Controlled Preemption: Amplifying Side-Channel Attacks from Userspace

Yongye Zhu

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2025-125
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2025/EECS-2025-125.html

May 20, 2025

Controlled Preemption: Amplifying Side-Channel Attacks from Userspace

by

Yongye Zhu

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Master of Science

in

Computer Science

in the

Graduate Division

of the

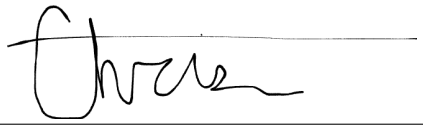
University of California, Berkeley

Committee in charge:

Associate Professor Christopher W. Fletcher, Chair
Associate Professor Yakun Sophia Shao

Spring 2025

The dissertation of Yongye Zhu, titled Controlled Preemption: Amplifying Side-Channel Attacks from Userspace, is approved:

Chair  Date 5/11/2025

 Date 5/20/2025

University of California, Berkeley

Controlled Preemption: Amplifying Side-Channel Attacks from Userspace

Abstract

Controlled Preemption: Amplifying Side-Channel Attacks from Userspace

by

Yongye Zhu

Master of Science in Computer Science

University of California, Berkeley

Associate Professor Christopher W. Fletcher, Chair

Microarchitectural side channels are an ongoing threat in today's systems. Yet, many side-channel methodologies suffer from low temporal resolution measurement, which can either preclude or significantly complicate an attack.

This paper introduces *Controlled Preemption*, an attack primitive enabling a *single unprivileged* (user-level) attacker thread to repeatedly preempt a victim thread after colocating with that victim thread on the same logical core. Between preemptions, the victim thread executes zero to several instructions—sufficiently few to enable high-resolution side channel measurements.

The key idea in Controlled Preemption is to exploit scheduler fairness heuristics. Namely, that modern thread schedulers give a thread A the ability to preempt another thread B until a fairness tripwire (signaling that A is starving B) fires. We show how this idea enables hundreds of short preemptions before tripping the fairness tripwire is robust to noise and applies to both the Linux CFS and EEVDF schedulers. We also develop a technique that helps colocate the attacker and victim threads onto the same logical core, an attacker capability overlooked by prior work.

Our evaluation tests Controlled Preemption in the context of several different victim programs, victim privilege levels (inside and outside of Intel SGX) and choices of side channel. In each attack, we demonstrate results that are competitive with prior work but make fewer assumptions (e.g., require only user-level privilege or require fewer colocated attacker threads).

To my parents, for their unwavering love and support.

Contents

Contents **ii**

List of Figures **iii**

List of Tables **v**

1 Introduction **1**

2 Background **5**
2.1 Linux Completely Fair Scheduler 5
2.2 Microarchitectural Side-Channel Attacks 7

3 Threat Model **8**

4 Controlled Preemption **9**
4.1 Controlled Preemption on the CFS 9
4.2 Controlled Wake Up 11
4.3 Evaluation 13
4.4 Achieving Core Colocation 19
4.5 EEVDF Scheduler 19

5 Proof-of-Concept Exploits **21**
5.1 Attacking T-Table AES 21
5.2 Attacking SGX Enclaves 23
5.3 Monitoring the BTB Side Channel 26

6 Mitigations **29**

7 Related Work **31**

8 Conclusion **33**

Bibliography **34**

List of Figures

1.1 Techniques to enable high temporal resolution channel-agnostic side-channel analysis from userspace. 2

4.1 Detailed illustration of Controlled Preemption where $\Delta = \tau_{victim} - \tau_{attacker}$. . . 10
4.2 Pseudo-code describing two methods to force the attacker to wake up at a precise time. 12
4.3 Temporal resolution of Controlled Preemption using different wake up methods and performance degradation techniques. Data is shown as a histogram, in terms of the number of victim instructions retired per preemption. Different lines correspond to different values of δ. 15
4.4 The number of repeated preemptions (y-axis) achieved by Controlled Preemption when varying $I_{attacker} - I_{victim}$ (x-axis). The plot assumes Method 1 (Chapter 4.2) although the result transfers to Method 2. Blue dots are observations (concrete settings of $I_{attacker}$ and I_{victim}) on our test machine. The curve indicates the expected number of consecutive preemptions. 16
4.5 The number of repeated preemptions (y-axis) is a function of the victim's `nice` value (x-axis). We set the attacker to the default `nice` value 0. $I_{attacker} - I_{victim}$ ranges between 10 µs and 15 µs. 17
4.6 vruntime progression in a 'noisy' system with a third noise thread. The zoom-in shows the vruntimes of the victim and noise thread at the point where they converge (Sample 54033). 18
4.7 Temporal resolution of Controlled Preemption on EEVDF. The figure represents the same experiment as that shown in Chapter 4.3b, but using EEVDF. 20

5.1 Heatmap illustrating Flush+Reload results for a single AES run. The y-axis denotes attacker measurements for each of the 16 cache lines making up the T-table (yellow indicates a hit during Reload; purple indicates a miss). Each column (x-axis) denotes a sample where the attacker preempts the victim. The first four accesses (those made in the first round) are circled in red. 23
5.2 Cache probe latency trace for the base64 decoding function running inside an SGX enclave. Blue and orange traces denote probe latencies for the two LUT eviction sets. Red is the latency to probe the instruction cache eviction set. The grey (white) area is the victim performing the validity (decode) loop. 24

5.3 BTB prime probe code 26
5.4 Victim control path when running `medtls_mpi_gcd` function for $a = 1001941$ and $b = 300463$. If the instruction latency is high, the victim has executed the block that invalidates the BTB entry. So the prefetcher won't prefetch the target fetching location. 27

List of Tables

2.1 Relevant CFS configurations. 7

Acknowledgments

This thesis is the culmination of a brief but transformative chapter in my life. Though my time as a master's student was short, it was marked by moments of challenge, growth, and quiet wonder. Each step—no matter how difficult or uncertain—has contributed to the shape of this work. And yet, this journey was never mine alone; it was made possible by the guidance, support, and generosity of many, without whom none of this would have been possible.

First and foremost, I want to thank my advisor, Chris Fletcher. Chris is not only a brilliant researcher but also a generous and thoughtful mentor. Even though I know nearly nothing about the topic of my research beforehand, Chris is patient and guides me through every step. Chris not only taught me the knowledge of a specific field, but he also taught me how to think as a researcher, how to formulate a problem, and how to effectively deliver a result through papers and presentations, the skills that can be carried on no matter which field I will be working in, and beyond research. I wish you all the best in your future as a professor, and I really hope we'll get a chance to work together again someday.

Next, I want to thank all of my collaborators, Boru Chen and Neil Zhao, for walking alongside me. Boru introduced me to my thesis problem after I first arrived at UIUC and had discussed it with me along the way. This work won't happen without Boru's gentle help and all the brainstorming. I can always learn a lot every time I talk with him. Moreover, through our discussion, Boru taught me how to actually think and ask questions concretely on a problem and how to prepare the results concretely to have an effective discussion with the collaborators each week. Neil, your sharp insights and ability to uncover what lies just beneath the surface strengthened this work immeasurably. I would still remember the discussion we had on how to properly flush the iTLB to get better single-step results. I hope you all the best in your future PhD and professor journey.

Next, I want to express my appreciation to all my labmate in FPSG: Sushant Dinesh, Rutvik Choudhary, Nandeeka Nayak, Hannah Leung, Boru Chen, Alan Wang, Timor Averbuch, Alice Wu, Alex Thomas, Yuxin Jin, Chenxi Wan, Anto Kam, Archie Lee, Yingchen Wang, Yan Zhu, Shashank Anand, Kris Dong, Flavien Solt, Ian Mclellan, SooHyuk Cho. It is absolutely enjoyable to be in the same lab and engage in fun research questions with you. And I want to express my special thanks to Sushant, who led me to a completely new field than my thesis work. His vast knowledge of formal methods amused me, and I learn a lot from both his knowledge of formal methods as well as thought process as a senior PhD student. For others, getting to work with you all has been an invaluable asset throughout my career. I hope you all the best in your professional careers.

I would also like to thank my girlfriend, Serena Gu, for your love and encouragement along the way. Although we have been long-distance for over a year, I always feel like you are by my side. You're such an amazing girlfriend, and I can't wait to join you on the East Coast later this year.

Last but not least, I would like to thank my parents for giving me life, for raising me, and for giving me the chance to get an education. Moving from China to the U.S. seven

years ago wasn't easy, especially when we didn't know what to expect. But your bravery and strength have inspired me every step of the way. I have learned to never be afraid of the unknown because of you. Your support has meant the world to me, and I'll keep doing my best to make you proud.

Chapter 1

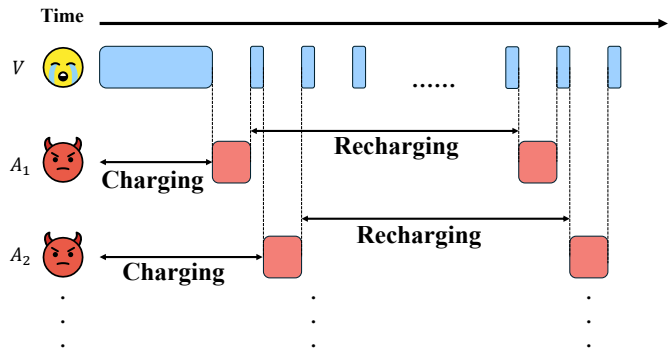
Introduction

Microarchitectural side-channel attacks exploit the hardware resources shared between mutually distrusting programs [41, 15, 24, 73, 25, 23, 2, 39, 72]. In such attacks, an attacker program monitors a victim program's utilization of shared resources to infer sensitive victim information like cryptographic keys [73, 15, 25, 20, 10, 74, 45, 24]. These attacks pose a serious threat to both commercial multi-tenant cloud environments [53, 29, 80] and local client environments [41, 56, 39].

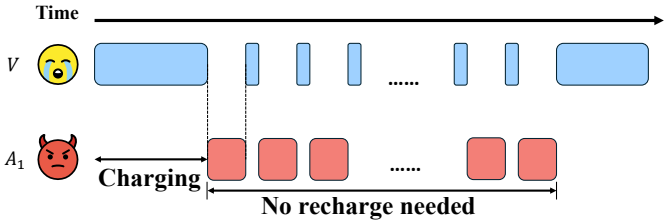
A fundamental characteristic in any microarchitectural side-channel attack is its *temporal resolution*, i.e., the number of victim instructions that execute/retire in between attacker measurements. Higher temporal resolution enables new attacks or otherwise significantly reduces the requirements of existing attacks. Consider for example the T-table AES first-round attack [48, 25]. Given perfect temporal resolution, a cache-timing attack on the T-Tables can extract the key after the AES routine runs a single time. By contrast, an attack with low temporal resolution (e.g., concurrently/continuously measuring the channel or measuring the channel once after the AES routine completes) requires 100s [52] to 1000s [77] of victim runs (due to cache pollution from later rounds and other sources of accumulated noise).

This paper advances a line of research that enables extremely high temporal resolution side channel analysis, i.e., giving the attacker the ability to nearly *single step* the victim program and perform side channel measurements in between each step. This line of work is based on thread preemption. In a nutshell, the attacker coerces the OS kernel to interleave the attacker and victim threads' executions onto the CPU, giving the attacker the opportunity to monitor shared states in between short periods of victim execution.

Existing work on malicious preemption has limitations. For example, works targeting the Enclave threat model [26, 45, 38, 60, 61] (e.g., SGX Step [61]) enable best-case temporal resolution and minimal noise, but require supervisor privilege. While there does exist a parallel line of work that enables similar capabilities from userspace [25, 7, 54, 6] (that started with Cache Games [25]), it requires a large number (e.g., from 10s to 100s) of attacker threads to assist in preemption. Requiring a large number of threads is disadvantageous for a number of reasons. More threads make the attack less stealthy and may even be disallowed depending on the system (e.g., through `ulimit`). Further, in practice, attacks



(a) Prior work [25, 54, 7, 6]



(b) Controlled Preemption (this paper)

Figure 1.1: Techniques to enable high temporal resolution channel-agnostic side-channel analysis from userspace.

involving multiple threads are difficult to implement (e.g., due to thread synchronization requirements) which itself results in coarser-grain temporal resolution [25, 7].

This paper. We present *Controlled Preemption*, a technique that enables a *single unprivileged* attacker thread to repeatedly preempt a victim thread once it has been colocated to the victim's logical core.

To understand Controlled Preemption, we first review why prior work on preemption from userspace requires many threads. See Chapter 1.1a. Assume a single-core system for simplicity. The idea in prior work is for the attacker to spawn and then sleep a thread (say A_1) so that its scheduling priority increases to the point where, on wake up, it preempts the victim thread V. Conceptually, a thread's scheduling priority is set to that of the thread that gets preempted on wake up. Thus, once an attacker thread preempts the victim, it needs to sleep again to "recharge" its priority. If the attacker wishes to preempt the victim multiple times, it requires multiple threads (A_1, A_2, etc.). Once thread A_1 preempts V and begins to recharge, thread A_2 is responsible for the next preemption, at which point it begins to recharge, and so on.

The key observation enabling Controlled Preemption is general and applies to multiple thread schedulers: *To improve system responsiveness, a scheduler will strive to allow a well-slept thread A to immediately preempt a running thread V, even if V has not completed its minimum scheduling quantum. Further, subject to fairness checks, a scheduler will al-*

low thread A to repeat the above process. The latter sentence is key to our single-thread attack. Through careful inspection of current scheduler designs, we find that scheduler fairness heuristics create what we call a *preemption budget* which enables an attacker thread to repeatedly preempt a victim up to the point where the amount of attacker CPU time exceeds a threshold (at which point the scheduler determines that the attacker is starving the victim).

Based on the notion of a preemption budget, we instantiate and thoroughly characterize a Controlled Preemption primitive and show that it enables a colocated single attacker thread to nearly single step[1] a victim thread hundreds of times without recharging, as shown in Chapter 1.1b. We show that the notion applies to both the Linux Completely Fair Scheduler (CFS) [47] and the more-recent Earliest Eligible Virtual Deadline First (EEVDF) scheduler [58, 12].

Note that in multicore systems, the attacker additionally needs a technique to colocate the attacker thread with the victim thread on the same logical core. Prior work ignores this step, either limiting their attacks to a single-core system [25] or assuming the attacker and victim threads are pinned to the same core throughout the attack [54, 7, 6]. Both of these assumptions are unrealistic. In this work, we develop a simple yet effective core-level colocation technique that exploits the scheduler's load-balancing logic.

Finally, we demonstrate three proof-of-concept attacks leveraging Controlled Preemption. These attacks cover a range of side channels, victim programs and victim program privilege levels. Specifically, we show that the attacker can use Controlled Preemption to mount an AES T-table first-round attack using Flush+Reload [73]. This attack achieves the same attack efficiency as prior work [7] (leaking the upper nibbles of each AES key byte in ~ 5 victim runs) but requires only one colocated attacker thread instead of 40 threads. We then use Controlled Preemption to demonstrate an attack on an SGX victim to extract sensitive information that can lead to full RSA key recovery using a Prime+Probe on the last-level cache [41]. To our knowledge, this is the first demonstration of an SGX Step-like attack mounted from userspace. Finally, we utilize Controlled Preemption to recover the secret-dependent control flow of an SGX victim using a BTB side channel [75], providing confidence that Controlled Preemption can be applied to multiple side channels.

In summary, this paper makes the following contributions:

- We develop Controlled Preemption, the first userspace framework that nearly single steps a victim thread using as few as one colocated attacker thread.
- We comprehensively characterize the Controlled Preemption primitive in the context of two widely-deployed popular thread schedulers (the Linux CFS and EEVDF schedulers). To our knowledge, we are the first to perform any characterization of user-level malicious preemption using the EEVDF scheduler.

[1] As detailed in Chapter 4.3, the victim thread executes only a few instructions between preemptions. In fact, we show that a majority of preemptions result in the victim executing *one* instruction when Controlled Preemption is combined with performance degradation.

- We develop an attack technique that colocates the attacker and victim threads onto the same logical core.
- We demonstrate three proof-of-concept attacks that show how Controlled Preemption can improve existing attacks on multiple victim programs, inside and outside of SGX and across multiple side channel types.

We have open sourced an implementation of Controlled Preemption, as well as this paper's evaluation, here: `https://github.com/FPSG-UIUC/Controlled-Preemption`.

Responsible disclosure. We disclosed our findings to the Linux kernel security team, who confirmed our findings and designated Controlled Preemption as a low-priority threat. They recommended enabling `NO_WAKEUP_PREEMPTION` to prevent the waking attacker thread from immediately preempting the victim thread at the cost of system responsiveness.

Parts of this thesis have been published in the following papers: [82].

Chapter 2

Background

2.1 Linux Completely Fair Scheduler

In Linux, threads that are ready to execute but not scheduled (or *runnable threads*) are stored in a per-logical core software structure named the *runqueue*. The *thread scheduler* is responsible for deciding when to preempt the current thread and switch to another in the runqueue. In this paper, we focus on the widely used Completely Fair Scheduler (CFS) [47, 32] and discuss the transferability of our techniques to the latest Earliest Eligible Virtual Deadline First (EEVDF) scheduler in Chapter 4.5.

In the CFS, each thread is assigned a virtual runtime (*vruntime*). When a thread executes, its vruntime is incremented by $\Delta\tau = \alpha\Delta t$, where Δt is the thread's real-world execution time and α is the increment rate determined by the thread's priority. $\alpha = 1$ under the default priority. A high priority thread has a small α value, allowing the thread to have a longer execution time for the same amount of $\Delta\tau$. In this paper, we denote vruntime by τ and real-world time by t.

The CFS achieves fair scheduling by ensuring that the difference between threads' vruntimes, in the local runqueue, does not exceed the kernel parameter *sysctl_sched_latency* (S_{bnd} for short). We call this policy the *fair scheduling invariant*. The exact value of S_{bnd} depends on the total number of cores in the system, and is set to 24 ms in our evaluated system (Chapter 2.1).

The CFS's exact scheduling behavior depends on many factors. Using Linux 6.5 as an example, we now briefly overview the characteristics that are important for our work. Complementary to the runqueue, threads that are not ready to execute, e.g., are waiting on blocking IO events or are asleep, are stored in a shared *waitqueue*.[1] Then, at a high level, the CFS can change which thread is running on a given logical core in three circumstances:

- *Scenario 1.* When, within the local runqueue, a thread's vruntime exceeds another thread's vruntime by S_{bnd}.

[1] To simplify the presentation, we assume a single system-wide waitqueue. In reality, the system uses different waitqueues depending on the event type that is blocking a given thread.

- *Scenario 2.* When a thread wakes up, i.e., is removed from the waitqueue and enters the local runqueue.
- *Scenario 3.* When a thread becomes blocked, i.e., is removed from the local runqueue and enters the waitqueue.

We discuss each in detail below.

Scenario 1: Runqueue stationary. First consider the case when no threads are moving between the runqueue and the waitqueue. Here, the CFS will select the thread with the smallest vruntime in the runqueue and schedule the thread onto the hardware. We call this thread A. To avoid excessive context switching, A is allowed to execute for a minimum time slice configured by *sysctl_sched_min_granularity* (or S_{min} for brevity). S_{min} is 3 ms in our evaluated system. Once A runs for S_{min} time, the CFS checks whether the fair scheduling invariant is violated. If so, A is descheduled, and the CFS chooses the thread with the smallest vruntime to schedule. Otherwise, A is scheduled again. As will be discussed in the next scenario, this minimum time slice is only enforced in Scenario 1. This detail will be critical for our work.

Scenario 2: A thread is waking up. Now consider when a thread B wakes up, B is removed from the waitqueue and is added to the runqueue. Since B was blocked by an IO request or sleep, B's vruntime can be significantly behind the vruntime of other threads. Consequently, B could monopolize the CPU for a long time before its vruntime catches up with others. To prevent this undesirable behavior when awakening B, the CFS assigns B an adjusted vruntime of

$$\tau_{wakeup} = \max(\tau_{min} - S_{slack}, \tau_{sleep}), \tag{2.1}$$

where τ_{min} is the smallest vruntime among existing threads in the local runqueue, S_{slack} is a fixed value determined by S_{bnd}, and τ_{sleep} is the vruntime of thread B at the moment when B was blocked. This heuristic prevents B from monopolizing the CPU and ensures that B's vruntime strictly increases. In the evaluated Linux kernel, S_{slack} is 12 ms.

Notably, after thread B's vruntime is adjusted to τ_{wakeup}, B can preempt the current running thread A *even if A has yet to complete its minimum time slice.* Our attack exploits this preemption logic. A preemption occurs when the following condition is true:

$$\tau_{curr} - \tau_{wakeup} > S_{preempt}, \tag{2.2}$$

where τ_{curr} is the thread A's vruntime and $S_{preempt}$ is a fixed threshold configured by the kernel parameter *sysctl_sched_wakeup_granularity*. $S_{preempt}$ is 4 ms in our system. Note that the CFS only decides between scheduling the running thread A and the waking thread B. Even if there is a third runnable thread C whose vruntime is smaller than the vruntime of A and B, C will not be considered for scheduling.

Scenario 3: A thread is blocked. Finally, consider when the current running thread is about to be blocked due to an IO request or a voluntary sleep. In this case, the CFS will de-schedule the current running thread and schedule the thread with the smallest vruntime from the local runqueue.

Chapter 2.1 summarizes the CFS configurations relevant to this paper and their default values. Note that their default values depend on the number of cores in the system.

Table 2.1: Relevant CFS configurations.

Config.	Default Value (Our System)	Description
S_{bnd}	λ×6 ms (24 ms)[1]	Upper bound of vruntime difference
S_{min}	λ×0.75 ms (3 ms)	Length of the minimum time slice
S_{slack}	$S_{bnd}/2$ (12 ms)[2]	A waking thread's max. vruntime lag
$S_{preempt}$	λ×1 ms (4 ms)	The vruntime threshold that the waking thread can preempt the current thread

[1] λ is a system-specific scaling factor and equals $\min(\log_2(\#\text{cores}) + 1, 4)$. $\lambda = 4$ in our 16-core system.
[2] This assumes the `GENTLE_FAIR_SLEEPERS` scheduler feature is set, which is the default configuration; otherwise, $S_{slack} = S_{bnd}$.

2.2 Microarchitectural Side-Channel Attacks

Microarchitectural side-channel attacks exploit shared hardware resources to exfiltrate sensitive information. Commonly exploited resources include CPU caches [24, 73, 41, 15, 52], TLBs [22, 59], coherence directories [72], on-chip interconnects [65, 49, 13], arithmetic ports [2], and BTBs [17, 38, 18, 75, 79].

We focus on stateful/persistent side-channel attacks [24, 73, 41, 15, 22, 59, 72, 17, 38, 79, 18, 75] where the victim encodes (transmits) a message into a channel/hardware structure that can be decoded (received) later. For these attacks, the attacker needs to interleave its execution with the victim to monitor the channel. Consider an example where the attacker runs on core A to perform an L1 cache Prime+Probe attack. During the attack, the attacker (1) preconditions the channel by priming specific L1 sets, (2) triggers the victim's execution on core A and (3) recovers the victim's memory access behavior by probing the L1 sets. Steps (1)–(3) can be repeated multiple times to extract more information from the victim execution.

A major complication is how much victim code runs during Step 2. Ideally, the attacker would like to interleave its execution at a fine grain, ideally performing Step 1 and Step 3 before and after each of the victim's sensitive memory accesses. This is usually impossible, due to coarse-grain thread scheduling, which adds significant noise that worst-case blocks the attack and best-case requires additional techniques to overcome [48, 27, 10, 66].

Chapter 3

Threat Model

We assume that the attacker and victim programs are colocated on the same physical machine, following many prior works [48, 25, 7]. We assume an unprivileged attacker that can interact with the Linux kernel using standard system calls like `fork` and `sleep` and can invoke the victim (start its execution) [45, 48, 77, 10, 74, 69, 44, 35, 20, 21]. The victim thread can be an unprivileged user-level thread or a thread inside an Intel SGX enclave.

We do not make additional assumptions about the execution environment or the priority of the victim thread, beyond that the kernel uses the Linux CFS or EEVDF schedulers to schedule the victim thread. Finally, we do not assume the availability of Simultaneous Multi-Threading (SMT).

Chapter 4

Controlled Preemption

In this section, we introduce *Controlled Preemption*, a series of techniques that enable a single unprivileged colocated attacker thread to interleave its execution with a victim's at a high frequency (e.g., one to tens of victim instructions per interleaving), enabling high-resolution side-channel observations.

Controlled Preemption is possible because thread schedulers enable blocked threads to reclaim the CPU *precisely* at the moment when they are due to become unblocked—as opposed to *only after* the currently running thread has exceeded some minimum scheduling quantum (e.g., S_{min}). This is by design and enables thread schedulers to be more responsive. For example, when a thread calls `sleep`(δ), it ideally wishes to be asleep for precisely δ time. Likewise, when data becomes available (e.g., network packets arrive), the thread responsible for processing that data should get CPU time immediately.

We show how the above characteristic can be exploited by an adversary to preempt a victim thread at a moment of the attacker's choosing. More subtly, we show how the above enables *a single attacker thread* to *repeatedly* preempt the victim at a high frequency, enabling fine-grain side channel measurements. Repeated preemption does not "come for free"—a scheduler's fairness heuristics (e.g., those discussed in Chapter 2.1) *should* prevent a malicious thread from denying service through repeated preemption. We show how schedulers' fairness heuristics can be avoided for a long enough period of time (enabling hundreds to thousands of preemptions) to attack security-critical software.

4.1 Controlled Preemption on the CFS

We now show how Controlled Preemption can be implemented using the CFS scheduler (Chapter 2.1). The key observation is that the $S_{slack} > S_{preempt}$ characteristic in the CFS creates an $(S_{slack} - S_{preempt})$-time *preemption budget* within which the attacker can repeatedly preempt the victim. Echoing earlier discussion, setting $S_{slack} > S_{preempt}$ is deliberate and important for ensuring system responsiveness. By 'preemption budget', we mean that the attacker can preempt the victim an arbitrary number of times, until the vruntime difference

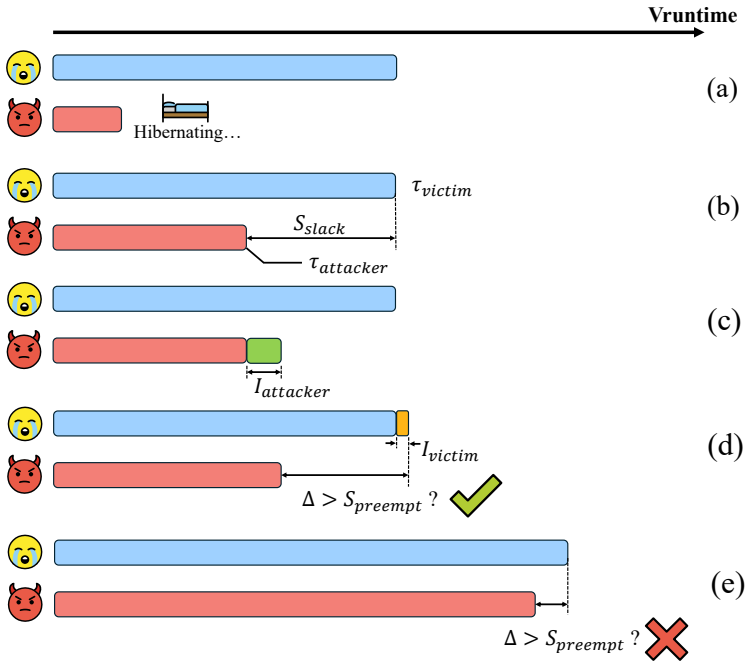


Figure 4.1: Detailed illustration of Controlled Preemption where $\Delta = \tau_{victim} - \tau_{attacker}$.

between the attacker and victim threads is smaller than $S_{preempt}$. Beyond this point, no more preemptions are possible until the victim's vruntime increases by a sufficient amount. For the parameters we use in the paper, $S_{slack} - S_{preempt} = 8$ ms which is sufficient time for the attacker to complete its attack. In the event that this is not sufficient, we discuss several methods to extract a longer trace of victim activity in Chapter 4.3 and Chapter 5.2.

We now explain how the attacker thread A can repeatedly preempt other threads in the same runqueue at a high frequency. For simplicity, we consider an example illustrated in Chapter 4.1 where a single logical core is exclusively shared between an attacker thread A and a victim thread V. We will discuss the implications of the core being shared with more (noisy) threads in Chapter 4.3 and how the attacker can colocate with the victim on the same logical core in Chapter 4.4.

Let $\tau_{attacker}$ and τ_{victim} denote the attacker's and victim's vruntimes, respectively. Let $I_{attacker}$ denote the time it takes for the attacker to perform a side-channel measurement. To enable repeated preemptions, we require that $I_{attacker}$ be small, i.e., $I_{attacker} < S_{slack} - S_{preempt}$. Let I_{victim} denote the amount of time that the attacker wishes the victim to run for in between preemptions. For simplicity, we consider both the attacker and the victim have the default priority and $\alpha = 1$, therefore $\Delta\tau_{victim} = I_{victim}$ and $\Delta\tau_{attacker} = I_{attacker}$. We will discuss the implications of priority in Chapter 4.3.

The attacker requires a method to wake up at a specific time and be considered for scheduling. We discuss two mechanisms for this task in Chapter 4.2 and assume the first of those methods (where the attacker uses `sleep`(δ)) here. The attacker will use `sleep` in two

contexts (explained in the following paragraphs). In the first context, it will set $\delta > 2 * S_{bnd}$ (any value $> 2 * S_{bnd}$ will do); in this case, we say the attacker is *hibernating*. In the second, it will set $\delta = I_{victim}$; which we refer to as the attacker *napping*.

The attack begins with the attacker hibernating (Chapter 4.1 (a)). When the attacker thread unblocks itself and enters the runqueue, its vruntime will be assigned to the left-hand argument of the max function in Equation 2.1, i.e., $\tau'_{attacker} = \tau_{victim} - S_{slack}$, where the tick mark $'$ (e.g., $\tau'_{attacker}$) denotes a new/updated vruntime. This is shown in Chapter 4.1 (b). Combining this with the relation $S_{slack} > S_{preempt}$ from earlier, Equation 2.2 says that the attacker immediately preempts the victim.

The attacker can now monitor and preempt the victim as follows. Once the attacker begins executing, it performs a side-channel measurement (i.e., the steps to pre-condition/receive on the channel it is measuring; c.f. Chapter 2.2) and then naps. Hence, it's vruntime increases by $I_{attacker}$ as shown in Chapter 4.1 (c). This triggers the CFS to schedule and execute instructions from the victim thread. Once the victim runs for I_{victim} time, the attacker wakes up. W.l.o.g. let $I_{attacker} > I_{victim}$. Then, the attacker's vruntime will be assigned to the right-hand argument to the max function in Equation 2.1, i.e., $\tau'_{attacker} = \tau_{attacker}$. Based on our earlier requirement that $I_{attacker} < S_{slack} - S_{preempt}$, we further have that Equation 2.2 holds, meaning that the attacker will preempt the victim as shown in Chapter 4.1 (d).

The above process (Chapter 4.1 (b)-(d)) can repeat, enabling repeated preemptions. An important detail is that because $I_{attacker} > I_{victim}$, $\tau_{victim} - \tau_{attacker}$ gradually shrinks and Equation 2.2 will eventually return false as shown in Chapter 4.1 (e). At this point, preemption will fail. This gives the attacker approximately $\left\lceil \frac{S_{slack} - S_{preempt}}{I_{attacker} - I_{victim}} \right\rceil$ preemptions to complete its attack. It is possible to stretch the 'preemption budget' by setting a larger I_{victim}, but doing so allows the victim to make more forward progress in between preemptions which may aggravate the side-channel analysis.

4.2 Controlled Wake Up

Following the example from the previous section, where we assumed only an attacker and victim thread, the attacker needs a way to wake up at a precise time and be considered for scheduling. We utilize high-resolution hardware timers for this purpose. We explore two methods to program these timers in userspace, enabling the attacker to interrupt the victim potentially after the victim has executed only a single instruction. Chapter 4.2 provides pseudo-code for both methods.

Method 1: Nanosleep.

The first method employs the *nanosleep* system call with a specified sleep duration δ. When the attacker invokes *nanosleep*, the operating system removes it from the runqueue (Scenario 3 from Chapter 2.1) and programs a one-shot hardware timer to trigger an interrupt after δ time passes. With the attacker blocked, the scheduler selects the victim thread to

```
// tweak timerslack
// to the lowest value 1ns
prctl(PR_SET_TIMERSLACK, 1)
// sleep to make
// vruntime lowest
sleep(5s)
while(true){
    // sleep for interval
    nanosleep(interval)
    // call attacker
    // procedure
    attacker()
}
```

(a) Method 1: Nanosleep

```
#define SIG SIGRTMIN
// register signal handler
// as attacker procedure
sigaction(SIG, attacker)
// create a timer that
// sends SIG when fires
timer_create(SIG)
// set periodic interval
timer_settime(interval)
// sleep to make
// vruntime lowest
sleep(5s)
// pause main routine
while(1) pause()
```

(b) Method 2: Timer

Figure 4.2: Pseudo-code describing two methods to force the attacker to wake up at a precise time.

run. After the interval δ elapses, the hardware timer fires, the victim's execution is interrupted, and control switches to the kernel's interrupt handler. Finally, the kernel wakes up the attacker thread, adds it back to the runqueue, sets its new vruntime and performs the preemption check (as described in Chapter 2.1).

An additional OS parameter, called *timer_slack*, controls the time allowed to pass beyond δ before the OS has to wake up the attacker thread. By default, *timer_slack* is 50μs, which is too coarse for our needs. Fortunately, we can reduce *timer_slack* to a small value (1 ns) using the non-privileged *prctl* syscall with *PR_SET_TIMERSLACK* as the argument.

Method 2: Timers.

The second method uses the POSIX timer API to create a periodic high-resolution timer. The attacker calls *timer_create* to create a timer and *timer_settime* to set it with interval δ. A signal handler is registered to handle timer expirations. After setting up the timer, the attacker blocks itself indefinitely by calling *pause* and waits for the timer signal. When the timer expires, a signal is sent to the attacker. The kernel adds the attacker back to the runqueue, sets the attacker's vruntime and performs the preemption check as usual. If the preemption condition is met, the attacker preempts the victim. The attacker's signal handler then executes the side-channel measurement routine. After the signal handler completes, the attacker blocks again, awaiting the next timer signal.

We note, when using the POSIX timer API, setting an analog to *timer_slack* is not necessary: timer interrupts are handled immediately by the kernel while the execution of the attacker's userspace handler is still subject to the preemption check.

Zero Stepping

Setting δ for both of the above methods is non-trivial because time continues to pass when the OS is in the process of scheduling/context switching the victim onto the hardware. That is, δ must be set in a "Goldilocks" fashion similar to SGX Step [61]:

- If δ is set to be too small, the timer fires when the victim is in the process of being scheduled onto the CPU, which interrupts and prevents the victim from making any forward progress before the CPU is yielded back to the attacker.
- If δ is set to be too large, the victim executes more instructions before being preempted, which leads to lower time resolution side-channel analysis.

The former case is called a *zero step* [61, 11]. We analyze the characteristics of this Goldilocks zone in the next section.

Zero steps are benign but waste preemption budget. An oracle can be constructed to filter out signals caused by instructions of interest (Chapter 4.3), and hence data collection when zero steps occur will be omitted. At the same time, each zero step still costs the attacker $I_{attacker} - I_{victim}$ time in the preemption budget.

4.3 Evaluation

We now evaluate the Controlled Preemption primitive along multiple axes: its temporal resolution, the number of preemptions, robustness to noise, techniques for colocation and extensibility to the EEVDF scheduler.

Experiment setup. For the rest of the paper, we run all of our experiments on a desktop machine with a 16-core Intel Core i9-9900K processor and 64 GB RAM. This machine runs Ubuntu 22.04.1 with Linux kernel version 6.5 for CFS and 6.12-rc1 for EEVDF.

Until Chapter 4.3, we perform experiments on a quiescent machine to minimize interference from other processes. As required by our attack, we ensure that the attacker and victim threads are colocated on the same logical core; we show how to achieve this in Chapter 4.4. At experiment launch, the attacker sleeps (hibernates) for 5 seconds to ensure it is assigned the vruntime $\tau_{victim} - S_{slack}$ upon wake-up.

Temporal resolution

We characterize how many instructions the victim executes between interrupts using an eBPF [16] program that records the PC of the first victim instruction whenever the victim is scheduled. To translate the change in the victim PC to the number of instructions retired, we use a victim program that runs a long sequence of same-Byte length instructions in an infinite loop.

Chapter 4.3a and Chapter 4.3c show histograms of the preemption resolution, in terms of how many victim instructions retire per preemption. Each histogram is over 80,000

preemptions. We show results for both wake up methods from Chapter 4.2, varying the sleep and timer interval value δ. As we increase δ, the victim executes more instructions per preemption on average. Notably, for small δ, a majority of preemptions occur after the victim has only executed a small number (< 10) instructions but a sizable percentage of preemptions result in zero steps (Chapter 4.2).

Combining Controlled Preemption with performance degradation. To improve the ratio of single steps to zero steps, we combine Controlled Preemption with well known *performance degradation* techniques [3, 37, 50]. Specifically, the attacker evicts the victim instruction page's translation from the TLB before napping. As the victim page's translation can be cached either in the L1 instruction TLB and the unified L2 TLB, we construct eviction sets for both TLBs using techniques from Gras et al. [22]. Chapter 4.3b shows that combined with performance degradation, one can set a higher δ while reliably making non-zero but still small amounts of victim forward progress per preemption, as desired. The cost of this technique is a small increase in $I_{attacker}$ for evicting the TLB entries, which is small compared to the main attacker measurement procedure.

Number of preemptions

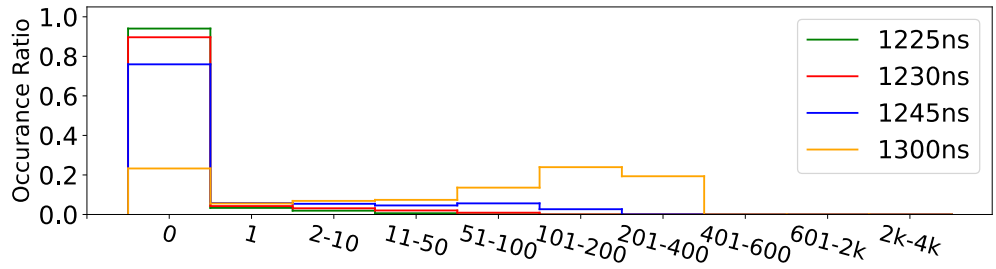
Next, we characterize the number of consecutive preemptions that the attacker can perform. Recall from Chapter 4.3, the expected number of preemptions is $\left\lceil \frac{S_{slack}-S_{preempt}}{I_{attacker}-I_{victim}} \right\rceil$. Note that $S_{slack} - S_{preempt} = 8$ ms is fixed based on the system. Thus, the attacker can increase the number of preemptions by decreasing its own measurement time $I_{attacker}$ or by increasing the victim time per preemption I_{victim} (by changing the blocking interval δ).

To count preemptions, we record the vruntime and process ID (PID) each time the kernel transfers control to userspace. Starting from when the attacker begins launching interrupts, we monitor until there are two consecutive kernel exits to the victim process without interleaving with the attacker. We vary $I_{attacker}$ by adjusting the attacker's execution length using different numbers of serialized cache-miss memory accesses, and run each experiment 50 times.

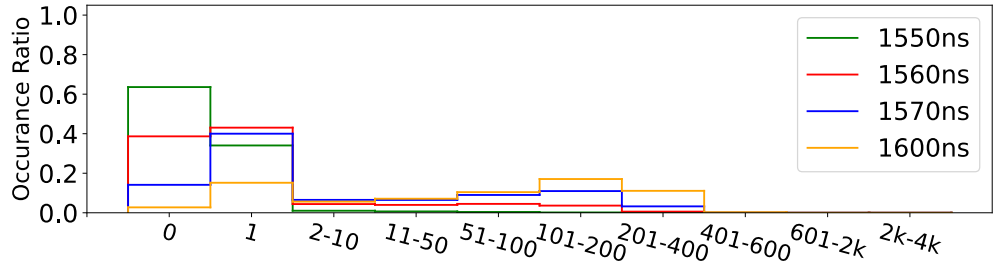
Chapter 4.4 shows the plot of the difference between $I_{attacker}$ and I_{victim} against the number of repeated interrupts achieved by the attacker. We also include a curve indicating the expected relation (given the ratio from before). The results demonstrate that the actual number of preemptions achieved closely matches up to the expected number.

Varying thread priority. Since a high priority thread increments its vruntime at a slower rate, we further examine the effect of the victim's scheduling priority. In this experiment, we vary the victim's priority by changing its *nice* value while keeping the attacker's priority at the default value of zero.[1] As shown in Chapter 4.5, decreasing the victim's `nice` value (increasing its priority) reduces the number of consecutive preemptions. Remarkably, even with the smallest `nice` value (highest victim priority), Controlled Preemption can still

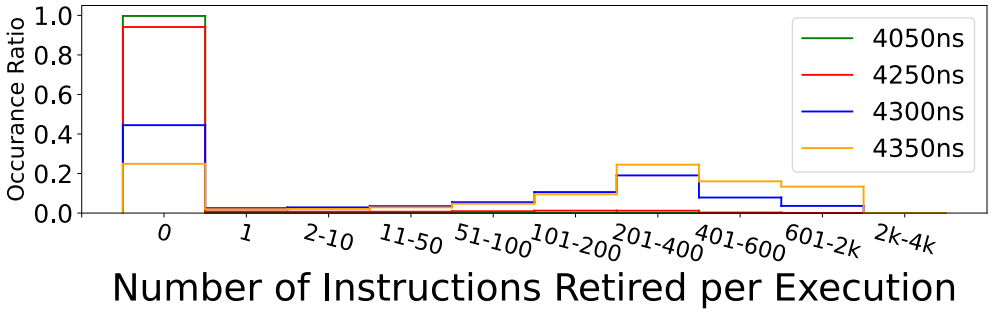
[1] We do not set the attacker's `nice` value below zero as this requires the attacker to be privileged. We do not set the attacker's `nice` value above zero as the attacker has no incentive to lower their priority.



(a) Nanosleep (Chapter 4.2)



(b) Nanosleep + Evict iTLB



(c) Timer (Chapter 4.2)

Figure 4.3: Temporal resolution of Controlled Preemption using different wake up methods and performance degradation techniques. Data is shown as a histogram, in terms of the number of victim instructions retired per preemption. Different lines correspond to different values of δ.

achieve hundreds of consecutive preemptions. This is because I_{victim} is near zero (regardless of `nice`). As a result, $I_{attacker} - I_{victim}$ is always dominated by $I_{attacker}$.

Orthogonally, we observe that temporal resolution is also largely unaffected by the victim `nice` setting.

Increasing the number of consecutive preemptions beyond the preemption budget. Controlled Preemption can borrow the idea of using multiple preemption threads from prior work [54, 7, 25] to increase the number of consecutive preemptions beyond the preemption budget. Similar to prior work, the attacker can launch n preemption threads A_1, A_2, ..., A_n. All these threads are well slept before the attack. During the attack, the

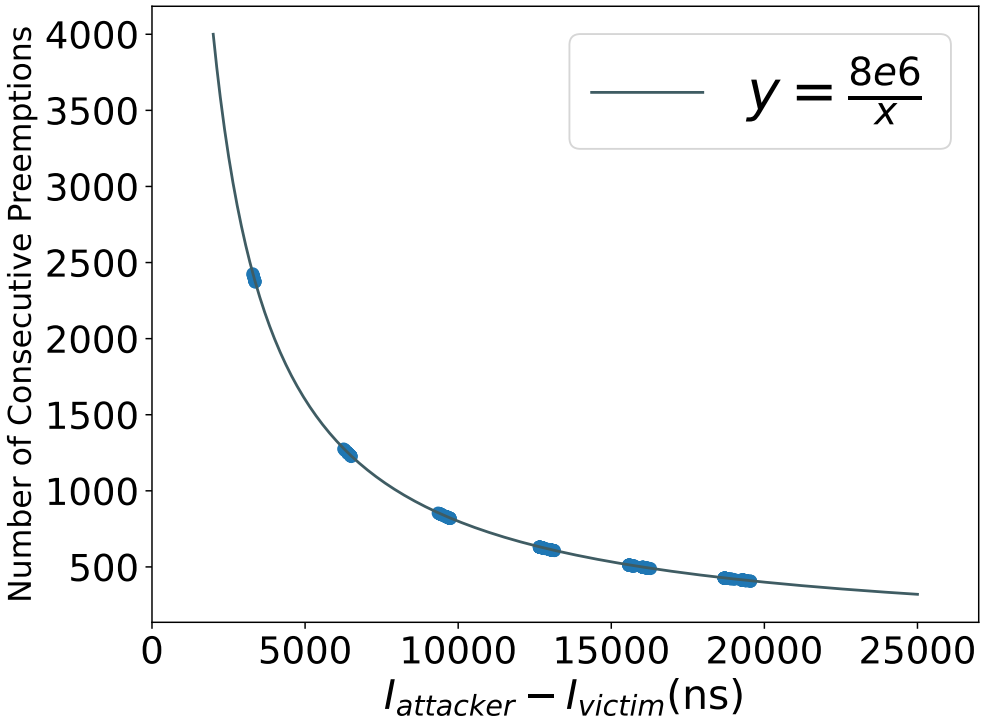


Figure 4.4: The number of repeated preemptions (y-axis) achieved by Controlled Preemption when varying $I_{attacker} - I_{victim}$ (x-axis). The plot assumes Method 1 (Chapter 4.2) although the result transfers to Method 2. Blue dots are observations (concrete settings of $I_{attacker}$ and I_{victim}) on our test machine. The curve indicates the expected number of consecutive preemptions.

attacker first uses A_1 to repeatedly preempt the victim thread. As A_1 is about to run out of preemption budget, the attacker wakes up A_2 to continue the attack, and so on. Since A_1 is sleeping while subsequent attacker threads continue the attack, it will be eligible to interrupt the victim after A_n completes its budget. With this round-robin strategy, the attacker achieves an effectively infinite preemption budget.

Measuring the impact of noise

We also test the robustness of Controlled Preemption to noise. We characterize two sources of noise. First, *scheduling noise* refers to noise caused by there being additional threads in the runqueue that are not owned by the victim or attacker. Second, *channel noise* refers to noise on the side channel (e.g., cache pollution). We conduct our experiment using Method 1 (Chapter 4.2), but the analysis is transferable to Method 2.

Scheduling noise. We study a system where the runqueue is shared between a victim thread V, an attacker thread A, and a third compute-bound noise thread N that does not

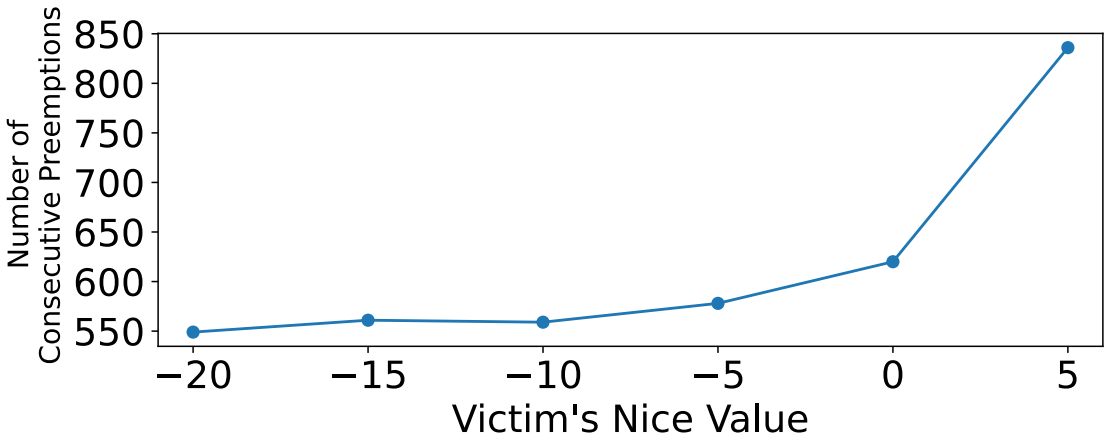


Figure 4.5: The number of repeated preemptions (y-axis) is a function of the victim's `nice` value (x-axis). We set the attacker to the default `nice` value 0. $I_{attacker} - I_{victim}$ ranges between 10 µs and 15 µs.

make system calls. See Chapter 4.6 for the results of an experiment that analyze how/when the three threads' vruntimes increase (which indicates when different threads get scheduled). Results generalize in a natural way for more noise threads.

Suppose the attacker (A) hibernates, i.e., has the smallest vruntime, at the start. We analyze two cases. First, suppose V's vruntime is initially less than N's vruntime. In this case, Controlled Preemption proceeds as usual between the A and V threads until either V's and A's vruntimes are such that Equation 2.2 returns false or until V's vruntime equals N's vruntime. Second, suppose N's vruntime is initially less than V's vruntime (not shown in Chapter 4.6). In this case, V will not be scheduled until, again, V's and N's vruntimes are equal.

Then, the remaining question is: can the attacker perform Controlled Preemption after the victim and noise threads' vruntimes become equal? This occurs at the dashed vertical line in Chapter 4.6. In this regime, we find that the attacker gets interleaved with either the victim or noise thread in an unpredictable fashion (see the zoom-in in the figure). That is, scheduling follows the pattern $((V|N)A)+$.

To continue Controlled Preemption in this regime, we adopt a well-known side-channel template attack (e.g., [23]) to construct a "victim ran last?" (or *victim presence*) oracle. Specifically, by pre-computing the victim's instruction trace at cache-line granularity, the attacker can monitor specific cache lines of interest during the measurement phase. By probing these cache lines, the attacker gathers information about the last executing thread, and only records data points if the victim thread ran last. We implement this oracle for our attack in Chapter 5.2.

We remark that the attacks we evaluate finish in several milliseconds. Further, the attacker can choose when to run the victim (per Chapter 3). Thus, it will likely be the case that any noise threads will be preexisting in the runqueue when the attack commences (as

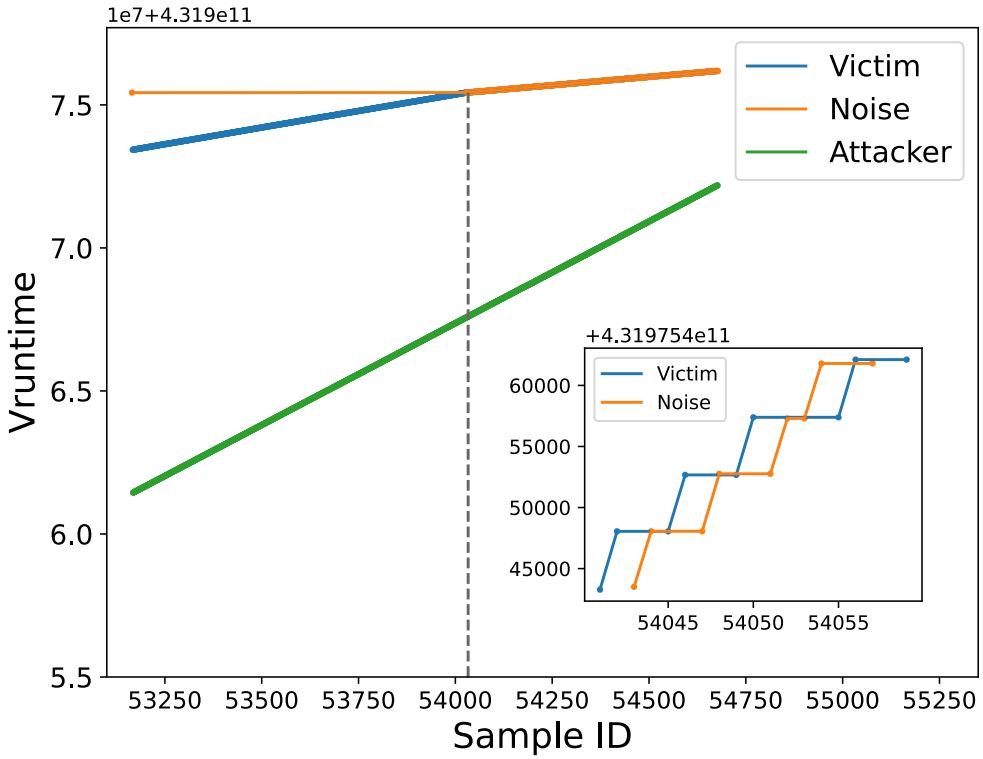


Figure 4.6: vruntime progression in a 'noisy' system with a third noise thread. The zoom-in shows the vruntimes of the victim and noise thread at the point where they converge (Sample 54033).

opposed to: are added to the runqueue when the attack is underway). In that case, noise thread vruntime will be higher than victim thread vruntime and Controlled Preemption will proceed between just A and V without wasting preemption budget on noise threads (until the victim's vruntime catches up with the noise threads).

Channel noise. We identify two types of channel noise that interfere with the attacker's side-channel measurements. The first type comes from the kernel's code/data footprint during context switches. We mitigate this noise by monitoring a sufficiently-large structure (e.g., the L2 Cache and LLC instead of the L1 cache) to not be polluted by the kernel. The second type of noise is random (non-systematic) noise coming from the other threads in other cores. To ameliorate this noise, we adopt two strategies: 1) we run the victim several times and take a majority vote; 2) alternatively, if possible, we measure private core structures like the BTB and TLB (which cannot be polluted by the activity of other cores). We demonstrate such an attack on the BTB in Chapter 5.3.

4.4 Achieving Core Colocation

Our attack requires that the attacker and victim threads reside in the same runqueue throughout the attack. Since the runqueue is a per-logical core structure, the attacker and victim threads need to colocate on the same logical core. One straightforward approach would be pinning the victim thread to an attacker-desirable logical core. However, pinning threads *owned by other users* requires supervisor privilege, which is not part of our threat model (Chapter 3).

We propose a simple strategy to achieve colocation without pinning, that works if the system has at least one idle logical core. We use this scheme in our evaluation (Chapter 5).

The idea is to exploit the load-balancing feature of the CFS [71]. The CFS periodically checks the load on each core's runqueue and migrates tasks from busier cores to idle cores.

Our approach starts with the attacker launching $N-1$ compute-intensive dummy threads, where N is the total number of logical cores in the system. The attacker then pins these $N-1$ dummy threads to $N-1$ logical cores, leaving one core C idle. Next, the attacker invokes the victim thread, which will be scheduled onto the idle core C to improve load balance. Finally, the attacker can launch the attack thread (as described previously) and pin it to C, colocating with the victim thread. Note that the victim is unlikely to migrate to another core during the attack. This is because the CFS load balancer observes that all other cores are occupied by the attacker's dummy threads and there are no idle cores to migrate the victim thread to.

The above scheme requires N threads total ($N-1$ dummy threads and 1 preemption/measurement thread) and is capable of monitoring either core-private or core-shared channels. It is also simple to implement: It does not require synchronization across attacker threads, and more generally presents the preemption/measurement thread with the illusion of living in a single-core system.

If the system is fully loaded, the above scheme will not work because there is no idle logical core. This situation is rare, and (in some cases) actively avoided. For example, Google Cloud Run tries to keep containers' CPU utilization below 60% [1]. Further, since the attacker can choose when to invoke the victim (Chapter 3), it can opt to run the attack when the system is not fully loaded.

4.5 EEVDF Scheduler

In this section, we show that Controlled Preemption is transferrable to the latest EEVDF scheduler [58, 12]. Intuitively, this is because the EEVDF scheduler also allows a well-slept thread to immediately preempt the current running thread for better system responsiveness.

To demonstrate the transferrability of our techniques, we repeat the temporal resolution experiment from Chapter 4.3 on EEVDF, using the same environment except changing the kernel version to 6.12-rc1. Chapter 4.7 reports the temporal resolution of the nanosleep method assisted by the TLB-flushing performance degradation technique. From the figure,

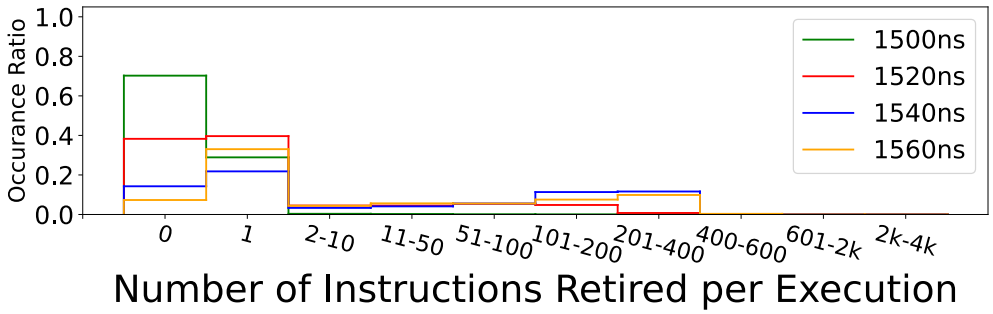


Figure 4.7: Temporal resolution of Controlled Preemption on EEVDF. The figure represents the same experiment as that shown in Chapter 4.3b, but using EEVDF.

it is clear that the victim retires only a few instructions between preemptions when using a small δ, a behavior closely resembling that of Chapter 4.3b.

We now discuss the preemption budget under the EEVDF scheduler. For simplicity, we consider the case where a single logical core is exclusively shared between an attacker thread A and a victim thread V. When selecting a thread to execute, the EEVDF first identifies *eligible* threads whose vruntime is smaller than the average vruntime of all the threads from the local runqueue. Among all the eligible threads, the EEVDF schedules the thread has the nearest *virtual deadline* to run. Since we assume only two threads in the runqueue, there is only one eligible thread; the scheduling algorithm is reduced to selecting the thread with the smallest vruntime. Therefore, we omit the details of the virtual deadline and refer interested readers to [58, 12].

Under such a scheduling policy, the attacker thread A can preempt the victim thread V as long as A's vruntime is smaller than V's. As a result, the preemption budget is simply the vruntime difference between V and A when A wakes up from hibernation. We repeat the experiment from Chapter 4.3 (that characterizes the number of preemptions) with EEVDF. When $I_{attacker} - I_{victim}$ ranges between 10,000 and 15,000 ns (i.e., the same setting in Chapter 4.5 with default `nice` value 0), we find that the attacker can repeatedly preempt the victim for a median number of 219 times from 165 repeated experiments. We leave an in-depth exploration of Controlled Preemption in the EEVDF as a future work.

Chapter 5

Proof-of-Concept Exploits

We now show how Controlled Preemption can facilitate breaking security-critical software. First, Chapter 5.1 characterizes a standard microarchitectural attack benchmark: Flush+Reload cache-timing attacks on the AES T-table algorithm. Second, Chapter 5.2 demonstrates how Controlled Preemption can also be used to conduct SGX-Step-like attacks from userspace and perform Prime+Probe cache-timing attacks. Finally, Chapter 5.3 demonstrates that Controlled Preemption can utilize other (non cache) hardware channels (specifically, the BTB).

All attacks follow the setup described in Chapter 4.3. To colocate the attacker and victim onto the same logical core, we spawn $N - 1 = 15$ dummy threads following Chapter 4.4. Then, a single attacker thread performs preemptions and side-channel measurements.

5.1 Attacking T-Table AES

To start, we demonstrate Controlled Preemption by collecting high temporal-resolution cache traces from a T-table Advanced Encryption Standard (AES) victim, which has been widely used to evaluate cache side-channel attack techniques [27, 48, 55, 25, 54, 7, 77]. We show that Controlled Preemption only requires 5 traces to conduct the first round attack, which is comparable to the state of the art [7], but only uses a single attacker thread for preemptions instead of 40 threads.

Overview

AES is a widely used symmetric block cipher with various key sizes. AES-128 uses a 16-byte secret key $\mathbf{k}$ to encrypt a 16-byte plaintext $\mathbf{p}$. We use the subscript to denote the 1-byte data slice, for example, p_0 is the first byte of $\mathbf{p}$, while $p_{14..15}$ is the last two bytes. The AES-128 encryption procedure involves 10 rounds of computation. Each round r mixes a 16-byte input $\mathbf{x}^{(r)}$ with a 16-byte round key $\mathbf{K}^{(r)}$ derived from the secret key $\mathbf{k}$ to generate the input

for the next round $\mathbf{x}^{(r+1)}$ or the final ciphertext. The first round input $\mathbf{x}^{(0)}$ is generated by $\mathbf{x}^{(0)} = \mathbf{p} \oplus \mathbf{k}$. Finally, the output of the last round $\mathbf{x}^{(10)}$ is the ciphertext.

To enhance performance, the OpenSSL T-table AES implementation simplifies the mixure computation in each round with table lookups using precomputed T-tables, denoted as $T_0, \ldots, T_3$, each containing 256 4-byte entries. Using T-tables, each round of computation can be represented as follows.

$$
\begin{aligned}
x_{0..3}^{(r+1)} &\leftarrow T_0[x_0^{(r)}] \oplus T_1[x_5^{(r)}] \oplus T_2[x_{10}^{(r)}] \oplus T_3[x_{15}^{(r)}] \oplus K_{0..3}^{(r)} \\
x_{4..7}^{(r+1)} &\leftarrow T_0[x_4^{(r)}] \oplus T_1[x_9^{(r)}] \oplus T_2[x_{14}^{(r)}] \oplus T_3[x_3^{(r)}] \oplus K_{4..7}^{(r)} \\
x_{8..11}^{(r+1)} &\leftarrow T_0[x_8^{(r)}] \oplus T_1[x_{13}^{(r)}] \oplus T_2[x_2^{(r)}] \oplus T_3[x_7^{(r)}] \oplus K_{8..11}^{(r)} \\
x_{12..15}^{(r+1)} &\leftarrow T_0[x_{12}^{(r)}] \oplus T_1[x_1^{(r)}] \oplus T_2[x_6^{(r)}] \oplus T_3[x_{11}^{(r)}] \oplus K_{12..15}^{(r)}
\end{aligned}
$$

Given the equation above, it is clear that the T-table AES implementation makes memory accesses depending on the value of $\mathbf{x}^{(r)}$. Since a single cache line fits 16 T-table entries, the attacker can recover the upper 4 bits of each byte of $\mathbf{x}^{(r)}$, known as the *upper nibble*. Now consider the first round computation ($r = 0$), where $\mathbf{x}^{(0)} = \mathbf{p} \oplus \mathbf{k}$. If the attacker can learn information about $\mathbf{x}^{(0)}$ with an attacker-controlled plaintext input $\mathbf{p}$, they can partially recover the secret AES key $\mathbf{k}$. This is known as the *first round attack*.

The main challenge of the first round attack is to distinguish T-table accesses made in the first round from other rounds. This is difficult because each round of encryption takes only about 120 cycles to complete on our system. Therefore, we employ Controlled Preemption to monitor the victim's execution at a fine temporal granularity.

Chapter 5.1 shows one measurement trace of one T-table T_0 over a single AES encryption execution, where the four lookup indexes $(x_0^{(0)}, x_4^{(0)}, x_8^{(0)}, x_{12}^{(0)})$ used in the first round have upper nibbles $(0, 4, 12, 8)$. Yellow blocks highlight the T-table access at one cache line (y-axis) captured by one attacker measurement/sample (x-axis). As seen in Chapter 5.1, the first four cache lines accessed by the victim are $(0, 4, 12, 8)$, which match the aforementioned upper nibbles of the secret indexes. Ideally, the attacker should see a single cache access (one yellow block) in each sample, with no overlap. In practice, the attacker sees smears. This is due to imperfect temporal resolution (stepping several vs. one instruction per preemption) and speculative execution (more instructions execute speculatively than are retired per preemption). We follow prior work to solve this by collecting more traces.

Evaluation

In our evaluation, the attacker program uses Flush+Reload [73] to monitor the victim's secret-dependent T-table accesses. Specifically, the attacker flushes the entire T-table before napping and times the reload latency to each entry after waking up. A short reload latency to an entry implies that the victim accessed the entry while the attacker is napping.

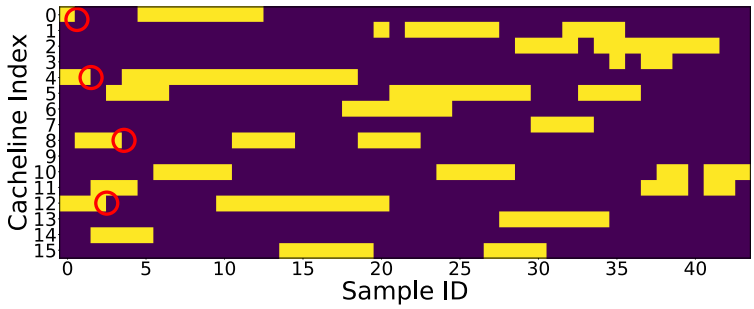


Figure 5.1: Heatmap illustrating Flush+Reload results for a single AES run. The y-axis denotes attacker measurements for each of the 16 cache lines making up the T-table (yellow indicates a hit during Reload; purple indicates a miss). Each column (x-axis) denotes a sample where the attacker preempts the victim. The first four accesses (those made in the first round) are circled in red.

Following the prior work [7], we collect multiple victim traces with randomized plaintexts to determine the upper nibble of each secret key byte and disable hardware prefetchers to reduce channel noise.[1]

We demonstrate our exploits with both the CFS and EEVDF schedulers. In each experiment, the victim is invoked 5 times, generating 5 side-channel traces that correspond to the same key. When repeating the experiment 100 times on CFS and EEVDF, where each repetition uses a different key, the attacker can infer the upper nibble of each key byte with an accuracy of 98.9% and 98.1%, respectively. Our recovery accuracy and the required number of victim executions are comparable to the state-of-the-art attack [7], which also exploits the CFS to single step the victim AES encryption process. But unlike Controlled Preemption, the prior attack requires 40 colocated attacker threads.

5.2 Attacking SGX Enclaves

Next, we demonstrate how Controlled Preemption enables high temporal resolution side-channel attacks on Intel SGX enclaves. This is akin to SGX Step [61, 60], but does not require the attacker to have supervisor privilege.

To start, we note that Controlled Preemption's temporal resolution when attacking SGX follows very similar trends as Chapter 4.3b but without explicit iTLB flushing (as SGX already performs TLB flushes on asynchronous enclave exit events [11]).

[1] Note that the need of disabling hardware prefetchers is a limitation of Flush+Reload and is not fundamental to Controlled Preemption. One can circumvent this limitation with Prime+Probe-based attacks (used in the next section). We use Flush+Reload to be apples-to-apples with prior work.

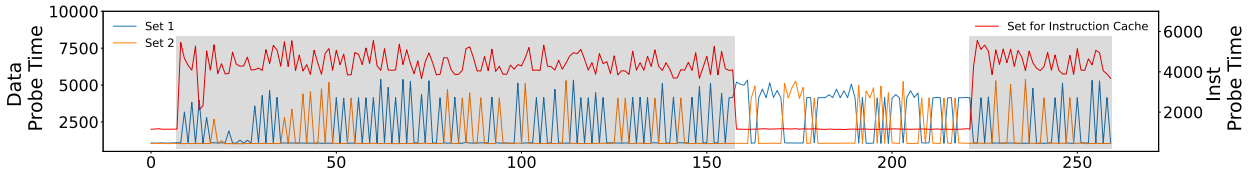


Figure 5.2: Cache probe latency trace for the base64 decoding function running inside an SGX enclave. Blue and orange traces denote probe latencies for the two LUT eviction sets. Red is the latency to probe the instruction cache eviction set. The grey (white) area is the victim performing the validity (decode) loop.

Overview

Cryptographic keys are often stored as base64-encoded PEM files for ease of transmission and are decoded into internal representations before use. OpenSSL uses a lookup table (LUT)-based approach to translate each base64 character into a 6-bit binary value. This process introduces secret-dependent access patterns, making it vulnerable to cache side-channel attacks.

OpenSSL's base64 decoding function *EVP_DecodeUpdate* groups 64 characters to parse at a time. First, it performs a *validity check* by looking up each character through the LUT. Second, it *decodes* valid characters to their binary representation. Both of the above are loops that read the LUT in a base64 character-dependent fashion. After that, the function returns the output and proceeds to decode the next chunk.

Sieck et al. [57] uses SGX Step to single step the victim enclave that runs the RSA PEM file decode procedure. They extract the victim's precise LUT access pattern via a last-level cache (LLC) Prime+Probe side channel. The LUT used for translation is 128 bytes in size and spans two consecutive cache lines. Knowing which cache set one LUT access touches shrinks the search space of one character. To complete the attack, they leverage prior RSA cryptanalysis to fully recover the RSA secret key. To reduce cache measurement noise caused by speculative or out-of-order execution, they compile the SGX program with Load Value Injection [62] mitigations by setting `MITIGATION-CVE2020-0551` to `LOAD` [31], which places load fences after every load instructions.

Challenges

Based on the prior base64 decoding attack, we replace SGX Step with Controlled Preemption. We monitor the LLC using Prime+Probe, targeting the LUT when it is accessed during the validity check. This is non-trivial compared to using SGX Step for several reasons:

Victim 'overlooping.' We are interested in monitoring one LUT access per loop iteration during the validity check. This presents a challenge. Ideally, we would like to set I_{victim} to exactly the length of one loop iteration. Setting I_{victim} smaller would result in extraneous preemptions, which consumes preemption budget. Yet, as can be seen in Chapter 4.3b, it is

difficult to guarantee a specific amount of forward progress per preemption, beyond single stepping.

To side-step this issue, we (once again) combine Controlled Preemption with performance degradation techniques. We construct an LLC eviction set that is congruent to the cache line that contains the LUT read instruction. This way, we can use a larger I_{victim} and use the instruction cache miss to stall the victim while waiting for the next preemption.

Intra-victim induced channel noise. The LUT is accessed during both the validity and decode loops. We must ensure that measurements correspond to the former not the latter. To address this, we dual-purpose the eviction set that evicts the LUT access load in the validity loop (see above) to also test whether the victim is in the validity or decode loop.

Insufficient preemption budget. Recall Chapter 4.3 which characterizes the number of repeated preemptions that can be achieved by Controlled Preemption. Since a 1024-bit RSA private key PEM file consists of nearly 900 base64 characters, the preemption budget only allows the attacker to recover about 60% of the LUT access trace.

To address this, the attacker invokes the victim twice using the same RSA key. During the first victim execution, the attacker starts preempting the victim as soon as the victim starts and captures the first half of the trace. During the second victim execution, the attacker times their hibernation and only starts preempting the victim when the victim is half way through their execution, recovering the second half of the trace. Finally, the attacker concatenates these two traces to form a complete trace.

Evaluation

We use the same victim setup (i.e., enables LVI mitigation) as the prior work [57]. The attacker constructs one LLC eviction set for the victim's load instruction (in the validity check) and two for the LUT. In the measurement phase, the attacker first probes the instruction eviction set and then, if successful, probes the LUT eviction sets.

Chapter 5.2 shows one segment of the Prime+Probe measurement trace. The grey area represents the victim's execution in the validity loop, while the white area represents the execution in the decode loop. As we can see, the Prime+Probe latency for the load instruction from the validity loop (the red line in Chapter 5.2) is high in the grey area, but low in the white area, indicating that it accurately detects when the victim is executing the validity loop.

We test on 30 randomized 1024-bit RSA private key files which contain on average 872 base64 characters. On average, our technique can recover the first 61.5% of the LUT access trace with a 99.2% accuracy from a single victim execution. With two victim executions and the trace concatenation method (above), the attacker can fully recover the LUT access trace with 98.9% accuracy.

5.3 Monitoring the BTB Side Channel

Finally, we demonstrate how Controlled Preemption can also be applied to monitor non-cache channels by exploiting the BTB. We reproduce an exploit first described in NightVision [75], which combines the fine-grain preemption primitive with a BTB side channel to leak fine-grain victim control-flow information. For the former, NightVision uses an SGX Step-like framework, and hypothesized (but did not verify) that a userspace preemption mechanism would also suffice. We verify this claim, enabling NightVision attacks from userspace.

Overview

NightVision found that the BTB can be updated by both control-transfer instructions (e.g., `jmp`) and non-control-transfer instructions (e.g., `nop`). As a result, if a non-control-transfer instruction A collides with the BTB entry of a control-transfer instruction C,[2] executing A will invalidate C's BTB entry, potentially leading to a control-flow misprediction when C is next fetched.

The above enables an attacker to infer the victim's control flow. For example, the attacker can execute a direct jump to create a BTB entry that collides with the victim's instruction of interest. Then the attacker triggers the victim execution, and later checks whether its branch mispredicts. If a misprediction occurs, the attacker learns that the victim's instruction of interest was executed. This attack works regardless of the type of the victim instruction.

```
1  btb_prime:
2      JMP T1
3      NOP (x1019)
4  T1:
5      RET
6  ...
7  ; 4 GB padding
8  ...
9  btb_probe:
10     RET
11     NOP (x1024)
12 T2:
13     NOP
```

```
1  ; flush target prefetch line
2  CLFLUSH T2
3  ; trigger prefetch
4  CALL    btb_probe
5  ; measure access time
6  RDTSCP
7  MOVQ    (T2), %rax
8  RDTSCP
9  ; allocate new BTB entry
10 ; before transition
11 ; back to victim
12 CALL    btb_prime
```

Figure 5.3: BTB prime probe code

[2] In our machine setup, the BTB entry is indexed by the lower 32 bits of the PC. Instructions with the same lower 32 bits will collide in the BTB.

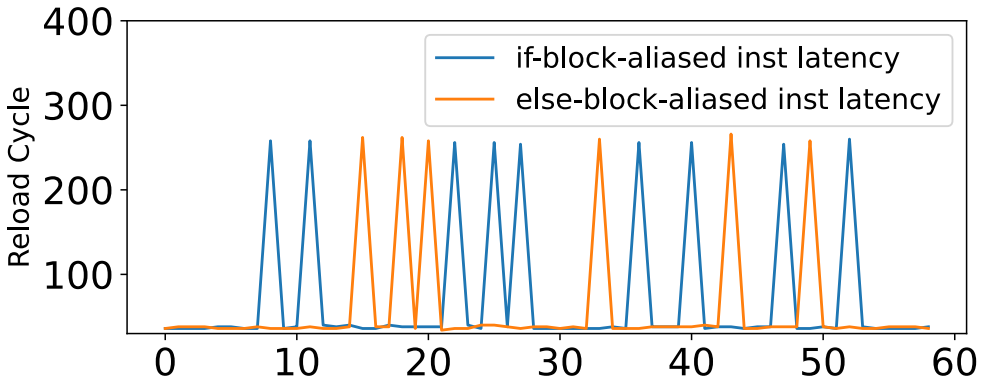


Figure 5.4: Victim control path when running `medtls_mpi_gcd` function for $a = 1001941$ and $b = 300463$. If the instruction latency is high, the victim has executed the block that invalidates the BTB entry. So the prefetcher won't prefetch the target fetching location.

Evaluation

As in NightVision, we attack the RSA key generation procedure in mbedTLS version 3.0 [42], which contains a secret-dependent branch in the Greatest Common Divisor (GCD) function `mbedtls_mpi_gcd`. Extracting the branch direction in each loop iteration is required to fully recover the RSA secret key [51]. Unlike NightVision relying on privileged performance counters to decode BTB states, we use BTB Train+Probe gadgets from prior work [79] (Chapter 5.3) to encode branch predictor state into cache state.[3]

Before the attack, the attacker identifies two instructions, each belonging to a separate direction of the secret-dependent branch. The attacker then creates two pairs of BTB Train+Probe gadgets that collide with these two victim instructions respectively. We use the same method as done in the previous PoC (Chapter 5.2) to interrupt the victim at least once per loop iteration. During each measurement phase, the attacker uses two BTB probes to infer which direction the victim most recently took.

Chapter 5.4 shows the attacker's attempt to recover the control-flow of the victim program by measuring the access latency of the prefetched locations. Suppose the victim has executed the `if` block during the attacker's sleep. In that case, the corresponding attacker-allocated BTB entry will be invalidated because it collides with the victim's non-branch instructions. Later, when the attacker executes `btb_probe` function, the CPU won't prefetch the target line because of the entry being invalidated, and the attacker will observe high latency. The attacker can use similar methods for detecting `else` branch.

We run our attack on 30 pairs of prime numbers, each of which results in 20∼30 loop iterations in the `GCD` function. We are able to extract all branch directions in a single victim

[3]Directly measuring branch prediction outcomes using the timestamp counter (`rdtsc`) is extremely noisy [38] and not well suited for our attack.

run with an average accuracy of 97.3%.

Chapter 6

Mitigations

We discuss two avenues to mitigate Controlled Preemption: (1) blocking the underlying side channel and (2) hardening the thread scheduler.

Blocking the underlying side channel.

Controlled Preemption increases the temporal resolution of existing side channels. Therefore, mitigating the encapsulated side-channel leakage can stop our attack. Per Chapter 2.2, Controlled Preemption can be used in conjunction with any stateful/persistent channel. Thus, we focus on defenses that nominally apply to any stateful/persistent channel.

In software, the most widely-deployed channel-agnostic defense is constant-time or data-oblivious programming [8, 9, 4, 35, 46]. Constant-time programming works by rewriting the program so that its observable execution trace is independent of secret data. The downside of this approach is that it may not be complete (secure). For example, recent work [10, 68, 67] has demonstrated how traditional constant-time programming guidelines are insufficient on modern microarchitectures. Thus, modest system/hardware support [76, 10, 5, 14] is likely also required.

In hardware, the peer to constant-time programming is spatial/temporal partitioning [81, 19, 40, 34]. Partitioning works by isolating microarchitectural resources, used by the victim, from the attacker. Before a partition is reused by programs belonging to a different security domain, its microarchitectural state is flushed [30, 19]. The downside to this approach is that it requires hardware support.

Finally, one can limit hardware timer resolution [28, 64], i.e., aggravate the side-channel receiver. This is effective because many attacks (regardless of channel) rely on high-resolution timers to detect microarchitectural events like cache misses and branch mispredictions. The downside to this approach is that there exists attack primitives that either do not rely on hardware timers to gather microarchitectural information [77, 79, 15] or otherwise can increase the resolution of a low-resolution timer [36, 43].

Hardening the thread scheduler.

Prior work [63] shows that setting minimum scheduling intervals for vCPUs in Xen hypervisors effectively mitigates frequent preemption-like attacks against hypervisors. Inside the Linux kernel, a similar strategy is to enable the `NO_WAKEUP_PREEMPTION` feature. When this feature is enabled, the victim thread can complete its minimum time slice before being preempted by the attacker's awakening thread. The downside is that this feature degrades system responsiveness.

Agnostic to enforcing a minimum time slice, Constable et al. [11] has proposed a software-hardware co-design approach to mitigate single-step attacks on Intel SGX. The idea is to use a special trusted prefetch handler after `ERESUME` to ensure that the victim makes significant progress before being preempted again. Unfortunately, this mechanism would require significant kernel modifications to be used by userspace programs. Moreover, this defense does not prevent the attacker from making relative coarse-grained (50-100 instructions/preemption) observations that are still sufficiently fine-grained to conduct certain attacks (e.g., the T-table AES attack from Chapter 5.1).

Chapter 7

Related Work

Improving the temporal resolution of side-channel analysis is an active area of research. Prior work splits into three categories, (1) those that induce preemption; (2) those that speed up the side-channel receiver; (3) those that slow down the victim (performance degradation). These three are not mutually exclusive. For example, Controlled Preemption can increase its preemption budget through approach (2) and Controlled Preemption already utilizes approach (3), e.g. TLB flushing (Chapter 4.3).

Preemption-based approaches.

There is a rich line of work on preempting victim threads in an Enclave (or otherwise privileged) threat model [26, 45, 38, 60, 61, 78]. Here, the attacker has supervisor privilege and uses supervisor capabilities to facilitate the attack.

Our goal is to enable similar temporal resolution side channel analyses in an unprivileged setting. The closest work is a series of papers [54, 7, 6, 25], starting with Cache Games [25], that also exploit the CFS to preempt the victim in a fine-grain manner. Controlled Preemption differs from these works in that they require multiple (10s to 100s) of threads to preempt the victim. This is because prior work overlooks Equation 2.2 in the CFS. They implicitly require attacker threads to always take the left-hand argument to the max function in Equation 2.1. This results in the following attack workflow. When an attacker thread performs a preemption, it is forced to "cool down" (sleep) for a significant period of time (S_{bnd}) before it can perform another preemption. If the attacker wishes to preempt the victim X times in fast succession, it requires X attacker threads—after the first preempts and is "cooling down", the second will preempt and so on. By contrast, our work performs a careful analysis of the CFS and enables repeated preemption with only a single attacker thread.

Speeding up the receiver or slowing down the victim.

Beyond preempting the victim, an attacker can optimize its receiver logic for a specific side channel (e.g., [52, 33, 27]). For example, Prime+Scope [52] reduces the accesses needed

to perform a Probe from the cache associativity to a single cache line; Spec-o-Scope [27] combines ideas from Prime+Scope and Katzman et al. [33] to achieve a 5 cycle resolution over the cache-timing channel. Preemption-based techniques, such as Controlled Preemption, differ from these works in that they are channel agnostic: they can improve the temporal resolution of any stateful channel.

Alternatively, the attacker can slow down the victim to achieve finer-grain measurements, e.g., by flushing the active instruction region [37, 3, 50], bus locking through cross cache line atomic memory accesses [70], and port contention [2]. Performance degradation, by itself, cannot achieve as fine-grain resolution as preemption-based methods, but can be combined with preemption for various purposes (as we do in Chapter 4.3 and Chapter 5.2).

Chapter 8

Conclusion

This thesis presented Controlled Preemption, the first userspace framework that enables a single colocated attacker thread to nearly single step a victim thread. We comprehensively characterized the Controlled Preemption primitive in the context of the widely-deployed CFS and EEVDF schedulers. We demonstrated three proof-of-concept attacks that show how Controlled Preemption can improve existing attacks on multiple victim programs, inside and outside of SGX, and across multiple side channel types.

Bibliography

[1] *About instance autoscaling in Cloud Run services*. `https://cloud.google.com/run/docs/about-instance-autoscaling`. 2025.

[2] Alejandro Cabrera Aldaya et al. "Port Contention for Fun and Profit". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2019.

[3] Thomas Allan et al. "Amplifying Side Channels through Performance Degradation". In: *Annual Conference on Computer Security Applications (ACSAC)*. 2016.

[4] Jose Bacelar Almeida et al. "Verifying Constant-Time Implementations". In: *USENIX Security Symposium (USENIX Security)*. 2016.

[5] *Arm Armv8-A Architecture Registers*. `https://developer.arm.com/documentation/ddi0595/2021-12`. 2021.

[6] C Ashokkumar et al. *" S-Box" Implementation of AES is NOT Side-channel Resistant*. 2018. URL: `https://ia.cr/2018/1002`.

[7] C. Ashokkumar, Ravi Prakash Giri, and Bernard Menezes. "Highly Efficient Algorithms for AES Key Retrieval in Cache Access Attacks". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2016.

[8] Gilles Barthe, Benjamin Grégoire, and Vincent Laporte. "Secure Compilation of Side-Channel Countermeasures: The Case of Cryptographic "Constant-Time"". In: *Computer Security Foundations Symposium (CSF)*. 2018.

[9] Gilles Barthe et al. "System-level Non-interference for Constant-time Cryptography". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014.

[10] Boru Chen et al. "GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers". In: *USENIX Security Symposium (USENIX Security)*. 2024.

[11] Scott Constable et al. "AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[12] Jonathan Corbet. *An EEVDF CPU Scheduler for Linux*. `https://lwn.net/Articles/925371/`. 2023.

[13] Miles Dai et al. "Don't Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects". In: *USENIX Security Symposium (USENIX Security)*. 2022.

[14] *Data Operand Independent Timing Instruction Set Architecture (ISA) Guidance*. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/best-practices/data-operand-independent-timing-isa-guidance.html`. 2023.

[15] Craig Disselkoen et al. "Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX". In: *USENIX Security Symposium (USENIX Security)*. 2017.

[16] *eBPF - Introduction, Tutorials & Community Resources*. `https://ebpf.io/`. 2024.

[17] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. "Jump over ASLR: Attacking Branch Predictors to Bypass ASLR". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016.

[18] Dmitry Evtyushkin et al. "BranchScope: A New Side-Channel Attack on Directional Branch Predictor". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2018.

[19] Qian Ge et al. "Time Protection: The Missing OS Abstraction". In: *Proceedings of the European Conference on Computer Systems (Eurosys)*. 2019.

[20] Daniel Genkin, Adi Shamir, and Eran Tromer. "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis". In: *International Cryptology Conference (CRYPTO)*. 2013.

[21] Daniel Genkin et al. "Stealing Keys from PCs Using a Radio: Cheap Electromagnetic Attacks on Windowed Exponentiation". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2015.

[22] Ben Gras et al. "Translation Leak-Aside Buffer: Defeating Cache Side-Channel Protections with TLB Attacks". In: *USENIX Security Symposium (USENIX Security)*. 2018.

[23] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. "Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches". In: *USENIX Security Symposium (USENIX Security)*. 2015.

[24] Daniel Gruss et al. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016.

[25] David Gullasch, Endre Bangerter, and Stephan Krenn. "Cache Games – Bringing Access-Based Cache Attacks on AES to Practice". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2011.

[26] Marcus Hähnel, Weidong Cui, and Marcus Peinado. "High-Resolution Side Channels for Untrusted Operating Systems". In: *USENIX Annual Technical Conference (USENIX ATC)*. 2017.

[27] Gal Horowitz, Eyal Ronen, and Yuval Yarom. "Spec-o-Scope: Cache Probing at Cache Speed". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2024.

[28] Wei-Ming Hu. "Reducing Timing Channels with Fuzzy Time". In: *Journal of computer security* (1992).

[29] Mehmet Sinan İnci et al. "Cache Attacks Enable Bulk Key Recovery on the Cloud". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2016.

[30] *Indirect Branch Predictor Barrier*. `https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/indirect-branch-predictor-barrier.html`. 2018.

[31] *Intel® Processors Load Value Injection Advisory*. `https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00334.html`. 2021.

[32] Nikita Ishkov. "A Complete Guide to Linux Process Scheduling". MA thesis. University of Tampere, 2015.

[33] Daniel Katzman et al. "The Gates of Time: Improving Cache Attacks with Transient Execution". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[34] Vladimir Kiriansky et al. "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018.

[35] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *International Cryptology Conference (CRYPTO)*. 1996.

[36] David Kohlbrenner and Hovav Shacham. "Trusted Browsers for Uncertain Times". In: *USENIX Security Symposium (USENIX Security)*. 2016.

[37] Andrew Kwong et al. "Checking Passwords on Leaky Computers: A Side Channel Analysis of Chrome's Password Leak Detect Protocol". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[38] Sangho Lee et al. "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing". In: *USENIX Security Symposium (USENIX Security)*. 2017.

[39] Moritz Lipp et al. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium (USENIX Security)*. 2016.

[40] Fangfei Liu et al. "CATalyst: Defeating last-level cache side channel attacks in cloud computing". In: *International Symposium on High-Performance Computer Architecture (HPCA)*. 2016.

[41] Fangfei Liu et al. "Last-Level Cache Side-Channel Attacks are Practical". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2015.

[42] *Mbed-TLS: An Open Source, Portable, Easy to Use, Readable and Flexible SSL Library*. `https://github.com/Mbed-TLS/mbedtls`. 2022.

[43] Ross Mcilroy et al. *Spectre is here to stay: An analysis of side-channels and speculative execution*. 2019. URL: `https://arxiv.org/abs/1902.05178`.

[44] Robert Merget et al. "Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E)". In: *USENIX Security Symposium (USENIX Security)*. 2021.

[45] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. "CacheZoom: How SGX Amplifies the Power of Cache Attacks". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2017.

[46] David Molnar et al. "The Program Counter Security Model: Automatic Detection and Removal of Control-Flow Side Channel Attacks". In: *USENIX Security Symposium (USENIX Security)*. 2005.

[47] Ingo Molnar. *sched-design-CFS.txt*. `https://www.kernel.org/doc/Documentation/scheduler/sched-design-CFS.txt`. 2007.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *The Cryptographers' Track at the RSA Conference (CT-RSA)*. 2006.

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

[50] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. ""Make Sure DSA Signing Exponentiations Really are Constant-Time"". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016.

[51] Ivan Puddu et al. "Frontal Attack: Leaking Control-Flow in SGX via the CPU Frontend". In: *USENIX Security Symposium (USENIX Security)*. 2021.

[52] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021.

[53] Thomas Ristenpart et al. "Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2009.

[54] Bholanath Roy et al. "Design and Implementation of an Espionage Network for Cache-based Side Channel Attacks on AES". In: *International Joint Conference on e-Business and Telecommunications (ICETE)*. 2015.

[55] Till Schlüter et al. "FetchBench: Systematic Identification and Characterization of Proprietary Prefetchers". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2023.

[56] Michael Schwarz, Moritz Lipp, and Daniel Gruss. "JavaScript Zero: Real JavaScript and Zero Side-Channel Attacks". In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[57] Florian Sieck et al. "Util::Lookup: Exploiting Key Decoding in Cryptographic Libraries". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021.

[58] I. Stoica and H. Abdel-Wahab. *Earliest Eligible Virtual Deadline First: A Flexible and Accurate Mechanism for Proportional Share Resource Allocation*. 1995. URL: `https://people.eecs.berkeley.edu/~istoica/papers/eevdf-tr-95.pdf`.

[59] Andrei Tatar et al. "TLB;DR: Enhancing TLB-based Attacks with TLB Desynchronized Reverse Engineering". In: *USENIX Security Symposium (USENIX Security)*. 2022.

[60] Jo Van Bulck and Frank Piessens. "SGX-Step: An Open-Source Framework for Precise Dissection and Practical Exploitation of Intel SGX Enclaves". In: *Annual Conference on Computer Security Applications (ACSAC)*. 2023.

[61] Jo Van Bulck, Frank Piessens, and Raoul Strackx. "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control". In: *Workshop on System Software for Trusted Execution (SysTEX)*. 2017.

[62] Jo Van Bulck et al. "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2020.

[63] Venkatanathan Varadarajan, Thomas Ristenpart, and Michael Swift. "Scheduler-based Defenses against Cross-VM Side-channels". In: *USENIX Security Symposium (USENIX Security)*. 2014.

[64] Bhanu C. Vattikonda, Sambit Das, and Hovav Shacham. "Eliminating fine grained timers in Xen". In: *Cloud Computing Security Workshop*. CCSW. 2011.

[65] Junpeng Wan et al. "MeshUp: Stateless Cache Side-channel Attack on CPU Mesh". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2022.

[66] Alan Wang et al. " Peek-a-Walk: Leaking Secrets via Page Walk Side Channels". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2025.

[67] Yingchen Wang et al. "DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2023.

[68] Yingchen Wang et al. "Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86". In: *USENIX Security Symposium (USENIX Security)*. 2022.

[69] Zixuan Wang et al. "NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[70] Zhenyu Wu, Zhang Xu, and Haining Wang. "Whispers in the hyper-space: high-speed covert channel attacks in the cloud". In: *USENIX Security Symposium (USENIX Security)*. 2012.

[71] Wei Xu. *Deep into Linux and Beyond*. `https://wxdublin.gitbooks.io/deep-into-linux-and-beyond/content/cfs_internals.html`. 2024.

[72] Mengjia Yan et al. "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World". In: *IEEE Symposium on Security and Privacy (IEEE S&P)*. 2019.

[73] Yuval Yarom and Katrina Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack". In: *USENIX Security Symposium (USENIX Security)*. 2014.

[74] Yuval Yarom, Daniel Genkin, and Nadia Heninger. "CacheBleed: A Timing Attack on OpenSSL Constant Time RSA". In: *Journal of Cryptographic Engineering* (2016).

[75] Jiyong Yu, Trent Jaeger, and Christopher Wardlaw Fletcher. "All Your PC Are Belong to Us: Exploiting Non-control-Transfer Instruction BTB Updates for Dynamic PC Extraction". In: *International Symposium on Computer Architecture (ISCA)*. 2023.

[76] Jiyong Yu et al. "Data Oblivious ISA Extensions for Side Channel-Resistant and High Performance Computing". In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[77] Jiyong Yu et al. "Synchronization Storage Channels (S2C): Timer-less Cache Side-Channel Attacks on the Apple M1 via Hardware Synchronization Instructions". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[78] Yinqian Zhang et al. "Cross-VM Side Channels and Their Use to Extract Private Keys". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012.

[79] Zhiyuan Zhang et al. "BunnyHop: Exploiting the Instruction Prefetcher". In: *USENIX Security Symposium (USENIX Security)*. 2023.

[80] Zirui Neil Zhao et al. "Last-Level Cache Side-Channel Attacks Are Feasible in the Modern Public Cloud". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2024.

[81] Zirui Neil Zhao et al. "Untangle: A Principled Framework to Design Low-leakage, High-performance Dynamic Partitioning Schemes". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2023.

[82] Yongye Zhu et al. "Controlled Preemption: Amplifying Side-Channel Attacks from Userspace". In: *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 2025.