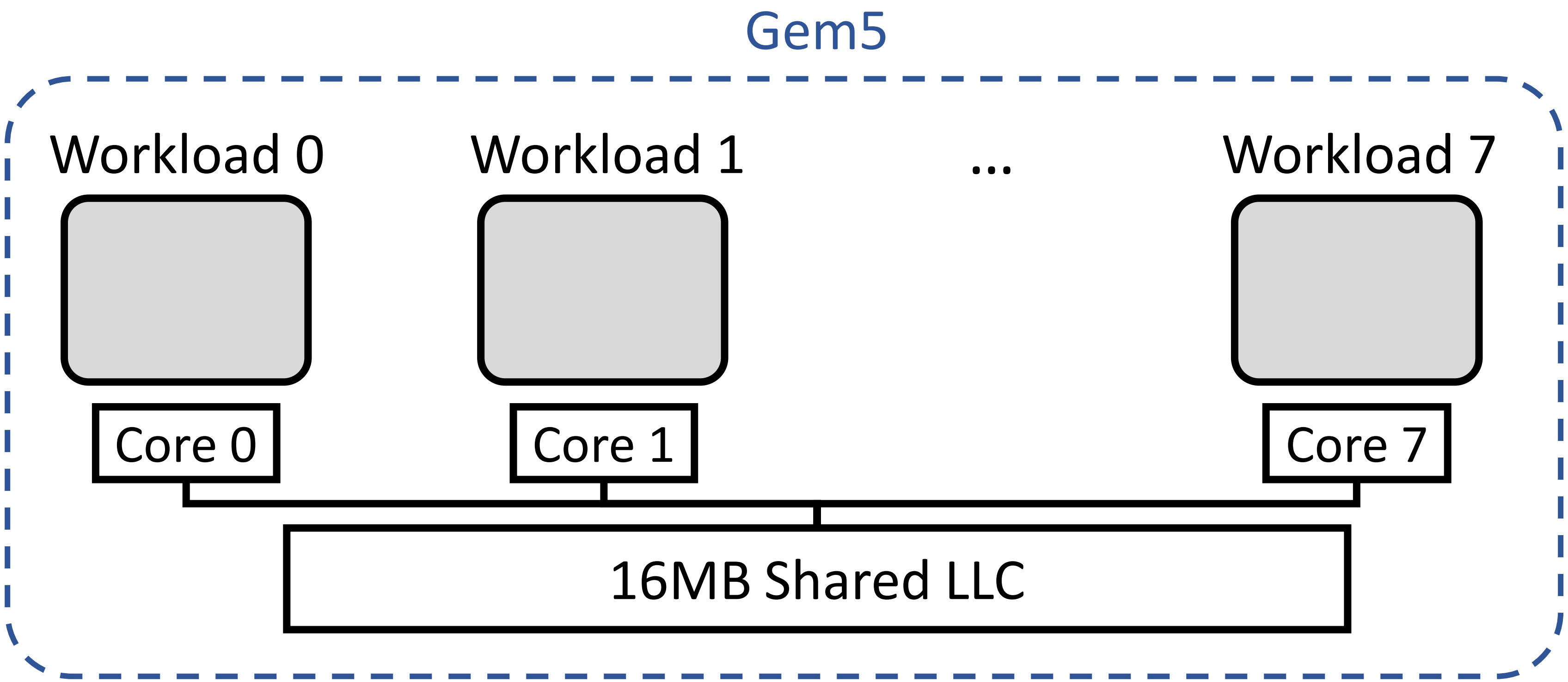


Cause bit errors and reduce the amount of information the attacker learns

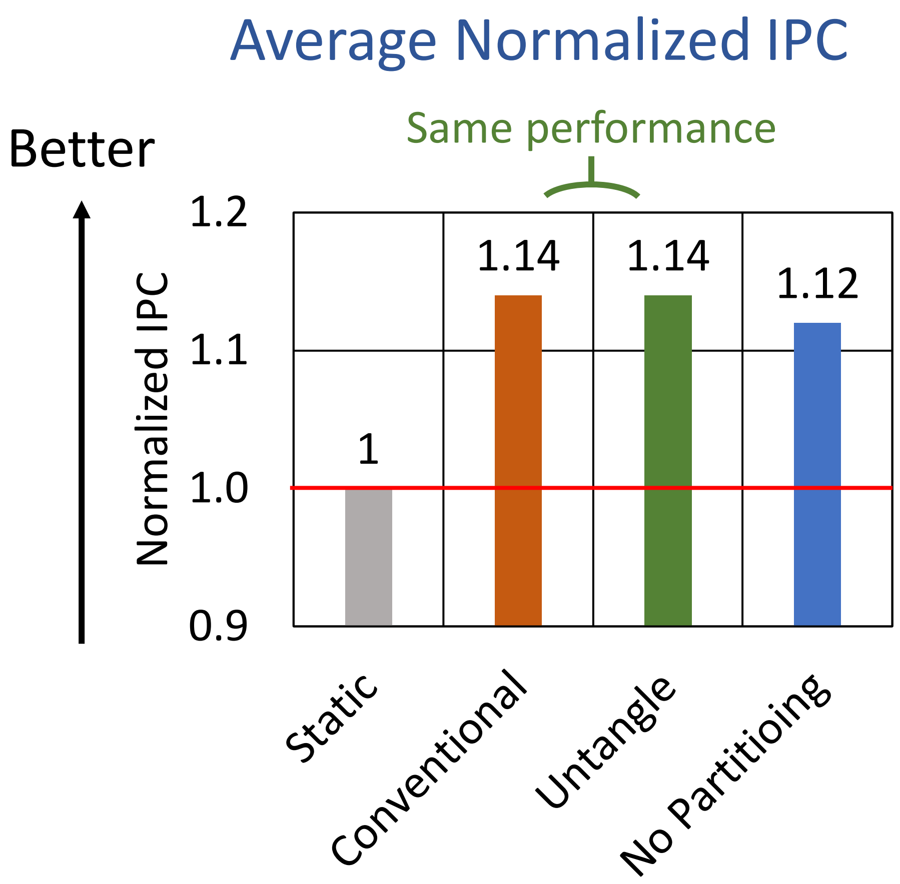
Check out our paper for more details on the covert channel model

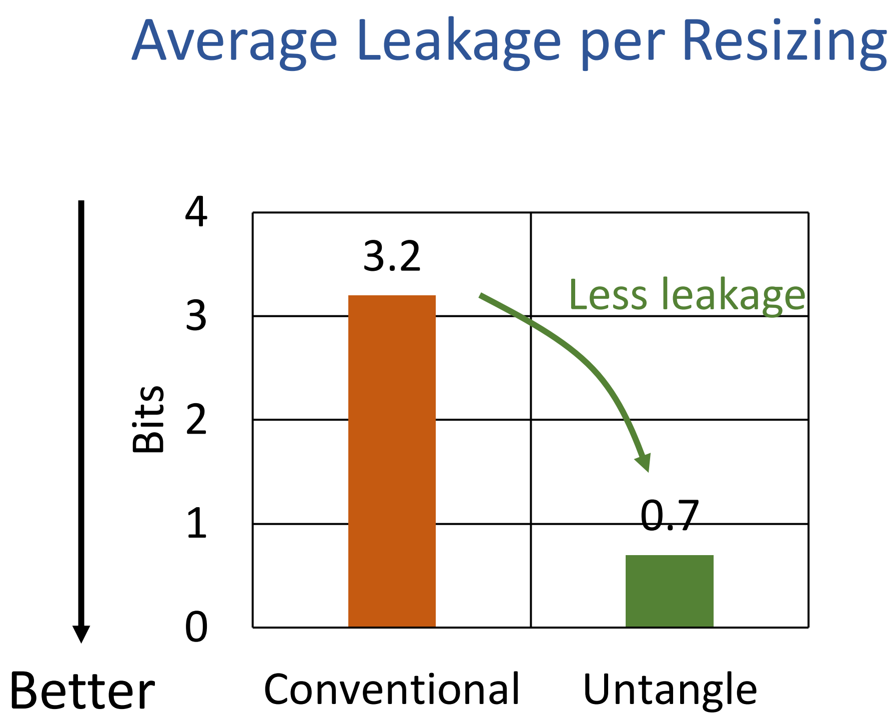
Evaluation Setup

Augment a conventional dynamic last-level cache (LLC) partitioning scheme



Evaluation Results





More resizings under a given leakage threshold

Conclusion

- Untangle is a **general framework** for constructing low leakage, high-performance dynamic partitioning schemes
- **Formally** split the leakage into:

  Action Leakage   Scheduling Leakage

- **Design principles** to *untangle* program timing from the action leakage
- Model the scheduling leakage **without analyzing program timing**
- Applied to dynamic LLC partitioning ⇒ **Same performance, less leakage**

Thanks for Listening!

"Untango"