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

*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](https://xkcd.com/2176) and [https://xkcd.com/1808/](https://xkcd.com/1808/)*
Static Resource Partitioning as a Defense

Victim

Attacker

No Peeking!

Shared Hardware Resources

Resource Starvation

Resource Wastage
Dynamic Partitioning and Its Leakage

Victim

No Peeking!

Attacker

High Demand

Low Demand

Shared Hardware Resources
Dynamic Partitioning and Its Leakage

No Peeking!

Victim

Attacker

Shared Hardware Resources
Dynamic Partitioning and Its Leakage

No Peeking!

Victim

Attacker

Shared Hardware Resources

😊 High Performance
Dynamic Partitioning and Its Leakage

Victim

Shared Hardware Resources

Attacker

I see your expansion

😊 High Performance

😊 Some Information Leakage
Quantify the Leakage

1. Measure information leakage
Quantify the Leakage

1. Measure information leakage
Quantify the Leakage

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 $\Rightarrow$ More resizings under the budget $\Rightarrow$ Better performance
Untangle: Contributions

Our Main Contributions:

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

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

- Core 0
  - Private Caches
- Core 1
  - Private Caches
- Core N
  - Private Caches

![Diagram](image)

Metric: LLC miss rate
Generalized Dynamic Partitioning

Component 2: Action Heuristic

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

- **High** LLC miss rate:
  - Expand the partition
  - Maintain the partition
  - Shrink the partition
Generalized Dynamic Partitioning

Component 3: Resizing Schedule

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

\[\text{Fixed-time:} \quad \text{Resizing 1} \quad \text{Resizing 2} \quad \text{Resizing 3} \quad \text{Time} \]

\[\text{Common choice} \]

\[\text{Fixed-progress:} \quad \text{Resizing 1} \quad \text{Resizing 2} \quad \text{Resizing 3} \quad \text{Time} \]

\[N \text{ retired instructions} \quad N \text{ retired instructions} \]
Split the Leakage

- Utilization Metric
- Attacker observable
- Action Heuristic & Resizing Actions
- Action Leakage
- Resizing Schedule
- Scheduling 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

Resizing 1

What
Expand? Maintain? Shrink?

When
\( t_1? \) \( t_2? \) \( t_3? \) \( t_4? \) \( t_5? \)

Resizing 2

What
Expand? Maintain? Shrink?

When
\( t_6? \) \( t_7? \) \( t_{10}? \) \( t_8? \) \( t_9? \)
“What” and “When” are Entangled

Hard to analyze!
Untangle It!

Static Program Analysis

Root Causes

- Secret-dependent demand

Observable Leakage

- Action leakage

Eliminate action leakage

Secret-dependent program timing

Impractical to analyze

- Scheduling leakage

Measure and reduce it without analyzing program timing

① ② ③ ④
Principle 1: Timing-Independent Metric

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

Example of what is \textit{not} a timing-independent metric for cache:

\textbf{Number of cache hits in the past }T\textbf{ 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)

Secret = 0

Low utilization

1s (slow)

High utilization

Secret = 1

1s (fast)

😊 Progress-based schedule avoids this problem
Eliminating Action Leakage

**Existing Static Analyses:** CacheAudit\(^1\), CaSym\(^2\), etc

- **Root Causes**
  - Secret-dependent demand
  - Secret-dependent program timing

- **Observable Leakage**
  - Action leakage

**Principle 1 + Principle 2**

\(^1\)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

- **Root Causes**
  - Secret-dependent demand
  - Secret-dependent program timing

- **Observable Leakage**
  - Action leakage

- **Annotation**
  - Principle 1 + Principle 2

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)
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

Victim

Attacker
Covert Channel

Victim cooperatively sends message to attacker using the scheduling “leakage”

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

\[ \rightarrow \text{A conservative upper bound of scheduling leakage rate} \]

😊 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

- $Resizing_1 \geq 1ms \rightarrow Resizing_2$

Maximum number of instructions the core can retire in 1ms
Mechanism 2: Add Random Noise

**Intuition:** delay each action by a random time $\delta$ 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

Gem5

Workload 0  Workload 1  ...  Workload 7
Core 0  Core 1  Core 7

16MB Shared LLC
Evaluation Results

**Average Normalized IPC**

- Static: 0.9
- Conventional: 1.14
- Untangle: 1.14
- No Partitioning: 1.12

**Average Leakage per Resizing**

- Conventional: 3.2
- Untangle: 0.7

**More resizings under a given leakage threshold**

**Less leakage**
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!

Tango, pls?

No!...

“Untango”