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

Zirui Neil Zhao
University of Illinois
Urbana-Champaign, USA
ziruiz6@illinois.edu

Adam Morrison
Tel Aviv University
Tel Aviv, Israel
mad@cs.tau.ac.il

Christopher W. Fletcher
University of Illinois
Urbana-Champaign, USA
cwfletch@illinois.edu

Josep Torrellas
University of Illinois
Urbana-Champaign, USA
torrella@illinois.edu

ABSTRACT

Partitioning a hardware structure dynamically among multiple security domains leaks some information but can deliver high performance. To understand the performance-security tradeoff of dynamic partitioning, it would be useful to formally quantify the leakage of these schemes. Unfortunately, this is hard, as *what* partition resizing decisions are made and *when* they are made are entangled.

In this paper, we present *Untangle*, a novel framework for constructing low-leakage and high-performance dynamic partitioning schemes. *Untangle* formally splits the leakage into leakage from deciding what resizing action to perform (*action leakage*) and leakage from deciding when the resizing action occurs (*scheduling leakage*). Based on this breakdown, *Untangle* introduces a set of principles that decouple program timing from the action leakage. Moreover, *Untangle* introduces a new way to model the scheduling leakage without analyzing program timing. With these techniques, *Untangle* quantifies the leakage in a dynamic resizing scheme more tightly than prior work. To demonstrate *Untangle*, we apply it to dynamically partition the last-level cache. On average, workloads leak 78% less under *Untangle* than under a conventional dynamic partitioning approach, for the same workload performance.

CCS CONCEPTS

• **Security and privacy → Side-channel analysis and countermeasures**.

KEYWORDS

Microarchitectural side-channel defense, resource partitioning, information leakage

ACM Reference Format:
Zirui Neil Zhao, Adam Morrison, Christopher W. Fletcher, Josep Torrellas. 2023. Untangle: A Principled Framework to Design Low-Leakage, High-Performance Dynamic Partitioning Schemes. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3582016.3582033

1 INTRODUCTION

Contemporary computer systems have many hardware structures that are shared by multiple processes. They include caches, TLBs, the reorder buffer, the branch predictor, and physical registers. Sharing enhances hardware efficiency and usually improves performance. Unfortunately, it often creates side-channel vulnerabilities (e.g., [1, 6, 21, 31, 32, 58]): a process may glean secret information from the way another process shares the structure with it.

A popular approach to ensure the secure use of a shared hardware structure is to partition it spatially [28]. Each process gets a static partition for the duration of its execution. Hence, information about a process' use of the shared resource cannot leak to processes that own other partitions of the structure. For example, in a shared cache, each process may get one way. While static partitioning is safe, it is undesirable for a dynamic environment where running processes and process resource demands change over time. In such an environment, any static partition is suboptimal and can lead to resource wastage or underprovisioning [51, 54].

An intermediate approach that can retain high performance while leaking a limited amount of information is *dynamic partitioning* [4, 36, 39, 43]. Here, individual processes can increase or decrease the size of their partition dynamically. For example, a process may be allowed to resize its partition at certain times and by a certain amount.

It would be useful to formally quantify the leakage of dynamic partitioning. One could then assess the trade-off between security lost and performance gained. Unfortunately, accurately quantifying the leakage with dynamic partitioning is hard. The precise way to do so is to enumerate all the possible inputs that the victim program can take (including their probabilities) and record all the resulting *Resizing Traces.* A resizing trace is the sequence of resizing actions (e.g., expand the partition, shrink it, or maintain it), and the time of each action. Then, the leakage of the program is calculated as the *entropy* (intuitively, the variability) of these resizing traces [12].

Clearly, this approach does not scale. Furthermore, in today's dynamic partitioning schemes, *what* resizing decisions are made (in 'space') and *when* they are made (in 'time') are *entangled.* For example, *when* a program reaches a given phase and triggers a resize does depend on its rate of forward progress up to that point (time), which in turn is based on previous partition decisions (space), and so on. Since program timing depends on low-level effects such as microarchitectural details, it is typically intractable to analyze. By implication, since the sequence of actions is entangled with timing, the sequence of actions is intractable to analyze.[1]

As a result, state-of-the-art leakage analysis typically assumes the worst case: all the resizing traces that could theoretically occur are realizable and, therefore, at each resizing decision point, all choices are equally likely. The result is *leakage overestimation.* Then, assuming that a user has a fixed leakage budget and that exhausting the budget at runtime will prohibit further resizings [2, 20, 61], overestimating leakage means that fewer partition resizings are allowed before the budget is reached—unnecessarily hurting performance. Therefore, if the leakage could be bound tightly, it would be possible to improve performance.

In this paper, we present *Untangle*, a novel framework for constructing low-leakage and high-performance dynamic partitioning schemes. *Untangle* formally splits the leakage into two parts: (i) leakage from deciding *what* resizing action to perform (*action leakage*) and (ii) leakage from deciding *when* each resizing action occurs (*scheduling leakage*).

Based on this breakdown, *Untangle* makes two advances. First, *Untangle* introduces a set of principles for constructing dynamic partitioning schemes that untangle program timing from the action leakage. As a result, the sequence of resizing actions only depends on the retired dynamic instruction sequence of the execution, and not on timing (e.g., the cycle when each instruction retires). Following these principles, the action leakage can be altogether eliminated with the help of annotations.

In a second advance, *Untangle* introduces a novel way to model the scheduling leakage without analyzing program timing. Overall, with these two contributions, *Untangle* is able to quantify the leakage of a dynamic resizing scheme more tightly than prior work.

The main focus of this paper is on describing the *Untangle* framework rather than presenting a detailed hardware implementation. Still, *Untangle* can be applied to a variety of hardware structures. In our evaluation, we apply it to dynamically partition the last-level cache. For a large set of workloads, we compare *Untangle* to a conventional dynamic partitioning approach. We show that, on average, workloads leak 78% less under *Untangle* than under the conventional dynamic approach, for approximately the same workload performance.

This paper makes the following contributions:

- Proposes *Untangle*, a novel framework for constructing low-leakage and high-performance dynamic partitioning schemes.
- Presents a set of principles to untangle program timing from action leakage.
- Introduces a way to model scheduling leakage without analyzing program timing.
- Applies *Untangle* to dynamic partitioning of the last-level cache and evaluates its performance and leakage under a large set of workloads.

[1]This issue is akin to taint explosion in traditional information-flow systems [30, 45].

2 BACKGROUND

2.1 Microarchitectural Side Channels and Defenses

Microarchitectural side channels rely on hardware resources shared between an attacker and a victim. The attacker exfiltrates a secret by monitoring the victim's secret-dependent utilization of the shared resource. Examples of commonly-exploited shared hardware resources are caches [23, 31, 58], TLBs [21, 44], and functional units [1, 6].

A popular way to defend against side-channel attacks is to partition the shared resource among different security domains. The partition can be *spatial* or *temporal.* Spatial partitioning divides the resource into non-overlapping sections used by different domains (e.g., way-partitioning in caches [28]). A temporal partitioning scheme splits the time into non-overlapping slices, and only one domain is allowed to use the resource in each time slice (e.g., interconnect traffic shaping [16]). When it is not ambiguous, we use the term *partition size* as the portion of the total resource assigned to one domain, regardless of spatial or temporal partitioning.

Depending on the partitioning policy, a scheme can either fix the partition size or dynamically resize it to adapt to a program's demand. The former schemes are *static*, while the latter are *dynamic* (e.g., [36, 39]).

Information leakage through side channels can be detected using a variety of techniques. One approach is to leverage taint analysis [41, 57]. In this case, secret data are annotated as taint sources. Then, taint propagation is used to detect instructions that have secret-dependent usage of the resource of interest, or instructions that are control-dependent on secrets. Other approaches include symbolic execution [3, 8, 50] or abstract interpretation [18, 19, 49]. These specific works formally model the behavior of a cache and find instructions that put the cache into a secret-dependent state.

2.2 Entropy and Mutual Information

Entropy is a quantitative measure of information, represented by the *uncertainty* of a random variable [12]. Let X be a discrete random variable that takes values in $\mathcal{X}$ and $p(x)$ be the probability of $\{X = x\}, x \in \mathcal{X}$. Then the entropy of X is

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log p(x). \tag{2.1}$$

When the log is to the base 2, the entropy is measured in bits. $H(X)$ has the property of $H(X) \leq \log |\mathcal{X}|$, where $|\mathcal{X}|$ is the number of elements in $\mathcal{X}$. The equality is achieved if and only if X follows a uniform distribution over $\mathcal{X}$. Intuitively, the more uniform the distribution of the variable is, the higher the entropy or information carried by the variable is.

The *joint entropy* of two random variables X and Y is

$$H(X, Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x, y), \tag{2.2}$$

where $p(x, y)$ is the probability of $\{X = x, Y = y\}, x \in \mathcal{X} \wedge y \in \mathcal{Y}$. The *conditional entropy* of X given Y is

$$H(X|Y) = -\sum_{x \in \mathcal{X}} \sum_{y \in \mathcal{Y}} p(x, y) \log p(x|y). \tag{2.3}$$

Table 1: Characteristics of some prior dynamic partitioning schemes.

Name	Resource	Utilization Metric	Action Heuristic	Resizing Schedule
UMON [36]	Last-level cache (LLC)	Number of LLC hits under different partition sizes	Pick partition sizes that maximize global LLC hits	Every 5 M cycles
Jigsaw [4]	LLC	Similar to UMON [36]	Peekahead algorithm in software	Every 50 M cycles
Jumanji [39]	LLC	Tail latency of network requests	Compare to static thresholds	Every 100 ms
SecSMT [43]	Pipeline structures shared between SMT threads	Number of "full" events	Increase the partition that has the most "full" events	Every 100 K cycles

The *mutual information* between variables X and Y is the amount of information we learn about one of the two variables when observing the other variable. It is defined by

$$I(X;Y) = -\sum_{x\in\mathcal{X}}\sum_{y\in\mathcal{Y}} p(x,y)\log\frac{p(x)p(y)}{p(x,y)}. \qquad (2.4)$$

$I(X;Y)$ is always *non-negative* and $I(X;Y) = 0$ if X and Y are independent. In all cases, $I(X;Y) = I(Y;X)$.

3 LEAKAGE OF DYNAMIC PARTITIONING SCHEMES

3.1 Generalizing Dynamic Partitioning Schemes

A dynamic partitioning scheme is typically characterized by three components (Table 2). One is the *Utilization Metric* for the resource of interest. This metric reflects a program's demand for the resource and guides resizing. For example, for the last-level cache (LLC), one possible utilization metric is the number of LLC misses per thousand instructions. Typically, improving the utilization metric translates into performance improvements.

Table 2: Components of a dynamic partitioning scheme.

Component	Description
Utilization Metric	Measure of the demand for the resource
Action Heuristic & Resizing Actions	How to pick what resizing action to perform (e.g., Expand, Shrink, Maintain)
Resizing Schedule	When to make a resizing assessment and perform the decided action

Another component is the *Action Heuristic* and the *Resizing Actions* (or *Actions* for short). Resizing actions are scheme-defined operations for adjusting the partition size. Common actions are "expand the partition" (Expand), "shrink it" (Shrink), and "maintain it" (Maintain). More generally, a scheme can define a set of actions, and each action consists of using a given partition size next (e.g., actions can be "set the cache partition size to 1 MB", or "to 2 MB", or "to 4 MB"). The role of the action heuristic is to pick one of the possible actions based on the utilization metric value. For example, an action heuristic is to compare the utilization metric to some utilization thresholds and, based on the result, decide which action to perform. We call this checking and decision process a *Resizing Assessment*.

A final component is the *Resizing Schedule* (or *Schedule* for short). It determines *when* to make a resizing assessment and perform the action. The action decided during the assessment is typically performed immediately, but it can also be performed later. Example schedules are to assess resizing at fixed time intervals or after retiring a fixed number of instructions. The choice of resizing schedule affects a scheme's responsiveness to the program's demands.

Table 1 lists the components of our framework for some prior dynamic partitioning schemes.

3.2 Leakage with Dynamic Partitioning

Dynamically adjusting the partition size of a program based on the program's resource demands can cause information leakage. Secrets can be leaked through *when* resizing assessments are made and *what* resizing actions are taken. More formally, the victim's *Resizing Trace*, which includes the *sequence of resizing actions* and the *timing of each action*, is secret-dependent and can leak information. Using cache partitioning as an example, Figure 1 demonstrates three ways of leaking a secret (similar to the three types of leakage in [5]).

```
if (secret)
   for i in 0..4M // traverse a 4MB array
      access(&arr[i]);
// Resizing assessment, expand?
```

(a) Resizing action depends on the secret through control flow.

```
for i in 0..4M // traverse a 4MB array
   access(&arr[i * secret]);
// Resizing assessment, expand?
```

(b) Resizing action depends on the secret through data flow.

```
if (secret)
   usleep(1000); // sleep for 1ms
for i in 0..4M // traverse a 4MB array
   access(&arr[i]);
// Resizing assessment, will expand
```

(c) Resizing timing depends on the secret.

Figure 1: Code snippets that demonstrate the leakage of a dynamic cache-partitioning scheme.

In Figure 1a, the secret controls the execution of a large-array traversal. If the secret is non-zero, the array is traversed, increasing the cache utilization and causing a partition expansion. The attacker can observe the expansion, hence exfiltrating the secret (Section 4 details our threat model). In Figure 1b, the secret influences the indexes used in the array traversal. Depending on the secret value, the array traversal may access a different number of cache lines, resulting in a different cache utilization and, possibly, a different resizing action. Finally, in Figure 1c, regardless of the secret value, the array traversal always executes and triggers a partition expansion. However, the secret is leaked based on *when* the expansion occurs.

The most accurate way to measure leakage in a dynamic partitioning scheme is to exhaustively enumerate all possible victim program inputs (including their probability) and the resulting resizing traces under the partitioning scheme. These are the set of resizing traces that are *realizable*. Then, the leakage of the program is calculated as the entropy of these traces using Equation 2.1. This is the amount of information that the victim program leaks under this scheme. Unfortunately, although this approach is accurate, it is not feasible in practice.

3.3 Limitations of Prior Work

To mitigate the leakage in dynamic partitioning schemes, most prior work either coarsens the granularity of resizing (i.e., resizes less frequently or reduces the number of possible actions), or fixes the timing of resizing actions to a publicly-known schedule [2, 16, 20, 61]. As a result, these approaches reduce the leakage at the cost of losing some of the adaptivity of dynamic schemes. Fundamentally, they trade off performance for better security.

Making matters worse, prior schemes often *overestimate* the leakage from resizing by implicitly assuming that all the resizing traces that could theoretically occur are realizable. For example, consider a dynamic partitioning scheme that makes resizing assessments every one millisecond and supports two resizing actions. In a one-second execution of the program, the scheme will make 1000 resizing assessments. Since the timing of the assessments is fixed at multiples of one millisecond, the leakage purely comes from what action is taken at each assessment. Hence, a common but over-conservative estimation is that the scheme can produce *any* of the 2^{1000} different traces in a one-second execution and that all traces have the same probability of occurring. Hence, the leakage is computed using Equation 2.1 to be $\log 2^{1000} = 1000$ bits, which is too conservative for most programs.

Overall, this conservative assumption results in further restricting the adaptivity allowed: given a target leakage budget, the budget will be consumed sooner because of the leakage overestimation, which will prohibit further resizings. The end result is to render dynamic schemes less appealing.

3.4 Our Approach and Challenges

An intuitive idea to reduce the leakage of dynamic resizing schemes is to make the scheme aware of which data is public and which is secret. Then, resizing assessments that only depend on public data can be performed without leaking secret information. Consequently, the scheme gains more resizing flexibility—which maximizes performance without impacting security.

Consider Figures 1a and 1b again. If the scheme knows that the array traversal is secret-dependent (e.g., through annotations inserted by the side-channel detection tools of Section 2.1), it can conveniently exclude the *secret-dependent cache demand* when measuring the cache utilization metric. Hence, the resulting resizing trace depends on only public cache utilization and does not reveal the secret.

However, these annotations alone cannot remove the leakage in Figure 1c. This is because the secret causes the *public* memory accesses to have *secret-dependent timing*. Note that the secret-dependent timing can manifest not only through *when* a resizing assessment is made, but also through *what* resizing action is taken at an assessment. To see how, consider Figure 1c but with a resizing schedule that makes an assessment at 1 ms (in contrast to making an assessment after the array traversal at Line 5). In this case, depending on the secret value, the point of assessment may be before or after the public array traversal, resulting in a different cache utilization and, consequently, a different resizing action.

Unfortunately, extending existing side-channel detection tools for this type of implicit flow through program timing is very hard—given the impracticality of fully determining program timing with modern processors and taint explosion in traditional information-flow systems [30, 45]. As a result, in the case when the victim has secret-dependent timing (which is the general case), it is hard to bound the leakage any tighter than the conservative approach discussed above does.

To address this problem, in Section 5, we build a novel framework named *Untangle* that helps reason about this entanglement of action and timing. Based on the framework, we develop design principles for dynamic resizing schemes and mitigations to achieve a tight bound on the leakage. *Untangle* enables us to attain high performance without compromising security.

4 THREAT MODEL

We consider a mutually-distrusting peer model where the attacker and the victim are in the same security level. The attacker and the victim share a hardware resource (e.g., a cache). The system can partition the resource into attacker and victim partitions. The system can dynamically change the partition sizes based on the resource utilization, which can be secret-dependent and reveals sensitive information of the victim.

We assume an idealized attacker that can directly observe the victim's exact resizing trace (i.e., what resizing actions are taken and when). In practice, an attacker can only indirectly estimate the victim's resizing trace by probing its own partition size and observing how it changes over time as a result of victim resizes. This estimation is not completely accurate because neither the resizing actions nor the attacker's probing are instantaneous. Hence, a realistic attacker would be less capable than the idealized attacker that we are assuming. Lastly, we assume that the partition scheme does not change utilization metric, action heuristic, resizing actions, or resizing schedule in the course of the victim's execution.

The victim sets a threshold for how much leakage from the victim program's run or runs is tolerable. The dynamic partitioning scheme (i.e., *Untangle*) measures the runtime leakage and *guarantees* it cannot exceed this threshold. If and when the threshold is reached, the victim is not allowed to perform further resizings—hurting the performance of its subsequent execution, *but not its security*.

We assume that there are sound approaches to annotate programs with secret-dependent usage of the resource being partitioned and secret-dependent control-flow. This is achievable with existing analyses [3, 8, 18, 49], or with a conservative approach that annotates all the instructions from the part of the program that handles secrets. Section 6.5 discusses the capabilities of these existing analyses.

5 *UNTANGLE*: SECURE DYNAMIC PARTITIONING

Recall from Section 3.1 that a dynamic partitioning scheme uses a resizing schedule to decide when to perform resizing assessments, and an action heuristic to decide what resizing actions to take. As a result, secrets are leaked through the observation of "when" and "what" actions are taken. While annotating instructions that have secret-dependent resource usage can help reduce the leakage in some special cases, annotations have limited use in general programs due to secret-dependent timing (Section 3.4).

In this section, we present *Untangle*, a novel framework that quantifies the leakage in a dynamic partitioning scheme with a tight bound. *Untangle* formally splits the leakage into two parts: (i) leakage from deciding what resizing action to perform (*action leakage*) and (ii) leakage from deciding when each resizing action occurs (*scheduling leakage*). Based on this breakdown, *Untangle* makes two contributions. First, it introduces a set of principles to disentangle program timing from the action leakage, and eventually remove the action leakage with annotations. Second, *Untangle* introduces a novel way to tightly-bound scheduling leakage without analyzing program timing.

Figure 2 shows a diagram with the action and scheduling leakages, and how both are affected by the same two root causes: secret-dependent demand and secret-dependent timing. In this section, we describe how *Untangle* allows us to untangle the different effects. First, Section 5.1 shows how we formally separate the two types of leakage. Then, Section 5.2 shows how we can eliminate action leakage. Finally, Section 5.3 presents an easy way to bound scheduling leakage. The result is a tight bound estimation of the total leakage in a dynamic partitioning scheme.

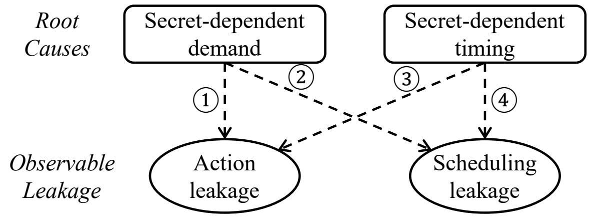


Figure 2: Action and scheduling leakages and their root causes. *Untangle* is able to eliminate Edges ① and ③, and simplify the analysis for Edges ② and ④.

5.1 Decoupling the Two Types of Leakage

Recall from Section 3.2 that a resizing trace is a sequence of tuples, where each tuple contains a resizing action and the time of the action. Further, the leakage of a specific victim program is the entropy of the realizable resizing traces for the program.

To understand how we decouple the leakages for a given program, let S be a discrete random variable that represents a sequence of resizing actions. S takes values in a set $\mathcal{S}$. If we denote the set of supported resizing actions by $\mathcal{A}$, then an action sequence $s \in \mathcal{S}$ is $a_1, a_2, ..., a_n$, where a_i is the ith action in the sequence, and $a_i \in \mathcal{A}$. Note that S only contains what resizing actions are taken but not when.

For an action sequence $s \in \mathcal{S}$, let T_s be a discrete random variable that represents the timing of action sequence s. T_s is a sequence of strictly-increasing timestamps $t_1, t_2, ..., t_n$, where t_i is the timestamp when the ith action occurs. T_s takes values in $\mathcal{T}[s]$. Without loss of generality, we assume that these timestamps have a finite resolution and therefore represent them as integers. Under a fixed time-interval resizing schedule, s has only one possible T_s (i.e., $|\mathcal{T}[s]| = 1$). However, under a more general resizing schedule, s can have many different T_s (i.e., $|\mathcal{T}[s]| > 1$). An example of one such resizing schedules is to make a resizing assessment every N retired instructions. T_s varies because the time to retire N instructions and then trigger an assessment depends on program timing.

We use tuple (S, T_S) to denote a random variable that represents the resizing trace. (S, T_S) takes values in $\{(s, \tau_s) \mid s \in \mathcal{S} \wedge \tau_s \in \mathcal{T}[s]\}$. Therefore, the leakage L, which is equal to the entropy of the realizable resizing traces, is the joint entropy of S and T_S (using Equation 2.2):

$$L = H(S, T_S) = -\sum_{s \in \mathcal{S}} \sum_{\tau_s \in \mathcal{T}[s]} p(s, \tau_s) \log p(s, \tau_s), \qquad (5.1)$$

where $p(s, \tau_s)$ is the probability of following a specific action sequence s with a specific timing sequence τ_s.

By the chain rule of joint entropy [12], we can rewrite Equation 5.1 as:

$$\begin{aligned} L = H(S, T_S) &= H(S) + H(T_S|S) \\ &= H(S) - \sum_{s \in \mathcal{S}} \sum_{\tau_s \in \mathcal{T}[s]} p(s, \tau_s) \log p(\tau_s|s) && (5.2) \\ &= H(S) - \sum_{s \in \mathcal{S}} \sum_{\tau_s \in \mathcal{T}[s]} p(s) p(\tau_s|s) \log p(\tau_s|s) && (5.3) \\ &= H(S) + \sum_{s \in \mathcal{S}} p(s) \underbrace{(-\sum_{\tau_s \in \mathcal{T}[s]} p(\tau_s|s) \log p(\tau_s|s))}_{\text{denoted by } H(T_s|S=s)} && (5.4) \\ &= H(S) + \sum_{s \in \mathcal{S}} p(s) H(T_s|S = s) && (5.5) \\ &= H(S) + E[H(T_s|S = s)]. && (5.6) \end{aligned}$$

Equation 5.2 applies the definition of conditional entropy from Equation 2.3. The term over the bracket in Equation 5.4 is the entropy of the timing sequences in a specific action sequence s, which is denoted as $H(T_s|S = s)$ in Equation 5.5. The second term in Equation 5.5 is the expected value of $H(T_s|S = s)$ for every possible action sequence s.

Equation 5.6 shows that the leakage is composed of two simple terms: (1) $H(S)$ is the entropy of resizing action sequences, which we call *action leakage*; and (2) $E[H(T_s|S = s)]$ is the expected value of the entropy of timing sequences T_s for every possible action sequence s, which we call *scheduling leakage*.

Example. Figure 3 illustrates the computation of leakage by decoupling action and scheduling leakages. For this example, assume a dynamic partitioning scheme with two supported resizing actions, Expand and Maintain, and three realizable traces. These three traces have two unique action sequences: (1) s_1 =

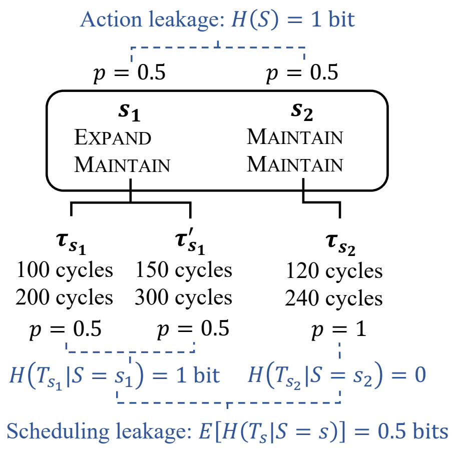


Figure 3: An illustration of decoupling the leakage.

EXPAND, MAINTAIN (i.e., s_1 performs EXPAND and then MAINTAIN), and (2) s_2 = MAINTAIN, MAINTAIN. Both sequences are equally likely (i.e., $p(s_1) = 0.5$ and $p(s_2) = 0.5$). Sequence s_1 has two equally probable timing sequences, τ_{s_1} = 100 cycles, 200 cycles and τ'_{s_1} = 150 cycles, 300 cycles. Sequence s_2 has only one possible timing sequence, τ_{s_2} = 120 cycles, 240 cycles.

With our framework, the action leakage is the entropy of the resizing action sequences. Since there are two resizing action sequences in total and they are equally likely, the action leakage $H(S)$ is $-(0.5 \log 0.5 + 0.5 \log 0.5) = 1$ bit. As for the scheduling leakage, since sequence s_1 has two equally likely timing sequences, $H(T_{s_1}|S = s_1) = 1$ bit; further, since sequence s_2 has only one possible timing sequence, $H(T_{s_2}|S = s_2) = 0$. Therefore, the scheduling leakage is $E[H(T_s|S = s)] = p(s_1)H(T_{s_1}|S = s_1) + p(s_2)H(T_{s_2}|S = s_2) = 0.5$ bits. In total, these three traces leak $L = H(S) + E[H(T_s|S = s)] = 1.5$ bits.

5.2 Eliminating Action Leakage

The action leakage $H(S)$ can be caused by both secret-dependent demand and secret-dependent timing (Edges ① and ③ in Figure 2). Unfortunately, it is challenging to reduce the bounds on the action leakage due to the impracticality of analyzing secret-dependent program timing (Section 3.4). To make the action leakage independent of program timing and, therefore, remove Edge ③, we propose two design principles. With these two principles, the action sequence will only depend on the retired dynamic instruction sequence in the execution, but not on program timing. Then, we will discuss how to remove Edge ① with annotations. By removing both edges, we have completely eliminated the action leakage.

Principle 1: Use a *timing-independent metric* to measure the resource utilization. A timing-independent metric means that it only depends on the architectural semantics of the executed program, such as its retired dynamic instruction sequence, and not on the actual instruction timing. An example of what is *not* a timing-independent metric for caches is the number of cache hits in the past T cycles (similar to the metric used in [36]). This metric is not timing-independent for two reasons. First, the performance statistic, i.e., the number of cache hits, is timing-dependent in modern out-of-order processors. The reason is that the program timing can change the order of memory accesses, resulting in different cache states and affecting the number of cache hits. Second, the profiling history included in the window of T cycles is also timing-dependent.

To define a timing-independent metric, we must only use timing-independent performance statistics. Also, if the metric is defined on a window of execution, the history included in the window must not depend on program timing. In the example of cache partitioning, a timing-independent metric can be the memory footprint (i.e., the number of unique memory lines accessed) of the past N *retired* memory instructions, regardless of what level in the cache hierarchy the memory requests were served from.

Principle 2: Use a resizing schedule based on the progress of instruction execution (or "progress-based schedule" for short). This means that we tie the assessment points to when the program has made a certain progress (e.g., after N retired instructions)—not to when a certain time has elapsed. The reason is that if assessment points are tied to elapsed time, e.g., making an assessment after T cycles, then the utilization metric value at the point of assessment depends on what instructions the program can execute in T cycles, which depends on program timing. As a result, secret-dependent timing can still influence the resizing action taken at an assessment, even if a timing-independent metric is used. Figure 4 illustrates a time-based schedule that assesses at every T cycles and a progress-based schedule that assesses at every N retired instructions.

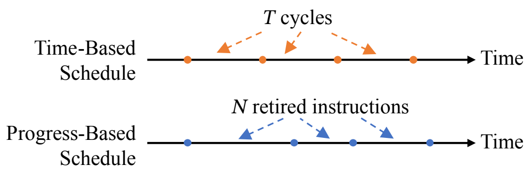


Figure 4: Comparison of a time-based schedule (used by prior work [4, 36, 39, 43]) and a progress-based schedule. Dots on the timelines are the times when assessments occur.

By following these two design principles, the resizing action sequence becomes timing-independent—i.e., it only depends on the sequence of retired dynamic instructions in an execution. This removes Edge ③ in Figure 2. With this property, if we can additionally ensure that the action sequence only depends on the *public* portion of the dynamic instruction sequence, we can also remove Edge ① and, therefore, completely eliminate the action leakage.

To make the action sequence only dependent on the *public* portion of the instruction sequence, we annotate all the instructions that use the resource under partitioning and are data- or control-dependent on secrets. Then, when measuring the utilization metric, we *exclude their contribution.* We also annotate any instructions that are control-dependent on secrets, irrespective of whether they use the resource. Then, the execution of these instructions is *not counted towards the execution progress.* Prior program analyses [3, 8, 18, 49] can be applied to find and annotate these two kinds of instructions. With this support, regardless of the values of secret inputs, the point in the execution where an assessment is made and the utilization metric value at that assessment point are independent of the said secrets. This removes Edge ①. Overall, the action sequence is now secret-independent. This means that, for a given public input, there is only one possible realizable action sequence s regardless of the secret inputs. Therefore, we have eliminated the action leakage.

5.3 Bounding Scheduling Leakage

The scheduling leakage $E[H(T_s|S = s)]$ is the expected value of the entropy of timing sequences T_s, for every possible action sequence s. Since there is only one possible action sequence s for a given public input due to the elimination of the action leakage in Section 5.2, then $E[H(T_s|S = s)] = H(T_s|S = s)$. Hence, the following discussion focuses on bounding $H(T_s|S = s)$ for the specific action sequence s that occurs at runtime for the given public input.

If we tied assessment points to elapsed time (e.g., a resize every T cycles), $H(T_s|S = s) = 0$ for any action sequence s and the scheme would not have scheduling leakage. But then it would have timing-dependent action leakage (Section 5.2). If, instead, we use a progress-based resizing schedule as discussed in Section 5.2, we eliminate the action leakage with the help of annotations. However, we still have timing-dependent scheduling leakage: when an assessment occurs leaks how much time the program takes to make a certain amount of execution progress.

It may seem that no matter whether a scheme ties assessment points to elapsed time or to progress, one cannot avoid analyzing program timing. To solve this problem, we propose a covert channel model that enables us to bound the *worst-case* scheduling leakage in an environment with a progress-based resizing schedule, without analyzing program timing. With this approach then, we have no action leakage and can compute a tightly bound of the scheduling leakage.

A covert channel assumes that both the sender (i.e., victim) and the receiver (i.e., attacker) are cooperative, while a side channel assumes that the sender is non-cooperative. Therefore, computing *the maximum data rate* of the more capable covert channel produces an upper bound of the scheduling leakage that occurs in the real environment with a non-cooperative victim. Overall, with this approach, we do not need to find exact realizable timing sequences nor consider Edges ② and ④ in Figure 2.

Next, we describe the model for the covert channel, the bound on the scheduling leakage, and optimizations to reduce the leakage. We assume a progress-based resizing schedule and that we have already eliminated the action leakage with annotations.

5.3.1 *Understanding the Covert Channel.*

We observe that the leaked information is encoded as the *duration* of remaining in a certain observable state (i.e., using a certain partition size). To illustrate this observation, we revisit the code snippet in Figure 1c. The code snippet always decides to EXPAND after finishing the array traversal, but the timing of EXPAND is secret-dependent, as shown in Figure 5. Therefore, the example can be modeled as a covert channel that changes the current state (i.e., performs EXPAND) after t ms to transmit a symbol "0", or after $t + 1$ ms to transmit a symbol "1".

The sender can try various transmission strategies to amplify the leakage. For example, the sender can use more than two input symbols per transmission to increase the amount of data being

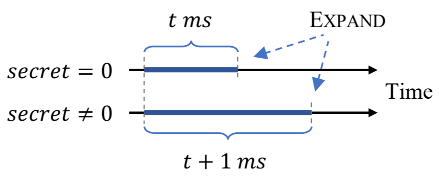


Figure 5: Timing of EXPAND for the code snippet in Figure 1c.

transferred each time. Each symbol will be assigned a different time duration. The sender can also increase the time duration differences between symbols, to make the channel more resilient to potential noise—e.g., instead of using t ms and $t + 1$ ms to encode "0" and "1", one can use t ms and $t + 10$ ms to make "0" and "1" more distinguishable under noise. Finally, the sender can tune the probability distribution of input symbols.

We do not limit the transmission strategy that a sender uses. Even in this case, the maximum data rate through the covert channel is still bounded because of a trade-off between the amount data per transmission and the average transmission time. Intuitively, this is because as the number of symbols increases or the time differences that distinguish these symbols increase, so does the average transmission time. It can be shown that, after a point, increasing the data per transmission results in a lower data transmission rate.

Section 5.3.3 presents a formal model of the covert channel. Then, Appendix A discusses how to calculate the maximum data rate of the covert channel, which is an upper bound of the scheduling leakage of the side channel.

Example. The following two strategies illustrate the trade-off: (i) STRATEGY 1 uses 1 ms, 2 ms, 3 ms, and 4 ms to represent an alphabet of four symbols; and (ii) STRATEGY 2 uses 1 ms, 2 ms, ..., 8 ms to represent an alphabet of eight symbols. To simplify the discussion, we assume that all symbols are equally likely in both strategies. Then, STRATEGY 1 transmits $\log 4 = 2$ bits per transmission (i.e., the entropy of the four symbols), and the average transmission time is $(1 + 2 + 3 + 4)/4 = 2.5$ ms. STRATEGY 2 transmits $\log 8 = 3$ bits per transmission and the average transmission time is $(1 + 2 + ... + 8)/8 = 4.5$ ms. Comparing the data rates for both strategies, we see that STRATEGY 1's data rate (2 bits/2.5 ms = 800 bits/s) is higher than STRATEGY 2's (3 bits/4.5 ms ≈ 667 bits/s), despite using fewer symbols.

5.3.2 *Limiting the Maximum Data Rate.*

Based on the previous intuitive explanation of the covert channel, and before presenting a formal model of it, we introduce two mechanisms to reduce the maximum data rate of the covert channel (i.e., the upper bound of the scheduling leakage rate). One mechanism lowers the transmission rate and the other reduces the amount of data that the receiver (i.e., the attacker) can learn per transmission.

Mechanism 1: Set a minimum wait time, called the *Cooldown Time* (denoted by T_c), between two consecutive resizing assessments. Specifically, if an assessment occurs at t, then the scheme enforces that the next assessment cannot occur before $t + T_c$. This cooldown time helps reduce the transmission rate.

Once T_c is picked, the resizing schedule has to be aware of the value of T_c, and guarantee that the time between two consecutive assessments is never below T_c. For example, using a resizing schedule that makes assessments every N retired instructions, a possible

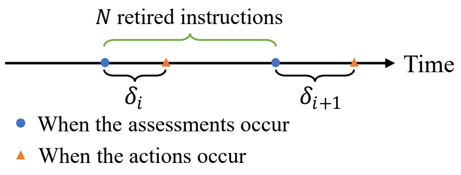


Figure 6: Delaying actions by a random amount of time.

approach is to set N to the maximum number of instructions that the core can possibly retire within T_c. If the core has a commit width of w, then $N = wT_c$ (assuming T_c is measured in cycles).

The cooldown time is set based on the security and performance goals: the longer the cooldown time is, the lower the leakage rate is, and the slower the program execution is.

Mechanism 2: Delay each resizing action by a random time δ after the corresponding assessment point. Intuitively, adding random delays "blurs" the differences between symbols and can introduce bit errors in the channel, thus reducing the amount of information learned by the receiver. This technique is shown in Figure 6. On the time axis, blue dots show when the assessments occur, and orange triangles show when the actions take place. Note that, right after Assessment i is made, we start counting progress towards Assessment $i + 1$. This ensures that the *action* taken at Assessment $i + 1$ is not influenced by program timing.

5.3.3 *Formal Analysis.*

This section formalizes the proposed covert channel model and the two data rate reduction mechanisms just described. To be conservative, we reason about the upper bound of $H(T_s|S = s)$ for the worst-case action sequence s. The worst-case action sequence is the one that changes the partition size at every action, thus making the timing of every action visible to the attacker. Later, in Section 5.3.4, we will discuss how the model can be optimized when the sequence includes MAINTAIN decisions—as usual, assuming that only one action sequence is possible.

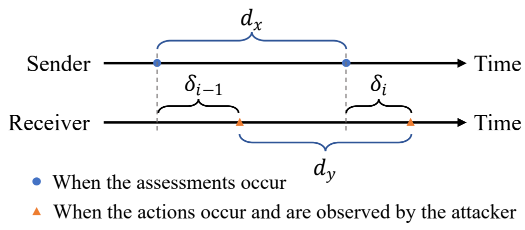
We assume that the resolution at which the attacker (i.e., receiver) can measure the time is finite. We represent timestamps with unitless integers, with 1 time unit being the resolution.

As per Section 5.3.1, the information is encoded as the time duration of remaining in a certain partition size, and the sender uses different durations to represent different input symbols. Therefore, let X be a random variable that represents an input symbol. X takes values in a discrete input alphabet $\mathcal{X}$. An input symbol $x \in \mathcal{X}$ follows the input distribution $p(x)$. For each input symbol x, we use a unique time duration d_x to represent it. Since we enforce that two assessments must be at least T_c apart, $d_x \geq T_c$ for any x. Then, the average time for one transmission is:

$$T_{avg} = \sum_{x \in \mathcal{X}} p(x) d_x. \tag{5.7}$$

Lastly, if there are multiple transmissions (one per each assessment), we denote the input symbol in the ith transmission as X_i, and the input sequence used in n transmissions as $X^n = X_1, X_2, ..., X_n$.

On the receiver side, let Y be a random variable that represents an output symbol. Note that Y is not the same as X because of the random delay δ. Specifically, Y takes values in a discrete output alphabet $\mathcal{Y}$, which is determined by $\mathcal{X}$ and the distribution of the random delay δ (i.e., $p(\delta)$). For a specific input symbol x represented by time duration d_x, the time duration observed by the receiver,



d_x
Sender
Time
δ_{i-1}
δ_i
Receiver
Time
d_y
● When the assessments occur
▲ When the actions occur and are observed by the attacker

Figure 7: Timeline of the sender and the receiver.

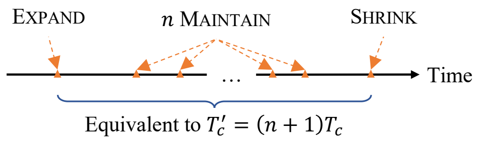


Figure 8: Optimizing the covert channel with MAINTAIN.

denoted by d_y, can be different from d_x due to the δ. If we denote the random delay in transmission i by δ_i, then

$$d_y = d_x + \delta_i - \delta_{i-1}, \tag{5.8}$$

as illustrated in Figure 7. Since each possible d_y is mapped to $y \in \mathcal{Y}$, the output distribution $p(y)$ can be computed from $p(x)$ and $p(\delta)$. Lastly, the output sequence received from n transmissions is denoted by $Y^n = Y_1, Y_2, ..., Y_n$.

The maximum amount of information that the receiver learns from n transmissions is $I(X^n; Y^n)$, the mutual information between X^n and Y^n (Equation 2.4). Also, the average time for n transmissions is nT_{avg}. Therefore, the data rate R of the covert channel is

$$R = I(X^n; Y^n)/nT_{avg}. \tag{5.9}$$

Different input distributions of $p(x)$ result in different values of $I(X^n; Y^n)$ and T_{avg}. Therefore, we are interested in the input distribution of $p(x)$ that produces the maximum data rate (R_{max}) of the covert channel. This value is an upper bound of the scheduling leakage rate of any victim program.

Computing a closed-form of R_{max} is complex. Consequently, in Appendix A, we show a numerical method that computes a tight upper bound of R_{max}.

5.3.4 *Optimized Covert Channel Model.*

In Section 5.3.3, the covert channel model conservatively assumes the worst-case action sequence, where every action changes the partition size, thus making the timing of every action visible to the attacker. However, in practice, most resizing assessments result in MAINTAIN (as shown in Section 9), whose timing is invisible to the attacker.

Since the victim only has one possible action sequence s under *Untangle* for a given public input, we can leverage the MAINTAIN actions to optimize the covert channel model to reduce the maximum data rate. The idea is illustrated in Figure 8. If the victim chooses MAINTAIN *n consecutive* times, the execution is equivalent to a case when the two visible resizing actions that occur right before and right after these n MAINTAIN actions are separated by a longer cooldown time $T_c' = (n + 1)T_c$. Therefore, the scheduling leakage *during this period* is reduced because of the increased cooldown time. Consequently, we can monitor the number of consecutive MAINTAINS performed during a victim execution and lower the upper bound of the scheduling leakage rate of the execution. In Section 7, we discuss how to obtain the optimized R_{max} at runtime.

6 DISCUSSION

In this section, we describe some aspects related to the operation of *Untangle*.

6.1 Timing-Dependent Dynamic Instruction Sequences

In Section 5.2, we removed Edge ③ in Figure 2 and made the resizing action sequence depend not on program timing but only on the retired dynamic instruction sequence of the execution. However, in some cases, this is not enough to eliminate the effect of timing because the dynamic instruction sequence *itself* depends on program timing. This case may happen, e.g., in parallel programs, where the timing of when a thread attempts a synchronization operation may result in different outcomes: repeated spinning or proceeding past the synchronization. It may also happen in single-threaded programs, where the thread may check the current time and take different paths based on the result.

To handle this case, the code regions with these timing-dependent dynamic instruction sequences need to be annotated, so that one can exclude their contribution when measuring the utilization metric and exclude their instructions when quantifying execution progress. The techniques used by existing analysis tools that detect secret-dependent control and data flow in a program can be used as a basis to identify and annotate these timing-dependent sequences. For example, one can treat the data read inside a critical section or the return value of a get-time system call as a secret. We consider any further analysis of this issue the subject of future work.

6.2 Other Attacks

A powerful attacker can replay the victim program many times, gaining additional information at every replay from the scheduling leakage. However, the operating system can use the upper bound of the victim program's leakage rate as computed by *Untangle* (Equation 5.9) to keep accumulating the victim program leakage across the multiple runs. When the accumulated leakage across runs reaches a user-defined threshold, the system prevents the victim program from performing any further resizes. From then on, the performance of the program will decrease, but there will be no more leakage.

The attacker can also actively interact with the victim, as illustrated in Figure 9. In this example, the victim is in a steady state and decides to MAINTAIN at assessment ①. Since MAINTAIN is invisible to an attacker, our optimized covert channel model can leverage it to further bound the scheduling leakage (Section 5.3.4). However, before the victim makes the next assessment, the attacker can put a high pressure on the shared resources to "squeeze" the victim partition. This strategy can force the victim to perform an attacker-visible action EXPAND at the next assessment (②), incurring a higher *scheduling* leakage rate. As a result, the user-defined leakage threshold will be reached sooner, which can disable further resizing and hurt execution performance, but cannot violate the security guarantees discussed in Section 4, as the leakage will not exceed the threshold. Note that an active attacker cannot cause *action* leakage in *Untangle*. The reason is that even if the action sequence changes, it is not due to secret values and, therefore, there is no action leakage. The change is due to the attacker actions, which cannot affect the victim's timing-independent resource utilization metric at the points of assessment.

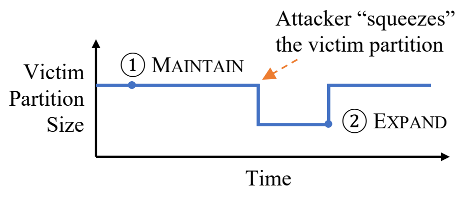


Figure 9: An active attacker that interacts with the victim.

6.3 Partitioning Other Hardware Resources

In Section 3 and 5, we mainly use the last-level cache (LLC) as an example of resource of interest. However, *Untangle* is a general framework and it can be applied to different hardware resources. To apply *Untangle* to a new type of resource, we first need to define a timing-independent utilization metric for that resource (Section 5.2). For example, we can trivially extend the LLC utilization metric to the TLB. Another example of resource of interest is functional units shared by two SMT threads [43, 47], where we can use the fraction of the retired instructions that utilize a certain type of function unit as a metric. Next, we also need to extend the static analysis to identify the secret-dependent usage of the new resource of interest. For the TLB, we can reuse the static analysis for caches [3, 8, 18, 49]; for function units, an analyzer that detects secret-dependent control flow suffices.

6.4 Extending the Threat Model

Untangle assumes a peer security model, where all programs are mutually distrusting and in the same security level (Section 4). It is possible to extend *Untangle* to support a more complex security lattice that is also tiered. Under this model, information flow from a lower-tiered program (L) to a higher-tiered program (H) is allowed, but not vice versa. As a result, program L can take resizing actions that claim resources from or free resources to H without counting towards the leakage thresholds of both programs. However, there is one caveat: L's resizing can affect the timing of H due to the change of available resources. This timing change in H can be secret-dependent and observed by L through other observable events (e.g., termination of H). *Untangle*'s covert channel model can be adapted to measure this type of leakage.

6.5 Using Existing Static Analyses for Annotation

Recall from Section 5.2 that we need to annotate all instructions that have secret-dependent resource usage and all instructions that are control-dependent on secrets. The main challenge of using existing analyses [3, 8, 18, 49] for annotation is their scalability, since they use static analysis to ensure soundness. According to the literature, all these analyses can analyze cryptography libraries. Besides that, Cacheaudit [18] also analyzes sorting primitives and Casym [8] analyzes database applications (e.g., PostgreSQL [22]). These libraries and applications can process sensitive information.

For applications that are beyond the capability of these tools, one can use manual inspection assisted by these tools to generate conservative but sound results for *Untangle*—e.g., by applying analyses on a manually-selected portion of the program. In this case, the performance of applications may decrease due to the conservativeness of the analysis.

7 HARDWARE IMPLEMENTATION

This section discusses a potential implementation of key aspects of the *Untangle* hardware. Similar to the previous discussion, we use LLC partitioning as an example. We do not explore a full implementation because the focus and the novelty of this paper is in the *Untangle* framework.

Transmitting the Annotations to the Hardware. There are several possible ways to transmit the annotations to the underlying *Untangle* hardware. Intuitively, we can re-purpose a currently-unused instruction prefix to mark the relevant instructions. A similar approach is used by Intel for lock elision [25]. However, this approach can generate bloated binaries if many instructions are annotated. An alternative approach is to introduce two new instructions that flag the start and the end of a secret-dependent code region. Finally, we can also introduce a special bit in the page table to coarsely annotate pages that contain secret-dependent code [38]. The latter approach does not require recompilation and can be applied to legacy programs.

Monitoring LLC Utilization. Many prior works have proposed various ways of monitoring LLC utilization [4, 36, 39]. We describe one possible mechanism that we use in the evaluation. The mechanism is similar to UMON [36], which assumes that the partition size is chosen from a pre-defined list of supported sizes. At a high level, for each domain, at runtime, the mechanism simulates memory accesses with each possible partition size, and measures the corresponding number of LLC hits. Then, during a resizing assessment, the monitor picks the size for each domain that maximizes the number of LLC hits across all domains. To satisfy the requirements of a timing-independent utilization metric, the monitor does not consider memory instructions that are data- or control-dependent on secrets. In addition, it only considers the memory accesses resulting from retired memory instructions and in program order (Section 5.2).

The proposed mechanism can be implemented with a set-associative hardware table that selectively simulates memory accesses to only certain cache sets. This hardware table only contains tags but not data. When a public load or store to one of the monitored sets retires, the table is accessed. Memory accesses that would hit in the private caches are filtered out.

Measuring Scheduling Leakage at Runtime. Recall from Section 5.3.4 that we leverage consecutive Maintain actions, which are invisible to the attacker, to further reduce the bound on scheduling leakage rate. However, it is impractical to compute the optimized scheduling leakage rate at runtime, since it needs to generate a new R_{max}, and this involves a computation-intensive algorithm (Appendix A). Therefore, we use a small hardware table that stores precomputed leakage rates. Specifically, table entry i stores the leakage rate R_{max_i}, corresponding to when i consecutive Maintains occur. At runtime, if the victim chooses Maintain m consecutive times, we conservatively assume that the next action is not Maintain and use the rate R_{max_m} to compute the leakage for that resizing. If the next action turns out to be another Maintain, we switch to the lower rate $R_{max_{m+1}}$. Finally, if m exceeds the table capacity, we conservatively use the rate of the entry for the maximum number of Maintains considered.

Table 3: Parameters of the simulated architecture.

Parameter	Value
Architecture	8 out-of-order x86 cores at 2.0 GHz
Core	8-issue, 8-commit, no SMT, 72 load queue entries, 56 store queue entries, 224 ROB entries, LTAGE branch predictor
Private L1-I & L1-D cache	32 kB, 64 B line, 8-way, 2 cycle round trip (RT) latency
Shared L2 cache (LLC)	16 MB (2 MB per slice), 64 B line, 16-way, 8 cycles RT latency
DRAM	50 ns RT latency after L2
Supported partition sizes for a domain	128 kB, 256 kB, 512 kB, 1 MB, 2 MB, 3 MB, 4 MB, 6 MB, 8 MB
Monitor window M_w	1 M memory instructions

Table 4: Partitioning schemes evaluated.

Scheme	Description
Static	Static partitioning. Each domain uses a 2 MB partition
Time	Dynamic partitioning. Assessing resizing every 1 ms
Untangle	Dynamic partitioning. Assessing resizing every 8 M retired instructions with a cooldown time of 1 ms
Shared	No partitions. All domains share the 16 MB LLC

8 EXPERIMENTAL METHODOLOGY

We use last-level cache (LLC) partitioning as an example to demonstrate that *Untangle* can offer flexibility and better performance than static partitioning, and significantly less leakage than prior dynamic partitioning schemes. We choose the LLC as the resource of interest because it is a commonly-exploited hardware resource, and there are many prior LLC partitioning schemes [15, 28, 36, 37, 46].

Following prior work [15, 37, 46], we use set partitioning. We assume a simple design where the size of a partition is chosen from a pre-defined list of 9 choices. We use the mechanism discussed in Section 7 to monitor LLC utilization and guide resizes. The monitor only considers the past M_w retired memory instructions at the time of an assessment, to focus on the program's most recent LLC utilization.

Configurations & Schemes. We model an 8-core system (Table 3) using cycle-level simulations with gem5 [7]. We consider four LLC partitioning schemes (Table 4). The baseline scheme is Static, which partitions the LLC statically, giving 2 MB to each domain. We evaluate two dynamic partitioning schemes: Time is similar to previous ones [4, 36, 39, 43] that make resizing assessments at a fixed time interval (1 ms interval in our configuration); Untangle applies our mitigations described in Section 5. Untangle makes resizing assessments every 8 M retired instructions and its minimum wait time between resizes is 1 ms. We use this configuration for Untangle to match the performance of Time by performing resizing

Table 5: OpenSSL [11] cryptographic benchmarks.

Name	Description
Chacha20	Stream cipher, encrypt 10 kB payloads
AES-128	Block cipher with a 128-bit key, encrypt 10 kB payloads
AES-256	Block cipher with a 256-bit key, encrypt 10 kB payloads
SHA-256	Digest function, compute 10 kB payloads
RSA-2048	RSA signing with a 2048-bit key
RSA-4096	RSA signing with a 4096-bit key
ECDSA	ECDSA signing using curve Secp256k1
EdDSA	EdDSA signing using curve Ed25519

assessments at a similar frequency. The random delay in Untangle follows a uniform distribution between [0, 1 ms). Both Time and Untangle start with a partition size of 2 MB. Finally, Shared is an insecure configuration that uses a shared LLC without partitioning.

Workloads. To evaluate workloads that have both secret-related and public parts, we build workloads composed of one SPEC17 benchmark [9] and one cryptographic benchmark from OpenSSL 3.0.5 [11] (Table 5). Both benchmarks share the same domain and hence use the same LLC partition. Since we target a typical workload that spends most of its execution time in the public part, we repeatedly run in a loop 1 M instructions from the cryptographic benchmark and then 10 M instructions from the SPEC17 benchmark. Both benchmarks make forward progress. We conservatively assume that all instructions from the cryptographic benchmark are secret-dependent. We do not set a leakage threshold for a workload; we allow it to freely resize and then measure its leakage.

For SPEC17, we use the reference input size. We simulate all 36 SPEC17 benchmarks.[2] For each SPEC17 benchmark, we use SimPoint [24] to select a representative slice of 500 M instructions. We study each SPEC17 benchmark's sensitivity to LLC size by running it with every supported partition size and normalize its instruction-per-cycle (IPC) to the IPC with an 8 MB partition (i.e., the maximum partition size). The study is detailed in Appendix B. We neglect the crypto benchmarks because they have much smaller LLC use. We then define the *adequate LLC size* of a benchmark as the *minimal* LLC size that allows the benchmark to reach a normalized IPC of at least 0.9. If a benchmark has an adequate LLC size higher than 2 MB (i.e., the Static partition size), we classify it as *LLC-sensitive* (8 benchmarks in total); otherwise, it is *LLC-insensitive* (28 benchmarks in total).

Since we simulate an 8-core system, we first randomly select a mix of eight workloads (2 LLC-sensitive and 6 LLC-insensitive). Then, from this base mix, we randomly replace two LLC-insensitive workloads with two LLC-sensitive ones to generate a new mix. We repeat this change until there are no LLC-insensitive workloads in the mix. We run our experiments on each mix. Next, we repeat this process with different base mixes to cover all possible workloads. For each experiment, we warm up the system for 5 ms. Then, we simulate the mix of workloads until each workload finishes its slice (500 M instructions from SPEC and 50 M from crypto). When a workload finishes, if there are other running workloads in the system, the finished workload maintains its pressure on the LLC, but does not update the statistics that we collect.

[2]The same SPEC application with another input is a different benchmark.

Measuring the Leakage. We measure the leakage in Time with $\log |\mathcal{A}|$ bits per assessment, where $|\mathcal{A}|$ is the number of supported resizing actions (Section 3.3). For Untangle, we use the leakage model proposed in Section 5 with the optimization discussed in Section 5.3.4.

We compare Time with 1 ms assessment interval against Untangle with $T_c = 1$ ms, which corresponds to 8 M retired instructions between assessments. We report the *leakage per assessment* of a workload under each scheme. Leakage per assessment determines the number of assessments that a workload is allowed under a leakage threshold. The lower the leakage per assessment is, the more assessments the scheme can make. Because Time and Untangle use different resizing schedules, the same workload can have different number of resizing actions under different schemes, in spite of running the same number of instructions. The total leakage from an execution is proportional to the number of resizing actions during the execution.

9 EVALUATION

We evaluate 16 workload mixes in total. Due to the similarity between mixes, and due to space limits, we only show in Figure 10 the results of 4 selected mixes. Appendix B includes the results of the rest of mixes.

In Figure 10, each group of three charts corresponds to one mix. The top-left group (Mix 1) is for a mix with 2 LLC-sensitive workloads (shown in bold). As we move from left to right (Mix 2), and then from top to bottom (Mix 3 and Mix 4), we replace 2 non LLC-sensitive SPEC17 benchmarks with 2 LLC-sensitive SPEC17 benchmarks, until we reach a mix with all 8 LLC-sensitive benchmarks. In the title of each group, we show the total *LLC demand* as the sum of the adequate LLC size of all the workloads in the mix. In a given group, the bottom-most chart shows the IPC of every workload and scheme—normalized to Static. The middle chart shows the leakage per assessment in bits for Time and Untangle. Finally, the topmost chart shows the distribution of partition size measured at intervals of 100 µs. In that chart, the thick short bar covers the first to third quartile range; the thin long bar is the minimum and maximum range; the white dot is the median of the partition sizes. Note that the figure has a non-linear y-axis, and each dashed horizontal line corresponds to a supported partition size listed in Table 3.

Consider the top-left group of charts in Figure 10. It shows Mix 1, which has 2 LLC-sensitive workloads and a total LLC demand from all the 8 workloads equal to 14.6 MB. There is enough LLC for every workload. Under both Time and Untangle, the two LLC-sensitive workloads, `gcc_2`[3] and `parest_0`, attain high speedups over Static, and even slightly outperform Shared. The remaining LLC-insensitive workloads experience no slowdown, in spite of some of them using partitions smaller than the 2 MB of Static (see top chart). Overall, the system-wide speedup (i.e., the geometric mean of IPCs) of Time and Untangle over Static is 1.14. Shared (i.e., no partitioning) has a lower speedup of 1.12 because of cache conflicts between workloads.

[3]To refer to a workload, we use the SPEC17 application name plus a number for the input.

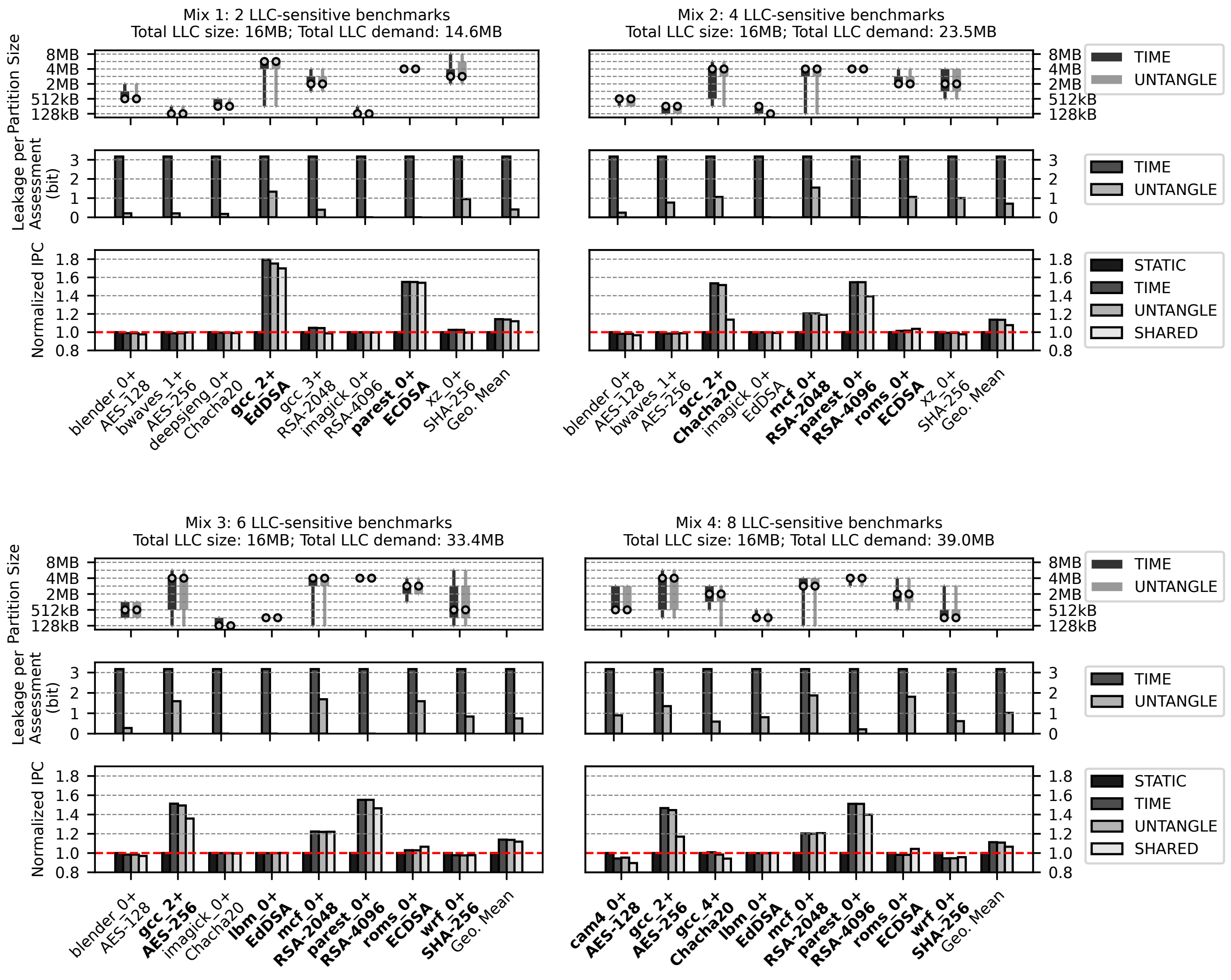


Figure 10: Comparing different partitioning schemes according to: IPC normalized to Static (bottom-most row of each group of three charts), leakage per assessment (middle row), and distribution of partition size (topmost row). The bars correspond to different workloads, where bolded workloads are LLC-sensitive.

Since the dynamic partitioning scheme in our evaluation supports 9 different actions (Section 8), Time leaks $\log 9 \approx 3.2$ bits per assessment for every workload (middle row chart). Based on our measurements, the leakage per assessment in Untangle is at most 1.3 bits per assessment, and on average 0.4 bits per assessment.

Consider now the top-right group of charts in Figure 10. It shows Mix 2, which replaces `deepsjeng_0` and `gcc_3` from Mix 1 with `mcf_0` and `roms_0`. The Mix 2 workloads demand 23.5 MB of LLC in total. Due to the increased total LLC demand, `gcc_2` receives a smaller partition than in Mix 1, with a median size of 4 MB (top chart). As a result, its speedup is now lower than in Mix 1, namely 1.54 under Time and 1.52 under Untangle. `parest_0` and `mcf_0` attain good speedups, but `roms_0` does not translate its higher cache use into performance. Overall, both Time and Untangle deliver a system-wide speedup of 1.14, while Shared has a speedup of 1.08. Lastly, Untangle has a leakage per assessment of 1.5 bits at most and 0.7 bits on average, while Time leaks 3.2 bits per assessment.

The bottom-left group of charts in Figure 10 shows Mix 3, which includes two more LLC-sensitive workloads: `lbm_0` and `wrf_0`. These 8 workloads demand 33.4 MB of LLC, which is more than twice the available LLC. In this over-committed setting, three LLC-sensitive workloads have their demand fulfilled under Time and Untangle: `gcc_2`, `parest_0`, and `mcf_0`. Also, the rest of workloads do not suffer slowdown when compared to Static. Both Time and Untangle deliver a system-wide speedup of 1.14, and Shared has a speedup of 1.12.

For the Mix 2 and 3 workloads, we start to see an increase of leakage per assessment under Untangle. The reason is that the workloads have more attacker-visible resizing actions that change the partition size, due to the high LLC pressure. In Mix 3, Untangle has a maximum leakage per assessment of 1.7 bits and 0.7 bits on average.

In Mix 4 (bottom-right group of charts), all 8 workloads are LLC-sensitive. They demand a total LLC of 39.0 MB. Under this extreme LLC pressure, Time and Untangle can still fulfill three LLC-sensitive workloads. However, some of the remaining workloads start to suffer slowdown. Overall, both Time and Untangle have a system-wide speedup of 1.11, while Shared has a lower speedup of 1.07. The leakage per assessment in Untangle increases to 1.9 bits in the worst-case workload and 1.0 bits on average. As usual, Time leaks 3.2 bits per assessment.

To summarize, Time and Untangle provide nearly the same speedups over Static, but Untangle leaks information at a significantly lower rate. Shared delivers slightly lower speedups due to cache conflicts between workloads. In Untangle, it can be shown that, of all reassessments across all mixes, 90% are Maintain.

Total Leakage. Table 6 summarizes the leakage of the selected mixes under Time and Untangle. The table shows both the average leakage per assessment and the average total leakage per workload. Across the mixes, the leakage per assessment under Untangle is 78% lower than under Time.

Table 6: Leakage of Mixes 1-4 under Time and Untangle.

	Time		Untangle	
	Avg. leakage per assessment	Avg. total leakage	Avg. leakage per assessment	Avg. total leakage
Mix 1	3.2 bits	637.6 bits	0.4 bits	38.5 bits
Mix 2	3.2 bits	829.7 bits	0.7 bits	65.5 bits
Mix 3	3.2 bits	979.9 bits	0.7 bits	70.0 bits
Mix 4	3.2 bits	1084.1 bits	1.0 bits	96.0 bits

Leakage of Untangle under an active attacker. A powerful active attacker can put high pressure on the shared LLC, forcing the victim to make an attacker-visible resizing action at every single assessment (Section 6.2). Then, the victim leaks at a higher rate. To study this environment, we measure the leakage under Untangle without the optimized covert channel model of Section 5.3.4. We find that the average leakage per assessment is 3.8 bits, averaged across all the workloads from all the mixes. This leakage is higher than with the optimization (0.7 bits). This worst-case leakage rate is very rare in a benign execution. However, even if it occurs, *Untangle still upholds the security guarantees*: at this increased leakage rate, the user-defined leakage threshold will be reached sooner, which will disable further resizings and, at worst, only hurt performance. This is not a limitation of *Untangle*, since an *active* attacker can always slow down the victim by forcing it to use the smallest partition.

10 RELATED WORK

Types of Hardware Defenses. Hardware techniques to block microarchitectural side-channels fall into two categories. Randomization-based schemes [13, 29, 34, 35, 42, 52, 54, 62] attempt to obfuscate victim resource usage. These schemes offer high performance, but not comprehensive security guarantees. Partitioning-based schemes [14, 27, 48, 51] provide comprehensive security guarantees, but static partitioning incurs significant performance overhead [51].

Secure Dynamic Resource Partitioning. SecDCP [51] dynamically partitions cache resources based on a *tiered* security model: behaviors of sensitive programs cannot influence resizing decisions; only non-sensitive programs do. This model does not apply to cases where all programs are mutually distrusting and in the same security level (i.e., *peers*). In contrast, *Untangle* has a peer security model.

SecSMT [43] dynamically partitions pipeline resources. It supports both the tiered and peer security models. In the peer model, however, SecSMT only loosely bounds the leakage to 1 bit per assessment (for 2 possible resizing actions). This is leakage overestimation. In contrast, *Untangle*'s leakage bounds are much tighter.

Quantifying the Leakage of Side Channels. Some works quantify the leakage of side channels in the absence of partitioning. Dynamic approaches examine specific victim executions and quantify their leakage trace [50, 53, 55, 56, 59, 60]. Thus, these approaches cannot produce a worst-case leakage bound. Other approaches use symbolic execution [3, 10] or abstract interpretation [18, 19, 49]. Although these approaches are sound, they cannot quantify leakage that depends on program timing.

11 CONCLUSION

This paper presented *Untangle*, a framework for constructing low-leakage, high-performance dynamic partitioning schemes. *Untangle* formally splits the leakage into leakage from deciding what resizing action to perform and leakage from deciding when each resizing action occurs. Based on this breakdown, *Untangle* makes two contributions. First, it introduces a set of principles for constructing dynamic partitioning schemes that untangle program timing from the action leakage. Second, *Untangle* introduces a novel way to model the scheduling leakage without analyzing program timing. With these techniques, *Untangle* is able to quantify the leakage in a dynamic resizing scheme in a tighter way than prior work.

We applied *Untangle* to dynamic partitioning of the last-level cache. On average, workloads leak 78% less under *Untangle* than under a conventional dynamic partitioning approach, for approximately the same workload performance.

ACKNOWLEDGMENTS

This research was funded by an Intel Resilient Architectures and Robust Electronics (RARE) gift; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by NSF grants 1942888 and 1954521. We thank the anonymous reviewers and the shepherd for their valuable feedback. We also thank Linyi Li for his suggestions on computing the maximum data rate R_{max}.

A COMPUTING THE MAXIMUM DATA RATE

In this appendix, we describe how to compute the maximum data rate R_{max} of the covert channel model from Section 5.3.3. Recall that the maximum data rate R_{max} is:

$$R_{max} = \max_{p(x)}\{I(X^n; Y^n)/nT_{avg}\}, \quad \text{(A.1)}$$

where the maximization is taken over all possible input distributions $p(x)$.

The following discussion assumes that the random delay δ across transmissions is *independent and identically distributed (IID)*, and that the input symbol X follows that same input distribution $p(x)$ across transmissions. As a result, the output symbol Y follows the same output distribution $p(y)$ across transmissions.

To find the maximum data rate R_{max}, the first step is to compute the mutual information $I(X^n; Y^n)$. However, computing it directly from the mutual information definition is not feasible, since the number of transmissions n can be unbounded. Hence, we perform the following conservative simplification and approximation. By the definition of mutual information (Equation 2.4), we have

$$I(X^n; Y^n) = H(Y^n) - H(Y^n|X^n). \tag{A.2}$$

For the first term $H(Y^n)$, which is the joint entropy of $Y_1, Y_2, ..., Y_n$, we apply the chain rule [12]

$$H(Y^n) = H(Y_1) + \sum_{i=2}^{n} H(Y_i|Y^{i-1}) \tag{A.3}$$

$$\leq \sum_{i=1}^{n} H(Y_i) = nH(Y). \tag{A.4}$$

For the second term $H(Y^n|X^n)$, by the definition of the conditional entropy (Equation 2.3), we have

$$H(Y^n|X^n) = -\sum_{x^n \in \mathcal{X}^n} \sum_{y^n \in \mathcal{Y}^n} p(x^n, y^n) \log p(y^n|x^n) \tag{A.5}$$

$$= -\sum_{x^n \in \mathcal{X}^n} \sum_{\delta^n \in \Delta^n} p(x^n, \delta^n) \log p(\delta^n|x^n) \tag{A.6}$$

$$= -\sum_{x^n \in \mathcal{X}^n} \sum_{\delta^n \in \Delta^n} p(x^n) p(\delta^n) \log p(\delta^n) \tag{A.7}$$

$$= -\sum_{\delta^n \in \Delta^n} p(\delta^n) \log p(\delta^n) \tag{A.8}$$

$$= H(\delta^n) = nH(\delta), \tag{A.9}$$

where Equation A.6 substitutes y^n with δ^n because y^n is a function of δ^n and x^n, Equation A.7 holds because δ^n and x^n are independent, and Equation A.9 holds because random delays are IID.

Therefore,

$$I(X^n; Y^n) = H(Y^n) - H(Y^n|X^n) \leq n(H(Y) - H(\delta)). \tag{A.10}$$

Using Equation A.10, we can conservatively approximate $I(X^n; Y^n)$ without considering the whole sequences of X^n and Y^n. Hence, the goal becomes finding

$$R'_{max} = \max_{p(x)} \{(H(Y) - H(\delta))/T_{avg}\} \tag{A.11a}$$

$$\text{subject to} \quad \sum_x p(x) = 1, p(x) > 0 \tag{A.11b}$$

over all possible input distributions $p(x)$. R'_{max} is an upper bound of R_{max}. The optimization problem A.11 fits the standard single-ratio *fractional programming* (FP) problem [40]. *Dinkelbach's transform* [17] can iteratively converge to the optimal input distribution $p(x)$ that achieves R'_{max}.

Dinkelbach's transform [17]. To simplify the discussion, we first consider a general FP problem

$$\underset{z}{\text{maximize}} \quad N(z)/D(z) \tag{A.12a}$$

$$\text{subject to} \quad z \in \mathcal{Z}, \tag{A.12b}$$

where $N(z)$ and $D(z)$ are continuous and real-valued functions of z. Moreover, $D(z) > 0$ for all $z \in \mathcal{Z}$.

To solve Problem A.12, Dinkelbach's transform introduces an auxiliary variable q and a helper function given by

$$F(q) = \max_{z \in \mathcal{Z}} \{N(z) - qD(z)\}. \tag{A.13}$$

The algorithm then iteratively updates q according to the following steps (with the value of q in the ith iteration denoted by q_i):

(1) Set $q_1 = 0$ and $i = 1$.
(2) Solve $F(q_i)$ for z_i over $z \in \mathcal{Z}$.
(3) Update $q_{i+1} = N(z_i)/D(z_i)$, increment i, and go to Step 2.

The algorithm iterates n times until either $F(q_n) < \epsilon$, where ϵ is a positive real number representing the tolerance, or n reaches the maximum number of iterations. Subsequently, z_n can be used as an approximate solution to Problem A.12, and q_n is an approximation of $\max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$.

To find a tight upper bound for $\max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$ using the iterative solution q_n, we observe that $F(q)$ is strictly monotonic decreasing with respect to q [17]. Furthermore, it can be proven that $q^* = \max_{z \in \mathcal{Z}}\{N(z)/D(z)\}$ if and only if $F(q^*) = 0$ [17]. Therefore, we can guess an upper bound $q' = q_n + \delta$, where δ is a small positive real number. If we verify that $F(q') \leq 0$, then we know that $q' \geq q^*$ since $F(q)$ is strictly monotonic decreasing. Otherwise, we increase δ and repeat the process.

Our implementation. We apply Dinkelbach's transform to solve Problem A.11 and find an upper bound of R'_{max}. To do so, we need to find a distribution $p(x)$ that maximizes $(H(Y) - H(\delta)) - q_i T_{avg}$ for each iteration (Step 2 of the algorithm). We begin with analyzing the concavity of the target function by examining its individual components. The first term, $H(Y)$, is the entropy of the output symbols and is a concave function of $p(y)$ [12]. Since $p(y)$ is a linear function of $p(x)$ (i.e., $p(y) = \sum_x p(y|x)p(x)$), $H(Y)$ is also a concave function of $p(x)$. The second term $H(\delta)$ is a constant for a given random noise distribution. The last term, T_{avg}, is a linear function of $p(x)$ by the definition of T_{avg} (Equation 5.7). Therefore, the target function $(H(Y) - H(\delta)) - q_i T_{avg}$ is concave and can be optimized with a standard concave programming method.

We implement the optimization using PyTorch's [33] Adam optimizer [26] and have observed convergence. We then guess a tight upper bound of R'_{max} with $q' = q_n + \delta$ and use the same optimizer to empirically verify that $F(q') < 0$ after 10,000 iterations. We leave proving $F(q') < 0$ as an open problem for future work.

B COMPLETE EVALUATION

The complete evaluation results are shown in Figures 11–17.

REFERENCES

[1] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida Garcia, and N. Tuveri. 2019. Port Contention for Fun and Profit. In *2019 IEEE Symposium on Security and Privacy (SP)*. 870–887.
[2] Aslan Askarov, Danfeng Zhang, and Andrew C Myers. 2010. Predictive black-box mitigation of timing channels. In *Proceedings of the 17th ACM conference on Computer and communications security*. 297–307.
[3] Qinkun Bao, Zihao Wang, Xiaoting Li, James R Larus, and Dinghao Wu. 2021. Abacus: Precise side-channel analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 797–809.
[4] Nathan Beckmann and Daniel Sanchez. 2013. Jigsaw: Scalable software-defined caches. In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*. IEEE, 213–224.

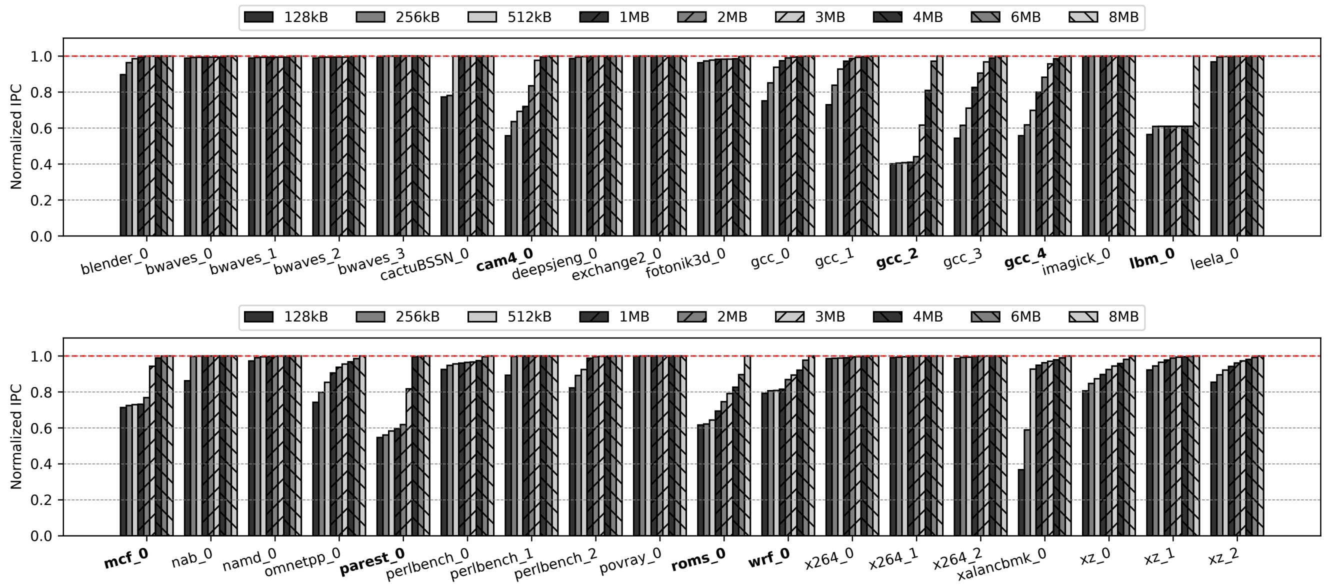


Figure 11: LLC sensitivity study of all 36 SPEC17 benchmarks. IPCs are normalized to the IPCs with an 8 MB partition. LLC-sensitive benchmarks are bolded.

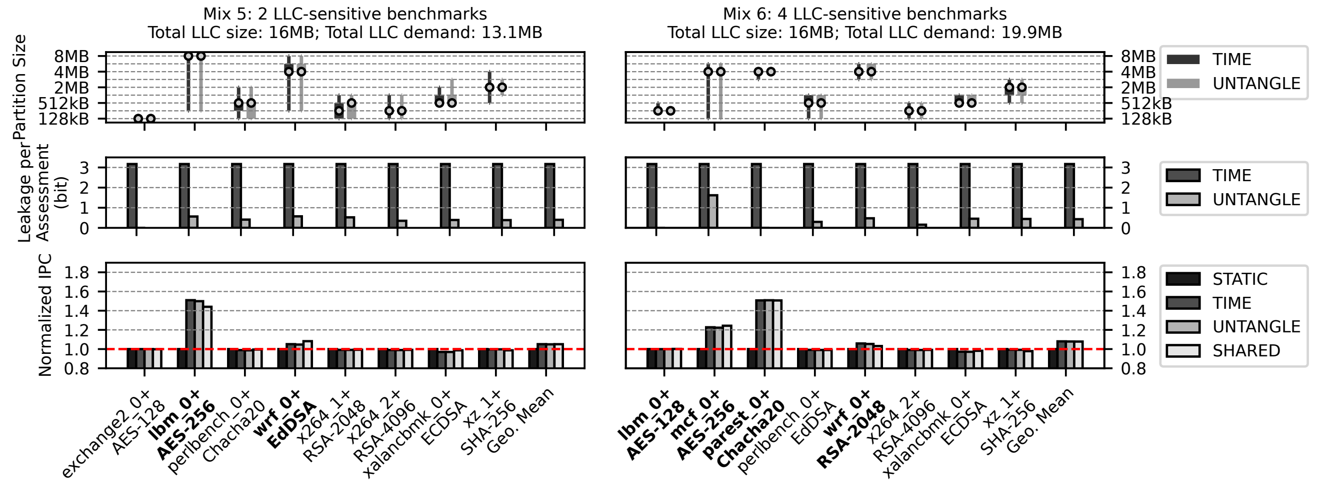


Figure 12: Comparing different partitioning schemes for workloads Mix 5 and Mix 6.

[5] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos V. Rozas, Adam Morrison, Frank McKeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa R. Alameldeen. 2021. Speculative interference attacks: breaking invisible speculation schemes. In *ASPLOS '21: 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 1046–1060.

[6] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.

[7] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *ACM SIGARCH Computer Architecture News* (2011).

[8] Robert Brotzman, Shen Liu, Danfeng Zhang, Gang Tan, and Mahmut Kandemir. 2019. CaSym: Cache aware symbolic execution for side channel detection and mitigation. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE.

[9] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*. 41–42.

[10] Sudipta Chattopadhyay, Moritz Beck, Ahmed Rezine, and Andreas Zeller. 2019. Quantifying the information leakage in cache attacks via symbolic execution. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 1 (2019), 1–27.

[11] OpenSSL Contributors. 2022. OpenSSL 3.0.5. https://github.com/openssl/openssl/releases/tag/openssl-3.0.5.

[12] Thomas M. Cover and Joy A. Thomas. 2006. *Elements of information theory (2nd edition).* Wiley.

[13] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *Proceedings of the 46th International Symposium on Computer Architecture (ISCA'19).*

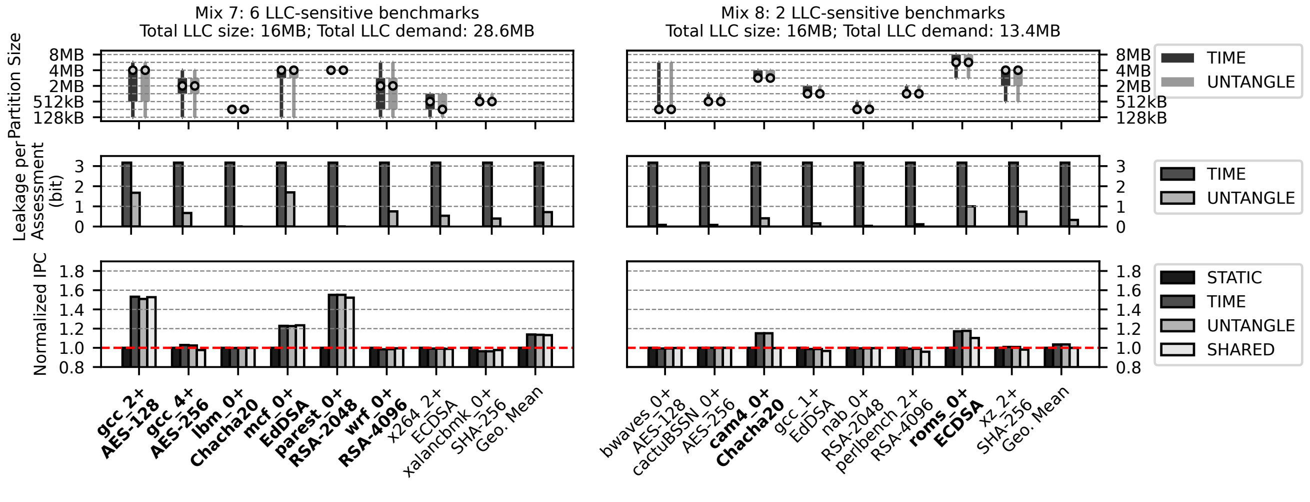


Figure 13: Comparing different partitioning schemes for workloads Mix 7 and Mix 8.

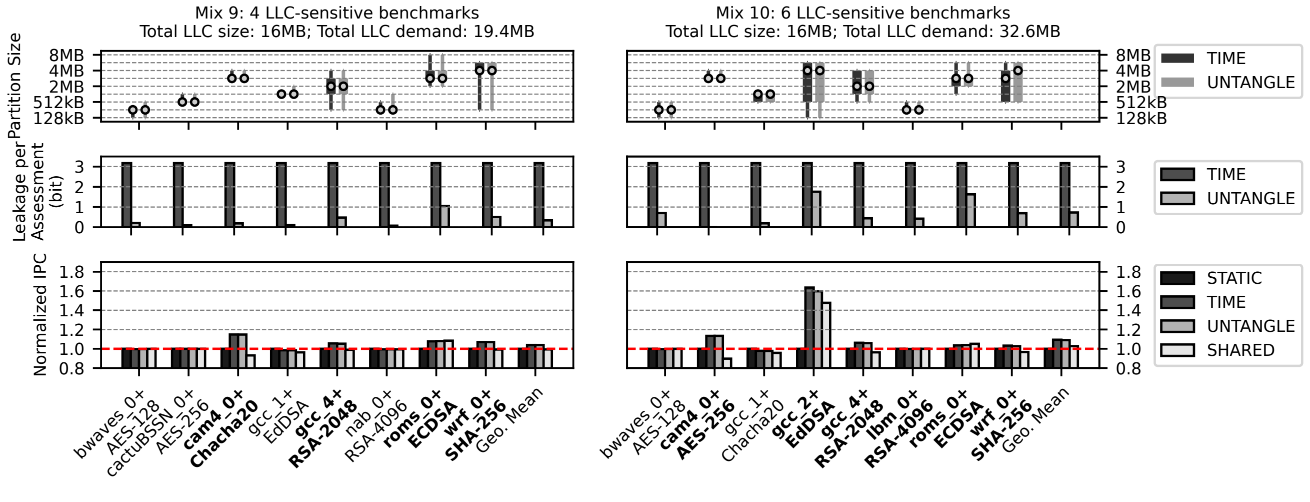


Figure 14: Comparing different partitioning schemes for workloads Mix 9 and Mix 10.

[14] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments. In *29th USENIX Security Symposium (USENIX Security 20)*.

[15] Ghada Dessouky, Emmanuel Stapf, Pouya Mahmoody, Alexander Gruler, and Ahmad-Reza Sadeghi. 2022. Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures. In *29th Annual Network and Distributed System Security Symposium, NDSS*.

[16] Peter W Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S Emer, and Mengjia Yan. 2022. DAGguise: mitigating memory timing side channels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 329–343.

[17] Werner Dinkelbach. 1967. On nonlinear fractional programming. *Management science* 13, 7 (1967), 492–498.

[18] Goran Doychev, Dominik Feld, Boris Köpf, Laurent Mauborgne, and Jan Reineke. 2013. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *Proceedings of the 22th USENIX Security Symposium*. 431–446.

[19] Goran Doychev and Boris Köpf. 2017. Rigorous analysis of software countermeasures against cache attacks. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 406–421.

[20] Christopher W. Fletcher, Ling Ren, Xiangyao Yu, Marten Van Dijk, Omer Khan, and Srinivas Devadas. 2014. Suppressing the oblivious RAM timing channel while making information leakage and program efficiency trade-offs. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 213–224.

[21] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security*.

[22] The PostgreSQL Global Development Group. 2023. PostgreSQL: The world's most advanced open source database. https://www.postgresql.org/.

[23] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: a fast and stealthy cache attack. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 279–299.

[24] Greg Hamerly, Erez Perelman, Jeremy Lau, and Brad Calder. 2005. Simpoint 3.0: Faster and more flexible program phase analysis. *Journal of Instruction Level Parallelism* 7, 4 (2005), 1–28.

[25] Intel. 2021. Intel 64 and IA-32 Architectures Software Developer's Manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

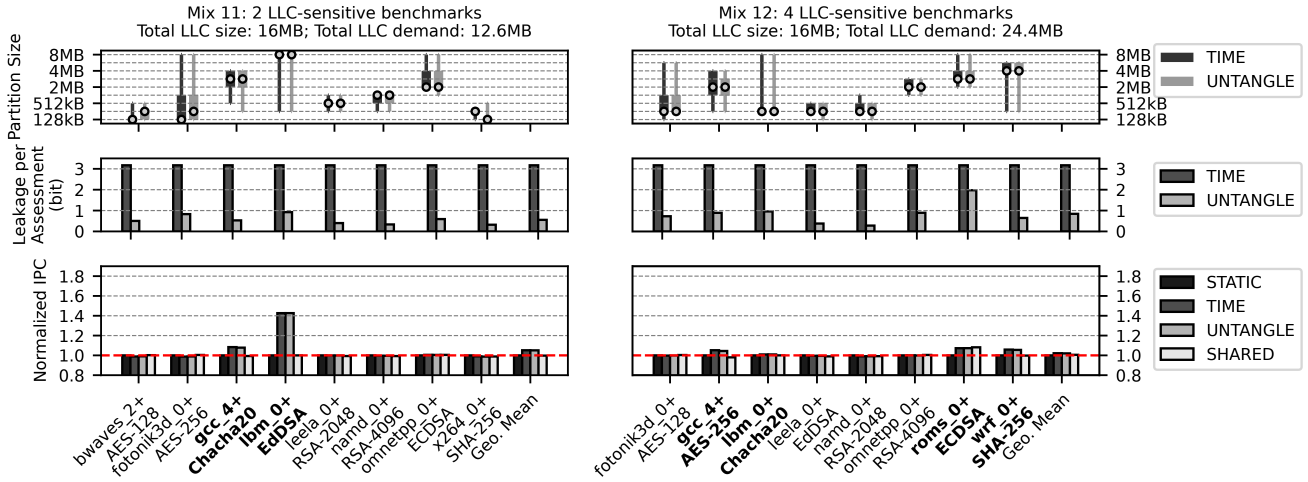


Figure 15: Comparing different partitioning schemes for workloads Mix 11 and Mix 12.

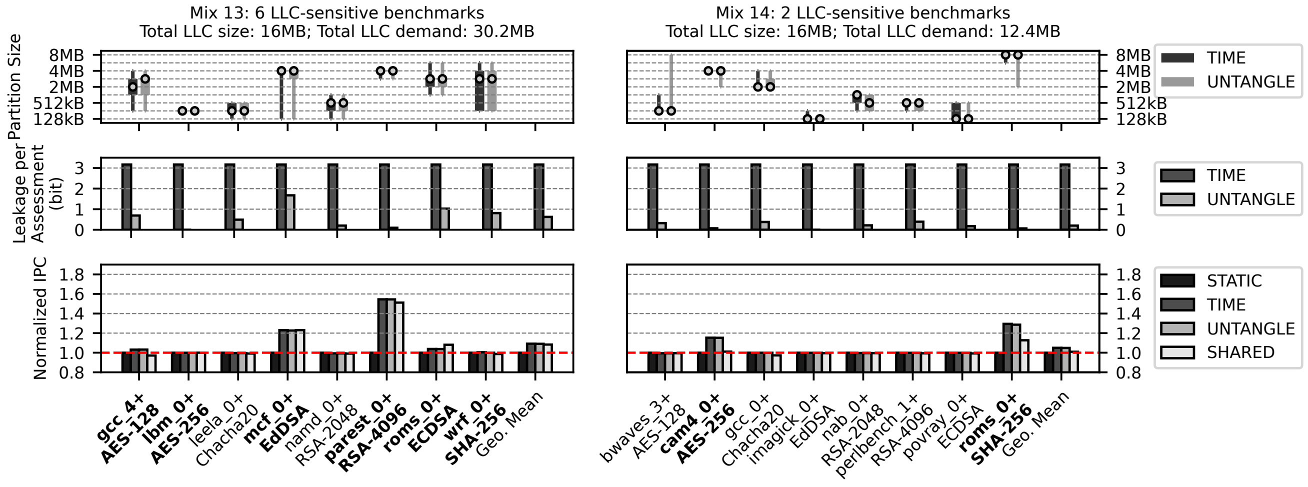


Figure 16: Comparing different partitioning schemes for workloads Mix 13 and Mix 14.

[27] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 974–987.

[28] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *2016 IEEE international symposium on high performance computer architecture (HPCA)*. IEEE, 406–418.

[29] Fangfei Liu and Ruby B. Lee. 2014. Random Fill Cache Architecture. In *47th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'14)*.

[30] Andrew C. Myers, Lantian Zheng, Steve Zdancewic, Stephen Chong, and Nathaniel Nystrom. 2006. *Jif 3.0: Java information flow*. http://www.cs.cornell.edu/jif

[31] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology – CT-RSA 2006*, David Pointcheval (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–20.

[32] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. In *30th USENIX Security Symposium (USENIX Security 21)*.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[34] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. 2021. Systematic Analysis of Randomization-based Protected Cache Architectures. In *IEEE Symposium on Security and Privacy (S&P'21)*.

[35] Moinuddin K Qureshi. 2019. New attacks and defense for encrypted-address cache. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 360–371.

[36] Moinuddin K Qureshi and Yale N Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. IEEE, 423–432.

[37] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. 2021. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-partitioning. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*. IEEE, 37–49.

[38] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. ConTExT: A Generic Approach for Mitigating Spectre. In *27th Annual Network and Distributed System Security Symposium (NDSS)*.

[39] Brian C Schwedock and Nathan Beckmann. 2020. Jumanji: The Case for Dynamic NUCA in the Datacenter. In *2020 53rd Annual IEEE/ACM International Symposium*

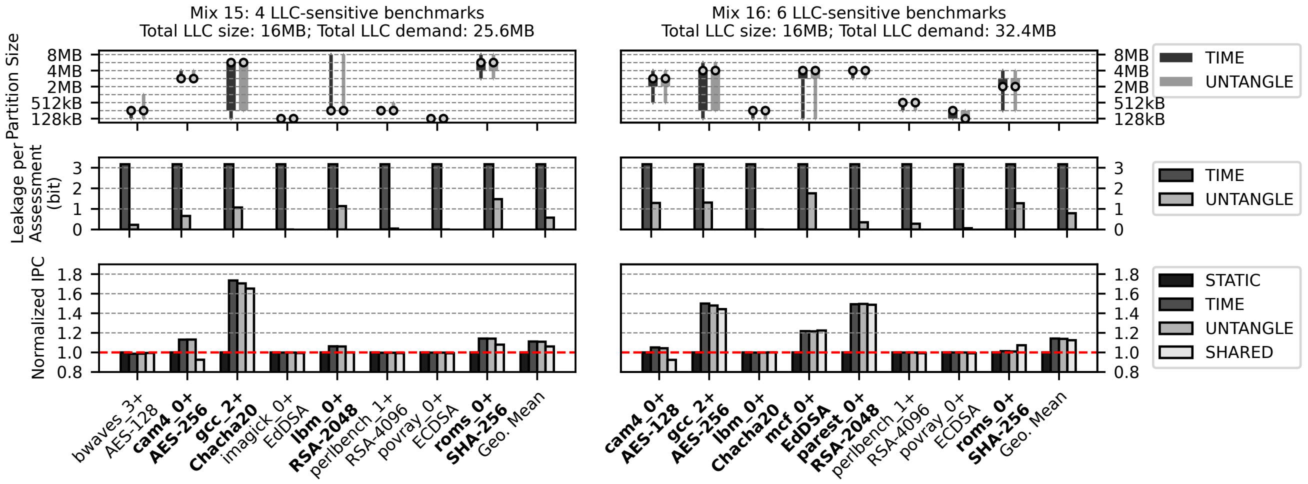


Figure 17: Comparing different partitioning schemes for workloads Mix 15 and Mix 16.

on Microarchitecture (MICRO). IEEE, 665–680.

[40] Kaiming Shen and Wei Yu. 2018. Fractional programming for communication systems—Part I: Power control and beamforming. *IEEE Transactions on Signal Processing* 66, 10 (2018), 2616–2630.

[41] Mingshen Sun, Tao Wei, and John CS Lui. 2016. Taintart: A practical multi-level information-flow tracking system for Android runtime. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 331–342.

[42] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2020. PhantomCache: Obfuscating Cache Conflicts with Localized Randomization. In *27th Annual Network and Distributed System Security Symposium (NDSS).*

[43] Mohammadkazem Taram, Xida Ren, Ashish Venkat, and Dean Tullsen. 2022. SecSMT: Securing SMT processors against contention-based covert channels. In *USENIX Security Symposium.*

[44] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. 2022. TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering. In *31st USENIX Security Symposium (USENIX Security 22).* 989–1007.

[45] Mohit Tiwari, Hassan M. G. Wassel, Bita Mazloom, Shashidhar Mysore, Frederic T. Chong, and Timothy Sherwood. 2009. Complete information flow tracking from the gates up. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS.* ACM, 109–120.

[46] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oguz Ergin. 2022. Composable Cachelets: Protecting Enclaves from Cache Side-Channel Attacks. In *2022 USENIX Security Symposium.*

[47] Daniel Townley and Dmitry Ponomarev. 2019. SMT-COP: Defeating Side-Channel Attacks on Execution Units in SMT Processors. In *28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19).* 43–54.

[48] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2019. BRB: Mitigating Branch Predictor Side-Channels. In *IEEE International Symposium on High Performance Computer Architecture (HPCA'19).*

[49] Shuai Wang, Yuyan Bao, Xiao Liu, Pei Wang, Danfeng Zhang, and Dinghao Wu. 2019. Identifying Cache-Based Side Channels through Secret-Augmented Abstract Interpretation. In *28th USENIX security symposium (USENIX security 19).* 657–674.

[50] Shuai Wang, Pei Wang, Xiao Liu, Danfeng Zhang, and Dinghao Wu. 2017. CacheD: Identifying Cache-Based Timing Channels in Production Software. In *26th USENIX Security Symposium (USENIX Security 17).* 235–252.

[51] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. 2016. SecDCP: secure dynamic cache partitioning for efficient timing channel protection. In *Proceedings of the 53rd Annual Design Automation Conference.* 1–6.

[52] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture.* 494–505.

[53] Samuel Weiser, Andreas Zankl, Raphael Spreitzer, Katja Miller, Stefan Mangard, and Georg Sigl. 2018. DATA–Differential Address Trace Analysis: Finding Address-based Side-Channels in Binaries. In *27th USENIX Security Symposium (USENIX Security 18).* 603–620.

[54] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX Security.*

[55] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. Microwalk: A framework for finding side channels in binaries. In *Proceedings of the 34th Annual Computer Security Applications Conference.* 161–173.

[56] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. 2017. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security.* 859–874.

[57] Zhemin Yang and Min Yang. 2012. Leakminer: Detect information leakage on Android with static taint analysis. In *2012 Third World Congress on Software Engineering.* IEEE, 101–104.

[58] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Security Symposium.*

[59] Yuanyuan Yuan, Zhibo Liu, and Shuai Wang. 2023. CacheQL: Quantifying and Localizing Cache Side-Channel Vulnerabilities in Production Software. In *32nd USENIX Security Symposium (USENIX Security 23).*

[60] Yuanyuan Yuan, Qi Pang, and Shuai Wang. 2022. Automated side channel analysis of media software with manifold learning. In *31st USENIX Security Symposium (USENIX Security 22).* 4419–4436.

[61] Danfeng Zhang, Aslan Askarov, and Andrew C Myers. 2011. Predictive mitigation of timing channels in interactive systems. In *Proceedings of the 18th ACM conference on Computer and communications security.* 563–574.

[62] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. 2021. A Lightweight Isolation Mechanism for Secure Branch Predictors. In *58th ACM/IEEE Design Automation Conference (DAC'21).*

Received 2022-10-20; accepted 2023-01-19