KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Cache Side-Channel Attacks on Existing and Emerging Computing Platforms

Antoon Purnal

Supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

June 2023

Cache Side-Channel Attacks on Existing and Emerging Computing Platforms

Antoon PURNAL

Examination committee:

Prof. dr. ir. Paul Sas
Chairman

Prof. dr. ir. Ingrid Verbauwhede
Supervisor

Prof. dr. ir. Frank Piessens

Dr. Benedikt Gierlichs

Assoc. Prof. Daniel Gruss
Graz University of Technology, Austria

Prof. Thomas Eisenbarth
University of Lübeck, Germany

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

June 2023

Preface

As I write the final words to this chapter of my life, I want to express my appreciation to some of the people who were along for the ride.

I want to start by thanking my supervisor, Ingrid Verbauwhede, for bringing me on board to pursue a PhD degree in her group. Ingrid, thank you for the opportunities and your sustained support. I will not forget your confidence and trust in me, granting me the academic freedom necessary to carve my own path.

I want to express my gratitude to Frank Piessens, Benedikt Gierlichs, Daniel Gruss, and Thomas Eisenbarth. I am humbled to be evaluated by such an eminent committee. I thank Paul Sas for chairing the final phase of the PhD.

During my time at COSIC, I saw many people come and go. I thank my current and former colleagues for contributing to a pleasant and enriching atmosphere. The group is too large and my cache too incoherent - I dare not enumerate you. I consider myself lucky to have had the pleasure of sharing coffee, cakes, beers, or other poisons with many of you. I hope our paths cross again. Throughout the whole PhD, I had Furkan as my office mate. I learned a lot from him and will long remember the *Daily Trouble* calls that powered our first paper together. In my first year in the office, I also enjoyed the pleasant company of Pieter.

COSIC is a nice and stimulating place to work, as evidenced by how swiftly I returned to the office when possible. I want to thank Lennert and Arthur for doing the same during what feels like the dark ages of the PhD. The hardware and cosinet groups deserve credit for being excellent gatherings for streamlining work in progress. For the cosinet meetings, in particular, it is cute how we keep pretending that a one-hour slot is sufficient for our discussions. I apologize to my fellow humans in the hardware group for the mind-numbing timer experiment.

I thank the many students over the years. I hope I impacted some of you (positively) as thesis supervisor, ombudsperson, or teaching assistant.

Several funding agencies made my research possible. I want to highlight the Research Foundation - Flanders (FWO) for granting me a PhD fellowship.

I want to recognize the delightful non-technical staff at COSIC. Péla, Elsy, Wim, Dana, Saartje, you are not on any paper, but COSIC would not be the same without you. I also want to acknowledge the gentlemen of the ESAT system group. Microarchitectural research without root privileges is tricky. Still, thank you for facilitating it in the best way possible.

Beyond COSIC, I had the privilege and pleasure to meet several bright and funny researchers. I want to thank the fine people from Daniel's group at TU Graz. I especially want to mention Lukas, the wingman on my first big paper, Andreas and Jonas. Thanks for selectively enriching my German vocabulary. Despite the short walk from COSIC to DistriNet, I only properly got to know several members in a van in California. I don't egret [sic] that trip at all. Furthermore, I thank Thomas Unterluggauer for the interesting discussions and brainstorms before and during my time at Intel.

Beyond research, I want to recognize some special people. From the early days, I could count on my buddies from Rotselaar: Jan-Willem & Elise, Jeffrey, Wannes, Vinnie, Daan & Tinneke. It fills me with joy to see that we are still going strong. After arriving in Leuven, through sheer coincidence, I met a fantastic group of people: Laura & Nick, Lucky & Annabel, Tuur & Iris, Ellen, Sofie. Spending more time in your presence is *more better*. I often get to enjoy the company of Bernd & An Sofie, Jonathan & Lotte, Simon & Nina, Adriaan & Mira, Evelien, Luuk, Ward. I look forward to the day we reach a PhD-majority. Special thanks go out to Anthony for being in the correct timezone to attend my defense, and to Robbert for being such a remarkable and inspiring *copain*.

I want to thank my family, especially my parents and brother Sam, for everything they have done for me. Mom, dad, this thesis would not have been written without your support. Thank you for encouraging me to pursue an engineering degree, and for dissuading me from trying to become a legendary guitar player. (Almost a decade later, I wonder what the alternative timeline looks like.)

The concluding words of praise are reserved for my favorite person. My dear Femke, I may not express it enough, but I feel privileged to share a life with someone as unselfish, kind, and supportive as you. No matter which chapters lie ahead, I look forward to writing them together. Thanks for being you.

I deeply appreciate the people who are close to me. I hope they know it. Though they may not read this thesis from cover to cover, it is dedicated to them.

Toon Purnal
Leuven, June 2023

Abstract

The act of executing a program on a computing platform produces inadvertent side effects that depend on the data being processed. Microarchitectural side-channel attacks leverage the side effects stemming from interference in shared hardware components to extract potentially sensitive data. Arguably the most important class of microarchitectural side-channel attacks are *cache attacks*, which target the shared cache hierarchy. This thesis advances the understanding of the capabilities of cache attacks in conventional and novel execution contexts. In addition, it contributes to the defensive landscape through a critical security assessment of promising, low-overhead mitigations.

The first line of research explored in this dissertation concerns advanced cache attack techniques. Our first contribution is the development of Prime+Scope, a low-requirement and cross-core cache contention attack that delivers the highest temporal precision to date. Our second contribution is a thorough exploration of the cache attack surface in emerging heterogeneous computing platforms, where an attacker may have access to one or more hardware accelerators. We show how a malicious FPGA accelerator may not just accelerate legitimate computations but also *attacks*, while consuming a negligible amount of resources.

The second line of research targeted by this dissertation advances the state of the art of cache attack mitigations. To this end, it critically examines two influential, transparent and low-overhead countermeasure classes. First, we perform a systematic analysis of cache randomization, which is a hardware countermeasure that injects entropy into the address-to-index mapping of the cache. Second, we study the effectiveness of restricting the availability of high-precision sources of time. Our findings indicate that minuscule timing differences can be converted and amplified to sidestep this restriction, ultimately enabling even a human observer to distinguish between a single cache hit or cache miss.

Beknopte samenvatting

Het uitvoeren van een computerprogramma veroorzaakt onbedoelde neveneffecten die afhankelijk zijn van de verwerkte gegevens. Microarchitecturale nevenkanaalaanvallen maken gebruik van de neveneffecten die ontstaan door interferentie in gedeelde hardwarecomponenten. De meest invloedrijke microarchitecturale nevenkanaalaanvallen zijn *cacheaanvallen*. Dergelijke aanvallen richten zich op de hierarchie van caches in moderne processoren. Deze thesis onderzoekt de eigenschappen en mogelijkheden van cacheaanvallen in conventionele en nieuwe uitvoeringsomgevingen. Bovendien draagt het bij aan het beschermen van toekomstige systemen door veelbelovende tegenmaatregelen aan een kritische analyse te onderwerpen.

Het eerste deel van dit proefschrift beschrijft nieuwe geavanceerde cacheaanvaltechnieken. Onze eerste bijdrage is de ontwikkeling van Prime+Scope, de kernoverschrijdende cacheaanval met 's werelds hoogste nauwkeurigheid in het tijdsdomein. Onze tweede bijdrage is een grondige exploratie van cachegebaseerde aanvalsmogelijkheden in opkomende heterogene computerplatformen. We demonstreren hoe een kwaadwillende FPGA-accelerator niet alleen legitieme berekeningen kan versnellen, maar ook kan assisteren bij cacheaanvallen.

Het tweede deel van deze dissertatie bestudeert defensieve maatregelen tegen cacheaanvallen. Hiertoe analyseren we twee invloedrijke en beloftevolle verdedigingstechnieken. Ten eerste voeren we een systematische analyse uit van cache-randomisatie, een hardwarematige maatregel die entropie toevoegt aan adres-naar-index-toewijzing in de cache. Ten tweede bestuderen we de effectiviteit van het uitschakelen van nauwkeurige bronnen van tijd. Onze bevindingen geven aan dat deze beperking kan worden omzeild door het omzetten en versterken van minuscule tijdsverschillen. Hierdoor kan zelfs een menselijke waarnemer het onderscheid maken tussen een enkele cache-hit of cache-miss.

List of Abbreviations

AES Advanced Encryption Standard.

AMD Advanced Micro Devices.

API Application Programming Interface.

BP Branch Prediction.

BPU Branch Prediction Unit.

CAT Cache Allocation Technology.

CD Coherence Directory.

CPU Central Processing Unit.

DCA Direct Cache Access.

DDIO Data Direct IO.

DES Data Encryption Standard.

DMA Direct Memory Access.

DRAM Dynamic Random Access Memory.

dTLB Instruction Translation Lookaside Buffer.

FPGA Field Programmable Gate Array.

GPU Graphics Processing Unit.

HPC Hardware Performance Counter.

IO Input-Output.

ISA Instruction Set Architecture.

iTLB Instruction Translation Lookaside Buffer.

KASLR Kernel Address Space Layout Randomization.

L1 Level 1 Cache.

L1d Level 1 Data Cache.

L1i Level 1 Instruction Cache.

L2 Level 2 Cache.

LLC Last-Level Cache.

MESI Modified Exclusive Shared Invalid.

MLC Mid-Level Cache.

NIC Network Interface Card.

NRE Non-Recurring Engineering.

OS Operating System.

PCIe Peripheral Component Interconnect Express.

PRNG Pseudorandom Number Generator.

RDTSC Read Time-Stamp Counter.

RISC Reduced Instruction Set Computer.

ROB Reorder Buffer.

RX Receiver.

SGX Software Guard Extensions.

SLA Service-Level Agreement.

SMT Simultaneous Multi-Threading.

SNR Signal-to-Noise Ratio.

TEE Trusted Execution Environment.

TLB Translation Lookaside Buffer.

TSX Transactional Synchronization Extensions.

TX Transmitter.

Contents

List of Figures

List of Tables

Part I

Cache Side-Channel Attacks and Defenses

Haste makes waste.

English Proverb

Chapter 1

Introduction

Our society increasingly relies on digital infrastructure. As we become surrounded by computing devices, we entrust them with our medical and financial data and depend on them to operate critical infrastructure. The heart of a computing device is its *processor*, an incredibly sophisticated piece of (mostly) silicon on which computer programs are executed. Modern processors are the result of decades of engineering and feature a wide array of performance optimizations to satisfy an increasing hunger for inexpensive computing power.

Today, computing devices often execute software originating from multiple, mutually distrusting sources. The emergence of cloud computing even embeds this trend as a core part of its value proposition, i.e., the amortization of the fixed acquisition and operational costs of computer infrastructure by sharing it among different customers. However, the viability of sharing a computing system among multiple parties relies on a proper separation between them.

More Than Meets The Eye. To a keen observer, there is often a greater amount of information available than that which is directly apparent on the surface. People may unintentionally reveal their preferences through their body language, a vault may reveal its access code through the clicks it produces, and the accumulation of mail in the mailbox may give away that its owners are on vacation. In computing systems, similar sources of incidental information leakage exist, and the technical term describing them is *side channels*.

The central side channel to this publication-based thesis is *time*. In the context of modern computing systems, the passage of time is software-observable and correlates with the activity of other processes running on a shared hardware

platform. An essential and ubiquitous source of timing variations is a cache, a buffer that holds particular data close to the processor under the expectation of it being used at a later point in time. A cache inadvertently produces a timing side channel on which elements are in the cache and which are not; accessing some data is fast, and accessing other data is slow. A *cache attack*, then, cleverly weaponizes the side channel to distinguish the former from the latter to spy on other processes across software-defined isolation boundaries.

Side channels arising from hardware processor optimizations are, in some sense, by design. Caches are instrumental for performance by reducing round trips to comparatively slow main memory. After half a century of performance engineering and intense competition between vendors, it should come as no surprise that many leaky optimizations exist. The vast attack surface of modern processors, together with the challenging nature of protecting performance-critical components, results in a flourishing area of research.

1.1 Main Contributions

Recognizing the inherent security-performance trade-off imposed by processor caches, this dissertation seeks to understand better the low-level capabilities of cache attacks on existing and emerging computing platforms. It also aims to contribute to the future defensive landscape by studying the effectiveness of promising, low-overhead, and state-of-the-art cache attack mitigations.

In alignment with these objectives, this dissertation unpacks and answers the following main research questions (RQs). For each research question, I selected a representative main-author publication that aims to answer it.

RQ1. What are the true limits to the precision of cache attacks?

Most cache attacks infer the memory access patterns of other programs by measuring the access latency of specific cache lines. Such measurements are believed to be intrinsically *destructive*, i.e., the side effects of each measurement need to be reverted before the next one can occur. Moreover, the most broadly-applicable attacks, i.e., those based on contention for shared cache resources, need to perform *several memory accesses* (e.g., 12 or 16) as part of every measurement. Both of these measurement properties adversely affect the time precision of the attack, and both are considered to be unavoidable.

In this thesis, I question the fundamental nature of these limitations, and ask: *What is the optimal time precision for cache attacks? Can an attacker perform non-destructive cache contention measurements with just one memory access?*

These questions form the subject of my CCS 2021 paper, which is included as Chapter 6 in this thesis.

Prime+Scope: Overcoming The Observer Effect for High-Precision Cache Contention Attacks
Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede
ACM Conference on Computer and Communications Security, 2021

RQ2. How are emerging computing platforms affected?

The increasing demand for processing power, together with the diminishing economic returns on technology scaling, fuels the trend of *hardware specialization*. By complementing general-purpose processing units with domain-specific accelerators, computing systems become heterogeneous. Heavily optimized for specific tasks, accelerators provide much better performance per unit of cost (or energy), making them attractive for cloud computing contexts. Particularly promising are Field Programmable Gate Arrays (FPGAs) which, due to their reconfigurability, recover much of the flexibility lost due to hardware acceleration.

As accelerators interface with the processor caches, I ask: *Are heterogeneous computing systems susceptible to attacks by malicious accelerators? Do combined software-hardware adversaries have more control over the shared cache?*

I explore this forward-looking research problem in my USENIX Security 2022 article, which is included as Chapter 7 in this thesis.

Double Trouble: Combined Heterogeneous Attacks on Non-inclusive Cache Hierarchies
Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede
USENIX Security Symposium, 2022

RQ3. How secure are state-of-the-art countermeasures?

There exists a complicated trade-off between the performance-enhancing capabilities of hardware components and their impact on security. Caches, especially, are so instrumental to the performance of modern processors that

modifications to them, however small, face strict practical constraints if they are ever to be implemented. Very recently, *cache randomization* was proposed as a scalable, transparent, and low-overhead countermeasure, which attracted considerable academic attention.

In light of these developments, I ask: *How secure are randomization-based secure caches? Are they susceptible to probabilistic attacks? How important is the cryptographic strength of the randomization function?*

My IEEE S&P 2021 publication, included as Chapter 8, examines this problem.

Systematic Analysis of Randomization-based Protected Cache Architectures
Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede
IEEE Symposium on Security and Privacy, 2021

RQ4. Do timing attacks require accurate timers?

Microarchitectural timing attacks infer secrets from observing whether specific actions are *fast* or *slow*. Often, the underlying timing differences are in the order of 10 ns-100 ns. Modern processors and environments expose high-precision timers that enable attackers to observe such minuscule timing differences. It is, therefore, natural to wonder whether simply disabling or restricting these timing sources would defeat microarchitectural side-channel attacks. Enticed by this potential, all major browsers have already adopted similar restrictions.

However, I ask: *Are microarchitectural side-channel attacks thwarted in the absence of high-precision timers? Can arbitrary timing differences be amplified? Is it still possible to leak memory access patterns across processor cores?*

I answer these questions as part of my AsiaCCS 2023 article, which is incorporated as Chapter 9.

ShowTime: Amplifying Arbitrary CPU Timing Side Channels
Antoon Purnal, Márton Bognár, Frank Piessens, and Ingrid Verbauwhede
ACM Asia Conference on Computer and Communications Security, 2023

1.2 Other Contributions

In addition to the publications incorporated in this thesis, I contributed to several other articles during my PhD research. A small selection of these papers is highlighted below. A complete list of publications can be found in Part II.

Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks. This work proposes a combined randomization-based and isolation-based secure cache that defends against the attacks I outline in Chapter 8, while retaining attractive performance. The design leverages a novel cryptographic construction to enable security domains to compete for randomized and partially-overlapping cache resources. Victim cache lines automatically become inaccessible to an attacker after a few observations. Afterward, an attacker can no longer evict them, no matter which and how many accesses they perform.

This article appeared at IEEE S&P 2023 in collaboration with Lukas Giner, Stefan Steinegger, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss.

Forkcipher: A New Primitive for Authenticated Encryption of Very Short Messages. Short messages are a previously-overlooked optimization target for authenticated encryption. This work proposes a novel cryptographic primitive called a *forkcipher* which, roughly speaking, computes two independent tweakable permutations of an input block at an amortized computational cost. The article presents a forkcipher instantiation called ForkSkinny, three provably secure modes of operation, and an evaluation for hardware implementations.

This article appeared at ASIACRYPT 2019 in collaboration with Elena Andreeva, Virginie Lallemand, Reza Reyhanitabar, Arnab Roy, and Damian Vizár.

1.3 Organisation of this Dissertation

There are two parts to this publication-based doctoral dissertation. The first part is concerned with providing the necessary preliminaries, outlining the state of the art, and positioning the contribution of this thesis in the context of microarchitectural attacks and defenses. Chapter 2 exposes the basic organization of modern processors, covers the cache hierarchy, and introduces basic notions on side-channel attacks. Chapter 3 covers state-of-the-art microarchitectural side-channel attacks. It establishes that the cache hierarchy is one of the most critical hardware components to examine due to its

combined impact on performance and security. Chapter 4 covers state-of-the-art defenses against microarchitectural side-channel attacks, again focusing on the cache hierarchy. Chapter 5 contains concluding remarks and provides an outlook on the future of microarchitectural side-channel research.

The second part of this doctoral thesis first provides a list of all publications, which is a superset of those presented in Sections 1.1 and 1.2. Chapters 6 to 9 contain the selection of peer-reviewed articles that constitute this dissertation. These articles are included without modifications from their peer-reviewed versions, except for editorial changes to accommodate a consistent single-column layout throughout this manuscript.

Chapter 2

Background

This chapter provides the necessary background for reasoning about microarchitectural side-channel attacks. Section 2.1 covers the organization of modern processors and the hardware components that underlie their phenomenal performance. One such component is the cache hierarchy, a fundamental part of the memory subsystem, which is the subject of Section 2.2. Section 2.3 introduces basic notions of microarchitectural side-channel attacks. As our exposition assumes some familiarity with computer architectures, novice readers are recommended to refer to introductory material when necessary [90, 148].

2.1 CPU Organization

2.1.1 Layers of Abstraction

Managing Complexity. The beating heart of present-day computing systems is their processor, or Central Processing Unit (CPU). Built from billions of tiny transistors, they are an incredibly sophisticated feat of engineering, powering much of the modern world economy. To make them perform a valuable function, a developer writes a piece of *software*. This piece of software describes an algorithm that needs to be executed on the processor (*hardware*). Software is written as code in a programming language which, through several compilation steps, is translated to assembly code and later to machine code. The machine code executes on the processor, which has a *microarchitecture* comprising several functional units and their interconnection. These functional units are built

from standard cells, which are composed of transistors. During a complicated manufacturing process, these transistors are constructed from physical materials.

To manage all this complexity, computing systems abide by well-defined abstraction layers that encapsulate the complexity of the layers below. The interaction between abstraction layers is determined by their *interface*, which essentially serves as a contract; it specifies the functional behavior that the layer above can expect from the layer below.

Instruction Set Architecture. The interface this thesis is concerned with is the one between *software* and *hardware*. This abstraction layer is referred to as the *Instruction Set Architecture (ISA)*. It specifies the contract between, on the one hand, the final piece of machine-readable code that specifies the instructions to be performed by the machine and, on the other hand, the microarchitecture of the processor that implements the execution of these instructions. The ISA facilitates the compilation of arbitrary software to a sequence of instructions that can be executed on any processor that implements this ISA. Examples of ISAs are x86, Arm-v8 and RISC-V. This thesis focuses on x86, which is used by Intel and AMD processors.

CPU Microarchitecture. The microarchitecture of a processor is its implementation of the ISA. Much of the performance advances throughout the twentieth century were achieved by increasing the operating frequency of the processor. However, the hardware manufacturer can include other optimizations that, transparently to the end user, enhance the average-case performance of many applications. The basic principles underlying many optimizations at the microarchitectural level are parallelism, i.e., the hardware's ability to perform actions simultaneously; locality, i.e., the tendency of algorithms to repeatedly touch the same data; and prediction, i.e., the ability to predict the future behavior of a program through its actions in the past [148]. Provided that the processor behaves in a functionally correct manner, as specified by the ISA, the manufacturer is free to introduce optimizations in the microarchitecture.

2.1.2 Hardware Organization

Modern CPU Pipeline. Figure 2.1 is a simplified depiction of a modern CPU. Each CPU core has an execution pipeline with a frontend and backend. The CPU frontend is responsible for fetching the instruction stream and decoding it into a stream of *micro-ops* that are executed in the CPU backend. The frontend features several components to speed up this process, such as caches to accelerate

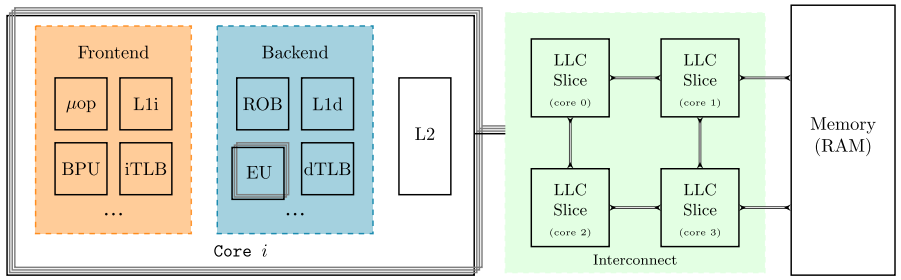


Figure 2.1: High-level processor organization.

address translation (iTLB in Figure 2.1) and retrieve instructions faster (L1i), a branch prediction unit (BPU) to predict the outcome of unresolved branches, and other components like the micro-op cache (μop) and loop stream decoder (not depicted). The output of the CPU frontend is a stream of micro-ops. In the backend, these micro-ops get executed on the *execution units* (EU), which are batched in *execution ports*. Data loads occur through the L1 data cache (L1d), and the dTLB accelerates the address translation.

Instruction-Level Parallelism. To exploit the available parallelism present in the micro-op stream, modern processors are typically *superscalar*, i.e., multiple micro-ops may be executed in parallel within one cycle. In addition, they typically employ *out-of-order execution*, i.e., micro-ops may execute ahead of older micro-ops so long as their dependencies are already resolved. Modern processors also implement *speculative execution*, i.e., execution may be steered along one of the outcomes of a branch before said branch is resolved.

The Reorder Buffer (ROB) is responsible for reordering the micro-ops and issuing them when they are ready to be executed. These micro-ops are then dispatched across the execution ports and executed in the execution units. However, micro-ops may only be retired, i.e., have their result committed to the architectural state, in program order. The ROB flushes the pipeline in case of executed but not retired instructions, thereby preventing these incorrectly executed instructions from affecting the software-visible state of the program.

Thread-Level and Task-Level Parallelism. Modern processors have more than one core, as depicted in Figure 2.1. This allows them to speed up the execution of different threads within a process, or different processes altogether, so long as these are independent enough to be executed in parallel. In modern computing contexts, different programs or even different tenants may

be executing simultaneously on different cores. Processors with *hyperthreading*, or *simultaneous multithreading (SMT)*, are able to process multiple independent instruction streams on a single (*physical*) core, giving software the impression of multiple (*logical*) cores at a lower cost. Logical cores share many of the resources in the processor frontend and backend. Some parts of the processor are shared by all the cores, such as the last-level cache (LLC), its interconnect, and main memory.

Modern high-performance computing servers may have multiple CPU sockets within the same system, each comprising a number of cores. Sockets may have accelerators attached to them, which are heavily optimized for specific tasks, e.g., networking or graphics processing.

2.1.3 Software Organization

System Software. In most cases, application software does not execute directly on the processor. Instead, it is mediated by a privileged piece of software called the *operating system (OS)*. The OS instantiates the execution of other programs as *processes*, and is responsible for scheduling their execution and allocating system resources for them. As an additional layer of abstraction, a *hypervisor* provides the ability to have different *virtual machines (VMs)* with their own OS and application software on the same physical device. The OS (or hypervisor) needs to ensure that different processes (or VMs) are *isolated* from each other. Today, this is implemented using a feature called *virtual memory*.

Virtual Memory. Instead of directly interacting with actual (physical) memory, programs experience the abstraction of their own exclusive virtual address space. The operating system or hypervisor maps virtual addresses to physical addresses, and stores this information in *page tables*. The translation from virtual to physical addresses occurs in parallel to the retrieval of the data from the L1d cache. Recently-used translations are buffered in cache-like structures called Translation Lookaside Buffers (TLBs). The unit of granularity is the *page*, which is typically 4 KiB in size but may differ across platforms. Modern processors typically support larger pages in addition to the default smallest page (e.g., 2 MiB and 1 GiB, which are larger to reduce the number of translations that are needed). Part of the virtual address, the page offset bits, is preserved across address translation.

2.2 Cache Hierarchy

2.2.1 Working Principle

Locality of Memory Accesses. The cache is one of the most important hardware components to improve average software performance [148]. A cache is a piece of storage that buffers data (including both instructions and actual data) that is expected to be used soon. It is *small*, relative to the total physical memory available, and *fast*, relative to the speed of contemporary main memory technology. The motivating observation underlying the working principle of cache is that a program's memory accesses are typically not uniformly randomly distributed across *time*, nor *space*. Instead, the vast majority of programs exhibit strong temporal locality, i.e., memory locations touched at a given point in time are likely to be accessed again at a later time, and spatial locality, i.e., accessing a memory location increases the posterior probability of accessing neighboring memory locations.

Cache Lines. When a program performs an access to memory, it may be transparently served from the cache instead, if it is available (i.e., a cache *hit*). If the data corresponding to the requested memory location is not cached, i.e., a cache *miss* occurs, it is served from memory and typically installed in the cache in anticipation of future accesses. Instead of only caching the word (e.g., 64 bits) corresponding to a memory address, the cache transfers data to and from memory in larger *cache lines*. A typical cache line size is 64 bytes.

Cache Associativity. Caches may differ in their *associativity*, i.e., the number of locations in which a specific memory address can be cached. If a cache line may be installed at any location in the cache, that cache is said to be *fully associative*. However, given that caches may contain several thousands of cache lines, a fully associative organization consumes a prohibitive amount of power due to the simultaneous lookup of *all* the cache lines. To avoid this, cache lines are often indexed by their memory address. In a *direct-mapped* cache, a memory address can only be installed in one location. Most caches are *set-associative*, as in Figure 2.2, which means that addresses are mapped into *sets* and may be installed in any of the cache lines (*ways*) of the set. The part of the address that serves to identify a cache line while it is cached is called its *tag*. Two addresses that are mapped to the same set are said to be *congruent*.

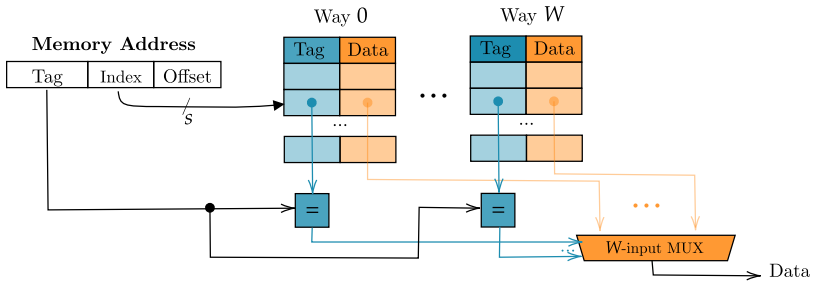


Figure 2.2: Set-associative cache organization (with 2^s sets and W ways).

2.2.2 Multi-level Cache Hierarchy

Cache Hierarchy. It turns out that a single large cache is not the most efficient use of resources. Instead, modern processors feature a multi-level cache hierarchy. On Intel processors, there are three levels: L1, L2 (or MLC, for mid-level cache) and L3 (or LLC, for last-level cache). The lower the level, the smaller and faster it is; the higher the level, the larger and slower it is. When a read request is issued by the core, first the L1 cache is queried. If this is a cache miss, the next-level cache is queried, and so on. If none of the cache levels hold a copy of the requested data, the request goes to the main memory directly.

On modern Intel processors, each CPU core has its own private L1 and L2 caches, whereas the last-level cache (LLC) is shared among all the cores. The L1 cache is separated into a separate instruction (L1i) and data (L1d) cache, whereas the L2 and LLC do not distinguish between instructions and data. The exact access latency depends on the processor, but it typically takes a handful of cycles to fetch data from L1, around ten cycles from L2, 40 to 50 cycles for the LLC, and a few hundred cycles to fetch data from main memory. Representative sizes, as of today, are 32 KiB for both L1i and L1d caches, 256 KiB-1 MiB for L2 caches, and several MiB for the LLC.

Addressing Modes. The cache set indexing function may be based on the memory location's virtual or physical address. The L1 cache is typically virtually indexed to allow its lookup to occur in parallel to the address translation, which produces the physical tag. This limits the number of L1 sets to the number of cache lines present in the smallest page size supported by the system. On x86 CPUs this is 4 KiB, so 64 sets; on Apple CPUs this is 16 KiB, so 256 sets. Additionally, it makes it susceptible to *aliasing*, where lines with identical page offset bits are conservatively considered to refer to the same physical address

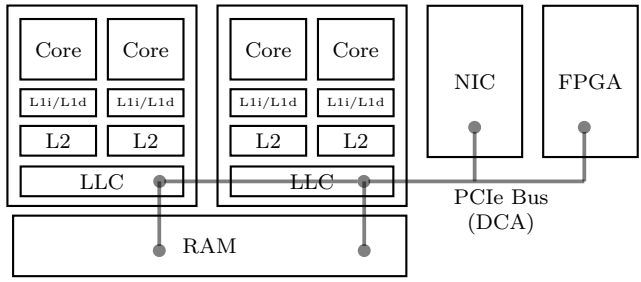


Figure 2.3: Multi-level cache hierarchy.

to avoid read-after-write and write-after-read hazards. The L2 and LLC are only queried upon an L1 cache miss, at which time the address translation has hopefully already been completed. Therefore, they are typically physically indexed and physically tagged.

Distributed LLC. Contemporary last-level caches (LLCs) are considerable in size. Therefore, they are implemented as *slices*, distributed across the cores. The reason for this, as opposed to a monolithic design, is to avoid contention hotspots and distribute the load across slices that may be queried in parallel. The assignment from (physical) memory addresses to slices is determined by a slice indexing function (referred to as *complex addressing*), which is static for a given processor. Traffic between the LLC slices happens over the LLC *interconnect*. Common implementations of the interconnect include a one-dimensional ring-bus topology and a two-dimensional mesh topology, where the latter scales better with increasing core counts. Note that the distributed nature of the LLC implies that its access time depends on the LLC slice to which the requested cache line is mapped.

Inclusion Invariants. There is typically a well-defined relationship between lines in different cache levels. A cache is said to be *inclusive* with respect to another cache if lines present in the latter cache must also be present in the former. A cache is *exclusive* with respect to another cache if any given line may only exist in one of both cache levels at any given time. A cache that satisfies none of these properties is said to be non-inclusive.

An inclusive LLC simplifies and speeds up lookups for data in other cores, as lines cannot be present in the core-private caches without a copy in the LLC. However, a clear drawback is the duplication of data in the L1 and L2 caches due to inclusion, limiting the size of the L2 cache in practice. An inclusive LLC

design also implies that when a line is evicted from an inclusive last-level cache, it is also automatically evicted from the lower-level cache. This mechanism, referred to as *back-invalidation*, will be relevant in later chapters.

Traditionally, Intel LLCs used to be inclusive. Since 2017, however, a non-inclusive LLC organization is becoming an increasingly prevalent design choice among Intel processors. Nonetheless, its role for cross-core lookups is typically taken up by another inclusive component [252], the *coherence directory (CD)*, or *core valid bits*.

Hardware Prefetching. A hardware prefetcher checks for memory accesses that match its activating pattern and, to prevent future cache misses, prefetches the memory locations that complete the extrapolated pattern. Several types of prefetchers exist, such as *next-line* (or *stream*) prefetchers, which prefetch one (or more) cache lines ahead of the latest access, or *stride* prefetchers, which continue accesses to memory locations separated by a constant stride. On Intel processors, the private L1 and L2 caches are equipped with hardware prefetchers, which can be configured by privileged software [97].

Direct Cache Access. In the context of high-performance computing, servers are increasingly equipped with domain-specific accelerators, like Network Interface Cards (NICs), Graphics Processing Units (GPUs), and Field Programmable Gate Arrays (FPGAs). As depicted in Figure 2.3, these accelerators are attached to the CPU over Peripheral Component Interconnect Express (PCIe). For performance reasons, they are tightly integrated with the CPU and access memory directly through the LLC interconnect, i.e., direct cache access (DCA) [92] as opposed to direct memory access (DMA). Intel's implementation of DCA is called Data Direct IO (DDIO) [99].

2.2.3 Cache Metadata

Replacement Metadata. When a new cache line is installed in a set-associative cache, another one in the set needs to be removed (or *evicted*). The optimal line to evict is determined by the future behavior of the program, which is unpredictable. Modern caches implement *replacement policies* at the cache-set level, to determine the sequence of evictions. Replacement policies are selected based on how they affect cache hit rates, and on the size and complexity of the circuit that implements them. We refer to the line to be evicted next as the *eviction candidate*. In some circumstances, specific cache ways may be empty, in which case they are typically preferentially filled over the eviction candidate.

Coherence Metadata. In a multicore processor, data may have local copies in several caches. A *cache coherence protocol* serves to keep this data coherent and typically associates cache lines with a *coherence state*. Most coherence protocols are variants of MESI [147], which describe modified, exclusive, shared and invalid states for each cache line. They also describe a state machine to transition between states depending on the reads and writes to a cache line. Cache coherence, being mainly relevant for cross-core lookups, is managed in the LLC and, if the latter is non-inclusive, in the LLC and CD together.

2.2.4 Interacting with the Cache

Most state changes in the cache occur implicitly through reading and writing to memory. However, modern ISAs endow the programmer with instructions that provide an explicit interface with the state of the cache, diluting the strict separation between architecture and microarchitecture.

Cache Invalidation. The x86 ISA contains cache line invalidation instructions, such as the unprivileged `clflush(opt)` instructions. These instructions explicitly remove lines from the cache hierarchy, even though there is no cache conflict that causes their eviction.

Software Prefetch. The compiler or developer may know that a specific cache line may be needed in the near future. An ISA may include *software prefetch* instructions to communicate this knowledge to the hardware. On x86, there are several prefetch instructions that install a memory location in the cache but differ in their implementation. For instance, `prefetchnta` indicates that the cache line may be preferentially evicted, and `prefetchw` changes the coherence state of a cache line ahead of time to speed up future writes to it. Note that software prefetch instructions are part of the compiled code and differ from the hardware prefetcher, which is part of the microarchitecture.

Other Interactions on Intel Processors. Intel TSX provides hardware support for encapsulating instructions in a transaction, satisfying the property that the transaction either executes completely or is aborted. Conditions for aborting a transaction include cache eviction events for memory addresses captured by the transaction. Intel CAT [98] is a quality-of-service feature to configure which LLC ways are available to which CPU cores. Intel DDIO [99] governs which LLC ways are available for use by PCIe devices (e.g., NICs, FPGAs and GPUs).

Hardware Performance Counters (HPCs) [97] expose processor-internal events, such as cache hits and misses, to inform performance engineering workflows.

2.3 Microarchitectural Timing Side Channels

Side Channel Attacks. A *side channel* (or *incidental channel*) refers to an unintended information flow that, through inference, reveals another piece of information. The name implies that it should be contrasted with a *main channel* which may be any intentional functional behavior. Some examples of intended functional behaviors include the computation of a cryptographic algorithm on a device, typing a password on a keyboard, and converting virtual addresses to physical addresses. Corresponding side channels may include the instantaneous power consumption of the device, the sound of the keys being pressed, and the time it takes to perform the translation. Over the years, several side channels in computing systems have been discovered, such as time [110], power consumption [112], electromagnetic emanations [59], acoustics [12], temperature [94], and many more.

Side channels are mainly useful when they allow one to infer a piece of information that is unavailable for direct observation. We refer to a *side-channel attack* when an *attacker* uses leakage through a side channel to infer secret information by an uncooperating *victim*.

Microarchitectural Side Channels. Microarchitectural side channels are sources of leakage at the hardware-software interface of a computing system. Indeed, a *functionally correct* program may exhibit a secret-dependent usage of microarchitectural resources, e.g., accesses to various caches or utilization of specific execution units. Through this usage, the program may expose its secret information to an attacker. The predominant mechanism by which an attacker can draw inferences on the usage of microarchitectural resources is *time*, which constitutes the focus of this thesis.

In general, we say that programs encode their control flow and data access patterns in the microarchitectural *context*. The microarchitectural context encompasses the microarchitectural *state*, e.g., which lines are cached and which are not, and the microarchitectural *utilization* at any point in time, e.g., the usage of a functional unit like a multiplier. Leakage of the former type is said to be *stateful*, whereas the latter type is referred to as *stateless* leakage.

Covert Channels. We distinguish between side-channel and covert-channel attacks. When two parties communicate covertly, despite the system's security policy dictating that no communication channel should exist between them, they are said to establish a covert channel. In the absence of a *main channel*, these parties establish a communication interface through the intentional use of side channels. Because of the implied intentionality of a covert channel, the communicating parties are referred to as *sender* and *receiver* instead of attacker and victim. In our setting, a sender process transmits information by deliberately encoding it into its usage of microarchitectural resources. The receiver then decodes this information through timing measurements.

Interacting with Time. Modern computing systems expose the passage of time to software. On x86, high-precision readouts of time are available through Read Time-Stamp Counter (`RDTSC`) instructions. Timing an action can be implemented by surrounding this action with calls to the timing source. However, care should be taken that the action is being timed, no more and no less. This is typically ensured by instrumenting serialization instructions [81] (e.g., `LFENCE/MFENCE` on x86) or inserting data dependencies [143]. In the absence of a dedicated timing interface, programs may implement one themselves using a software counter [177]. In the browser, timers are available with `performance.now()`.

2.4 Conclusion

This chapter introduced the necessary background on the relevant parts of modern computer organization. The cache hierarchy was identified as a performance-critical but complex hardware component, which encodes a large state that is shared across processor cores. We then introduced basic notions of microarchitectural timing channels.

Chapter 3

Cache Side-Channel Attacks

In this chapter, we first provide a broad overview of microarchitectural timing attacks (Section 3.1). Next, we turn to cache-based timing attacks, arguably the most important subclass, and perform an in-depth exploration (Section 3.2). Microarchitectural (cache-based) side-channel attacks are the subject of several surveys [27, 61, 129, 192, 232] which, unfortunately, rapidly become out of date as new attacks get discovered. For this reason, we do not attempt to provide a complete survey and instead focus our efforts on recent trends and state-of-the-art techniques.

3.1 Microarchitectural Timing Attacks

3.1.1 Attack Targets

Microarchitectural side-channel attacks leak the memory access patterns of other processes. Therefore, if a program's access patterns (i.e., its control flow or data access patterns) depend on secret information, it is vulnerable to these attacks. Several applications have been demonstrated to carry such dependencies.

Cryptography. Cryptographic implementations have been, and still are, a typical attack target because they (repeatedly) operate on small and well-defined pieces of secret data, i.e., cryptographic *keys*. Moreover, naive implementations of cryptographic algorithms typically directly encode their keys in the microarchitectural context through secret-dependent control flow or

data accesses. As a result, minor side-channel leakage proves to be sufficient to extract these cryptographic keys [106, 110, 146], compromising the system-level security properties that are based on their confidentiality.

Few branches in cryptography have been spared from microarchitectural side-channel attacks. Attacks have been applied to symmetric-key cryptography, most notable on block cipher implementations (e.g., DES and AES) [21, 81, 84, 143, 203, 204] but stream cipher implementations can also be vulnerable [64]. They have also been applied to implementations of public-key cryptosystems [20, 33, 128, 150, 151, 241, 250], where even a tiny amount of leakage may still be exploitable [9, 22, 63]. Other attack examples include the revival of classic padding oracle attacks [103, 169, 170] and attacks on a vulnerable implementation of a pseudorandom number generator (PRNG) [39].

Other Targets. Besides cryptographic algorithms, other programs may also operate on sensitive information. In this context, microarchitectural side-channel attacks have enabled researchers to break kernel address space layout randomization (KASLR) [75, 93, 120]. These attacks can also be used to spy on user input, such as keystrokes or touchscreen swipes [121, 142, 165].

Using microarchitectural side channels, an attacker can determine which website a user may be visiting, i.e., *website fingerprinting* [82, 142, 186, 196]; which other applications are running on the system, i.e., *application fingerprinting* [83]; or which specific device is executing the attacker program i.e., *device fingerprinting* [168, 200]. These side channels have also been used to establish high-bandwidth covert channels in the cloud [136, 191] and infer when different virtual machines are co-resident in the cloud infrastructure [165, 249]. Recently, they were employed to leak machine learning hyperparameters [91, 234], to reconstruct private databases [182], to track autonomous vehicles [130], to implement stealthy computations [52], and to perform targeted online deanonymization [245].

3.1.2 Threat Models

In a security evaluation, an *attacker model* (or *threat model*) captures the capabilities that realistic adversaries are assumed to have. For microarchitectural timing attacks, the attacker model is determined by two main classification axes. The first is the *proximity* of the attacker's execution w.r.t. the victim. The second is the *type* of execution the attacker is assumed to have on the computing platform on which the victim code is running.

Execution Proximity. In general, the closer the attacker's execution is to the victim, the more resources are shared and the more interference (i.e., leakage) exists between attacker and victim. In the most extreme case, the attacker and victim run on the same physical core, either in a time-sliced or simultaneous fashion. In this case, they share resources like the L1 data [143], L1 instruction [2] and unified L2 [233] caches, and the core's execution ports [6, 23]. More commonly, the attacker and victim may execute on the same machine but on different CPU cores [101, 128, 241]. If they execute on different CPU sockets of the same server, there are still some resources in which they can interfere [100, 152, 155]. An attacker may also have execution capabilities outside of, but attached to, the computing device on which the victim is running, e.g., an FPGA connected over PCIe [227]. Finally, an attacker may not have code execution on the target system at all. In this case, she does not directly share any hardware resources with the victim and has to interact with the microarchitecture of the machine through an interface, e.g., by sending network packets [116, 176].

Execution Type. Regarding the execution capabilities of the adversary, the most common assumption in the microarchitectural attack literature is that of unprivileged (native) execution. This assumption applies to co-located tenants in a cloud context or a malicious (or compromised) program on a personal computing device. Another common execution context is a malicious web source (i.e., person-in-the-browser) with sandboxed execution of, e.g., JavaScript or WebAssembly code. These languages do not expose all instructions in the ISA to the attacker. This restricts them from using, for instance, `clflush`, fencing, or software prefetch instructions. The strongest execution type is native execution with elevated privileges, which is relevant in the threat model of trusted execution environments (TEEs), such as Intel SGX [28, 139, 208, 209]. Finally, peripheral devices vary widely in the expressiveness of their execution. On an FPGA, for instance, an adversary can implement arbitrary logic; on a NIC, an attacker does not have, in principle, direct code execution.

3.1.3 Shared Microarchitectural Resources

Microarchitectural side-channel attacks observe how other processes modulate shared microarchitectural resources. We now classify some of these attacks by the shared microarchitectural (sub-)component they target.

Cache. As outlined in Chapter 2, caches maintain several kinds of state, both in data and metadata. Almost all of these may generate cross-process leakage. If a cache line belongs to shared memory between attacker and victim, its *caching*

status may reveal accesses by the victim to the attacker. Memory access patterns may also leak through *cache contention*, i.e., attacker and victim compete for resources in the same cache set, slice or full cache. Programs may also encode information in *cache replacement* or *cache coherence metadata*. Section 3.2 describes these leakage mechanisms in more detail.

Non-cache. Processes running simultaneously on the same physical core may compete for resources in the processor *frontend* [161, 195] and *backend*, such as execution units [3, 176] or their ports [6, 23, 70, 166]. These processes also share branch prediction units [53, 55], TLBs [71], and they may introduce false-dependencies on each other [140]. Processes running on any core within the CPU socket may compete for bandwidth on the on-chip interconnect between slices [45, 144, 220], or leak through hardware prefetchers [184]. Processes may even interfere across connected sockets, e.g., in the DRAM controller [152] or on the PCIe bus [193].

3.1.4 Comparing Microarchitectural Leakage Sources

Executing software on a modern processor exposes it to leakage through many different timing side channels. These channels are not equal in their utility to an attacker. We consider the following main classification metrics.

Cross-Core. A leakage source is more valuable to an attacker if it is shared across processor cores. If it is not, an attacker is required to co-locate on the same core as the victim in order to exploit it. This requirement may be challenging to satisfy in practice, e.g., in public clouds [132]. Several OSes and hypervisors disable multi-threading by default [68], allow to disable it [89], or schedule processes at the level of physical instead of logical CPU cores [42].

Visibility. We refer to a hardware component as providing *full visibility* to an attacker if all the monitored victim activity produces interference in this component. A component only provides *partial visibility* if some of the victim activity is filtered by other components (e.g., a memory access that hits in the L1 cache does not affect the state of the higher-level caches). Full visibility, therefore, requires either the absence of such filtering components or a mechanism to circumvent them. Inclusive LLCs provide full visibility, as lines evicted from the LLC are automatically invalidated in the lower-level caches, forcing future accesses to them to affect the LLC state.

Not all leakage sources provide full visibility. For instance, memory accesses only generate cache interconnect traffic if the data is not cached in L1 or L2 [144], and only generate DRAM traffic if the data being queried does not have a copy in any of the cache levels [152].

Spatial Granularity. The *spatial granularity* of a microarchitectural component refers to the quantity of information it reveals to an attacker. We distinguish between memory-indexed components, i.e., those that take a memory address as input, and operation-indexed components, i.e., those for which the usage depends on the micro-op (or instruction) that triggers it.

For memory-indexed components, the spatial granularity is higher the more it restricts the space of memory addresses that cause the same interference. Some leakage mechanisms only provide coarse-grained information, i.e., they do not restrict the address space at all [134, 186], restrict it to a specific memory page [71], or restrict it to the LLC slice that served the request [144]. Other mechanisms reveal interference in the set to which the address maps, e.g., in the cache [143], in DRAM [152], or in the branch predictor [53]. Some components reveal the specific cache line that contains the memory address [81], or even address information within a cache line [243].

For operation-indexed components, the spatial granularity is higher the more it restricts the space of possible instructions that cause the interference, e.g., the execution port [6, 23] or the execution unit itself [3, 176]. There is no total ordering between memory- and operation-based granularities. The relative utility to an attacker depends on the program under attack.

Channel Capacity. The channel capacity is a measure of the magnitude of the leakage per unit of time and incorporates the channel bandwidth and error rate (cf. Shannon's information theory [183]). While not a perfect quantitative comparison for a practical attack, it is a good qualitative measure of the power of a leakage source. For this reason, characterizing the covert channel capacity is common practice in microarchitectural research as part of the discovery of a new leakage source (e.g., [77, 128, 144, 152, 161, 231]).

Comparison. Table 3.1 summarizes the most significant sources of microarchitectural timing leakage at the time of writing, and how they fare on the metrics outlined before. We focus on Intel processors, omitting some recent works that target hardware components only present in products by other vendors [60, 122, 214]. For each leakage source, we include the fastest covert-channel implementation in the literature, if one exists. The covert channel

Table 3.1: Microarchitectural side-channel attacks. Those that are based on core-shared cache resources are highlighted.

Microarchitectural Component	Cross Core	Full Visibility	No Shared Memory	Spatial Granularity	Channel Capacity	
Cache Bank: Contention [243]	✗	✓	✓	<line	/	/
4K-Aliasing [140, 191, 253]	✗	✓	✓	<line	8.9 Mbps	[253]
Translation Lookaside Buffer [71]	✗	✓	✓	page	11 Mbps	[197]
Branch Prediction (BP) [53, 55]	✗	✓	✓	BP set	0.8 Mbps	[54]
Execution Ports [6, 23]	✗	✓	✓	port	1.2 Mbps	[195]
CPU Frontend [161, 195]	✗	✓	✓	various	1.6 Mbps	[195]
Variable-Latency Instr. [176]	✗	✗	✓	instr.	/	/
Hardware Prefetcher [184]	✓	✗[a]	✗[b]	line	0.3 Mbps	[184]
DRAM Row Buffer [152]	✓	✗	✓	row	2 Mbps	[152]
Slice Interconnect [45, 144, 220]	✓	✗	✓	slice	4.1 Mbps	[144]
PCIe Contention [193]	✓	✓	✓	none	/	/
Cache Status [81, 241]	✓	✓	✗	line	13.9 Mbps	[171]
Cache Coherence [44, 202, 239]	✓	✓	✗	line	6.6 Mbps	[87]
Cache Replacement [31, 231]	✓	✗[a]	✓	set	4 Mbps[c]	[231]
Cache Contention [128, 143, 150]	✓	✓	✓	set	3.5 Mpbs	[156]

[a] Full visibility requires complementary mechanism (e.g., cache status or contention)
[b] Existing works use the cache status leakage source for extraction
[c] Leakage rate reported for L1 cache (not LLC) and with shared memory

bandwidth is expected to increase with the degree of sharing between attacker and victim (e.g., same-core execution or shared memory).

3.1.5 Focus of this Dissertation

As revealed by Table 3.1, the cache hierarchy is an attractive and versatile target for side-channel attacks. In this thesis, our main focus lies on attacks and defenses for the last-level cache (LLC) and the coherence directory (CD).

Rationale. First, the LLC (or CD) is shared across cores, obviating the need for an attacker to co-locate with the victim on the same CPU core. Second, the LLC (or CD) has a large number of sets, resulting in a high spatial resolution. Third, attacks on the LLC provide full visibility, which is a property that other fine-grained cross-core leakage sources do not provide unless they *additionally* manipulate the cache hierarchy. Fourth, the cache hierarchy is critical for performance, so modifications to the cache hierarchy itself, or to the use of it by software, need to take this fact into account. Fifth, the cache hierarchy is a core component in emerging computing contexts, as PCIe accelerators (for heterogeneous computing) get direct access to it. Sixth, novel cache topologies feature several undocumented features [237]. Finally, capabilities to manipulate the cache hierarchy are an important building block for other important classes of microarchitectural attacks (cf. Section 3.2.5).

3.2 Cache Attacks

3.2.1 Cache Attack Techniques

The cache hierarchy is a complex hardware component that stores both program data and operational metadata. All cache-timing attacks share the same fundamental property: the time it takes to retrieve a piece of memory depends on the state of the cache, i.e., which lines are cached, and in which level(s). Still, there are several different *leakage mechanisms* through which a difference in a victim's memory accesses may lead to an attacker-observable time difference. Note that co-located attackers, i.e., those that share the computation platform with their victim, are not limited to measuring the execution time of the victim. Indeed, they can directly interact with the cache state through memory accesses and measure how long *their own* actions take [150], resulting in attack techniques that are considerably more powerful. We now classify the main sources of leakage in modern cache hierarchies and summarize them in Figure 3.1.

Cache Collisions. *Cache collision* attacks exploit that the execution time of a victim routine is correlated with the number of cache misses it experiences. By measuring the execution time of the victim routine, these attacks essentially obtain a noisy estimate of the number of *cache collisions* or, equivalently, the number of *different* cache lines that are touched during a particular execution. Then, the targeted secret information is extracted by statistically aggregating information on a large number of response times and correlating those with the inputs of the algorithm. Cache collision attacks do not fundamentally require any local cache preparation, so they can be mounted remotely [21]. A drawback is that the information obtained from a single execution is very coarse, both in space and in time, limiting the scope of cache collision attacks to particular implementations. A notorious class of algorithms that leaks through cache collisions are table-based implementations of block ciphers [21].

Cache Status. *Cache status* leakage reveals whether a specific cache line is used by a victim program. If its access latency for the attacker is low, the line was brought into the cache by some other process. The most straightforward implementation of cache status leakage is the Flush+Reload [81, 241] technique. First, the attacker prepares the cache state by ensuring that the cache line in question is removed from the cache. Then, at a later point in time, she measures whether its cache status has changed. If it has, another process has used this cache line.

Variations of the Flush+Reload attack have surfaced with slightly different properties. Evict+Reload [78] replaces the cache flushing with eviction for contexts where cache flushing primitives are not exposed to the attacker (e.g., on Arm platforms [121]). The Flush+Flush [77] technique integrates the measurement and preparation steps of the attack, as the execution time of the flush instruction depends on the caching status of a cache line.

Cache Contention. Memory accesses by a process also affect the cache status of *other cache lines* due to their competition for shared cache resources. Attacks exploiting this source of leakage are referred to as *cache contention* or *cache conflict* attacks. These attacks implement the Prime+Probe [101, 128, 142, 143, 237, 250] technique, where an attacker exhausts a full cache set of interest with her own lines, and later measures the access times of these lines. In this manner, she determines whether any of these lines was evicted by competing memory accesses by another process. To mount a Prime+Probe attack, an attacker must first obtain a set of memory addresses that, through address translation, the slicing function, and the set indexing function, map to the same cache set. Such a collection of addresses is referred to as an *eviction set*, and we will discuss how to construct it in Section 3.2.3. Note that cache contention leakage pertains to the cache itself, as well as its directory [237].

Typically, the attacker measures cache contention through *timing measurements* of accesses to *her own lines*. There are two main exceptions. First, on Intel processors with TSX (cf. Section 2.2.4), an attacker may implement the Prime+Abort [48] technique. Instead of using a timing measurement, this technique establishes a hardware transaction that is *aborted* in case contention occurs. Second, an attacker capable of measuring the execution time of the relevant *victim code* can mount an Evict+Time [143] attack. This technique infers contention by exhausting specific cache sets and correlating those with the victim's execution time.

Cache Occupancy. A particular case of leakage through cache contention is *cache occupancy* [134, 185, 186], where the object of contention is the cache as a whole. A cache occupancy measurement consists of measuring the time it takes an attacker to traverse a cache-sized buffer. As such, it does not require the construction of eviction sets, nor does it burden the attacker with finding the cache sets through which the victim is leaking. However, it sacrifices the temporal and spatial granularity of the channel.

Cache Replacement. Not only the data that is cached at a given point in time may encode memory access patterns, but also the metadata that is stored to

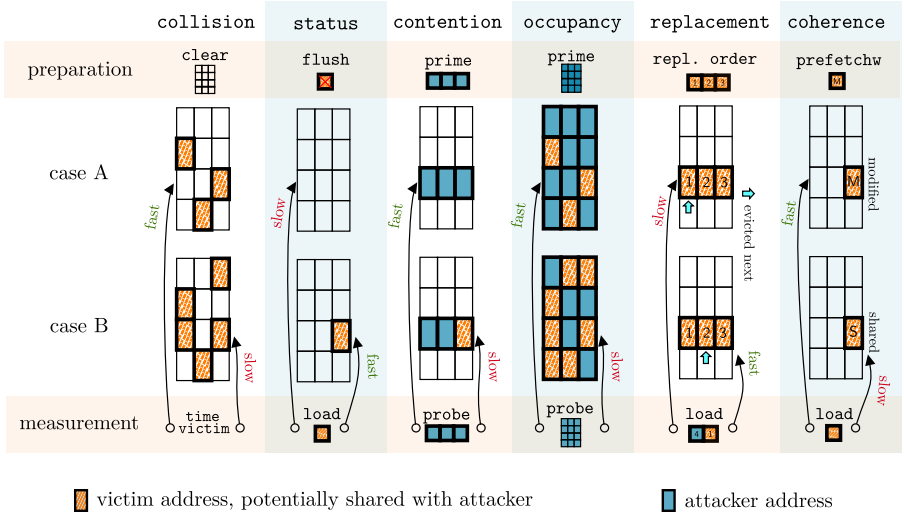


Figure 3.1: Cache attack techniques. Each technique (i.e., cache collision, status, contention, occupancy, replacement and coherence) comprises a preparation stage and a measurement stage. Typically, a timing measurement is used to distinguish between cases A and B.

make caching as efficient as possible. In particular, memory accesses influence the *cache replacement policy*, which may be exposed to an attacker through the sequence of future cache evictions [31, 109, 231]. Attacks targeting cache replacement leakage require intricate knowledge of the replacement policy being used. Additionally, as the cache replacement state is stored and acted on at the cache-set level, it requires the construction of eviction sets to influence and observe it, similar to cache contention attacks.

Cache Coherence. Another piece of metadata affected by memory accesses is the *coherence state* associated with a given cache line. The coherence state of a cache line may affect its access latency [239]. Most techniques exposing cache coherence metadata can only detect deliberate changes to the coherence state, e.g., writes. This limits their applicability to covert channels [44, 202, 239]. One exception is the Prefetch+Prefetch [87] attack on Intel processors. By using the `PREFETCHW` instruction to mark a cache line as "modified", future reads of this cache line by other processes can be detected, as those change the coherence state to "shared". All cache coherence attacks require shared memory, which is a drawback they share with cache status attacks.

3.2.2 Cross-Core Cache Attacks

Cross-Core Visibility. To satisfy the *full visibility* property from Section 3.1.4 in a cross-core attack, an attacker needs to be able to evict the target data from the victim's core-private caches. Otherwise, the private caches act as a filter for future memory accesses by the victim. For FLUSH-based attacks, this visibility is obtained automatically; the flush instruction invalidates a cache line in all cache levels that are kept coherent, i.e., across cores [241] and even across sockets [100]. For attacks that use eviction or contention, an attacker can achieve this through back-invalidation if the last-level cache is inclusive [128]. If the LLC is not inclusive but comes with an inclusive coherence directory (CD) (cf. Section 2.2.2), full visibility can be regained by targeting that component instead [237].

Shared Memory. A foundational limitation of cache status and cache coherence leakage is their structural dependence on shared memory with the victim. That is, the physical page that contains the cache line of interest needs to be mapped in the virtual address space of both the attacker and the victim. Cache replacement leakage does not strictly require shared memory, but the side-channel signal is much stronger if memory is shared [31, 231]. In the past, shared (read-only) memory with libraries like OpenSSL [81] could readily be obtained. Memory deduplication features (in several operating systems [10, 25]) produce shared mappings for virtual pages that have identical contents. However, such features are actively discouraged in multi-tenant scenarios [218], and obtaining shared mappings for arbitrary pages is not possible in several contexts (e.g., person-in-the-browser, cf. Section 3.1.2).

Rapid Measurements. The *temporal precision* of a cache attack technique captures how accurately an attacker can attribute a victim's memory access to a specific point in time. It is determined by the rate at which cache state measurements can be performed. This rate is often limited in practice. First, many cache attack techniques suffer from the observer effect [156, 240, 241], i.e., the act of measuring the cache state perturbs the cache state itself. Therefore, the state change produced by a measurement needs to be undone before it can be repeated. Second, even if the measurement is repeatable, the speed of a measurement depends on how many memory accesses it encompasses. We refer to a cache attack technique as providing *rapid measurements* when its measurements can be repeated and are very short (i.e., no more than a few hundred CPU cycles). Our own PRIME+SCOPE technique [156] achieves the highest temporal precision to date, i.e., about 70 CPU cycles.

Table 3.2: Cross-core cache attack comparison.

Attack Technique	Mechanism			Prerequisites	
	Leakage Source	Fine Grained	Rapid Measure	No Shared Memory	ISA Agnostic
Flush+Reload [241]	status	line✓	✗	✗	✗
Flush+Flush [77]	status	line✓	✓	✗	✗
Evict+Reload [78]	status	line✓	✗	✗	✓
Reload+Refresh [31]	replacement	line✓	✗	✗	✗
Prefetch+Prefetch [87]	coherence	line✓	✓	✗	✗
Cache Occupancy [186]	occupancy	none✗	✗	✓	✓
Prime+Probe [101, 128]	contention	set✓	✗	✓	✓
Prime+Abort [48]	contention	set✓	✓	✓	✗
Evict+Time [143]	contention	set✓	✗	✓	✗
Prime+Scope [156]	contention	set✓	✓	✓	✓

Comparison of Techniques. Table 3.2 compares the existing cross-core cache attack techniques. Some attacks have specific requirements on the system configuration (e.g., shared memory between attacker and victim) or on parts of the ISA being exposed to the attacker (e.g., the flush and prefetch instructions, or Intel TSX). For an attack to have fine-grained spatial and temporal precision in the absence of shared memory, it requires the up-front construction of eviction sets, which is a topic we turn to next.

3.2.3 Routines for Constructing Eviction Sets

Motivation. The eviction set problem was first introduced and solved to mount cross-core Prime+Probe attacks [101, 128]. However, finding addresses that map to the same LLC set and slice is also a necessary prerequisite for the Evict+Reload [78], Evict+Time [143], Reload+Refresh [31], and Prime+Abort [48] techniques.

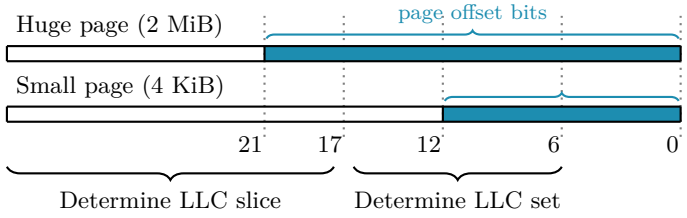


Figure 3.2: LLC set and slice index bits (Intel Xeon Platinum 8280).

Search Problem. Two aspects of modern computing systems complicate the construction of LLC eviction sets. The first challenge is that the LLC is physically indexed, while unprivileged processes interact with virtual addresses. The only address bits for which the knowledge is preserved across address translation are the page offset bits (cf. Figure 3.2). The second challenge is that LLCs are organized in slices. The processor may use an undocumented and processor-specific slice indexing function, incorporating many of the physical address bits, not just the least significant bits of the address. Both challenges imply that it is rarely possible to *statically*, i.e., without information gathered at runtime, determine whether two addresses are congruent. Therefore, researchers have developed routines to find eviction sets *dynamically*, i.e., informed by cache hit and miss behavior during the execution of the attacker process.

Reduction Methods. Early methods are tailored to inclusive LLCs and rely on huge pages [101, 128] or reverse engineering the slicing function [96, 102, 135, 242]. Oren et al. [142] are the first to construct eviction sets from the browser, without huge pages. Yan et al. [238] find congruent addresses for the coherence directory of non-inclusive Intel LLC. Islam et al. [104] learn additional physical address bits through observing false dependencies. All mentioned procedures have in common that they start from a large collection of addresses, which is a superset of an eviction set with high probability. Then, they conditionally remove a single element from this superset, depending on whether the eviction persists after it is removed. The runtime complexity of this routine is quadratic in the size of the initial superset. Other researchers [189, 217] propose to conditionally remove *groups* of addresses, resulting in a linear-time algorithm.

Expansion Methods. In our research, we developed an even more powerful strategy to construct eviction sets (cf. Chapter 6). It starts from an empty set and gradually adds addresses that demonstrate congruence in the LLC (or CD). The strategy applies to inclusive and non-inclusive Intel processors, works without huge pages, and is oblivious to the slice-index mapping. It is two orders of magnitude faster than the previous state of the art. Such a significant speedup reduces the runtime of the overall attack and enhances its stealth. Indeed, it may be necessary to construct eviction sets for a large number of cache sets and then isolate the cache sets that carry the secret-dependent signal [128, 136].

3.2.4 Practical Considerations

Reverse Engineering. CPU microarchitectures are opaque, as silicon vendors are not incentivized to expose any details beyond the specification of the

ISA. However, these details are necessary in order to validate, optimize and characterize side-channel leakage. Reverse engineering is, therefore, an effortful but important part of microarchitectural research. In the latest years, significant progress has been made in uncovering undocumented microarchitectural behavior. Some recent works focus specifically on modern cache hierachies [155, 156, 237] and their replacement policies [1, 31, 216]. Reverse engineering the microarchitectural properties of specific instructions may lead to optimizations of existing leakage channels (e.g., [77, 86]) or the discovery of undocumented cache fill policies [155]. Some implementation choices seem to apply generally across the industry (e.g., speculative execution [111]), while others are specific to CPU vendors [73, 121] or processor generations [237, 243].

Automation. To accelerate future developments in the field, automation may assist the microarchitectural security researcher. Already today, some degree of automation helps to find leakage in binaries to prototype attacks [78, 95, 179], or assists in the attack development process itself [51]. Automation may also help to find new sources of leakage [95, 225], and machine-learning techniques such as reinforcement learning may help to synthesize improved attack patterns [131].

Performance Degradation. As an alternative to increasing the time precision of the cache attack technique, the victim application can sometimes be slowed down [5, 7] or frequently interrupted [81, 139] instead. However, these techniques may have non-general system requirements (e.g., shared memory, specific OS scheduling policies, or elevated privileges) or may suffer from the absence of stealth because they significantly affect the victim's execution time.

Sources of Noise. In microarchitectural side-channel attacks, there are three main sources of *noise*, i.e., activity on the system that modulates CPU resources but that is not of interest to the attacker. First, noise may be produced by the attacker program itself, as it also executes on the system and hence affects the microarchitectural context. Second, some processor features, like speculative execution [111] or hardware prefetchers [184, 221] can generate spurious memory accesses, i.e., accesses that do not occur as part of the intended control flow of the attacker and victim programs. Third, there may be noise from other processes or the operating system.

3.2.5 Relation to Other Microarchitectural Attacks

In the last decade, two additional and important classes of software-based microarchitectural attacks have been discovered. While they do not constitute the focus of this thesis, advances in these attacks are strongly correlated with an increased understanding of the cache hierarchy and the techniques that can be used to manipulate it.

Transient Execution Attacks. Publicly disclosed in 2018, *transient execution attacks* [111, 124] are an emerging class of microarchitectural attacks. These attacks exploit the fact that, on a processor with out-of-order [124] or speculative [111] execution, instructions may be executed despite them not being part of the architecturally specified control flow. These instructions, referred to as *transient instructions*, are never architecturally committed. However, during their execution, they affect the microarchitectural context, which may be exposed using the leakage mechanisms covered in this chapter.

The cache hierarchy plays an integral role in the existing body of transient execution attacks for two main reasons. First, for most published attacks [66, 111, 124, 159, 160, 173, 174, 206, 207, 212], the cache hierarchy is used to establish a covert channel to exfiltrate the information from the transient domain. Second, these attacks often selectively remove data from the cache to ensure that optimal conditions are obtained for the processor to execute instructions transiently, which may require the construction of eviction sets.

Rowhammer. The main memory technology used in present-day computing systems is predominantly Dynamic Random Access Memory (DRAM). It stores the memory contents electrically as a capacitive charge. Over time, this charge leaks away automatically, requiring a so-called *refresh* to preserve the encoded value. In 2014, it was observed that this degenerative process may be adversarially accelerated to occur within a refresh interval, through a large number of DRAM activations to neighboring rows [108]. Such a capability was quickly shown to have severe security implications [180], in different execution contexts [76, 210], and despite recent mitigations by DRAM vendors [58, 113].

Knowledge of the cache hierarchy internals is a crucial prerequisite for Rowhammer attacks, especially when considering restricted execution contexts [76, 164], since the cache has to be bypassed to successfully trigger DRAM activations.

3.3 My Contributions

Contribution 1: Optimize Cache Contention Attacks

Context. In this chapter, leakage through cache contention was identified as an important source of microarchitectural leakage due to its full visibility across cores, fine spatial granularity, and very limited assumptions on attacker capabilities. However, the time precision of Prime+Probe is structurally limited by its working principle; each probing measurement accesses as many cache lines as the associativity of the microarchitectural element, i.e., more than ten ways for LLCs or CDs on modern Intel CPUs.

Research Outcomes. We developed the Prime+Scope technique [156] to optimize leakage through cache contention. Its cache measurement is *repeatable*, i.e., it can occur in a back-to-back fashion without interleaving a cache preparation step; and *optimally short*, i.e., it measures just one access to the L1 data cache. Because of this, Prime+Scope measurements can fire every 70 cycles (25 nanoseconds), far out of reach for existing techniques. Prime+Scope is generally applicable like Prime+Probe, works on LLCs and CDs, and applies to at least a decade of Intel CPUs. We confirmed its power with the fastest cross-core cache contention covert channel to date (3.5 Mbps capacity), an improved attack on AES T-tables, and a simple and portable eviction set construction procedure that outperforms previous techniques by 100-600x.

Enablers. To wield Prime+Scope on inclusive caches, we developed an automated gray-box search methodology. For non-inclusive caches, we discovered a novel eviction primitive for the coherence directory. During our research, we also developed portable routines to construct eviction sets in mere milliseconds. We open-sourced these artifacts to foster future research.

Publication. My CCS 2021 paper on this topic [156], entitled *"Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks"*, is included as Chapter 6 in this thesis. I am the principal author of this work.

Contribution 2: Explore Emerging Computing Contexts

Context. High-end computing environments, including those in multi-tenant clouds, are becoming increasingly heterogeneous through the deployment of domain-specific hardware accelerators (e.g., NICs, FPGAs, and GPUs). These accelerators interface directly with the processor's last-level cache to minimize the communication overhead with the CPU.

Research Outcomes. New trends in computing bring forth new threat models. In our Double Trouble work [155], we examined *combined cache attacks*, which arise when traditional co-tenancy is complemented with control over accelerators (such as FPGAs). These accelerators access the LLC directly [92, 99] but are severely constrained in their influence on the cache state. Our study showed that these constraints, despite being perceived as limitations in other threat models [116, 227], are actually *beneficial* to a combined attacker. Our observations challenge common assumptions in the field, such as the minimal size for eviction sets, the feasibility of cross-socket Evict+Reload attacks, and the accuracy of amplitude-modulated cache covert channels. We built a compact and extensible FPGA hardware accelerator to demonstrate our findings and used it to further shatter speed records for eviction set construction.

Enablers. To arrive at our results, we conducted a thorough FPGA-assisted reverse-engineering effort. This process revealed previously undocumented properties of Intel DDIO technology, such as an additional region of independent interest for security (e.g., [116, 196, 227]) and performance (e.g., [56, 57]) research. In addition, we uncovered previously unknown details about non-inclusive caches that contradict and complement those of prior work [237, 238] and allow for more efficient attacks. Finally, to reduce the friction for other researchers to start working on this topic, which has quite some barriers to entry, we developed an easy-to-use API for software to use the FPGA hardware accelerator. We also open-sourced our software and hardware implementations.

Publication. My USENIX Security 2022 paper on this topic [155], entitled *"Double Trouble: Combined Heterogeneous Attacks on Non-inclusive Cache Hierarchies"*, is included as Chapter 7 in this thesis. I am a main author together with Furkan Turan, who developed the FPGA implementation.

3.4 Conclusion

This chapter exposed microarchitectural side-channel attacks, and cache attacks in particular, as a threat to the security of modern multi-tenant computing systems. Cache attacks were argued to have high spatial and temporal precision, to be widely shared across computing contexts, and to concern one of the most performance-critical hardware components. We contributed Prime+Scope, a novel state-of-the-art cache attack technique with unprecedented time precision. Observing the trend of increasing complexity and heterogeneity in modern computing systems, we comprehensively studied the implications of sharing the cache hierarchy with potentially malicious accelerators.

Chapter 4

Defenses Against Cache Side-Channel Attacks

This chapter covers the state of the art in defenses against microarchitectural timing attacks, with a focus on defenses against cache side-channel attacks. Again, our emphasis lies on countermeasures that have received significant academic attention in recent years. For a complete survey, we refer to existing works (e.g., [61, 129, 192]).

Metrics for Countermeasures. Mitigations against microarchitectural timing attacks may differ in the security properties they provide. However, this is not the only basis on which they should be compared. A key consideration is how the mitigation affects, both directly and indirectly, the performance of the microarchitectural component or the system at large. For instance, flushing a cache upon each context switch is time-consuming, representing a direct cost, but also incurs the indirect cost of resuming execution with an empty cache.

Countermeasures also benefit from being customizable, i.e., whether they can be tweaked for different applications or security guarantees. Another essential factor is the invasiveness of the countermeasure in software or hardware, and the complexity and scale of its non-recurring engineering (NRE) costs. Another property that determines the adoption rate is the degree of manual effort for the software developer, whether or not it requires compiler or operating system changes, and whether the mitigation transparently applies to existing (unmodified) application software. A final consideration is that if the countermeasure disables existing interfaces, it also affects benign uses of such

interfaces, which may be undesirable or impossible to enforce because of, e.g., service-level agreements (SLAs).

4.1 Countermeasure Strategies

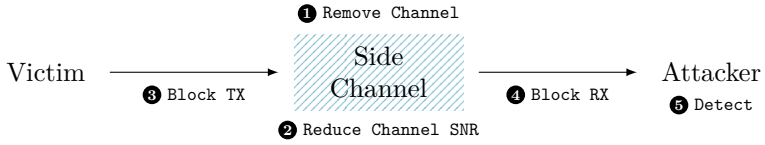


Figure 4.1: Strategies to defend against cache timing attacks.

In a side-channel attack, a victim can be considered to unwillingly encode (i.e., transmit) information on an incidental communication channel, which is then decoded by an attacker on the receiving end. Following such a representation, Figure 4.1 indicates the five main anchor points for defenses.

The first, and conceptually the simplest, countermeasure is to remove the existence of the channel altogether (❶). In many cases, this boils down to disabling a performance-enhancing feature, or partitioning it so it is no longer shared. The second strategy is to preserve the channel but severely reduce the quality of the information that can be transmitted on it (❷). The third strategy is to protect programs from transmitting secrets (❸), i.e., from *encoding* them into the microarchitectural context. The fourth strategy focuses on the receiving end of the channel, trying to preclude information from being *read out* of the microarchitectural context (❹). The fifth strategy is to attempt to detect when an attacker is performing a microarchitectural side-channel attack (❺).

In the remainder of this section, we elaborate on all five defense strategies and indicate their strengths and pitfalls based on lessons learned in recent research.

4.1.1 Remove the Channel

To eliminate side-channel leakage stemming from the shared usage of a microarchitectural component (strategy ❶), three main approaches exist that, at least in theory, completely remove the leakage.

Stop Sharing. The first approach is to disable the shared use of the performance-enhancing feature. Examples of this approach include disabling

page sharing (e.g., through memory deduplication). This eliminates all channels that exist because of the sharing (e.g., cache status and cache coherence leakage, cf. Table 3.2). A similar consideration holds for Simultaneous Multi-Threading (cf. Section 2.1.2), whose existence enables several high-bandwidth covert channels in the CPU pipeline [195], as well as the L1 caches [150]. Naturally, disabling hardware optimizations may come with severe performance penalties.

Temporal Multiplexing. The second strategy is temporal multiplexing, i.e., to maintain the shared usage of the component by multiple security domains, but not simultaneously. In this manner, the channel disappears during the exclusive access of the component, leaving only leakage through the state preserved across the *transition*. Implementations of state destruction across transitions are periodic flushes [143] of time-multiplexed private caches [251]. Note that temporal partitioning cannot be applied to inherently concurrent hardware resources, such as the last-level cache [61].

Spatial Multiplexing. The third strategy is spatial multiplexing, i.e., to partition the shared component between security domains, giving each their own exclusive share [145]. For caches, their typical two-dimensional set-associative structure permits two straightforward high-level spatial partitioning strategies: partitioning along the *sets* (e.g., [43, 47]) or along the *ways* (e.g., [15, 49, 109, 126, 254]). Another strategy is to lock lines in the cache to prevent their eviction by other processes [107, 224]. Spatial partitioning has the drawback of flexibility, i.e., rapid on-the-fly claiming and relinquishing of resources is hard. Under-utilization of the cache at any point in time corresponds to a performance penalty. Additionally, it may be hard to scale spatial partitioning to many domains.

Caveats. Disabling or multiplexing shared hardware components has the advantage of providing clear security guarantees. However, both temporal and spatial partitioning come with caveats revealed by recent work. Spatial partitioning needs to be implemented rigorously, ensuring that all software, including system software, is assigned to a partition. No single piece of software, not even a privileged one, should have access to all partitions. Otherwise, it can be used as a confused deputy [211]. In the context of caches, it needs to be ensured that all its leakage sources are partitioned, including the cache metadata. For instance, cache replacement information should not persist across temporal boundaries after flushing [215], nor leak across spatial boundaries [109].

4.1.2 Decrease the Signal-to-Noise Ratio of the Channel

The second main strategy to mitigate cache side-channel attacks is to allow a channel's existence but severely degrade its signal-to-noise ratio (strategy ❷). Therefore, it reduces the amount of information that can be obtained through shared usage of the component while maintaining the benefits of sharing.

Inject Additional Accesses. Some works propose to actively inject noise into the system [32, 118, 246], i.e., generate activity that is uncorrelated or anti-correlated [38] to the signal of interest. Preloading sensitive data into the cache [81] has been proposed to obfuscate subsequent secret-dependent accesses. With hardware transactional memory (cf. Intel TSX, Section 2.2.4), it is possible to additionally abort and restart the execution of a piece of code if one of the victim lines is evicted from the cache [74].

Remove Spatial Information. The cache contention leakage channel would be severely degraded, both in its temporal and spatial granularity, if all caches were to be implemented in a fully-associative manner, with a random replacement policy. Indeed, as such caches do not exhibit any evictions that correlate with the address of newly inserted cache lines, they only (at best) reveal the victim's working set size [46, 186]. Unfortunately, the power consumption of such a cache is prohibitively high, so it is not a viable solution for most real systems.

Preclude Full Visibility. Another approach is to remove the full visibility of the channel (cf. Section 3.1.4). With only partial visibility, the attacker needs to rely on activity on the victim core to evict the cache lines of interest, which degrades the time precision and weakens the side-channel signal. For the LLC and CD, full visibility can be disabled by preventing the inclusion property from invalidating lines in the lower-level caches. In this context, researchers propose a novel replacement policy across the cache hierarchy to avoid invalidations due to inclusion [235], or modify the coherence directory to provide it with a per-core private directory that avoids attacker-controllable evictions [238]. Some commercially available processors feature a mechanism that avoids LLC evictions if they would produce inclusion victims [73].

Cache Randomization. A new and promising line of work to defend against cache-side channel attacks is cache randomization. The idea is to make it impossible for an attacker to detect cache set contention, prompting them to resort to a cache occupancy attack, losing all spatial information. If successful,

it can effectively emulate the security properties of a fully associative cache without incurring the prohibitive overheads associated with them. Initial cache randomization proposals [127, 224] target the L1 cache and use an indirection table to implement a randomized memory-to-cache mapping. However, they scale poorly with cache size, making them prohibitively expensive for large caches and therefore unfit to be deployed for last-level caches.

The last-level cache is the most important cache to protect, as it is shared across cores. At the same time, it has the highest access latency, so the sensitivity of its performance to slight tweaks is much lower than for the other cache levels. This presents an opportunity exploited by recent designs. These designs use low-latency cryptographic mechanisms to compute a pseudorandom address-to-index mapping on the fly, in hardware, for every LLC memory access [157, 158, 201, 228]. Most other parts of the memory hierarchy actually do not need to be changed. We revisit LLC cache randomization in more detail in Section 4.2.

Caveats. Injecting unnecessary memory accesses does not remove the presence of the initial side-channel signal. Therefore, the signal may be uncovered with additional measurements [38]. Preloading sensitive data may not block all sources of leakage (e.g., cache replacement metadata) or be circumvented by rapid cache manipulation strategies [29]. Eliminating full visibility may not be sufficient to block all practical attacks [73]. For cache randomization, there may exist advanced techniques to recover spatial information on memory accesses and increase the signal-to-noise ratio again (cf. Section 4.3). There may also exist other side channels that reveal cache set congruence [198].

4.1.3 Block the Encoding of the Secret

The third high-level strategy to defend against side-channel attacks is to stop programs from unintentionally encoding their secrets into the microarchitectural context (strategy ❸). Most proposals adhering to this strategy are centered around the notion of *constant-time programming*.

Constant-Time Programming. A program is said to satisfy the constant-time property if it does not branch on secrets, does not perform secret-dependent memory accesses, and does not allow secrets to act as operands to variable-latency instructions [21, 105, 110]. Producing software that satisfies this property requires considerable expertise and undermines the abstractions provided by the ISA. Therefore, several machine-assisted techniques have been developed to analyze programs for leakage [16, 105], using methods such as dynamic

instrumentation (e.g., [117, 222]), formal analysis (e.g., [8, 50]), or statistical techniques (e.g., [162]). Verifying the constant-time property for a piece of code may occur at the level of compiled and optimized (LLVM) assembly code [8], at the binary level [229], or experimentally for the execution of a given binary on a given platform [162]. Some tools are able to eliminate classes of leakage automatically (e.g., [41, 230]).

Discussion. An advantage of the constant-time programming paradigm is that, when fully abided by, it blocks the transmission on many known channels, more so than ad-hoc countermeasures. However, while burdening the developer to adhere to constant-time programming practices is a reasonable assumption for highly sensitive code, it is unreasonable to expect such care and effort to become part of general software engineering practices. Many platforms now provide constant-time hardware support for core cryptographic primitives [79, 80].

Caveats. The constant-time property is not necessarily portable across platforms due to operand-dependent instruction timings [11, 41]. Portability issues can be avoided by exposing the constant-time property of an instruction as part of the ISA [244]. Special care should be taken to preserve constant-time properties across the compilation chain [17, 187]. As constant-time programming protects potential victims from unintentionally transmitting their secrets through the modulation of microarchitectural resources, it does not protect against covert channels, where secrets are intentionally transmitted.

4.1.4 Block the Decoding of the Secret

The fourth high-level strategy to defend against timing side-channel attacks is to thwart the readout of secret information from the microarchitectural context (strategy ❹). This strategy mainly manifests itself as restrictions on the timing sources present in the system.

Restricting Timers. Timing sources may be completely virtualized, detaching them from the actual passage of time (e.g., [14, 119]). To defend against remote attackers, servers can schedule their response to be transmitted only at discrete times (e.g., [115]). A popular academic proposal [114, 133, 213] is to drastically reduce the resolution of available timers (e.g., to 100 µs), since the decoding of microarchitectural side channels requires timing precisions in the order of 10 to 100 nanoseconds. Today, all major browsers have adopted some form of timer

granularity restrictions [67, 138, 219, 226], as it is a straightforward and broad countermeasure against CPU timing attacks.

Caveats. Virtualizing time may incur severe overheads [119] and affect legitimate uses of time, e.g., in interacting with other systems [61]. For environments without fine-grained timers, part of the lost precision may be recovered by fabricating new timers [72, 167, 175]. If the attacker thread is networked, it may also communicate with a timing server, resulting in a timer granularity that depends on the jitter in the network latency [178]. Additionally, the time difference between microarchitectural events may be amenable to amplification (cf. [137] and Section 4.3).

4.1.5 Detect the Attack at Runtime

The fifth major defense strategy is to detect and stop attacks as they occur (strategy ❺). The motivating observation is that CPU timing attacks exhibit signature behavior that allows one to distinguish them from benign applications.

On-the-fly Detection. The detection strategy is appealing, as it does not adversely affect the performance of the processor in normal operating regimes. Instead, the performance overhead is limited to the response upon detection of an attack. Several works propose to leverage existing hardware performance counters (HPCs) [4, 30, 37, 77, 85, 141, 149, 248]. Note that, on modern platforms, system-level privileges are required to access these counters, and only a limited number of counters may be queried simultaneously. Other works propose to add dedicated detection hardware, e.g., to pattern-match for recurrent contention patterns in important CPU components [36], to compare two executions of a program with different address-to-index mappings [236], or to detect cyclic interference between security domains [88].

Caveats. The main challenge for the adoption of on-the-fly detection is achieving the extremely low false positive rates needed for real-world systems. False positive rates need to be extremely low to avoid benign activity from being flagged. Another challenge is that detection-based countermeasures are prone to cat-and-mouse games; attacks can be modified to become less detectable (e.g., [77]) or explicitly avoid the metric used for detection (e.g., [31]). Finally, the action that should be undertaken upon detecting an attack is subject to a system-dependent trade-off between usability, performance and security.

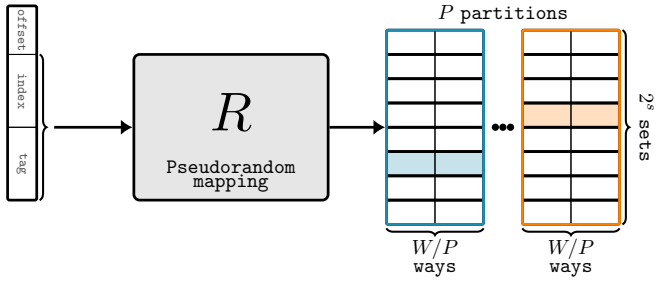


Figure 4.2: Randomized skewed cache.

4.2 Randomization-based Protected Caches

This section revisits cache randomization, a defense against cache attacks that aims to severely decrease their signal-to-noise ratio at low overhead (❷). We focus on recent and state-of-the-art designs that protect the shared LLC.

Deterministic Randomization Mapping. The first generation of randomized LLC proposals [157, 201] tweak the LLC address-to-index mapping to increase its unpredictability without affecting the sliced, set-associative structure of the LLC. By instantiating the set index mapping as a pseudorandom function, i.e., one that is random-looking but deterministic, these proposals eradicate all prior knowledge an attacker has about LLC-congruence (cf. Section 3.2.3).

However, even if such a randomization mapping uniformly randomly distributes cache lines among sets, its deterministic nature implies that LLC eviction sets still exist, albeit slightly harder to construct. The proposed solution to prevent an adversary from finding and using eviction sets is to change (or *rekey*) the randomization mapping when a particular condition is satisfied, e.g., the number of LLC accesses exceeds a predetermined threshold. Rekeying neutralizes any eviction sets an adversary has already constructed, forcing her to start over. However, the proposed rekeying rates significantly underestimate the potential for speeding up eviction set construction (cf. Section 3.2.3). To maintain security against these novel techniques, the rekeying rates of first-generation designs would need to be increased to impractical levels [158].

Probabilistic Randomization Mapping. The second generation of randomized LLCs [158, 228] inserts a *probabilistic* component into the randomized set indexing function. In particular, the designs in this category leverage a skewed

cache architecture [181]. Each cache set is subdivided into a fixed number of skews or *partitions*, where each partition is indexed by a distinct pseudorandom function. Upon a cache line fill request, one of the partitions is randomly selected to host the incoming line, and the cache set index is determined by that partition's randomization mapping. Upon a cache lookup, the mapping for each partition is evaluated, and the original address of the matching cache lines is compared to the requested line to assess whether there is a cache hit.

Randomized skewed caches cleverly inject entropy on every cache line fill while keeping the resulting lookup tractable. The implied non-determinism enforces a complete overhaul for attacks, with two main implications. First, existing routines for eviction set construction [157, 217] do not tolerate probabilistic conflicts. Worse, eviction sets in the traditional sense no longer *exist* for these designs, as it is highly unlikely for addresses to be congruent in all partitions simultaneously. Second, assuming an adversary *does* manage to find addresses that contend in one or more partitions, the following exploitation phase is also adversely affected by the now-probabilistic nature of cache contention.

Low-Latency Cryptography. A crucial requirement for cache randomization is that its performance overhead is small. An essential and challenging consideration is minimizing the latency to compute the index mapping. Some designs propose bespoke low-latency functions [157, 158, 194] which, as we will show in Section 4.3, risk shortcut attacks that target the randomization mapping directly. For future-proof security properties, the cache randomization field may draw from advances in low-latency cryptography [13, 18, 19, 24], or develop custom but cryptographically sound functions [34, 65].

4.3 My Contributions

Deploying defenses against microarchitectural side-channel attacks incurs a considerable cost, whether that be in performance, energy, hardware utilization, modifications to existing software, or compatibility breaks. Therefore, defenses must be evaluated well in advance of their deployment. In this dissertation, two promising and influential countermeasures are evaluated and shown to be insufficient to thwart all attacks.

Contribution 3: Evaluate Cache Randomization

Context. In Section 4.2, randomized cache architectures were identified as a promising countermeasure against cache side-channel attacks, due to their transparency, low overhead, and considerable reduction of the quality of the side channel. After efficient eviction set construction techniques defeated the first generation of cache randomization designs, stronger designs were proposed at flagship conferences (e.g., CEASER-S at ISCA, ScatterCache at USENIX Security). Prior to our work, it was an open question whether these designs stood up to sophisticated and previously unknown statistical attacks.

Research Outcomes. We consolidated the existing cache randomization designs in a generic model and systematically evaluated their security. The resulting Prime+Prune+Probe attack applies to all designs and is generic enough [62] to affect PhantomCache [194], a different type of probabilistic randomized cache that appeared after our analysis was completed. Orthogonally, with a devastating shortcut attack on CEASER [157] and CEASER-S [158], we show that the strength of the randomization function is essential to security.

Publication. My S&P 2021 paper on this topic [154], entitled *"Systematic Analysis of Randomization-based Protected Cache Architectures"*, is included as Chapter 8 in this thesis. I am the principal author of this work.

Follow-up. Related to this work, I co-authored an S&P 2023 paper [65] which proposes an innovative combination of randomization and isolation. By orchestrating partially overlapping cache utilization profiles for different security domains, our cache design is able to wield useful properties of isolation without facing the problem of cache underutilization. At the same time, in addition to the obfuscation provided by randomization, it removes the *full visibility* property of cache contention; over time, cache lines become hidden from the attacker. This not only thwarts my Prime+Prune+Probe attack [154], it also defends against cache occupancy attacks, which were previously considered to be out of scope for randomization-based secure caches [157, 158, 172, 228]. My main contribution to this article is the security analysis of the design.

Follow-up by the Community. Figure 4.3 shows a selection of related work that followed up on my evaluation in the two years since its publication. Several new designs were proposed that explicitly try to defend against the techniques I have uncovered [40, 65, 172, 188, 199, 205]. My statistical analysis techniques were

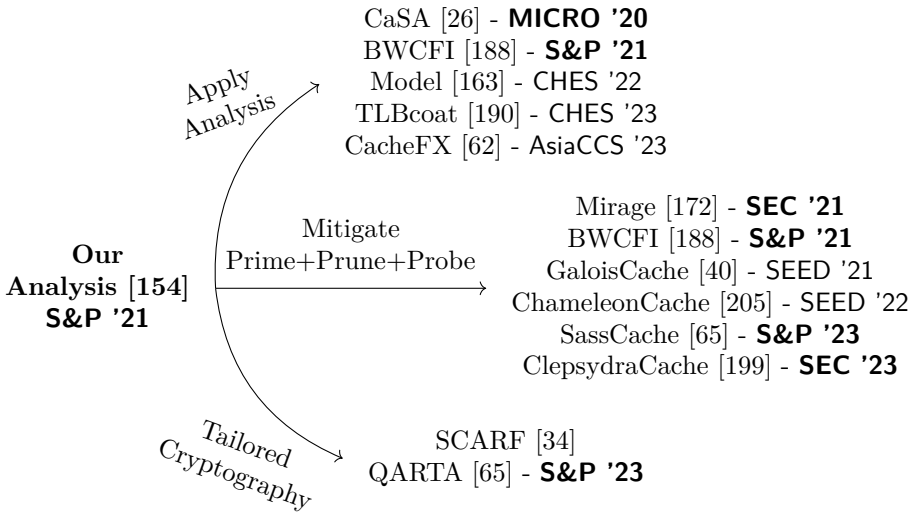


Figure 4.3: Positioning of my cache randomization research and follow-up work (tier-1 venues in **bold**).

applied by Bourgeat et al. [26], Song et al. [188], Ribes-González et al. [163], and Stolz et al. [190]. In response to my shortcut attacks exploiting the randomization itself, novel low-latency ciphers [34, 65] were specifically developed for randomization-based protected caches.

Contribution 4: Evaluate Timer Restrictions

Context. Restricting the granularity of available timers is a promising countermeasure against CPU timing side-channel attacks. It does not require hardware changes, does not constrain developers to produce constant-time code, and does not affect the performance-enhancing behavior of microarchitectural components. Moreover, it influences the readout phase of all types of timing attacks, not just cache attacks. For these reasons, it has received attention in academia [114, 133, 213] and has been deployed in all major browsers [67, 138, 219, 226]. Although basic cache attack primitives were shown without high-precision timers, large classes of microarchitectural attacks were thought to be mitigated, including cross-core and stateless side-channel attacks.

Research Outcomes. We showed that restricted timers are an incomplete defense against CPU timing attacks. First, we proposed new *single-shot amplifiers*, i.e., pieces of code that produce a large timing difference from a single difference in the microarchitectural context. Our most potent amplifier produces timing differences discernable by the human eye with 99% accuracy (median for fifteen participants with 100 observations each). Second, we developed conversion routines to convert side-channel information from one leakage type to another. Combining these two primitives, we proposed a generic framework for single-shot amplification of arbitrary microarchitectural timing leaks. We performed several case studies, e.g., the generation of cache eviction sets, both in real-world restricted browser environments (i.e., the latest version of Chrome) and natively using timers with precisions ranging from microseconds to *seconds*.

Publication. My AsiaCCS 2023 paper on this topic [153], entitled *"ShowTime: Amplifying Arbitrary CPU Timing Side Channels"*, is included as Chapter 9 in this thesis. I am the principal author of this work.

4.4 Conclusion

Eliminating microarchitectural side channels without tarnishing the performance optimization that causes them is a challenging problem. In this chapter, we identified and evaluated two influential and promising proposals to defend against cache attacks. Both promise a severe reduction of the side-channel attack potential without significantly affecting performance. With our work on randomization-based secure caches, we showed that sophisticated statistical attacks still apply to randomization-based secure caches and that, to maintain security, rekeying rates should be much higher than anticipated. Additionally, we demonstrated that cryptographic weaknesses in the randomization function may allow an adversary to completely bypass all intended security mechanisms. With our work on side-channel conversion and amplification, we arrived at the conclusion that timer restrictions alone, even when implemented beyond practical limits, provide insufficient protection against CPU timing attacks.

Chapter 5

Conclusion

At the risk of misprediction, this chapter identifies broad research trends in microarchitectural security and potentially promising branches moving forward.

Microarchitectural Exploration. Over the years, the research community has steadily increased its understanding of the leakage mechanisms that plague modern microarchitectures. Going forward, we expect this trend to continue. Novel microarchitectural features still make their way into commercial processors [214, 247], computing architectures adapt over time [155, 237], and research is expanding toward other processor vendors than Intel [60, 121, 214]. Even in the absence of new leakage sources, our Prime+Scope work [156] showed that order-of-magnitude improvements are still possible even for side channels that were thought to be well understood. Finally, one can observe an encouraging tendency in the community toward publicly sharing research artifacts, which democratizes academic microarchitectural research.

Complexity and Integration. The complexity and heterogeneity of computing systems keep increasing. Learning from our examination of heterogeneous systems [155], we foresee it to be even more challenging to defend against such attacks in the future, as the associated performance-security trade-offs are complicated, countermeasures are fragile, and improvements to attacks are easy to overlook. Automated testing is desirable to characterize new channels and avoid regressions, which becomes easier with increasing transparency. An antidote against complexity and opacity can be found in a clean-slate approach, for which the growing RISC-V ecosystem could act as a catalyst.

Security Specialization. As argued in Chapter 4, practically eliminating all side channels in modern computing systems is a challenging undertaking. Even assuming that all leakage sources are documented and characterized, the economic incentives of CPU vendors and cloud service providers remain firmly pointed towards increased multi-tenancy, heterogeneity, and aggressive performance enhancements. A pragmatic solution may be on-die specialization, where different CPU cores in the same package differ in their security-performance design choices, similar to existing performance and efficiency cores. This way, at least high-security processes can be holistically shielded from most of the microarchitectural attack surface.

High-Leverage Mitigations. In the absence of holistic countermeasures, low-overhead mitigations for the most significant side channels may still be a promising short-term approach. Chapter 3 argued that the shared cache hierarchy is one of the most high-leverage microarchitectural components, while Chapter 4 established that mitigations for the LLC and CD may have reasonable performance overheads. Combined with a strict core-based scheduling policy [42], they may pragmatically eliminate the strongest leakage. However, additional analysis, such as ours [154], is required to increase confidence in the security of probabilistic mitigations, similar to how the confidence in cryptographic primitives is established through its resilience against third-party analysis.

Physical Side Channels. Recently, microarchitectural side-channel attacks traversed another abstraction layer, i.e., the interaction of the computing device with the physical world. Indeed, some processor features measure physical properties, like the instantaneous power consumption, and expose it to unprivileged processes [123] or actuate based on them [125, 223]. In contrast to most of the side-channel attacks described in this dissertation, the physical leakage mechanism may directly leak the data processed by instructions. For now, software-observable physical side-channel leakage is fairly coarse-grained, resulting in low leakage rates. However, it would be naive to underestimate the signal-enhancing capabilities and motivation of the research community. Opportunities for low-overhead hardware mitigations may lie in established defensive practices for embedded devices (e.g., masking [35, 69]).

Bibliography

[1] A. Abel and J. Reineke. “uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures”. In: *ASPLOS*. 2019 (p. 33).

[2] O. Aciiçmez. “Yet another microarchitectural attack: exploiting I-cache”. In: *ACM Workshop on Computer Security Architecture*. 2007 (p. 23).

[3] O. Aciiçmez and J.-P. Seifert. “Cheap hardware parallelism implies cheap security”. In: *Fault Diagnosis and Tolerance in Cryptography (FDTC)*. 2007 (pp. 24, 25).

[4] M. Alam, S. Bhattacharya, D. Mukhopadhyay, and S. Bhattacharya. “Performance counters to rescue: A machine learning based safeguard against micro-architectural side-channel-attacks”. In: *IACR Cryptol. ePrint Arch. 2017/564*. 2017 (p. 45).

[5] A. C. Aldaya and B. B. Brumley. “HyperDegrade: From GHz to MHz Effective Cpu Frequencies”. In: *USENIX Security Symposium*. 2022 (p. 33).

[6] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. P. García, and N. Tuveri. “Port contention for fun and profit”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (pp. 23–26).

[7] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom. “Amplifying Side Channels Through Performance Degradation”. In: *Annual Conference on Computer Security Applications (ACSAC)*. 2016 (p. 33).

[8] J. B. Almeida, M. Barbosa, G. Barthe, and F. Dupressoir. “Verifying Constant-Time Implementations.” In: *USENIX Security Symposium*. 2016 (p. 44).

[9] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom. “Ladderleak: Breaking ECDSA With Less Than One Bit of Nonce Leakage”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020 (p. 22).

[10] A. Arcangeli, I. Eidus, and C. Wright. “Increasing Memory Density by using KSM”. In: *Proceedings of the Linux Symposium*. 2009 (p. 30).

[11] Arm. *ARM7TDMI Technical Reference Manual r4p1 - Multiply and multiply accumulate*. `https://developer.arm.com/documentation/ddi0210/c/Instruction-Cycle-Timings/Multiply-and-multiply-accumulate`. 2001 (p. 44).

[12] D. Asonov and R. Agrawal. “Keyboard acoustic emanations”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2004 (p. 18).

[13] R. Avanzi. “The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes”. In: *IACR Transactions on Symmetric Cryptology (ToSC)*. 2017 (p. 47).

[14] A. Aviram, S. Hu, B. Ford, and R. Gummadi. “Determinating timing channels in compute clouds”. In: *ACM Workshop on Cloud Computing Security (CCSW)*. 2010 (p. 44).

[15] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf. “CURE: A Security Architecture with CUstomizable and Resilient Enclaves”. In: *USENIX Security Symposium*. 2021 (p. 41).

[16] M. Barbosa, G. Barthe, K. Bhargavan, B. Blanchet, C. Cremers, K. Liao, and B. Parno. “SoK: Computer-aided cryptography”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (p. 43).

[17] G. Barthe, B. Grégoire, and V. Laporte. “Secure compilation of side-channel countermeasures: the case of cryptographic “constant-time””. In: *IEEE Computer Security Foundations (CSF)*. 2018 (p. 44).

[18] C. Beierle, J. Jean, S. Kölbl, G. Leander, A. Moradi, T. Peyrin, Y. Sasaki, P. Sasdrich, and S. M. Sim. “The SKINNY family of block ciphers and its low-latency variant MANTIS”. In: *CRYPTO*. 2016 (p. 47).

[19] Y. Belkheyar, J. Daemen, C. Dobraunig, S. Ghosh, and S. Rasoolzadeh. “BipBip: A Low-Latency Tweakable Block Cipher with Small Dimensions”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2023 (p. 47).

[20] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom. ““Ooh Aah... Just a Little Bit”: A Small Amount of Side Channel can go a Long Way”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2014 (p. 22).

[21] D. J. Bernstein. *Cache-timing attacks on AES*. 2005 (pp. 22, 27, 43).

[22] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom. "Sliding Right into Disaster: Left-to-right Sliding Windows Leak". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2017 (p. 22).

[23] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus. "SMoTherSpectre: Exploiting Speculative Execution through Port Contention". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019 (pp. 23–26).

[24] J. Borghoff, A. Canteaut, T. Güneysu, E. B. Kavun, M. Knezevic, L. R. Knudsen, G. Leander, V. Nikov, C. Paar, C. Rechberger, P. Rombouts, S. S. Thomsen, and T. Yalçın. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications". In: *ASIACRYPT*. 2012 (p. 47).

[25] E. Bosman, K. Razavi, H. Bos, and C. Giuffrida. "Dedup Est Machina: Memory Deduplication as an Advanced Exploitation Vector". In: *IEEE Symposium on Security and Privacy (S&P)*. 2016 (p. 30).

[26] T. Bourgeat, J. Drean, Y. Yang, L. Tsai, J. Emer, and M. Yan. "CaSA: End-to-end Quantitative Security Analysis of Randomly Mapped Caches". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2020 (p. 49).

[27] R. Branco and B. Lee. "Cache-related Hardware Capabilities and Their Impact on Information Security". In: *ACM Computing Surveys* (2022) (p. 21).

[28] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi. "Software grand exposure:{SGX} cache attacks are practical". In: *Workshop On Offensive Technologies (WOOT)*. 2017 (p. 23).

[29] S. Briongos, I. Bruhns, P. Malagón, T. Eisenbarth, and J. M. Moya. "Aim, Wait, Shoot: How the CACHESNIPER Technique Improves Unprivileged Cache Attacks". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021 (p. 43).

[30] S. Briongos, G. Irazoqui, P. Malagón, and T. Eisenbarth. "Cacheshield: Detecting Cache Attacks Through Self-observation". In: *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2018 (p. 45).

[31] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth. "RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks". In: *USENIX Security Symposium*. 2020 (pp. 26, 29–31, 33, 45).

[32] R. Brotzman, D. Zhang, M. Kandemir, and G. Tan. “Ghost Thread: Effective User-Space Cache Side Channel Protection”. In: *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2021 (p. 42).

[33] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom. “Flush, Gauss, and Reload–a Cache Attack on the BLISS Lattice-based Signature Scheme”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2016 (p. 22).

[34] F. Canale, T. Güneysu, G. Leander, J. Thoma, Y. Todo, and R. Ueno. “SCARF: A Low-Latency Block Cipher for Secure Cache-Randomization”. In: *IACR Cryptol. ePrint Arch. 2022/1228*. 2022 (pp. 47, 49).

[35] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi. “Towards sound approaches to counteract power-analysis attacks”. In: *CRYPTO*. 1999 (p. 52).

[36] J. Chen and G. Venkataramani. “CC-hunter: Uncovering covert timing channels on shared processor hardware”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2014 (p. 45).

[37] M. Chiappetta, E. Savas, and C. Yilmaz. “Real time detection of cache-based side-channel attacks using Hardware Performance Counters”. In: *IACR Cryptol. ePrint Arch. 2015/1034*. 2015 (p. 45).

[38] D. Cock, Q. Ge, T. Murray, and G. Heiser. “The last mile: An empirical study of timing channels on seL4”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2014 (pp. 42, 43).

[39] S. Cohney, A. Kwong, S. Paz, D. Genkin, N. Heninger, E. Ronen, and Y. Yarom. “Pseudorandom black swans: Cache attacks on CTR_DRBG”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2020 (p. 22).

[40] S. Constable and T. Unterluggauer. “Seeds of SEED: A Side-Channel Resilient Cache Skewed by a Linear Function over a Galois Field”. In: *IEEE Symposium on Secure and Private Execution Environment Design (SEED)*. 2021 (pp. 48, 49).

[41] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter. “Practical mitigations for timing-based side-channel attacks on modern x86 processors”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2009 (p. 44).

[42] J. Corbet. *Many uses for Core scheduling*. `https://lwn.net/Articles/799454/`. 2019 (pp. 24, 52).

[43] V. Costan, I. Lebedev, and S. Devadas. “Sanctum: Minimal Hardware Extensions for Strong Software Isolation”. In: *USENIX Security Symposium*. 2016 (p. 41).

[44] Y. Cui, C. Yang, and X. Cheng. “Abusing Cache Line Dirty States to Leak Information in Commercial Processors”. In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2022 (pp. 26, 29).

[45] M. Dai, R. Paccagnella, M. Gomez-Garcia, J. McCalpin, and M. Yan. “Don’t Mesh Around: Side-Channel Attacks and Mitigations on Mesh Interconnects”. In: *USENIX Security Symposium*. 2022 (pp. 24, 26).

[46] G. Dessouky, T. Frassetto, and A.-R. Sadeghi. “HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments”. In: *USENIX Security Symposium*. 2020 (p. 42).

[47] G. Dessouky, A. Gruler, P. Mahmoody, A.-R. Sadeghi, and E. Stapf. “Chunked-Cache: On-Demand and Scalable Cache Isolation for Security Architectures”. In: *Network and Distributed System Security Symposium (NDSS)*. 2022 (p. 41).

[48] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen. “Prime+Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX”. In: *USENIX Security Symposium*. 2017 (pp. 28, 31).

[49] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev. “Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks”. In: *ACM Transactions on Architecture and Code Optimization (TACO)*. 2012 (p. 41).

[50] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. “Cacheaudit: A tool for the static analysis of cache side channels”. In: *ACM Transactions on Information and System Security (TISSEC)*. 2015 (p. 44).

[51] C. Easdon, M. Schwarz, M. Schwarzl, and D. Gruss. “Rapid Prototyping for Microarchitectural Attacks”. In: *USENIX Security Symposium*. 2022 (p. 33).

[52] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh. “Computing with time: Microarchitectural weird machines”. In: *ASPLOS*. 2021 (p. 22).

[53] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Jump over ASLR: Attacking branch predictors to bypass ASLR”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016 (pp. 24–26).

[54] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. “Understanding and mitigating covert channels through branch predictors”. In: *ACM Transactions on Architecture and Code Optimization (TACO)*. 2016 (p. 26).

[55] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. “BranchScope: A New Side-Channel Attack on Directional Branch Predictor”. In: *ASPLOS*. 2018 (pp. 24, 26).

[56] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. “Make the Most Out of Last Level Cache in Intel Processors”. In: *EuroSys Conference*. 2019 (p. 36).

[57] A. Farshin, A. Roozbeh, G. Q. Maguire Jr, and D. Kostić. “Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks”. In: *USENIX Annual Technical Conference (ATC)*. 2020 (p. 36).

[58] P. Frigo, E. Vannacci, H. Hassan, V. Van Der Veen, O. Mutlu, C. Giuffrida, H. Bos, and K. Razavi. “TRRespass: Exploiting the many sides of target row refresh”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2020 (p. 34).

[59] K. Gandolfi, C. Mourtel, and F. Olivier. “Electromagnetic analysis: Concrete results”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2001 (p. 18).

[60] S. Gast, J. Juffinger, M. Schwarzl, G. Saileshwar, A. Kogler, S. Franza, M. Köstl, and D. Gruss. “SQUIP: Exploiting the Scheduler Queue Contention Side Channel”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023 (pp. 25, 51).

[61] Q. Ge, Y. Yarom, D. Cock, and G. Heiser. “A survey of microarchitectural timing attacks and countermeasures on contemporary hardware”. In: *J. Cryptographic Engineering*. 2018 (pp. 21, 39, 41, 45).

[62] D. Genkin, W. Kosasih, F. Liu, A. Trikalinou, T. Unterluggauer, and Y. Yarom. “CacheFX: A framework for evaluating cache security”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2023 (pp. 48, 49).

[63] D. Genkin, L. Valenta, and Y. Yarom. “May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017 (p. 22).

[64] B. Gierlichs, L. Batina, C. Clavier, T. Eisenbarth, A. Gouget, H. Handschuh, T. Kasper, K. Lemke-Rust, S. Mangard, A. Moradi, et al. “Susceptibility of eSTREAM candidates towards side channel analysis”. In: *State of the Art of Stream Ciphers Workshop*. 2008 (p. 22).

[65] L. Giner, S. Steinegger, A. Purnal, M. Eichlseder, T. Unterluggauer, S. Mangard, and D. Gruss. “Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2023 (pp. 47–49).

[66] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida. "Speculative Probing: Hacking Blind in the Spectre Era". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2020 (p. 34).

[67] Google. *Align performance API timer resolution to cross-origin isolated capability - Chrome Platform Status*. `https://chromestatus.com/feature/6497206758539264`. 2021 (pp. 45, 49).

[68] Google. *Product Status: Microarchitectural Data Sampling (MDS)*. `https://support.google.com/faqs/answer/9330250?hl=en`. 2019 (p. 24).

[69] L. Goubin and J. Patarin. "DES and differential power analysis the "Duplication" method". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 1999 (p. 52).

[70] B. Gras, C. Giuffrida, M. Kurth, H. Bos, and K. Razavi. "ABSynthe: Automatic Blackbox Side-channel Synthesis on Commodity Microarchitectures." In: *Network and Distributed System Security Symposium (NDSS)*. 2020 (p. 24).

[71] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. "Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks". In: *USENIX Security Symposium*. 2018 (pp. 24–26).

[72] B. Gras, K. Razavi, E. Bosman, H. Bos, and C. Giuffrida. "ASLR on the Line: Practical Cache Attacks on the MMU". In: *Network and Distributed System Security Symposium (NDSS)*. 2017 (p. 45).

[73] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth. "AutoLock: Why cache attacks on ARM are harder than you think". In: *USENIX Security Symposium*. 2017 (pp. 33, 42, 43).

[74] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. "Strong and Efficient Cache Side-channel Protection Using Hardware Transactional Memory". In: *USENIX Security Symposium*. 2017 (p. 42).

[75] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard. "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016 (p. 22).

[76] D. Gruss, C. Maurice, and S. Mangard. "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016 (p. 34).

[77] D. Gruss, C. Maurice, K. Wagner, and S. Mangard. "Flush+Flush: A Fast and Stealthy Cache Attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2016 (pp. 25, 28, 31, 33, 45).

[78] D. Gruss, R. Spreitzer, and S. Mangard. “Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches”. In: *USENIX Security Symposium*. 2015 (pp. 28, 31, 33).

[79] S. Gueron. “Intel’s new AES instructions for enhanced performance and security”. In: *Fast Software Encryption (FSE)*. 2009 (p. 44).

[80] S. Gueron and M. E. Kounavis. “Intel®carry-less multiplication instruction and its usage for computing the GCM mode”. In: *Intel White Paper 323640-001, Revision 2.0*. 2010 (p. 44).

[81] D. Gullasch, E. Bangerter, and S. Krenn. “Cache Games–Bringing Access-based Cache Attacks on AES to Practice”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2011 (pp. 19, 22, 25–27, 30, 33, 42).

[82] B. Gulmezoglu. “XAI-based Microarchitectural Side-Channel Analysis for Website Fingerprinting Attacks and Defenses”. In: *IEEE Transactions on Dependable and Secure Computing (TDSC)* (2021) (p. 22).

[83] B. Gulmezoglu, T. Eisenbarth, and B. Sunar. “Cache-based application detection in the cloud using machine learning”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2017 (p. 22).

[84] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar. “A Faster and More Realistic Flush+Reload Attack on AES”. In: *Constructive Side-Channel Analysis and Secure Design (COSADE)*. 2015 (p. 22).

[85] B. Gulmezoglu, A. Moghimi, T. Eisenbarth, and B. Sunar. “Fortuneteller: Predicting microarchitectural attacks via unsupervised deep learning”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2019 (p. 45).

[86] Y. Guo, X. Xin, Y. Zhang, and J. Yang. “Leaky Way: A Conflict-Based Cache Covert Channel Bypassing Set Associativity”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2022 (p. 33).

[87] Y. Guo, A. Zigerelli, Y. Zhang, and J. Yang. “Adversarial prefetch: New cross-core cache side channel attacks”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (pp. 26, 29, 31).

[88] A. Harris, S. Wei, P. Sahu, P. Kumar, T. Austin, and M. Tiwari. “Cyclone: Detecting contention-based cache information leaks through cyclic interference”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2019 (p. 45).

[89] R. Hat. *Simultaneous Multithreading in Red Hat Enterprise Linux*. `https://access.redhat.com/solutions/rhel-smt`. 2019 (p. 24).

[90] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011 (p. 9).

[91] S. Hong, M. Davinroy, Y. Kaya, S. N. Locke, I. Rackow, K. Kulda, D. Dachman-Soled, and T. Dumitraş. "Security analysis of deep neural networks operating in the presence of cache side-channel attacks". In: *arXiv:1810.03487*. 2018 (p. 22).

[92] R. Huggahalli, R. R. Iyer, and S. Tetrick. "Direct Cache Access for High Bandwidth Network I/O". In: *32st International Symposium on Computer Architecture (ISCA)*. 2005. DOI: 10.1109/ISCA.2005.23 (pp. 16, 36).

[93] R. Hund, C. Willems, and T. Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR". In: *IEEE Symposium on Security and Privacy (S&P)*. 2013 (p. 22).

[94] M. Hutter and J.-M. Schmidt. "The temperature side channel and heating fault attacks". In: *Smart Card Research and Advanced Application Conference (CARDIS)*. 2014 (p. 18).

[95] A. Ibrahim, H. Nemati, T. Schlüter, N. O. Tippenhauer, and C. Rossow. "Microarchitectural Leakage Templates and Their Application to Cache-Based Side Channels". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022 (p. 33).

[96] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar. "Cache Attacks Enable Bulk Key Recovery on the Cloud". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2016 (p. 32).

[97] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual*. https://cdrdv2-public.intel.com/671200/325462-sdm-vol-1-2abcd-3abcd.pdf. 2022 (pp. 16, 18).

[98] Intel. *Intel CAT: Improving Real-Time Performance by Utilizing Cache Allocation Technology*. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html. 2015 (p. 17).

[99] Intel. *Intel Data Direct I/O Technology Overview*. https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf. 2012 (pp. 16, 17, 36).

[100] G. Irazoqui, T. Eisenbarth, and B. Sunar. "Cross Processor Cache Attacks". In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2016 (pp. 23, 30).

[101] G. Irazoqui, T. Eisenbarth, and B. Sunar. “S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2015 (pp. 23, 28, 31, 32).

[102] G. Irazoqui, T. Eisenbarth, and B. Sunar. “Systematic reverse engineering of cache slice selection in Intel processors”. In: *Euromicro Conference on Digital System Design (DSD)*. 2015 (p. 32).

[103] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar. “Lucky 13 strikes back”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2015 (p. 22).

[104] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar. “SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks”. In: *USENIX Security Symposium*. 2019 (p. 32).

[105] J. Jancar, M. Fourné, D. D. A. Braga, M. Sabt, P. Schwabe, G. Barthe, P.-A. Fouque, and Y. Acar. ““They’re not that hard to mitigate”: What Cryptographic Library Developers Think About Timing Attacks”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (p. 43).

[106] J. Kelsey, B. Schneier, D. Wagner, and C. Hall. “Side channel cryptanalysis of product ciphers”. In: *European Symposium on Computer Security (ESORICS)*. 1998 (p. 22).

[107] T. Kim, M. Peinado, and G. Mainar-Ruiz. “StealthMem: System-Level Protection against Cache-based Side Channel Attacks in the Cloud”. In: *USENIX Security Symposium*. 2012 (p. 41).

[108] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. “Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors”. In: *ACM SIGARCH Computer Architecture News*. 2014 (p. 34).

[109] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. “DAWG: A defense against cache timing attacks in speculative execution processors”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018 (pp. 29, 41).

[110] P. Kocher. “Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems”. In: *CRYPTO*. 1996 (pp. 18, 22, 43).

[111] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. “Spectre Attacks: Exploiting Speculative Execution”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (pp. 33, 34).

[112] P. Kocher, J. Jaffe, and B. Jun. “Differential power analysis”. In: *CRYPTO*. 1999 (p. 18).

[113] A. Kogler, J. Juffinger, S. Qazi, Y. Kim, M. Lipp, N. Boichat, E. Shiu, M. Nissler, and D. Gruss. "Half-Double: Hammering From the Next Row Over". In: *USENIX Security Symposium*. 2022 (p. 34).

[114] D. Kohlbrenner and H. Shacham. "Trusted Browsers for Uncertain Times". In: *USENIX Security Symposium*. 2016 (pp. 44, 49).

[115] B. Köpf and M. Dürmuth. "A provably secure and efficient countermeasure against timing attacks". In: *IEEE Computer Security Foundations (CSF)*. 2009 (p. 44).

[116] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi. "NetCAT: Practical Cache Attacks From the Network". In: *IEEE Symposium on Security and Privacy (S&P)*. 2020 (pp. 23, 36).

[117] A. Langley. "ctgrind—checking that functions are constant time with Valgrind, 2010". In: *URL https://github.com/agl/ctgrind*. 2010 (p. 44).

[118] H. Li, N. Niu, and B. Wang. "Cache Shaping: An Effective Defense Against Cache-Based Website Fingerprinting". In: *ACM Conference on Data and Application Security and Privacy (CODASPY)*. 2022 (p. 42).

[119] P. Li, D. Gao, and M. K. Reiter. "Stopwatch: a cloud architecture for timing channel mitigation". In: *ACM Transactions on Information and System Security (TISSEC)*. 2014 (pp. 44, 45).

[120] M. Lipp, D. Gruss, and M. Schwarz. "AMD Prefetch Attacks through Power and Time". In: *USENIX Security Symposium*. 2022 (p. 22).

[121] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. "ARMageddon: Cache Attacks on Mobile Devices". In: *USENIX Security Symposium*. 2016 (pp. 22, 28, 33, 51).

[122] M. Lipp, V. Hadžić, M. Schwarz, A. Perais, C. Maurice, and D. Gruss. "Take a way: Exploring the security implications of AMD's cache way predictors". In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2020 (p. 25).

[123] M. Lipp, A. Kogler, D. Oswald, M. Schwarz, C. Easdon, C. Canella, and D. Gruss. "PLATYPUS: Software-based power side-channel attacks on x86". In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (p. 52).

[124] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. "Meltdown: Reading Kernel Memory from User Space". In: *USENIX Security Symposium*. 2018 (p. 34).

[125] C. Liu, A. Chakraborty, N. Chawla, and N. Roggel. "Frequency throttling side-channel attack". In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2022 (p. 52).

[126] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee. "Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing". In: *IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2016 (p. 41).

[127] F. Liu, H. Wu, K. Mai, and R. B. Lee. "Newcache: Secure Cache Architecture Thwarting Cache Side-Channel Attacks". In: *IEEE Micro*. 2016 (p. 43).

[128] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee. "Last-Level Cache Side-Channel Attacks Are Practical". In: *IEEE Symposium on Security and Privacy (S&P)*. 2015 (pp. 22, 23, 25, 26, 28, 30–32).

[129] X. Lou, T. Zhang, J. Jiang, and Y. Zhang. "A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography". In: *ACM Computing Surveys*. 2021 (pp. 21, 39).

[130] M. L. Luo, A Myers, and G Suh. "Stealthy Tracking of Autonomous Vehicles with Cache Side Channels". In: *USENIX Security Symposium*. 2020 (p. 22).

[131] M. Luo, W. Xiong, G. Lee, Y. Li, X. Yang, A. Zhang, Y. Tian, H.-H. S. Lee, and G. E. Suh. "AutoCAT: Reinforcement Learning for Automated Exploration of Cache-Timing Attacks". In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2023 (p. 33).

[132] A. Marshall, M. Howard, G. Bugher, B. Harden, C. Kaufman, M. Rues, and V. Bertocci. "Security best practices for developing windows azure applications". In: *Microsoft Corp*. 2010 (p. 24).

[133] R. Martin, J. Demme, and S. Sethumadhavan. "Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks". In: *International Symposium on Computer Architecture (ISCA)*. 2012 (pp. 44, 49).

[134] C. Maurice, C. Neumann, O. Heen, and A. Francillon. "C5: Cross-Cores Cache Covert Channel". In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2015 (pp. 25, 28).

[135] C. Maurice, N. L. Scouarnec, C. Neumann, O. Heen, and A. Francillon. "Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters". In: *Research in Attacks, Intrusions, and Defenses (RAID)*. 2015 (p. 32).

[136] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer. "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud". In: *Network and Distributed System Security Symposium (NDSS)*. 2017 (pp. 22, 32).

[137] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. “Spectre is here to stay: An analysis of side-channels and speculative execution”. In: *arXiv:1902.05178*. 2019 (p. 45).

[138] MDN. *performance.now() - Web APIs | MDN*. `https://developer.mozilla.org/en-US/docs/Web/API/Performance/now`. 2022 (pp. 45, 49).

[139] A. Moghimi, G. Irazoqui, and T. Eisenbarth. “CacheZoom: How SGX Amplifies the Power of Cache Attacks”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2017 (pp. 23, 33).

[140] A. Moghimi, J. Wichelmann, T. Eisenbarth, and B. Sunar. “Memjam: A false dependency attack against constant-time crypto implementations”. In: *International Journal of Parallel Programming*. 2019 (pp. 24, 26).

[141] M. Mushtaq, A. Akram, M. K. Bhatti, M. Chaudhry, V. Lapotre, and G. Gogniat. “Nights-watch: A cache-based side-channel intrusion detector using hardware performance counters”. In: *Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. 2018 (p. 45).

[142] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis. “The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2015 (pp. 22, 28, 32).

[143] D. A. Osvik, A. Shamir, and E. Tromer. “Cache Attacks and Countermeasures: The Case of AES”. In: *Cryptographers’ Track at the RSA Conference on Topics in Cryptology (CT-RSA)*. 2006 (pp. 19, 22, 23, 25, 26, 28, 31, 41).

[144] R. Paccagnella, L. Luo, and C. W. Fletcher. “Lord of the Ring(s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical”. In: *USENIX Security Symposium*. 2021 (pp. 24–26).

[145] D. Page. “Partitioned cache architecture as a side-channel defence mechanism”. In: *IACR Cryptol. ePrint Arch. 2005/280*. 2005 (p. 41).

[146] D. Page. “Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel”. In: *IACR Cryptol. ePrint Arch. 2002/169*. 2002 (p. 22).

[147] M. S. Papamarcos and J. H. Patel. “A low-overhead coherence solution for multiprocessors with private cache memories”. In: *International Symposium on Computer Architecture (ISCA)*. 1984 (p. 17).

[148] D. A. Patterson and J. L. Hennessy. *Computer organization and design ARM edition: the hardware software interface*. Morgan kaufmann, 2016 (pp. 9, 10, 13).

[149] M. Payer. “HexPADS: a platform to detect “stealth” attacks”. In: *Engineering Secure Software and Systems (ESSos)*. 2016 (p. 45).

[150] C. Percival. “Cache Missing for Fun and Profit”. In: *BSDCan*. 2005 (pp. 22, 26, 27, 41).

[151] P. Pessl, L. G. Bruinderink, and Y. Yarom. “To BLISS-B or not to be: Attacking strongSwan’s Implementation of Post-Quantum Signatures”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2017 (p. 22).

[152] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard. “DRAMA: Exploiting DRAM Addressing for Cross-cpu Attacks”. In: *USENIX Security Symposium*. 2016 (pp. 23–26).

[153] A. Purnal, M. Bognar, F. Piessens, and I. Verbauwhede. “ShowTime: Amplifying Arbitrary CPU Timing Side Channels”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2023 (p. 50).

[154] A. Purnal, L. Giner, D. Gruss, and I. Verbauwhede. “Systematic Analysis of Randomization-based Protected Cache Architectures”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (pp. 48, 49, 52, 153).

[155] A. Purnal, F. Turan, and I. Verbauwhede. “Double Trouble: Combined Heterogeneous Attacks on Non-inclusive Cache Hierarchies”. In: *USENIX Security Symposium*. 2022 (pp. 23, 33, 36, 51, 119).

[156] A. Purnal, F. Turan, and I. Verbauwhede. “Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2021 (pp. 26, 30, 31, 33, 35, 51).

[157] M. K. Qureshi. “CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping”. In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018 (pp. 43, 46–48).

[158] M. K. Qureshi. “New Attacks and Defense for Encrypted-address Cache”. In: *International Symposium on Computer Architecture (ISCA)*. 2019 (pp. 43, 46–48).

[159] H. Ragab, E. Barberis, H. Bos, and C. Giuffrida. “Rage against the machine clear: A systematic analysis of machine clears and their implications for transient execution attacks”. In: *USENIX Security Symposium*. 2021 (p. 34).

[160] H. Ragab, A. Milburn, K. Razavi, H. Bos, and C. Giuffrida. “Crosstalk: Speculative Data Leaks Across Cores are Real”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (p. 34).

[161] X. Ren, L. Moody, M. Taram, M. Jordan, D. M. Tullsen, and A. Venkat. “I see dead μops: Leaking secrets via Intel/AMD micro-op caches”. In: *International Symposium on Computer Architecture (ISCA)*. 2021 (pp. 24–26).

[162] O. Reparaz, J. Balasch, and I. Verbauwhede. “Dude, is my code constant time?” In: *Design, Automation & Test in Europe (DATE)*. 2017 (p. 44).

[163] J. Ribes-González, O. Farràs, C. Hernández, V. Kostalabros, and M. Moretó. “A Security Model for Randomization-based Protected Caches”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2022 (p. 49).

[164] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi. “SMASH: Synchronized Many-sided Rowhammer Attacks from JavaScript”. In: *USENIX Security Symposium*. 2021 (p. 34).

[165] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. “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 (p. 22).

[166] T. Rokicki, C. Maurice, M. Botvinnik, and Y. Oren. “Port Contention Goes Portable: Port Contention Side Channels in Web Browsers”. In: *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*. 2022 (p. 24).

[167] T. Rokicki, C. Maurice, and P. Laperdrix. “Sok: In Search of Lost Time: A Review of JavaScript Timers in Browsers”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021 (p. 45).

[168] T. Rokicki, C. Maurice, and M. Schwarz. “CPU Port Contention Without SMT”. In: *European Symposium on Computer Security (ESORICS)*. 2022 (p. 22).

[169] E. Ronen, R. Gillham, D. Genkin, A. Shamir, D. Wong, and Y. Yarom. “The 9 lives of Bleichenbacher’s CAT: new cache attacks on TLS implementations”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (p. 22).

[170] E. Ronen, K. G. Paterson, and A. Shamir. “Pseudo constant time implementations of TLS are only pseudo secure”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018 (p. 22).

[171] G. Saileshwar, C. W. Fletcher, and M. Qureshi. “Streamline: a fast, flushless cache covert-channel attack by enabling asynchronous collusion”. In: *ASPLOS*. 2021 (p. 26).

[172] G. Saileshwar and M. Qureshi. “MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design”. In: *USENIX Security Symposium*. 2021 (pp. 48, 49).

[173] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom. “CacheOut: Leaking data on Intel CPUs via cache evictions”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (p. 34).

[174] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss. “ZombieLoad: Cross-Privilege-Boundary Data Sampling”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2019 (p. 34).

[175] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. “Fantastic Timers and Where to Find Them: High-Resolution Microarchitectural Attacks in JavaScript”. In: *Financial Cryptography and Data Security*. 2017 (p. 45).

[176] M. Schwarz, M. Schwarzl, M. Lipp, J. Masters, and D. Gruss. “Netspectre: Read arbitrary memory over network”. In: *European Symposium on Computer Security (ESORICS)*. 2019 (pp. 23–26).

[177] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard. “Malware Guard Extension: Using SGX to Conceal Cache Attacks”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*. 2017 (p. 19).

[178] M. Schwarzl, P. Borrello, A. Kogler, K. Varda, T. Schuster, M. Schwarz, and D. Gruss. “Robust and Scalable Process Isolation Against Spectre in the Cloud”. In: *European Symposium on Computer Security (ESORICS)*. 2022 (p. 45).

[179] M. Schwarzl, E. Kraft, and D. Gruss. “Layered Binary Templating”. In: *ACNS*. 2023 (p. 33).

[180] M. Seaborn and T. Dullien. “Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges”. In: *Black Hat*. 2015 (p. 34).

[181] A. Seznec. “A Case for Two-way Skewed-associative Caches”. In: *International Symposium on Computer Architecture (ISCA)*. 1993 (p. 47).

[182] A. Shahverdi, M. Shirinov, and D. Dachman-Soled. “Database Reconstruction from Noisy Volumes: A Cache {Side-Channel} Attack on {SQLite}”. In: *USENIX Security Symposium*. 2021 (p. 22).

[183] C. E. Shannon. “A Mathematical Theory of Communication”. In: *ACM SIGMOBILE Mobile Computing and Communications Review* (2001) (p. 25).

[184] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur. “Unveiling hardware-based data prefetcher, a hidden source of information leakage”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018 (pp. 24, 26, 33).

[185] A. Shusterman, A. Agarwal, S. O’Connell, D. Genkin, Y. Oren, and Y. Yarom. “Prime+Probe 1, JavaScript 0: Overcoming Browser-based Side-Channel Defenses”. In: *USENIX Security Symposium*. 2021 (p. 28).

[186] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom. "Robust Website Fingerprinting Through the Cache Occupancy Channel". In: *USENIX Security Symposium*. 2019 (pp. 22, 25, 28, 31, 42).

[187] L. Simon, D. Chisnall, and R. Anderson. "What you get is what you C: Controlling side effects in mainstream C compilers". In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018 (p. 44).

[188] W. Song, B. Li, Z. Xue, Z. Li, W. Wang, and P. Liu. "Randomized Last-Level Caches Are Still Vulnerable to Cache Side-Channel Attacks! But We Can Fix It". In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (pp. 48, 49).

[189] W. Song and P. Liu. "Dynamically Finding Minimal Eviction Sets Can Be Quicker Than You Think for Side-Channel Attacks against the LLC". In: *Research in Attacks, Intrusions, and Defenses (RAID)*. 2019 (p. 32).

[190] F. Stolz, J. P. Thoma, P. Sasdrich, and T. Güneysu. "Risky Translations: Securing TLBs against Timing Side Channels". In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2023 (p. 49).

[191] D. Sullivan, O. Arias, T. Meade, and Y. Jin. "Microarchitectural Minefields: 4K-Aliasing Covert Channel and Multi-Tenant Detection in Iaas Clouds." In: *Network and Distributed System Security Symposium (NDSS)*. 2018 (pp. 22, 26).

[192] J. Szefer. "Survey of microarchitectural side and covert channels, attacks, and defenses". In: *Journal of Hardware and Systems Security*. 2019 (pp. 21, 39).

[193] M. Tan, J. Wan, Z. Zhou, and Z. Li. "Invisible Probe: Timing Attacks with PCIe Congestion Side-channel". In: *IEEE Symposium on Security and Privacy (S&P)*. 2021 (pp. 24, 26).

[194] Q. Tan, Z. Zeng, K. Bu, and K. Ren. "PhantomCache: Obfuscating Cache Conflicts with Localized Randomization". In: *Network and Distributed System Security Symposium (NDSS)*. 2020 (pp. 47, 48).

[195] M. Taram, X. Ren, A. Venkat, and D. Tullsen. "SecSMT: Securing SMT processors against contention-based covert channels". In: *USENIX Security Symposium*. 2022 (pp. 24, 26, 41).

[196] M. Taram, A. Venkat, and D. Tullsen. "Packet Chasing: Spying on Network Packets over a Cache Side-channel". In: *International Symposium on Computer Architecture (ISCA)*. 2020 (pp. 22, 36).

[197] A. Tatar, D. Trujillo, C. Giuffrida, and H. Bos. *TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering*. 2022 (p. 26).

[198] J. P. Thoma and T. Güneysu. “Write Me and I’ll Tell You Secrets–Write-After-Write Effects On Intel CPUs”. In: *Research in Attacks, Intrusions, and Defenses (RAID)*. 2022 (p. 43).

[199] J. P. Thoma, C. Niesler, D. Funke, G. Leander, P. Mayr, N. Pohl, L. Davi, and T. Güneysu. “ClepsydraCache–Preventing Cache Attacks with Time-Based Evictions”. In: *USENIX Security Symposium*. 2023 (pp. 48, 49).

[200] L. Trampert, C. Rossow, and M. Schwarz. “Browser-Based CPU Fingerprinting”. In: *European Symposium on Computer Security (ESORICS)*. 2022 (p. 22).

[201] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla. “Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems”. In: *Design Automation Conference (DAC)*. 2018 (pp. 43, 46).

[202] C. Trippel, D. Lustig, and M. Martonosi. “MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols”. In: *arXiv:1802.03802*. 2018 (pp. 26, 29).

[203] E. Tromer, D. A. Osvik, and A. Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. In: *Journal of Cryptology*. 2010 (p. 22).

[204] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi. “Cryptanalysis of DES implemented on computers with cache”. In: *Cryptographic Hardware and Embedded Systems (CHES)*. 2003 (p. 22).

[205] T. Unterluggauer, A. Harris, S. Constable, F. Liu, and C. Rozas. “Chameleon Cache: Approximating Fully Associative Caches with Random Replacement to Prevent Contention-Based Cache Attacks”. In: *IEEE Symposium on Secure and Private Execution Environment Design (SEED)*. 2022 (pp. 48, 49).

[206] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx. “Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution”. In: *USENIX Security Symposium*. 2018 (p. 34).

[207] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lippi, M. Minkin, D. Genkin, Y. Yarom, B. Sunar, D. Gruss, and F. Piessens. “LVI: Hijacking transient execution through microarchitectural load value injection”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2020 (p. 34).

[208] J. Van Bulck, F. Piessens, and R. Strackx. “Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2018 (p. 23).

[209] J. Van Bulck, F. Piessens, and R. Strackx. “SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control”. In: *Workshop on System Software for Trusted Execution (SysTEX)*. 2017 (p. 23).

[210] V. Van Der Veen, Y. Fratantonio, M. Lindorfer, D. Gruss, C. Maurice, G. Vigna, H. Bos, K. Razavi, and C. Giuffrida. “Drammer: Deterministic Rowhammer attacks on mobile platforms”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016 (p. 34).

[211] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi. “Malicious management unit: Why stopping cache attacks in software is harder than you think”. In: *USENIX Security Symposium*. 2018 (p. 41).

[212] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. “RIDL: Rogue In-flight Data Load”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (p. 34).

[213] B. C. Vattikonda, S. Das, and H. Shacham. “Eliminating Fine Grained Timers in Xen”. In: *ACM Workshop on Cloud Computing Security (CCSW)*. 2011 (pp. 44, 49).

[214] J. R. S. Vicarte, M. Flanders, R. Paccagnella, G. Garrett-Grossman, A. Morrison, C. W. Fletcher, and D. Kohlbrenner. “Augury: Using data memory-dependent prefetchers to leak data at rest”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (pp. 25, 51).

[215] P. Vila, A. Abel, M. Guarnieri, B. Köpf, and J. Reineke. “Flushgeist: Cache leaks from beyond the flush”. In: *arXiv:2005.13853*. 2020 (p. 41).

[216] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf. “CacheQuery: Learning replacement policies from hardware caches”. In: *ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020 (p. 33).

[217] P. Vila, B. Köpf, and J. F. Morales. “Theory and Practice of Finding Eviction Sets”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (pp. 32, 47).

[218] VMWare. *Security considerations and disallowing inter-Virtual Machine Transparent Page Sharing (2080735)*. `https://kb.vmware.com/s/article/2080735`. 2014 (p. 30).

[219] W3C. *High Resolution Time*. `https://www.w3.org/TR/hr-time-3/`. 2022 (pp. 45, 49).

[220] J. Wan, Y. Bi, Z. Zhou, and Z. Li. “MeshUp: Stateless cache side-channel attack on CPU mesh”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2022 (pp. 24, 26).

[221] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy. “Papp: Prefetcher-aware Prime and Probe Side-channel Attack”. In: *Design Automation Conference (DAC)*. 2019 (p. 33).

[222] S. Wang, P. Wang, X. Liu, D. Zhang, and D. Wu. “CacheD: Identifying Cache-Based Timing Channels in Production Software”. In: *USENIX Security Symposium*. 2017 (p. 44).

[223] Y. Wang, R. Paccagnella, E. T. He, H. Shacham, C. W. Fletcher, and D. Kohlbrenner. “Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86”. In: *USENIX Security Symposium*. 2022 (p. 52).

[224] Z. Wang and R. B. Lee. “New Cache Designs for Thwarting Software Cache-based Side Channel Attacks”. In: *International Symposium on Computer Architecture (ISCA)*. 2007 (pp. 41, 43).

[225] D. Weber, A. Ibrahim, H. Nemati, M. Schwarz, and C. Rossow. “Osiris: Automated Discovery of Microarchitectural Side Channels”. In: *USENIX Security Symposium*. 2021 (p. 33).

[226] WebKit. *What Spectre and Meltdown Mean For WebKit*. `https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/`. 2018 (pp. 45, 49).

[227] Z. Weissman, T. Tiemann, D. Moghimi, E. Custodio, T. Eisenbarth, and B. Sunar. “JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems*. 2020 (pp. 23, 36).

[228] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard. “SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization”. In: *USENIX Security Symposium*. 2019 (pp. 43, 46, 48).

[229] J. Wichelmann, A. Moghimi, T. Eisenbarth, and B. Sunar. “MicroWalk: A Framework for Finding Side Channels in Binaries”. In: *Annual Computer Security Applications Conference (ACSAC)*. 2018 (p. 44).

[230] H. Winderix, J. T. Mühlberg, and F. Piessens. “Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks”. In: *IEEE European Symposium on Security and Privacy (EuroS&P)*. 2021 (p. 44).

[231] W. Xiong and J. Szefer. “Leaking Information Through Cache LRU States”. In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2020 (pp. 25, 26, 29, 30).

[232] W. Xiong and J. Szefer. “Survey of transient execution attacks and their mitigations”. In: *ACM Computing Surveys*. 2021 (p. 21).

[233] Y. Xu, M. Bailey, F. Jahanian, K. Joshi, M. Hiltunen, and R. Schlichting. "An exploration of L2 cache covert channels in virtualized environments". In: *ACM Workshop on Cloud Computing Security (CCSW)*. 2011 (p. 23).

[234] M. Yan, C. Fletcher, and J. Torrellas. "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures". In: *USENIX Security Symposium*. 2020 (p. 22).

[235] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas. "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks". In: *International Symposium on Computer Architecture (ISCA)*. 2017 (p. 42).

[236] M. Yan, Y. Shalabi, and J. Torrellas. "ReplayConfusion: Detecting cache-based covert channel attacks using record and replay". In: *IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2016 (p. 45).

[237] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell, and J. Torrellas. "Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World". In: *IEEE Symposium on Security and Privacy (S&P)*. 2019 (pp. 26, 28, 30, 33, 36, 51).

[238] M. Yan, J.-Y. Wen, C. W. Fletcher, and J. Torrellas. "SecDir: a secure directory to defeat directory side-channel attacks". In: *International Symposium on Computer Architecture (ISCA)*. 2019 (pp. 32, 36, 42).

[239] F. Yao, M. Doroslovacki, and G. Venkataramani. "Are Coherence Protocol States Vulnerable to Information Leakage?" In: *IEEE Symposium on High Performance Computer Architecture (HPCA)*. 2018 (pp. 26, 29).

[240] Y. Yarom and N. Benger. "Recovering OpenSSL ECDSA Nonces Using the FLUSH+ RELOAD Cache Side-channel Attack." In: *IACR Cryptol. ePrint Arch. 2014/140*. 2014 (p. 30).

[241] Y. Yarom and K. Falkner. "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack". In: *USENIX Security Symposium*. 2014 (pp. 22, 23, 26, 27, 30, 31).

[242] Y. Yarom, Q. Ge, F. Liu, R. B. Lee, and G. Heiser. "Mapping the Intel last-level cache". In: *IACR Cryptol. ePrint Arch. 2015/905*. 2015 (p. 32).

[243] Y. Yarom, D. Genkin, and N. Heninger. "CacheBleed: a Timing Attack on OpenSSL Constant-time RSA". In: *Journal of Cryptographic Engineering*. 2017 (pp. 25, 26, 33).

[244] J. Yu, L. Hsiung, M. El Hajj, and C. W. Fletcher. "Data oblivious ISA extensions for side channel-resistant and high performance computing". In: *Network and Distributed System Security Symposium (NDSS)*. 2018 (p. 44).

[245] M. Zaheri, Y. Oren, and R. Curtmola. “Targeted Deanonymization via the Cache Side Channel: Attacks and Defenses”. In: *USENIX Security Symposium*. 2022 (p. 22).

[246] R. Zhang, X. Su, J. Wang, C. Wang, W. Liu, and R. W. Lau. “On mitigating the risk of cross-VM covert channels in a public cloud”. In: *IEEE Transactions on Parallel and Distributed Systems* (2014) (p. 42).

[247] R. Zhang, T. Kim, D. Weber, and M. Schwarz. “(M) WAIT for It: Bridging the Gap between Microarchitectural and Architectural Side Channels”. In: *USENIX Security*. 2023 (p. 51).

[248] T. Zhang, Y. Zhang, and R. B. Lee. “Cloudradar: A real-time side-channel attack detection system in clouds”. In: *Research in Attacks, Intrusions, and Defenses (RAID)*. 2016 (p. 45).

[249] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter. “Homealone: Co-residency detection in the cloud via side-channel analysis”. In: *IEEE Symposium on Security and Privacy (S&P)*. 2011 (p. 22).

[250] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart. “Cross-VM Side Channels and their use to Extract Private Keys”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2012 (pp. 22, 28).

[251] Y. Zhang and M. K. Reiter. “Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2013 (p. 41).

[252] L. Zhao, R. R. Iyer, S. Makineni, D. Newell, and L. Cheng. “NCID: a Non-inclusive cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies”. In: *Conference on Computing Frontiers*. 2010 (p. 16).

[253] Z. N. Zhao, A. Morrison, C. W. Fletcher, and J. Torrellas. “Binoculars: Contention-Based Side-Channel Attacks Exploiting the Page Walker”. In: *USENIX Security Symposium*. 2022 (p. 26).

[254] Z. Zhou, M. K. Reiter, and Y. Zhang. “A software approach to defeating side channels in last-level caches”. In: *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. 2016 (p. 41).

Part II

Publications

List of publications

International Conferences and Workshops with Proceedings

1. Jesse De Meulemeester, **Antoon Purnal**, Lennert Wouters, Arthur Beckers and Ingrid Verbauwhede, "SpectrEM: Exploiting Electromagnetic Emanations During Transient Execution". ***USENIX Security Symposium***, 2023

2. **Antoon Purnal**, Marton Bognar, Frank Piessens and Ingrid Verbauwhede, "ShowTime: Amplifying Arbitrary CPU Timing Side Channels". *ACM SIGSAC Asia Conference on Computer and Communications Security (**AsiaCCS**)*, 2023

3. Lukas Giner, Stefan Steinegger, **Antoon Purnal**, Maria Eichlseder, Thomas Unterluggauer, Stefan Mangard, and Daniel Gruss, "Scatter and Split Securely: Defeating Cache Contention and Occupancy Attacks". *IEEE Symposium on Security and Privacy (**S&P**)*, 2023

4. **Antoon Purnal**, Furkan Turan, and Ingrid Verbauwhede, "Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies". ***USENIX Security Symposium***, 2022

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

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

7. Arne Deprez, Elena Andreeva, Jose Maria Bermudo Mera, Angshuman Karmakar, and **Antoon Purnal**, "Optimized Software Implementations for the Lightweight Encryption Scheme ForkAE". *International Conference on Smart Card Research and Advanced Applications (**CARDIS**)*, 2020

8. Elena Andreeva, Virginie Lallemand, **Antoon Purnal**, Reza Reyhanitabar, Arnab Roy, and Damian Vizár, "Forkcipher: a New Primitive for Authenticated Encryption of Very Short Messages". *Advances in Cryptology (**ASIACRYPT**)*, 2019

9. **Antoon Purnal**, Victor Arribas, and Lauren De Meyer, "Trade-offs in Protecting Keccak Against Combined Side-Channel and Fault Attacks". *International Workshop on Constructive Side-Channel Analysis and Secure Design (**COSADE**)*, 2019

Technical Reports

10. Elena Andreeva, Virginie Lallemand, **Antoon Purnal**, Reza Reyhanitabar, Arnab Roy, and Damian Vizár, "ForkAE v1". *Submission to the NIST Lightweight Cryptography Standardization*, 2019

11. **Antoon Purnal** and Ingrid Verbauwhede, "Advanced profiling for probabilistic Prime+ Probe attacks and covert channels in ScatterCache". *arXiv preprint arXiv:1908.03383*, 2019

12. **Antoon Purnal**, Elena Andreeva, Arnab Roy, and Damian Vizár, "What the Fork: Implementation Aspects of a Forkcipher". *NIST Second Lightweight Cryptography Workshop*, 2019

Unpublished Manuscripts

13. Elena Andreeva, Benoit Cogliati, Virginie Lallemand, Marine Minier, **Antoon Purnal**, and Arnab Roy, "Masked Iterate-Fork-Iterate: A New Design Paradigm for Tweakable Expanding Pseudorandom Function". *Under submission*, 2023

Chapter 6

Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks

Publication data

Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede, "Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks". *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021, pp. 2906–2920.

Contributions

Principal author.

Prime+Scope: Overcoming the Observer Effect for High-Precision Cache Contention Attacks

Antoon Purnal, Furkan Turan and Ingrid Verbauwhede

imec-COSIC, KU Leuven

Abstract. Modern processors expose software to information leakage through shared microarchitectural state. One of the most severe leakage channels is cache contention, exploited by attacks referred to as Prime+Probe, which can infer fine-grained memory access patterns while placing only limited assumptions on attacker capabilities.
In this work, we strengthen the cache contention channel with a near-optimal time resolution. We propose Prime+Scope, a cross-core cache contention attack that performs back-to-back cache contention measurements that access only a single cache line. It offers a time resolution of around 70 cycles (25ns), while maintaining the wide applicability of Prime+Probe. To enable such a rapid measurement, we rely on the deterministic nature of modern replacement policies and their (non-)interaction across cache levels. We provide a methodology to, essentially, prepare multiple cache levels simultaneously, and apply it to Intel processors with both inclusive and non-inclusive cache hierarchies. We characterize the resolution of Prime+Scope, and confirm it with a cross-core covert channel (capacity up to 3.5 Mbps, no shared memory) and an improved attack on AES T-tables. Finally, we use the properties underlying Prime+Scope to bootstrap the construction of the eviction sets needed for the attack. The resulting routine outperforms state-of-the-art techniques by two orders of magnitude.
Ultimately, our work shows that interference through cache contention can provide richer temporal precision than state-of-the-art attacks that directly interact with monitored memory addresses.

1 Introduction

Modern processors comprise many hardware components that, transparently to the programmer, enhance average software performance. Several such components, with the cache hierarchy as the leading example, may be shared across

software-defined security boundaries. Through their access patterns, programs may unwillingly encode secret information into shared cache state, which can be extracted by a co-located adversary using a timing side channel. In particular, she first prepares the cache state and afterwards measures it to infer the changes produced by other programs.

Several techniques have been proposed to prepare and measure the cache state. While some can only monitor accesses to shared memory between attacker and victim, or require specific processor features, other techniques have no such prerequisites and are purely based on contention for cache resources. By targeting core-shared cache levels, such as the last-level cache (LLC) [1, 2], or the coherence directory (CD) [3], an attacker can measure *cache contention* for victim programs running on other processor cores.

Known mostly as PRIME+PROBE, cache contention attacks are widely applicable. The contention channel leaks information across virtual machine boundaries [4, 1, 2], to and from sandboxed code (e.g., in the browser [5, 6]), or even over the network [7]. It has been used to extract sensitive information of various kinds, such as cryptographic keys [8], user input [4, 7], kernel information [9] and browsing behavior [6]. It also enables establishing covert channels [1, 10, 11] and, recently, transient execution attacks [12, 13].

An important metric of cache attack techniques is their *temporal resolution*, i.e., the precision with which they can localize victim memory accesses in the time domain. The finer the resolution of the attack, the greater the visibility into data accesses and control flow of victim applications. This is of special importance in the general setting, where the attacker monitors victim behavior asynchronously and victim accesses may occur at any given time.

In case of insufficient time precision, prior works slow down the victim application (e.g., [14, 15, 16, 17]), or interrupt it heavily (e.g., [18, 19]), to amplify secret-dependent time differences. Instead of such performance degradation of the victim, which may not always be possible, this work pursues the opposite direction and investigates whether the time precision of cache attacks can be improved (and optimized). In particular, we identify two key challenges to enhance the time resolution of state-of-the-art cache contention attacks.

First, the main challenge to improve the precision of cache attacks, in general, is the *observer effect*, i.e., the phenomenon where the act of measuring a system affects its state. Many techniques suffer from it, often requiring the state change of every measurement to be undone before the next one can be performed. To minimize the influence of this effect, cache attacks are often discretized along the time axis in *windows* of fixed duration (e.g., [20, 21, 1, 14, 3]). However, this places fundamental limits on their time resolution.

Second, the cache contention channel, in particular, faces another bottleneck. For PRIME+PROBE, each probe accesses as many cache lines as the associativity of the target structure, e.g., at least 11 ways for core-shared caches on modern

Intel CPUs. Therefore, the time resolution is structurally bounded by the time it takes to access all these lines, even if the observer effect were to be overcome.

In this paper, we seek to optimize the resolution of Prime+Probe-style attacks. To this end, we ask the following main questions:

Is it possible to bypass the observer effect? Can contention be inferred by repeatedly measuring the access latency of a single cache line?

In this work, we make the surprising observation that the cache contention channel can have a higher time resolution than techniques that access monitored addresses directly. We propose Prime+Scope, a high-precision cross-core cache contention attack, whose measurement is both *repeatable* (i.e., the cache state does not need to be reinstated after every measurement), and essentially *optimal* (i.e., it performs a single memory access). Prime+Scope can monitor events asynchronously with a precision in the order of 25ns, significantly outperforming comparable techniques. At the same time, Prime+Scope inherits the general applicability of Prime+Probe.

Prime+Scope prepares the cache more precisely than traditional cache contention attacks. We obtain fast and effective Prime patterns using both an automated and a handcrafted methodology (resp. for inclusive and non-inclusive Intel LLCs). In the end, we find Prime+Scope to apply to all tested Intel CPUs of the last decade.

To confirm the superior time precision of Prime+Scope, we perform a quantitative comparison with state-of-the-art techniques. We also implement a cross-core covert channel on a last-level cache (LLC) and a coherence directory (CD). Symbols are encoded temporally in slots of no more than 80-120 processor cycles. The LLC/CD channels reach a capacity of 3.5 Mbps and 3.1 Mbps, respectively.

We evaluate Prime+Scope on a known-vulnerable AES implementation. With its fine temporal precision, it can extract the key material with 5x-25x fewer encryptions than Prime+Probe.

Finally, we bootstrap our newly discovered primitive to create a straightforward, portable and linear-time eviction set construction routine, which outperforms previous techniques by 100-600x.

Summarized, this paper makes the following main contributions:

- We present Prime+Scope, a generic cross-core cache contention primitive with near-optimal temporal resolution.
- To prepare the cache for continuous measurement, we propose PrimeTime, a methodology to find efficient Prime patterns.
- We evaluate Prime+Scope using micro-benchmarks, a high-capacity covert channel, and a high-precision attack on AES.
- Using the principles underlying Prime+Scope, we present fast and simple routines to construct LLC/CD eviction sets.

We have disclosed our findings to Intel. To facilitate reproduction of our research, artifacts are made available at

`https://www.github.com/KULeuven-COSIC/PRIME-SCOPE`

This article is organized as follows. Section 2 provides the necessary background. Section 3 explores the conditions for back-to-back cache measurements. Section 4 exposes PRIME+SCOPE, our main result. Section 5 covers the efficient preparation of the cache state, and Section 6 evaluates PRIME+SCOPE for micro-benchmarks and concrete examples. Section 7 positions our findings, Section 8 discusses limitations and countermeasures, and Section 9 concludes.

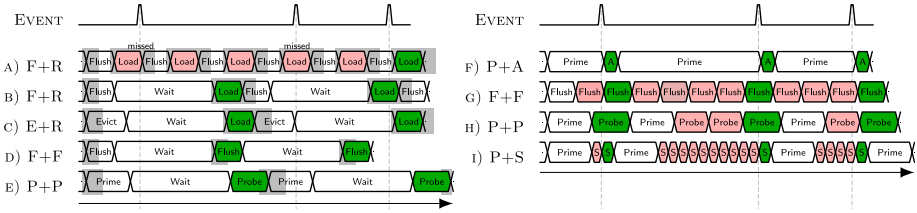
2 Preliminaries

2.1 Caches

To overcome the comparatively high latency of memory lookups, caches are buffers that keep soon-to-be used data close to the CPU. Caches operate on fixed-size (e.g., 64 bytes) memory blocks called *cache lines*, and are typically set-associative, referring to their organization along *sets* and *ways*. Cache lines are mapped to sets based on their memory address, and addresses mapping to the same set are called *congruent*. The maximal number of congruent lines that can reside in the cache at any given time is determined by the number of ways W, also referred to as the *associativity*.

When caching a new line exceeds the associativity, one line in the set is *evicted*; in this paper, we refer to that line as the *eviction candidate (EVC)*. The EVC is determined by the *replacement policy*, which is implemented by a (complex) state machine at the set-level, for which the state transitions depend on the accesses to the set.

Contemporary Intel processors feature a three-level cache hierarchy, where the access latency increases along with the distance from the CPU. When a CPU core references a memory address, the cache line containing this address is retrieved from the closest cache level that has a valid copy. The first two levels (L1 and L2) are organized separately for every core, while the last-level cache (L3, or LLC) is shared among all the cores. The majority of Intel processors have *inclusive* LLCs, meaning that cache lines present in the L1 and L2 caches must also be present in the LLC. However, recent Intel servers feature higher core counts and larger private caches, prompting the adoption of *non-inclusive* LLCs. In such cache hierarchies, the LLC may or may not contain lines that are present in L1 or L2, reducing the storage overhead due to inclusion.



(i) Windowed techniques cope with blind spots by waiting between preparation and measurement, lowering their time precision

(ii) Windowless techniques measure the cache state continuously without waiting, and revert the state only after detecting an event

Figure 1: Cache Manipulation techniques in the windowed vs. windowless paradigms.

2.2 Cache Side-Channel Attacks

Over the last years, several techniques have been proposed to infer memory access patterns by other programs through observation of shared cache state. The two most prominent attack classes are represented, respectively, by Flush+Reload and Prime+Probe. In this overview, and in the paper, we focus on attacks across cores.

Flush+Reload-style techniques. If the memory address to be monitored exists in memory shared with the attacker, she is able to access it directly, allowing to infer memory activity by other processes at cache-line granularity. The representative technique for this attack class is Flush+Reload [18, 20], which removes (flushes) the cache line containing a target address from the cache, and later determines if the victim accessed it by its reload time. If cache flushing is not available, Evict+Reload [22] replaces it with eviction.

Provided that the time until completion of the `clflush` instruction depends on the presence of the target in the cache, it can be used to both prepare and measure the cache state. This technique, referred to as Flush+Flush [23], has a higher time resolution than Flush+Reload. However, the more subtle time-dependence of cache flushes results in a comparatively low cross-core accuracy.

The main drawback of Flush+Reload-style attacks is their structural dependence on shared memory with a victim, which is harder to obtain for an attacker than only co-location. Moreover, a cache flush instruction may also not be available in restricted contexts, e.g., in the browser [24, 23], or generally for unprivileged processes [25].

Prime+Probe-style techniques. Cache contention attacks, often synonymously referred to as PRIME+PROBE attacks, prime a full cache set and measure the time it takes. Activity in this cache set by other processes will evict one or more of the attacker's lines, which is reflected in a higher latency to complete the prime.

For a cross-core attack, an adversary generally targets an inclusive structure shared with the victim. The inclusive property guarantees the eviction of congruent addresses from the victim's private caches, ensuring that future victim accesses to them indeed generate contention on the measured set. In cache hierarchies with an inclusive last-level cache (LLC), this requirement is readily obtained [1, 2]. For non-inclusive Intel caches, a suitable structure has been found in the coherence directory (CD) [3], which keeps track of lines in all the L2 caches. It is organized in sets, like the LLC, with the same function to index memory addresses into sets and slices. Instead of priming the LLC, the attacker primes the CD, evicting the target address from the CD and, due to its inclusion property, also from the victim's private L1/L2 caches. When lines are evicted from the CD, they are moved to the LLC [3].

To measure contention on the LLC (or CD), an attacker needs to obtain memory addresses that are mapped to the same set. In the presence of unknown physical address bits or cache slices, these so-called *eviction sets* need to be obtained at runtime [1, 26, 3].

As a variant of PRIME+PROBE, PRIME+ABORT [27] is a contention-*informed* attack using Intel TSX. As TSX transactions are aborted upon eviction of certain lines from the LLC, it is amenable to measure LLC contention, as attack [27] or defense [28]. Intel TSX may not be exposed to an attacker (e.g., from the browser), may be disabled for security reasons [29], or may not be available at all.

Cache contention attacks offer a spatial granularity of sets, which is inferior to the cache-line granularity of shared-memory attacks. However, due to the large number of sets in modern LLCs/CDs, the spatial information encoded in cache contention is still quite large.

3 Cache Manipulation Paradigms

Assume an attacker wants to spy on a cross-core event, i.e., one or more memory accesses by a victim program running on another CPU core. All cache attack techniques first *prepare* the cache state, and then *measure* it to infer the presence or absence of an event.

This section first revisits why some techniques need to allocate a waiting period between preparation and measurement, essentially partitioning the time axis into *windows*. Then, it examines the conditions under which a windowless paradigm can be adopted.

3.1 Windowed Paradigm

Blind Spots. To see why cache attacks are often organized in discrete time windows, consider the traces in Figure 1i. The first Flush+Reload trace (Figure 1i-A) continuously flushes and reloads a target. Such an application of Flush+Reload fails to detect many events. In particular, events that occur during the period slightly before the Reload, until the Flush has evicted the target, remain undetected [20, 14]. We refer to such a period as a *blind spot*.

To reduce the effect of blind spots on the detection rate, a wait stage may be inserted, i.e., a predetermined idle period between preparation and measurement. As in Figure 1i-B, such an organization detects the events that occur during the wait stage.

Other techniques, like Evict+Reload, Flush+Flush, and Prime+Probe, can also be instantiated like this (cf. Figure 1i-C-D-E). Evict+Reload behaves similarly to Flush+Reload, but has a larger blind spot as cache eviction is slower than flushing. In Section 3.2, we will see which techniques can be used without blind spots.

Resolution. The temporal resolution of windowed techniques is limited by the combined duration of the preparation, wait and measurement stages. In particular, the waiting period marks a trade-off between the accuracy and resolution of the attack. The larger the blind spot, the lower the resolution for the same detection accuracy.

Despite this limitation, windowed techniques such as Flush+Reload can be very powerful in practice. For instance, blind spots can be bypassed when the attacker controls the timing of the event (e.g., by synchronizing [8, 30, 31, 32] or interleaving [18, 33] with the victim). The limitation is also attenuated for infrequent events (e.g., user behavior [22, 5]), or when lower detection rates are tolerable (e.g., to profile a binary [22] or capture traces [34]). The miss probability may also be reduced by targeting events that reference the same line multiple times, e.g., loops [20] or function calls [35].

3.2 Windowless Paradigm

To understand how some techniques can increase the time resolution by avoiding windows [27, 36], we identify the two sources of blind spots. Both sources are an expression of the observer effect, i.e., the attacker perturbs the cache state by measuring it.

#1: Non-preserving. We refer to a cache measurement as *preserving* when, in the absence of an event, the relevant cache state before and after the measurement is equivalent. If the measurement is not preserving, monitoring cannot

continue without undoing the changes caused by the measurement. Hence, non-preserving measurements trigger a repeated preparation phase, which naturally introduces a period of time in which victim events are missed (cf. Figure 1i). For instance, the RELOAD measurement in FLUSH+RELOAD is non-preserving, so it needs to be followed by a FLUSH, and events occurring at the beginning of the FLUSH are missed [20, 14]).

#2: Non-concurrent. We refer to a cache measurement as *concurrent* when it detects events that temporally overlap with it. Depending on the degree of overlap between event and measurement, an event coinciding with measurement j may be detected during measurement j or $j+1$, but will not be missed, roughly speaking. For instance, the RELOAD in FLUSH+RELOAD is non-concurrent, as events occurring right before or during the RELOAD are missed [20].

Non-preserving measurements cannot be concurrent, as the necessary preparation phase erases all relevant state changes, rendering them unobserved. Non-concurrent measurements, even if they are preserving, are a source of blind spots, resulting in the need for a waiting interval to obtain the desired detection accuracy. It should also be noted that measurements can be concurrent on one processor and non-concurrent on another (e.g., FLUSH, cf. Section 6.1).

Going Windowless. Cache measurements that are preserving and concurrent can be performed back-to-back while maintaining their detection accuracy. As a result, they enable a windowless paradigm that maximizes their time resolution. This paradigm first prepares the relevant cache state, and then continuously measures it until an event is observed. Only upon detection of an event, the preparation phase is repeated to continue monitoring for events.

In PRIME+ABORT [27], the cache measurement occurs implicitly, through the TSX abort. Hence, it is preserving and concurrent, and has a natural windowless instantiation (cf. Figure 1ii-F). Although it is advertised as a distinguishing feature for PRIME+ABORT, other cache attack techniques can also avoid intermittent wait stages.

Van Bulck et al. [36] demonstrate a windowless FLUSH+FLUSH [23] (cf. Figure 1ii-G). On some platforms, FLUSH measurements are non-concurrent (cf. Section 6.1). If the detection accuracy is unsatisfactory, one can resort to a windowed instantiation, as in Figure 1i-D.

We note that even PRIME+PROBE can be windowless [37] (cf. Figure 1ii-H), provided that the PROBE measurement does not access more congruent addresses than the associativity W of the target structure. Indeed, it is preserving (if all W lines are simultaneously in the target structure, they will still be after a repeated access) and concurrent (an event will cause a miss on at least one of the attacker's lines at some point, regardless of overlap.)

Time Resolution. The advantage of windowless techniques is that their time precision is only fundamentally determined by the throughput of the measurement phase. Therefore, the duration of the preparation phase is of secondary importance for the resolution, as it only needs to be performed right after detecting an event.

3.3 This Work: Prime+Scope

This work sets out to optimize the resolution of cache-timing attacks, while maintaining only the basic requirements of cache contention to ensure that the technique is future-proof and suitable for restricted environments. In particular, we do not rely on shared memory between attacker and victim, or special ISA or processor features (e.g., `clflush` or Intel TSX). We achieve this by organizing the cache state such that the contention measurement is repeatable, i.e., it is preserving and concurrent, and optimally short, i.e., it consists of a single cache access. We call this technique Prime+Scope, and depict it in Figure 1ii-I. In the following section, we outline its core principles and instantiate it for different cache hierarchies.

4 Prime+Scope

4.1 Threat Model

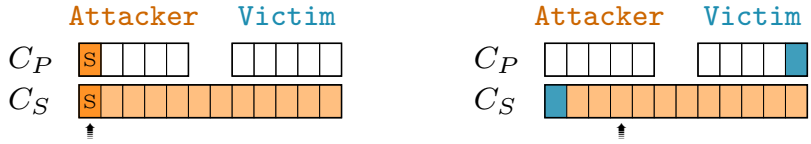
The adversary assumed in this work is able to execute unprivileged code on the same physical processor as a victim program. The attacker code need not be executed on the same core as the victim code, and the attacker is not assumed to be able to interrupt or otherwise control the victim program. Furthermore, we do not assume that attacker and victim have a shared memory region.

4.2 General Description

As described in Section 2.1, modern cache hierarchies comprise different levels. In what follows, C_S denotes the shared and inclusive cache structure in which contention is to be measured, and C_P denotes one of the attacker's private caches (e.g., the L1 cache).

Compared to existing cache contention channels, Prime+Scope has two additional core requirements:

① The eviction candidate in the shared and inclusive target structure (C_S) can be accurately predicted.
② Reads served from a lower-level cache (C_P) do not influence the replacement state of the target structure (C_S).



(i) PRIME fixes S as the EVC in C_S, which remains the case for following SCOPE operations.

(ii) The victim access evicts S (=EVC) from C_S and C_P, resulting in high access latency for S.

Figure 2: Working principle of PRIME+SCOPE

① **Eviction Candidate.** When a new line is to be installed into a cache set, among all available lines (ways) in the set, a chosen one is replaced with the new line. In this paper, we call that chosen line the Eviction Candidate (EVC). The candidate is determined by the cache replacement policy, which is implemented at the cache-set level as a state machine. For instance, the eviction candidate for the LRU policy is the cache line that has least recently been used. Though modern processors implement more sophisticated replacement policies, they are often deterministic [38, 39, 40]. Therefore, specific access patterns can mold the replacement policy machinery into a state where a chosen cache line is the eviction candidate [32, 40].

Awareness of the EVC in C_S permits to observe contention by only measuring EVC latency, as a new cache line fill evicts the EVC by definition. However, the attacker suffers from the observer effect, i.e., measuring the access latency of the EVC may change it to another line. To make the measurement preserving, PRIME+SCOPE relies on another common property of multi-level caches.

② **Low-Level Reads.** Prior work observed that the replacement state of inclusive Intel LLCs only depends on memory requests served by the LLC, not those served by the lower-level caches [39, 41, 40]. Instead of bypassing this filtering property (e.g., by enforcing L1/L2 misses), our work explicitly relies on it to make the cache measurement preserving, and thus, overcome the observer effect.

Prime+Scope. Based on these two key ingredients, we propose PRIME+SCOPE as a windowless technique to monitor cache contention. It allows an attacker to monitor victim accesses to a predetermined target address in two steps. The PRIME step serves two purposes, as in Figure 2i. First, it evicts the target from the C_S using an eviction set. Second, it performs the eviction with a specific access pattern that fixes a chosen line from the eviction set, denoted as scope line (S), as the EVC in C_S (the shared and inclusive high-level structure), *while maintaining its presence in C_P (the lower-level caches).* Afterwards, the preserving and concurrent SCOPE step continuously fetches S from C_P, and

measures the access latency. As it overcomes the observer effect, the relevant cache state remains intact both in C_P and C_S after each Scope.

The described cache state is destroyed when the victim accesses the target address. When this happens, as in Figure 2ii, the newly-allocated target replaces S, as it is the EVC. Because C_S is inclusive of C_P, the copy of S is also evicted from C_P. The next Scope will detect this event through a high access latency to S.

4.3 Instantiation

Cache Hierarchy. For processors with inclusive last-level caches (LLC), such as the majority of Intel's desktop CPUs or server CPUs until 2018, the core-shared and inclusive LLC itself can instantiate C_S, and the core-private L1 caches can instantiate C_P. Most Intel servers since 2018 have non-inclusive LLCs. For such processors, the CD is shared and inclusive [3], and can hence instantiate C_S.

Measurement: Scope. On all tested platforms (cf. Section 5), we found that requests served by C_P indeed preserve the EVC of C_S. The Scope continuously measures the access latency of the scope line S (=EVC), and terminates as soon as the access time exceeds a predetermined threshold, indicating the occurrence of the event. As in Figure 3, Prime+Scope measurements need to detect whether one cache line is served from L1, vs. from RAM (inclusive) or LLC (non-inclusive). In comparison, Prime+Probe measurements must distinguish "W lines in C_S" from "less than W lines in C_S".

Preparation: Prime. Prime+Scope is predicated on the existence and knowledge of a memory access pattern that prepares the cache state for repeated, single-access measurements. Concretely, we are looking for Prime patterns, consisting of accesses to W different addresses that satisfy the following requirements simultaneously:

$\mathbf{R}_A$. have high eviction rate ($> 99.5\%$)
$\mathbf{R}_B$. install a specific line S as the eviction candidate in C_S
$\mathbf{R}_C$. keep the line S in C_P

Requirement $\mathbf{R}_A$ is a traditional requirement for cache contention attacks; otherwise, the victim access might not evict any of the attacker's lines. Requirements $\mathbf{R}_B$ and $\mathbf{R}_C$ are unique to Prime+Scope, so we cannot rely on patterns established in prior work. In particular, we identify the following challenges.

Challenge-LLC: Keeping the EVC in L1. Taken at face value, requirements $\mathbf{R}_B$ and $\mathbf{R}_C$ are contradictory. Assume we want to install line S as the EVC in C_S. While requirement $\mathbf{R}_B$ suggests to access S *less frequently* than the other

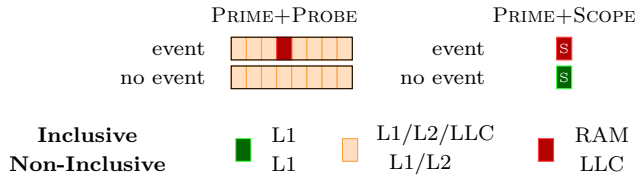


Figure 3: PRIME+PROBE monitors a full set in LLC (incl.) / CD (non-incl.) and detects eviction to RAM/LLC. PRIME+SCOPE monitors one line in L1, and detects eviction to RAM/LLC.

lines in the eviction set, to ensure it becomes the EVC in C_S, requirement $\mathbf{R}_C$ suggests to access S *more frequently* than the others, to ensure it is kept in C_P.

Challenge-CD: Controlling the EVC Prior work [3] observed that traditional eviction strategies do not perform well on the CD ($\mathbf{R}_A$). This poses a challenge for PRIME+SCOPE, as controlling the EVC ($\mathbf{R}_B$) is strictly harder than only evicting the target.

5 Finding Efficient Prime Patterns

This section covers the preparation of the cache state such that subsequent measurements can be performed with a single repeatable cache access. Although the PRIME duration has limited impact on the time precision (cf. Section 3.2), we opt to implement the PRIME step with fast and accurate access patterns. To find them, we propose PRIMETIME, an automated gray-box search methodology.

To understand the nomenclature of PRIME patterns, and how PRIMETIME finds them, an example pattern is shown in Figure 4 together with its translation into a code snippet. It encodes the access sequence of lines in the eviction set, along with the stride (gap) between indices, and the amount of repetitions. This snippet uses the first line of the eviction set (`evset[0]`) as the scope line S, which becomes the EVC after a successful PRIME with the snippet.

5.1 Last-Level Cache (LLC)

Main Idea. The key idea of our solution to **Challenge-LLC** relies on property ②. Assume the scope line S to be the first line in the set (line 0). Then, the PRIME patterns comprise accesses to W congruent lines, like other prime strategies, but accesses to lines 1 to $W-1$ are interleaved with accesses to S. Due to its frequent usage, S is always served from L1, so it keeps its insertion age in the LLC. The other lines, in contrast, evict each other from L1, and when they are read from the LLC, their age decreases, making them progressively

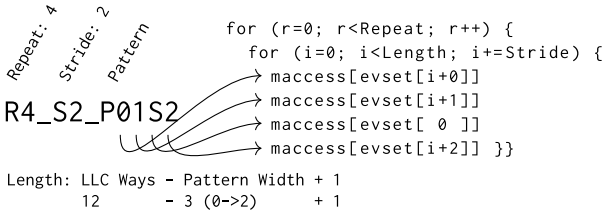


Figure 4: Translation of PRIME access patterns into code snippets. After a successful prime, the scope line `S=evset[0]` is the EVC in the LLC, while remaining present in L1.

younger. As soon as all other lines become younger than S, the latter is the EVC in the LLC without ever leaving the L1 cache.

Prime Properties. For each candidate PRIME pattern, we assess the *eviction rate (EVr)*, i.e., the fraction of successful evictions of the target line. More importantly, we also record the *eviction candidate rate (EVCr)*, i.e., the fraction of attempts where the target line is evicted, *and* the line that will be evicted next is the intended S, and it is still in L1. Finally, we also record the *duration*, i.e., the number of cycles to complete the accesses indicated by the pattern.

It is clear that $EVCr \leq EVr$. While the EVr is the success rate of preparing the cache set for PRIME+PROBE, the EVCr is the success rate of preparing S for a continuous SCOPE. Understandably, prior efficient patterns for PRIME+PROBE typically have a low EVCr, because these patterns are oblivious to the EVC. Hence, good PRIME patterns for PRIME+SCOPE differ from those in prior work.

Methodology of PrimeTime. The high-level description of PRIMETIME is shown in Algorithm 1. It starts with known access pattern templates, e.g., [42], and mutates them according to given directives. Mutations consist of repeated access to certain (sub-)patterns, permuting access orders, or interleaving accesses to S.

To limit execution time, PRIMETIME tests patterns in stages, gradually becoming more restrictive on the patterns that pass to the next stage, both in EVCr and cycle count. In the first stage, we test each pattern with 10 000 repetitions, with loosely defined success criteria. Later stages perform up to a million repetitions, while filtering for the best-performing patterns. A run for a specific microarchitecture takes approximately one hour under our configuration, but this can be scaled in either direction (i.e., speed vs. accuracy). Furthermore, PRIMETIME can be extended to cover a larger search space.

Algorithm 1 PrimeTime

Output: PRIME patterns with high EVCr and low cycle count

```
Patterns ← GenerateAccessPatterns()
Patterns ← Mutate(Patterns, with Repeated Access)
Patterns ← Mutate(Patterns, with interleaved S Accesses)
...
Measurements ← TestEviction(Patterns, 10 000 times)
Patterns ← Filter(Patterns, Measurements, Highest EVCr 7 000 )
Patterns ← Filter(Patterns, Measurements, Fastest 5 000 )
...
Measurements ← TestEviction(Patterns, 1 000 000 times)
Patterns ← Filter(Patterns, Measurements, Highest EVCr 150 )
Patterns ← Filter(Patterns, Measurements, Fastest 100 )
return Patterns
```

PrimeTime on Various Processor Generations. As shown in Table 1, PRIMETIME is able to construct effective PRIME access patterns on all tested generations of Intel CPUs, though their duration differs across microarchitectures. For each CPU, we indicate the target cache and one top-ranking pattern. To select this pattern, we consider EVCr, worst-case durations (99th percentile), and whether variants of the pattern are also successful. All patterns shown achieve $>99.9\%$ EVCr. In fact, many patterns exist with similar EVCr.

For Sandy Bridge (2011), the necessary conditions for PRIME+SCOPE still hold, but the PRIME patterns we have found are less efficient. We hypothesize

Table 1: Applicability of PRIME+SCOPE to various CPU microarchitectures, along with a top-ranking access pattern as discovered by PRIMETIME. Each pattern achieves (median) EVCr of $> 99.9\%$ at the indicated (median) cycle cost.

CPU	Year	Microarchitecture	LLC type	C_S	W_{C_S}	PRIME+SCOPE	Prime Access Pattern	Cycles
Intel Core i7-9700K	2018	Coffee Lake	inclusive	LLC	12	✓	`R4_S4_P01SS2301233210`	1 332
Intel Core i7-7700K	2017	Kaby Lake	inclusive	LLC	16	✓	`R2_S4_P01SS2SS301230123`	1 255
Intel Core i5-7500	2017	Kaby Lake	inclusive	LLC	12	✓	`R3_S4_P32SS1SS00123`	1 074
Intel Core i7-6700	2015	Skylake	inclusive	LLC	16	✓	`R3_S4_P01SS2SS301230123`	1 694
Intel Core i5-6500	2015	Skylake	inclusive	LLC	12	✓	`R4_S4_P3SS2SS100123`	1 266
Intel Core i7-4790	2013	Haswell	inclusive	LLC	16	✓	`R3_S4_P3SS2SS100123`	1 149
Intel Core i5-4590	2013	Haswell	inclusive	LLC	12	✓	`R2_S1_P01S2S012`	1 221
Intel Core i7-3770	2012	Ivy Bridge	inclusive	LLC	16	✓	`R3_S4_P3SS2SS1032103210`	1 517
Intel Core i5-3450	2012	Ivy Bridge	inclusive	LLC	12	✓	`R2_S1_P2SS10012`	1 216
Intel Core i5-2400	2011	Sandy Bridge	inclusive	LLC	12	✓	`R5_S1_P0S12012`	3 708
Intel Xeon Platinum 8280	2019	CascadeLake-SP	non-incl.	CD	12	✓	alternating pointer-chase	2 970
Intel Xeon Platinum 8180	2017	Skylake-SP	non-incl.	CD	12	✓	alternating pointer-chase	2 750

that this is because this generation of processors uses the MRU replacement policy in the LLC [39], for which the insertion age is already young to begin with (and the PrimeTime strategy works best when the insertion age is old).

Serialization. PrimeTime avoids processor-specific (reverse-) engineering work. As an alternative to PrimeTime, one can obtain Prime patterns by handcrafting patterns (e.g., [43, 40, 44]) that leverage on the knowledge of the exact cache replacement policy, and the interaction between cache levels. In the end, such a strategy may lead to efficient primes with minimal memory accesses. However, such handcrafted patterns generally need to serialize accesses [40, 43] to prevent out-of-order execution from destroying the intended effects. Such serialization is implemented with pointer-chasing or memory fences, rendering the Prime patterns slower. PrimeTime avoids serialization by executing on the target architecture to incorporate hard-to-predict runtime effects directly. Still, there may exist handcrafted patterns that are more effective than the unordered patterns found by PrimeTime. However, the patterns obtained with PrimeTime are sufficient for Prime+Scope.

5.2 Coherence Directory (CD)

To enable Prime+Scope on the coherence directory, we again need a suitable Prime pattern. Unfortunately, Yan et al. [3] showed that achieving a high eviction rate with known eviction patterns is hard, especially when limited to W addresses. For instance, they report that repeated accesses to $W = 12$ congruent lines require more than 10 iterations to fully prime the CD. This is **Challenge-CD**.

Slow Prime patterns, consisting of many accesses, are not a fundamental problem for Prime+Scope, as the ultimate time resolution is decoupled from the duration of the Prime (cf. Section 3.2). However, our PrimeTime tool indicates that such patterns fail to fix the EVC with high accuracy ($\mathbf{R}_B$), prohibiting Prime+Scope.

On the bright side, non-inclusive Intel caches have the advantage that lines in the CD always reside in one of the lower-level caches, satisfying $\mathbf{R}_C$ by design. Thus, what remains is to find a pattern that installs the desired eviction candidate in the CD ($\mathbf{R}_B$). We first cover a slow but universal solution. Then, we discuss our hypothesis for why traditional patterns do not work well on the CD, leading to a more efficient Prime pattern that leverages this information.

Fill-Flush-Fill. Prior work has used a fill-flush-fill approach to reset and simplify the replacement policy state [41, 39, 40]. Transposed to the CD, it would first fill the CD set, e.g., through many repetitions of an inefficient eviction pattern [3], flush all lines of the eviction set, and finally load them again in order. We

```
function() {
 ...
 before
 operation(secret);
 after
 ...
```

⏲1 `after` (data or instruction)
⏲2 `before` (data or instruction)

(i) Variable-Time Operation

```
function() {
  ...
  load(array[secret*1024]);
  ... // other operations
  load(array[!secret*1024]);
  ...
```

⏲1 `function`
⏲2 `array[0]`

(ii) Variable-Time Access

Figure 5: Uses of differential time: ⏲2 - ⏲1

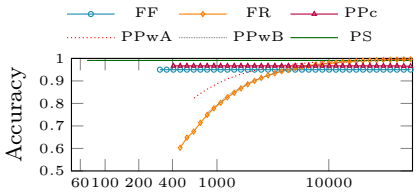
confirm that such patterns successfully prime the CD set (with EVCr > 99.9%), provided that the initial set filling is successful. However, such patterns are relatively slow. Moreover, the `clflush` instruction may not be available in restricted environments (cf. Section 4.1).

CD Replacement Policy. We believe that the poor performance of traditional eviction patterns on the CD is caused by property ②. The reason why this effect is more pronounced for the CD than for inclusive LLCs is the large associativity of private caches in current non-inclusive Intel hierarchies, and that lines in the CD are also cached in L1 and/or L2 [3]. If reading such lines does not influence the replacement state of the CD, many accesses are required for every attacker line to become younger than the lines to be evicted. Thus, for many access patterns, the CD *behaves* like a first-in-first-out (FIFO) queue, irrespective of the actual replacement policy.

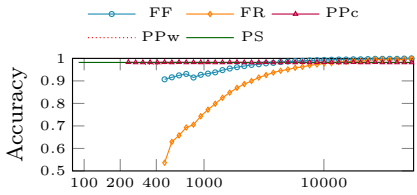
Based on this hypothesis, a straightforward way to prime the CD is to access W congruent lines that are currently not in the CD. Indeed, we find such an access pattern to simultaneously achieve a near-perfect EVr and EVCr (the first element of the set being the scope line `S`), making it a suitable PRIME pattern for PRIME+SCOPE. On all non-inclusive platforms under consideration, our successive PRIMEs alternate between two eviction sets of W addresses. As FIFO is very sensitive to ordering, and insensitive to repeated accesses, we enforce serialization by using a pointer-chasing approach [45].

6 Case Studies

Micro-benchmarks. PRIME+SCOPE bypasses the observer effect of the cache contention side channel, and reduces the cache measurement to a single memory access. Consequently, PRIME+SCOPE is able to monitor victim behavior with



(i) Precision on LLC (Core i5-7500, Kaby Lake)



(ii) Precision on CD (Xeon Platinum 8280, Cascade Lake)

Tech.	WL	Prepare	Measure	Res_{min}	Res_{95}
FR	✗	`clflush T`	`load T`	420	4 350
PPwA	✗	`R1_S1_P0`	`R1_S1_P0`	580	2 500
PPwB	✗	`R3_S1_P012012`	`R1_S1_P0`	790	4 350
FF	✓	/	`clflush T`	300	300
PPc	✓	/	`R1_S1_P0`	390	390
PS	✓	/	`load EVC`	70	70

(iii) Techniques (LLC). WL denotes windowless; Res_{min} and Res_{95} denote max. resolution and resolution for 95% accuracy (cycles)

Tech.	WL	Prepare	Measure	Res_{min}	Res_{95}
FR	✗	`clflush T`	`load T`	440	5 000
PPw	✗	`simple ptr-chase`	`simple ptr-chase`	300	300
FF	✗	`clflush T`	`clflush T`	430	1 600
PPc	✓	/	`simple ptr-chase`	210	210
PS	✓	/	`load EVC`	80	80

(iv) Techniques (CD). WL denotes windowless; Res_{min} and Res_{95} denote max. resolution and resolution for 95% accuracy (cycles)

Figure 6: Accuracy and resolution as function of window size

high temporal precision, even asynchronously, without having to cope with missed accesses. Section 6.1 quantifies this precision and compares it to other techniques, and Section 6.2 characterizes the influence of noise.

Differential Time. By scoping multiple sets simultaneously, PRIME+SCOPE can estimate the temporal separation between two (or more) events with fine precision. Figure 5 shows two classes of timing leaks for which PRIME+SCOPE is particularly well-suited.

The first class is that of *variable-time operations*, where the duration of an `operation` depends on a secret value. Such a code pattern encodes the secret in the time difference between memory accesses before and after `operation`, as in Figure 5i. Several attacks exploit leakage of this kind, e.g., for a secret-dependent number of loop iterations (e.g., [46]), or non-constant-time arithmetic (e.g., modular reduction [16, 17]). Cache attacks can only decode the secret if their precision is sufficient to detect the secret-dependent time difference of `operation`. Often, however, the resolution is too low, prompting the use of performance degradation of the victim [14, 16, 17].

The second class is that of *variable-time accesses*, where memory accesses occur at a secret-dependent *time* (or, as a special case, in a secret-dependent *order* [47]). In Figure 5ii, the elements of `array` are always accessed, but the time relative to the start of `function` depends on a (binary) secret. Again, the attack needs sufficient precision to detect the secret-dependent time differences.

In this paper, we focus on *variable-time access* leakage. Section 6.3 demonstrates a high-capacity covert channel that works with such temporal encoding of the data. In Section 6.4, we show that AES T-tables, a well-studied cache attack target, also exhibits variable-time access leakage. Too fine-grained to be properly harnessed by prior techniques, with PRIME+SCOPE, we exploit it to significantly reduce the number of traces needed for the attack.

Congruence Detection. The repeatable measurement of a single cache line is also useful to determine congruence in the target cache. In Section 6.5, we demonstrate this capability with a simple, efficient and portable eviction set construction methodology.

6.1 Temporal Precision

We now quantify the time resolution of PRIME+SCOPE (PS) to detect cross-core asynchronous events for an inclusive LLC and CD. For reference, we include the most prominent techniques; PRIME+PROBE (PP) for cache contention, i.e., the most comparable technique, and FLUSH+RELOAD (FR) and FLUSH+FLUSH (FF) for shared memory.

For Prime+Probe, the experiment includes windowed (PPw) and windowless (PPc) variants, where we consider two windowed versions for the LLC (PPwA and PPwB), as in Figure 6iii. For the CD, we use the accurate eviction patterns as discovered in Section 5.2, though they were unknown prior to this work.

On our non-inclusive processors, the Flush+Flush side channel also exists although inverted, i.e., lines present in the hierarchy have *lower* flush latency than those that do not. Moreover, the difference is quite large (200 vs. 330 cycles), unlike the subtle difference on our inclusive testbed. Prior work [48] reports that flushing an uncached line on multi-socket Intel systems triggers an access to memory for cross-socket coherence, which would clarify this behavior.

The measurement thresholds are calibrated dynamically and individually for every technique, based on timing histograms and the threshold selection regime with the best results. We note that some techniques (e.g., Flush+Reload, Prime+Scope) are less sensitive to the specific threshold value than others (e.g., Prime+Probe).

Methodology. We consider the following micro-benchmark for detecting asynchronous events. The event to be detected is an access to a specific cache line, by a process pinned to another core. To model an asynchronous event, the process first yields the CPU (`sched_yield`), before waiting for a randomly sampled number of `nops`. Then, the event is triggered with probability 1/2.

We consider the instances listed in Figure 6iii and Figure 6iv. All instances start from an already-prepared state, using the top-ranking Prime from Table 1 for both Prime+Probe and Prime+Scope. The windowless instances (FF, PPc, PS) perform back-to-back measurements, so the preparation phase does not need to be repeated (indicated with `/`). In contrast, the windowed instances (FR, PPwA, PPwB) comprise a measurement, a preparation phase, and a waiting period until the end of the window. All instances run iteratively, and they terminate either when an event is detected, or when there was no event and the random process has terminated.

This experiment is repeated for 1 000 runs of 10 000 events for each window size and each technique, and the global accuracy (true positives and true negatives divided by total) is recorded. We also record the fundamental maximal resolution (i.e., the minimal window size that is able to contain one measurement iteration), as well as the maximal resolution that delivers an accuracy of 95%.

Note that this micro-benchmark serves to quantify, for each technique, the maximal probing resolution for reliable cross-core cache event detection. It should not be interpreted as a comparison of these techniques in a general setting, where more error sources are at play that are not captured here (e.g., noise). However, a poor resolution in this experiment implies a poor resolution in practice.

Also, the experiment assumes that the initial cache preparation is already successfully performed, which may paint an optimistic picture for windowed techniques. For instance, for the CD, the EVr of a single unordered probe of W lines is quite low [3]. Hence, a windowed PRIME+PROBE (PPw) has lower accuracy than in this experiment, due to false negatives incurred by the imperfect EVr, but its temporal resolution is adequately estimated by this experiment.

Results. For both the LLC (Figure 6i) and the CD (Figure 6ii), the resolution of PRIME+SCOPE can be seen to tower above the other techniques, i.e., around 70 cycles or 25ns, while correctly detecting the majority of events (>98%). Figure 6iii and Figure 6iv indicate the maximal resolution (both fundamentally and for 95% accuracy).

As expected, windowed techniques have poor accuracy for small window sizes, with many events landing in blind spots (i.e., false negative errors). This is especially apparent for FLUSH+RELOAD, where small-window instances miss almost all events (cf. [20, 14]).

The resolution for windowless PRIME+PROBE (PPc) is already fairly high (390 cycles for the LLC, and 210 cycles for the CD). In contrast to typical applications of PRIME+PROBE [45, 1, 2], the windowless paradigm decouples PRIME and PROBE. This permits to optimize the PRIME stage for high EVr, and PROBE stage for speed.

The inflated time difference for FLUSH+FLUSH on the CascadeLake server makes it more accurate than FLUSH+RELOAD for all window sizes. However, the accuracy increases with the window size, indicating that the FLUSH measurement on this platform has a blind spot, i.e., it is not concurrent (cf. Section 3.2).

6.2 Susceptibility to Noise

Like PRIME+PROBE, PRIME+SCOPE is susceptible to noise resulting from activity in the targeted cache set, other than the event which is to be monitored. This limitation is fundamental to the cache contention leakage mechanism. It is natural to ask whether the more precise PRIME patterns for PRIME+SCOPE make it more fragile in the presence of noise. We explore it in the following experiment.

We consider two threads pinned to different cores of an Intel Core i7-7700K (Kaby Lake, 16 ways), where one thread monitors the other's memory accesses under different levels of noise. The `stress` tool is used to generate heavy memory load on one or more other cores (e.g., as in [11]). One thread accesses a predetermined address periodically (every 10 000 cycles), as ground truth, while the other thread continuously monitors the cache set for events, and records the timestamps at which the events are detected. Timestamps are obtained

Method	Stress	Correct	Miss	Multi
PS	0	98.24	1.44	0.32
PP_{PS}		99.42	0.45	0.12
PP_{CST}		98.72	1.15	0.13
PS	1	79.71	2.35	17.94
PP_{PS}		83.83	0.54	15.63
PP_{CST}		81.74	0.56	17.70
PS	5	78.38	2.42	19.20
PP_{PS}		82.80	0.68	16.51
PP_{CST}		81.94	0.00	18.06

	Pattern	EVr	EVCr	Duration	Precision
PS	`R3_S4_P01SS2SS301230123`	100%	99.9%	1810	70
PP_{PS}	`R2_S4_P01SS2SS301230123`	100%	99.9%	1255	1170
PP_{CST}	`R2_S1_P01`	100%	NA	1190	700

Figure 7: Distribution of time slots along *correct*, *miss* and *multi* categories for Prime+Scope and Prime+Probe (averages over 200 runs of more than 25 000 timeslots). Stress indicates the amount of `stress` workers, pinned to different cores, that are active in the background. The properties of the Prime patterns are as follows:

using the CPU's time stamp counter, which is synchronized across cores. After execution, the collected timestamps are analyzed to evaluate the detection accuracy of the techniques.

In the ideal case, only one event is detected in each time slot, being the ground-truth periodic access. In the experiment, three cases are distinguished: *correct* when only one event is detected, and it was detected right after the event occurred; *miss* when the ground-truth event was not detected in the time slot (false-negative error); and *multi* when events were detected that did not correspond to the ground-truth access (false-positive error). If a time slot contains both error types, which is uncommon, it is classified as *multi*.

Prime+Scope (PS) is compared with two windowless Prime+Probe instances. As indicated in Figure 7, the first one (PP_{PS}) inherits the Prime access pattern of Prime+Scope. The second one (PP_{CST}) uses a custom Prime+Probe pattern, which is also obtained with PrimeTime, but optimized for EVr instead of EVCr.

For the PRIME+PROBE instances, the indicated PRIME is repeated continuously and serves both as preparation (where *duration* is the number of cycles needed to prepare the cache after an event) and measurement (where *precision* is the number of cycles between successive measurements in the absence of an event). Note that PP_{PS} performs much more accesses than PP_{CST}, which is almost completely hidden in the preparation stage in the shade of cache misses, but is clearly visible in the measurement precision. As in Figure 1ii-H, the PRIME right after detection of an event is ignored, as its execution time may still be affected by that event.

A naive implementation of PRIME+SCOPE performs the PRIME just once for every detected event. However, suppose that the PRIME is unsuccessful in fixing the EVC, e.g., due to noise. This will blind the following SCOPE operations, as they may be fast even if some elements of the cache set have been evicted. To overcome this issue, the PRIME step is repeated when no events were detected within a chosen period (in this experiment, roughly 12 000 cycles).

Results For each technique and noise level, Figure 7 indicates the distribution of time slots along *correct*, *miss* and *multi* rates. This micro-benchmark provides a rough indication of how noise translates to false-positive and false-negative errors for the different windowless techniques. We can draw the following conclusions:

- The *miss* rates of PRIME+SCOPE are slightly (a few p.p.) higher than PRIME+PROBE. The main cause of such false-negative errors are accesses during the preparation phase of the attack, which may result in an imperfectly prepared set [40]. Hence, the observed behavior is clarified by imperfect preparation affecting the EVCr slightly more than the EVr. If high noise levels are to be expected, PRIME+SCOPE fares well with an upwards correction of the PRIME repetitions compared to the output of PRIMETIME (e.g., as in this experiment, where `R2_*` → `R3_*`).
- In terms of *multi* rates, all instances are comparable. The main cause of such false-positive errors is noise during the measurement phase, evicting the EVC. As this leads to high access latencies for both PROBE and SCOPE, this source of errors is expected to affect PRIME+SCOPE and PRIME+PROBE equally.

6.3 Cross-Core Covert Channel

To show that PRIME+SCOPE can discern fine-grained temporal cache activity, we build a high-capacity cross-core covert channel based on *variable-time access* leakage (cf. Figure 5ii). It temporally encodes m-bit symbols by performing a memory access in one of 2^m slots, where slots may be as short as 80 processor cycles.

As a representative sample, we implement it on the LLC of a Kaby Lake

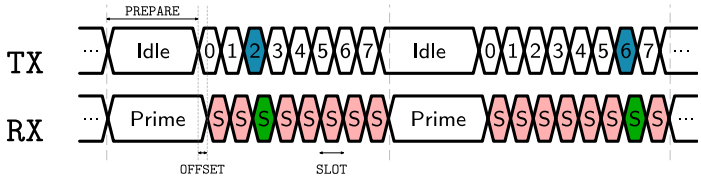


Figure 8: Covert Channel Operation ($m = 3$ bits per symbol).

processor, and on the CD of CascadeLake-SP. For our proof-of-concept implementation, we assume a synchronized transmitter and receiver that have agreed on a contention set (e.g., as in [49, 3, 50]).

Figure 8 visualizes the working mechanism of the covert channel, as well as its defining parameters (duration of preparation stage, transmission slots, and transmitter-receiver offset). First, the receiver primes the set. Then, the transmitter sends an m-bit symbol M by accessing a congruent line in slot number $i = M$. At the same time, the receiver scopes the set every `SLOT` cycles, decoding M as the slot number in which the scope line `S` is evicted.

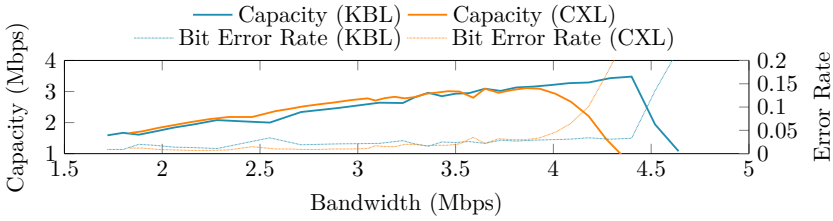
Optimizations. We perform a few modifications to improve the channel bandwidth. Instead of the canonical encoding, we encode the bitstream into m-bit symbols with reflected binary Gray codes to ensure that off-by-one symbol errors only lead to single-bit errors.

For the LLC channel, the receiver uses the Prime patterns of Section 5.1. We find that if the transmitter flushes the line right after accessing it, it slightly speeds up the prime for the receiver.

For the CD channel, the Prime stages consist of alternating pointer chases (cf. Section 5.2). To amortize the latency arising from serialization, four sets are primed simultaneously with their accesses interleaved (e.g., as in [51]). After the combined Prime, there are four rounds of 2^m slots, where each round encodes m bits.

Evaluation. Figure 9 gives capacity and error rate as a function of bandwidth, and summarizes the parameters for which the LLC- and CD-based channels obtain peak capacity. Respectively, the capacities are 3.5 Mbps and 3.1 Mbps, which is much higher than Prime+Probe on the LLC (e.g., 500 Kbps at 1% bit error rate [23]). Furthermore, they are in the same order of magnitude as state-of-the-art stateless channels without shared memory, such as Pessl et al. [49] (DRAM row buffer contention, 2.1 Mbps capacity) and Paccagnella et al. [50] (LLC ring contention, 4.1 Mbps capacity).

To our knowledge, the only other covert channel using the CD is due to



Platform	C_S	m	Capacity	`PREPARE`	`OFFSET`	`SLOT`
Core i7-7500 (KBL)	LLC	4	3.5 Mbps	1 400	90	100
Xeon Pl. 8280 (CXL)	CD	3	3.1 Mbps	4 750	125	100

Figure 9: Covert Channel Capacities and error rates for the Kaby Lake (KBL) and CascadeLake-SP (CXL) platform. For the peak capacities, the configuration in the following table are used, where `PREPARE`, `OFFSET` and `SLOT` are in cycles.

Yan et al. [3], with a bandwidth of 0.2 Mbps (error rate not reported). The order-of-magnitude capacity improvement of our channel stems from both a fast and efficient PRIME pattern (cf. Section 5.2), and the precision of PRIME+SCOPE (cf. Section 6.1).

As the goal is to characterize the temporal precision of PRIME+SCOPE, we limit the study of this covert channel to synchronized parties on idle systems. In practice, further engineering challenges need to be overcome (e.g., as undertaken in [1, 10, 11]).

6.4 Side-Channel Attack on AES

We now revisit the seminal first-round known-plaintext attack on the T-table implementation of AES [8], a standard benchmark for cache attack techniques (e.g., [23, 27, 52]). The time precision of PRIME+SCOPE allows a novel attack technique against AES, based on *variable-access time* leakage (cf. Figure 5ii), rather than traditional *access* leakage. As it can learn more information from each encryption, much fewer traces are needed to extract the secret. Although a windowless PRIME+PROBE can also absorb some of this information, PRIME+SCOPE requires 10-70x fewer traces. We first give a high-level outline of the traditional attack (for details, refer to [8, 45]). Like prior work, we attack OpenSSL 1.0.1e (or similar).

Traditional Attack The implementation features four precomputed tables Te_j, of 16 cache lines each. The attacker monitors accesses to such table lines $Te_j[M]$ which, on CPUs with 64-byte cache lines, leak the upper four bits (nibble) of every key byte k_i. We implement this attack with PRIME+PROBE (for

comparison) and Flush+Reload (for reference), where the attacker prepares the cache, triggers an encryption with known plaintext, and measures afterwards. For plaintexts where $\lceil p_i \rceil^4 = \lceil k_i \rceil^4 \oplus M$, cache line $Te_{i \bmod 4}[M]$ is accessed in the first round, and hence, in 100% of encryptions. For other p_i, it is accessed in 92.5% of encryptions, so each monitored $Te_j[M]$ carries information in 7.5% of encryptions.

Variable-Time Access: Prime+Scope. Consider the code snippet in Figure 10. Indeed, not only the access to a table encodes information, but also the encryption round in which it happens. We now show that, through its time precision, Prime+Scope is able to capture such leakage. Information is obtained through *differential time* between the start of the `AES_encrypt` function and one or more table entries.

We use the cache attack as an oracle for accesses to table entries during the first AES round. We spin up a thread for each monitored line (including the first instruction cache line of `AES_encrypt`). The adversary triggers encryptions, and each thread records the timestamp at which the access is detected (if any) for the monitored table entry. Then, the differential times are used to score the key nibble hypotheses. The larger the differential time, the larger the penalty for the key nibble, as the probability is lower that it corresponds to a first-round access. For a table access in the first round, we observe the differential time to be around 200-300 cycles.

An advantage of this attack is that every trace carries information for each monitored table entry, as opposed to only 7.5% for the traditional first-round attack. Note that a single-threaded Prime+Scope can also record differential times, but the temporal resolution decreases linearly with the number of lines scoped in one thread.

Variable-Time Access: Prime+Probe? For comparison, we explore whether Prime+Probe can also learn from the differential time. To capture the maximal performance of Prime+Probe, we consider an optimal, windowless configuration; the Prime is the same as for Prime+Scope, and the Probe

```
void AES_encrypt(...) {
 ... // s0-s3 contain p_i xor k_i
 // round 1:
 t0 = Te0[s0>>24] ^ Te1[(s1>>16) & 0xff]
    ^ Te2[(s2>>8) & 0xff] ^ Te3[s3 & 0xff] ^ rk[4];
 t1 = ... ; t2 = ...; t3 = ...; // similar to t0
 ... // rounds 2-10 (similar to round 1)
```

⏲1 AES_encrypt
⏲2 Te0/Te1/Te2/Te3

Figure 10: Variable-time access leakage for AES

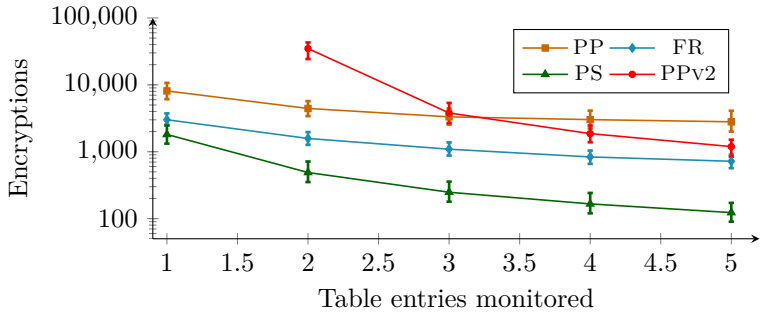


Figure 11: Median encryptions for AES T-tables (bars indicate 10-90th percentiles). Comparison of PRIME+SCOPE (PS) with traditional PRIME+PROBE (PP) and FLUSH+RELOAD (FR), as well as differential-time PRIME+PROBE (PPv2).

is the simple, unordered traversal of the set (pattern `R1_S1_P0`). According to Figure 6iii), we expect a precision of approx. 400 cycles (cf. 70 cycles for PRIME+SCOPE).

Results. Figure 11 presents the results on the LLC of an Intel Core i7-7700K (Kaby Lake, 16 ways). It shows the number of encryptions needed to mount the full first-round attack, which recovers 64 of the 128 key bits. We consider the key nibble found as soon as the hypothesis converges (i.e., it reaches the correct value and does not diverge from it). We perform 1 000 iterations and indicate the median and 10th and 90th percentiles to convey the variance. Note that these results are obtained without degrading victim performance (other than indirectly through the cache sets that are monitored).

PRIME+SCOPE retrieves the secret information with fewer traces (between 5-25x) than the traditional PRIME+PROBE. The differential-time PRIME+PROBE is also able to capture some of the temporal information, but again more slowly, with more traces than PRIME+SCOPE (10-70x). When only a single table entry is monitored in every encryption, we find that it fails to recover the secret even with as many as 100 000 traces, which may indicate that the timing differences are too small to be distinguished by PRIME+PROBE.

6.5 Finding Congruent Addresses

Cache contention attacks require the adversary to find eviction sets, i.e., sets of congruent addresses in the target cache. This practical challenge has been investigated thoroughly [1, 2, 27, 26, 3]. However, the principles underlying PRIME+SCOPE enable an efficient congruence test, resulting in a faster and simpler

routine that, counter-intuitively, requires fewer platform-specific parameters.

Algorithm 2 Eviction Set Construction

Input: `TARGET`: address for which an LLC eviction set is desired
Output: `ES`: eviction set

```
1: ES ← empty list
2: length ← 0
3: while length < LLC_WAYS do
4:     access(TARGET)
5:     do
6:         GUESS ← a line possibly congruent to the TARGET
7:         access(GUESS)
8:     while access(TARGET) is fast
9:     ES[length++] ← GUESS
10: end while
```

Algorithm: LLC. The foundation of the proposed LLC eviction set construction routine is given in Algorithm 2. It repeatedly measures the access latency of the `TARGET` address and, between each measurement, accesses a guess. As `TARGET` is continuously accessed, it is always served from the L1 cache, which does not influence its LLC replacement state. Guesses that turn out to be congruent with the `TARGET` are installed in the LLC, and each time this happens, the EVC in the LLC changes. After enough congruent guesses, the `TARGET` becomes the EVC. The next congruent guess then evicts `TARGET` from the LLC and, due to the inclusion property, also from the private caches. Therefore, the next access to `TARGET` is slow, indicating the congruence of the latest guess. The attacker repeats this procedure until she has obtained enough congruent addresses.

To speed up the routine, between lines 4 and 5 in Algorithm 2, we access already-obtained congruent addresses to accelerate `TARGET` becoming the LLC eviction candidate. Thus, the number of inner-loop iterations is expected to decrease as the algorithm proceeds.

To increase the robustness, we test whether the resulting set successfully evicts the `TARGET`. If not, an extra address is found, and the test is repeated. The number of failures until the test succeeds reveals the number of false positives in the set, which can be removed through a short reduction phase, akin to prior work [1, 26, 3].

The algorithm is identical for huge and small virtual memory pages, but the availability of huge pages speeds up the runtime significantly, as the guesses are more likely to be congruent due to the increased control over physical address bits [1, 26].

Table 2: Runtime (median) and accuracy (%) for eviction set construction (1 000 runs for randomly selected targets)

Processor	Vila et al. [26]		Ours	
Cache	Huge*	Small*	Huge	Small
Skylake	165.2 ms	316.3 ms	0.25 ms	2.80 ms
12 Way LLC	99%	100%	99%	99%
Skylake	113.2 ms	643.8 ms	0.55 ms	4.03 ms
16 Way LLC	98%	100%	96%	100%
Skylake-SP	NA	NA	3.15 ms	35.40 ms
12 Way CD	NA	NA	100%	93%

∗ Initial set size for $^{12}_{16}$ Way LLC is $^{65}_{90}$ for huge pages, $^{3500}_{4000}$ for small.

Algorithm: CD. On non-inclusive Intel caches, the set index mapping for the LLC and CD is identical. Hence, eviction sets constructed for one may be used for the other. Finding congruent addresses through contention on the CD is challenging, as congruence in the CD implies congruence in L2 [3], and `TARGET` may be evicted due to contention on L2, leading to false positives. Thus, like prior work [3], we perform the construction on the LLC.

The routine is similar to in Algorithm 2. However, recall that the LLC is non-inclusive, so the memory accesses on lines 4 and 7 do not guarantee the installation of the `TARGET` and the `GUESS`es in the LLC. We replace them with joint accesses by the attacker thread and a helper thread on another CPU core, as we observed that accesses from two cores place a copy of the line into the LLC[1].

Platforms. We tested the eviction set construction on all the machines in Table 1, as we had to obtain eviction sets for PRIMETIME. To compare with other work, we perform a detailed comparison on the Skylake microarchitectures in Table 2. Apart from the differences for inclusive and non-inclusive LLCs and a parameter for LLC associativity, it requires no adaptation to the processor.

Comparison. Vila et al. [26] study eviction set construction in detail, and propose a linear-time algorithm that improves over the quadratic-time baseline [1]. These routines iteratively remove one or more lines from a big initial set, measuring whether the residual set still evicts the target. In contrast, Algorithm 2 starts from an empty set, and adds congruent lines to it. It overcomes practical problems identified by previous works, such as the dependence

[1] For more information, we refer the reader to `https://www.github.com/KULeuven-COSIC/PRIME-SCOPE/evsets`.

Table 3: Classification of cross-core cache attack techniques in terms of prerequisites and features

Attack Technique	Mechanism		Prerequisites			Features			
	Leakage Source	Spatial Granularity	No Shared Mem.	No `clflush`	No TSX	Window-less	Measure Size	Multi-Target	Shown on CD
Flush+Reload [20]	load latency	line	✗	✗	✓	✗	$1_{✓}$	✓	✓
Flush+Flush [23]	load latency	line	✗	✗	✓	✓	$1_{✓}$	✓	✓
Evict+Reload [22]	load latency	line	✗	✓	✓	✗	$1_{✓}$	✓	✓
Reload+Refresh [40]	repl. state	line	✗	✗	✓	✗	$2_{✓}$	✓	✗
Prime+Probe [1, 2]	contention	set	✓	✓	✓	✓	$W_{✗}$	✓	✓
Occupancy [6]	contention	none	✓	✓	✓	✗	huge$_{✗}$	✗	✓
Prime+Abort [27]	TSX abort	set	✓	✓	✗	✓	$\varnothing_{✓}$	✗	✗
Prime+Scope	**contention**	**set**	✓	✓	✓	✓	$\mathbf{1}_{✓}$	✓	✓

on replacement policies (and their adaptivity) [26], TLB thrashing [53], and hardware prefetchers [45]. Similar to prior techniques [1, 26, 3], it does not require knowledge of the slicing function.

Because our implementation does not require any preparation steps, such as organizing the memory space in a linked list, or selecting a suitable starting set, we take into account the *total* execution time of the construction routine, which includes the time spent for failed preparation steps in addition to the last successful reduction step. As shown in Table 2, our implementation executes up to 660x faster than the one by Vila et al. [26], while achieving the same success rate (where success is defined as a set of W addresses that consistently evicts the target). Furthermore, the default configuration is adequate for successful execution on all tested Intel processors, while containing only a few configuration parameters.

For non-inclusive caches, only the initial study by Yan et al. [3] describes how to find LLC/CD eviction sets. They adapt the congruence test of earlier work [1] to overcome the challenges provided by non-inclusive LLCs. Compared to ours, their routine has the advantage of being single-threaded. As performance metrics are not provided in [3], we are unable to directly compare our work with theirs. However, it has quadratic complexity, and is so far unsuccessful when huge pages are not available. Even if their routine is adapted to linear time (e.g., [26]), we expect our algorithm to outperform it, in accordance with the findings for inclusive caches.

7 Related Work

7.1 Classification of Attack Techniques

Complementing the quantitative study in Section 6, Table 3 positions Prime+Scope with respect to existing cross-core cache attack techniques on the basis of prerequisites and features.

Prerequisites The most basic requirement is co-tenancy, where the attacker can run unprivileged code (native or otherwise) on the same physical machine as a victim. As long as both parties share at least one cache level, an attacker can measure contention on shared cache resources (as is done for PRIME+PROBE and PRIME+SCOPE).

Some techniques are predicated on additional capabilities, such as shared memory with the victim, the presence of a `clflush` instruction, or special processor features like Intel's TSX. These extra capabilities can increase the power of the technique, e.g., in terms of spatial resolution or reliability. However, the additional prerequisites limit the applicability of these techniques. For instance, shared memory is discouraged in multi-tenant clouds, and `clflush` may not be available to code that is not running natively on the system (e.g., in the browser). Intel TSX is not available on all Intel CPUs and, for those where it is available, Intel has added support to disable it [29] in response to recent transient execution attacks [54, 55].

Features. The most relevant features to this work are whether a technique can be instantiated in a *windowless* paradigm, and the number of cache accesses for each measurement.

In terms of spatial granularity, techniques based on shared memory can infer accesses to specific cache lines, whereas cache contention attacks are fundamentally limited to set-granularity. PRIME+SCOPE belongs to the latter category. Table 3 also indicates which techniques have been shown on CDs of non-inclusive LLCs [3].

Measuring *multiple events* enriches the information content of the channel and is an essential requirement to record differential times (cf. Section 6). PRIME+ABORT cannot monitor multiple events while maintaining the ability to distinguish between them [27]. Other techniques can do so, but may have to take the influence of spatial hardware prefetching into account [56, 22, 32].

Other Properties Some techniques are tailored to overcome specific system-level constraints. Cache occupancy attacks [6] forego the search for congruent addresses and instead measure contention on a cache-sized buffer. This makes them amenable for deployment in (very) restricted environments [57], at the cost of all spatial granularity and significant time precision. Some techniques offer stealth against runtime detection [23, 40], or bypass software-based countermeasures with indirect cache accesses [52]. Exploring PRIME+SCOPE in these system models is beyond the scope of this work.

7.2 Cache Attacks and Replacement Policies

Cache replacement policies were long perceived as obstacles, leading to techniques that minimize their influence (e.g., double pointer-chasing [45] or black-box eviction strategies [42]). However, enabled by reverse-engineering advances [38, 40, 41, 39], some works use replacement properties to the advantage of the attacker.

Same-Core. Xiong and Szefer [43] use the PLRU policy of the L1 cache to leak information between processes through LRU states. Recently, Röttger and Janc [13] use it to amplify the time difference between presence and absence of a speculative memory access.

Cross-Core. Reload+Refresh [40] detects accesses to a shared address by monitoring changes in the EVC. In this context, our efficient Prime patterns may be useful to prepare the EVC. Wang et al. [32] probe the L2 EVC to limit the impact of the aggressive hardware prefetcher on low-end, in-order Intel CPUs. Their Prime pattern consists of $2W$ ordered accesses (all cache misses), making it comparatively slow. Briongos et al. [44] detect the start of a victim routine and exploit LLC replacement to evict prefetched lines at the right time. As Prime+Probe lacks the required precision, they rely on Prime+Abort for detection. Future work should investigate the use of Prime+Scope to remove the dependency on Intel TSX.

To enable Rowhammer attacks without flushing, Gruss et al. [42] find efficient eviction strategies for unknown replacement policies. Aweke et al. [24] develop a pattern predicated on the Sandy Bridge MRU policy, which De Ridder et al. [51] modernize and improve for browser-based Rowhammer in the presence of DRAM mitigations.

8 Limitations and Countermeasures

Requirements. Prime+Scope does not work on processors for which the key properties (cf. Section 4) do not hold. For instance, it fails when the shared structure C_S has a random replacement policy (as it eliminates predictability of the `EVC`), or if the lower-level caches do not act as a filter for C_S (as it eliminates repeatability of the measurement). We believe these two properties are the only anchor points for the deployment of countermeasures to reduce Prime+Scope to Prime+Probe. However, invalidating these properties may adversely affect multi-level cache performance.

Leakage Types. As demonstrated in Section 6, Prime+Scope is able to extract information from fine-grained timing leaks. However, if time differences

are more coarse-grained (e.g., RSA square-and-multiply [20, 1]), the increased precision of PRIME+SCOPE does not directly lead to a more efficient attack. Still, we note that the windowless nature of PRIME+SCOPE eliminates false-negative errors due to overlap between measurement and event, which may help to reduce the number of required observations to retrieve the secret.

High-Frequency Events. Recall that for PRIME+SCOPE (and PRIME+ABORT and PRIME+PROBE), even for windowless instances, the cache state needs to be prepared after every detected event. If the event rate is very high, i.e., when the temporal separation of accesses to the same address is in the order of the PRIME duration, the preparation step becomes dominant for the time precision. We note that, although PRIME+SCOPE places more demands on cache state preparation than its counterparts, the PRIME patterns obtained with PRIMETIME are still fairly competitive, with most of them in the range of 1000-1300 cycles (cf. Table 1).

Generic Countermeasures. FLUSH+RELOAD and FLUSH+FLUSH can be thwarted by disallowing shared memory across security boundaries, but countermeasures to mitigate the cache contention channel are far more invasive. However, in recent years, this defensive avenue has attracted attention in the research community. The main lines of work are based on *isolation*, i.e., partitioning the cache along isolated portions (e.g., [58, 59, 60, 61, 62]), or *randomization*, i.e., obfuscating interference by modifying the set index mapping (e.g., [63, 64, 65, 66, 67, 68, 69]). By strengthening cache contention attacks, our work motivates further research in this direction.

9 Conclusion

This paper introduced PRIME+SCOPE, a high-resolution primitive to measure contention on shared cache resources. It can target last-level caches and directories alike, and we found it to apply to all tested Intel processors of the last decade. Roughly speaking, PRIME+SCOPE is a high-resolution successor to PRIME+PROBE, assuming the same attacker capabilities that make the latter so widely applicable. The fast and repeatable SCOPE measurement essentially optimizes the resolution of cache contention attacks, delivering a cross-core time precision that even flush-based techniques cannot provide.

We believe that PRIME+SCOPE is a valuable addition to the microarchitectural attack toolbox. We quantitatively evaluated its properties, and illustrated them with a high-bandwidth covert channel, a new fine-grained attack on AES T-tables, and a simple, efficient, and portable routine to construct eviction sets.

Acknowledgments

We thank the anonymous CCS 2021 reviewers, as well as Frank Piessens and Márton Bognár, for their valuable feedback. This research is partially funded by the European Research Council (ERC - #695305) and the Flemish Government through the FWO project TRAPS. It was also supported by the CyberSecurity Research Flanders (#VR20192203). Additional funding was provided by a generous gift from Intel. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[2] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES," in *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[3] M. Yan, R. Sprabery, B. Gopireddy, C. W. Fletcher, R. H. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[4] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "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.

[5] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The spy in the sandbox: Practical cache attacks in javascript and their implications," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[6] A. Shusterman, L. Kang, Y. Haskal, Y. Meltser, P. Mittal, Y. Oren, and Y. Yarom, "Robust website fingerprinting through the cache occupancy channel," in *USENIX Security Symposium*, 2019.

[7] M. Kurth, B. Gras, D. Andriesse, C. Giuffrida, H. Bos, and K. Razavi, "Netcat: Practical cache attacks from the network," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[8] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[9] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[10] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-cores cache covert channel," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2015.

[11] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the other side: SSH over robust cache covert channels in the cloud," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[12] E. Göktas, K. Razavi, G. Portokalidis, H. Bos, and C. Giuffrida, "Speculative probing: Hacking blind in the spectre era," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[13] S. Röttger and A. Janc, "A Spectre proof-of-concept for a Spectre-proof web." https://github.com/google/security-research-pocs/tree/master/spectre.js, 2021.

[14] T. Allan, B. B. Brumley, K. Falkner, J. Van de Pol, and Y. Yarom, "Amplifying side channels through performance degradation," in *Annual Conference on Computer Security Applications (ACSAC)*, 2016.

[15] D. J. Bernstein, J. Breitner, D. Genkin, L. G. Bruinderink, N. Heninger, T. Lange, C. van Vredendaal, and Y. Yarom, "Sliding right into disaster: Left-to-right sliding windows leak," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[16] D. Genkin, L. Valenta, and Y. Yarom, "May the fourth be with you: A microarchitectural side channel attack on several real-world applications of curve25519," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017.

[17] D. F. Aranha, F. R. Novaes, A. Takahashi, M. Tibouchi, and Y. Yarom, "Ladderleak: Breaking ECDSA with less than one bit of nonce leakage," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2020.

[18] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *IEEE Symposium on Security and Privacy (S&P)*, 2011.

[19] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "CacheZoom: How SGX amplifies the power of cache attacks," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2017.

[20] Y. Yarom and K. Falkner, "Flush+reload: A high resolution, low noise, l3 cache side-channel attack," in *USENIX Security Symposium*, 2014.

[21] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.

[22] D. Gruss, R. Spreitzer, and S. Mangard, "Cache template attacks: Automating attacks on inclusive last-level caches," in *USENIX Security Symposium*, 2015.

[23] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[24] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "Anvil: Software-based protection against next-generation rowhammer attacks," *ASPLOS*, 2016.

[25] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices," in *USENIX Security Symposium*, 2016.

[26] P. Vila, B. Köpf, and J. F. Morales, "Theory and practice of finding eviction sets," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[27] C. Disselkoen, D. Kohlbrenner, L. Porter, and D. M. Tullsen, "Prime+abort: A timer-free high-precision L3 cache attack using intel TSX," in *USENIX Security Symposium*, 2017.

[28] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and efficient cache side-channel protection using hardware transactional memory," in *USENIX Security Symposium*, 2017.

[29] Intel, "Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort." https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asyn 2019.

[30] G. Irazoqui, M. S. Inci, T. Eisenbarth, and B. Sunar, "Wait a minute! a fast, cross-vm attack on aes," in *Research in Attacks, Intrusions, and Defenses (RAID)*, 2014.

[31] L. G. Bruinderink, A. Hülsing, T. Lange, and Y. Yarom, "Flush, gauss, and reload–a cache attack on the BLISS lattice-based signature scheme," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[32] D. Wang, Z. Qian, N. Abu-Ghazaleh, and S. V. Krishnamurthy, "Papp: Prefetcher-aware prime and probe side-channel attack," in *Design Automation Conference (DAC)*, 2019.

[33] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2012.

[34] B. Gülmezoglu, M. S. Inci, G. I. Apecechea, T. Eisenbarth, and B. Sunar, "A faster and more realistic flush+reload attack on AES," in *Constructive Side-Channel Analysis and Secure Design (COSADE)*, 2015.

[35] N. Benger, J. Van de Pol, N. P. Smart, and Y. Yarom, ""ooh aah... just a little bit": A small amount of side channel can go a long way," in *Cryptographic Hardware and Embedded Systems (CHES)*, 2014.

[36] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *USENIX Security Symposium*, 2017.

[37] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[38] A. Abel and J. Reineke, "Measurement-based modeling of the cache replacement policy," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2013.

[39] A. Abel and J. Reineke, "nanobench: a low-overhead tool for running microbenchmarks on x86 systems," in *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[40] S. Briongos, P. Malagon, J. M. Moya, and T. Eisenbarth, "RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks," in *USENIX Security Symposium*, 2020.

[41] P. Vila, P. Ganty, M. Guarnieri, and B. Köpf, "Cachequery: Learning replacement policies from hardware caches," in *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[42] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A remote software-induced fault attack in javascript," in *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[43] W. Xiong and J. Szefer, "Leaking information through cache lru states," in *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[44] S. Briongos, I. Bruhns, P. Malagón, T. Eisenbarth, and J. M. Moya, "Aim, wait, shoot: How the cachesniper technique improves unprivileged cache attacks," in *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[45] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on aes, and countermeasures," *Journal of Cryptology*, 2010.

[46] L. De Feo, B. Poettering, and A. Sorniotti, "On the (in) security of elgamal in openpgp," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[47] M. Behnia, P. Sahu, R. Paccagnella, J. Yu, Z. Zhao, X. Zou, T. Unterluggauer, J. Torrellas, C. Rozas, A. Morrison, F. Mckeen, F. Liu, R. Gabor, C. W. Fletcher, A. Basak, and A. Alameldeen, "Speculative interference attacks: Breaking invisible speculation schemes," *ASPLOS*, 2021.

[48] L. Cojocar, J. Kim, M. Patel, L. Tsai, S. Saroiu, A. Wolman, and O. Mutlu, "Are we susceptible to rowhammer? an end-to-end methodology for cloud providers," in *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[49] P. Pessl, D. Gruss, C. Maurice, M. Schwarz, and S. Mangard, "Drama: Exploiting dram addressing for cross-cpu attacks," in *USENIX Security Symposium*, 2016.

[50] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the cpu on-chip ring interconnect are practical," in *USENIX Security Symposium*, 2021.

[51] F. de Ridder, P. Frigo, E. Vannacci, H. Bos, C. Giuffrida, and K. Razavi, "Smash: Synchronized many-sided rowhammer attacks from javascript," in *USENIX Security Symposium*, 2021.

[52] S. Van Schaik, C. Giuffrida, H. Bos, and K. Razavi, "Malicious management unit: Why stopping cache attacks in software is harder than you think," in *USENIX Security Symposium*, 2018.

[53] D. Genkin, L. Pachmanov, E. Tromer, and Y. Yarom, "Drive-by key-extraction cache attacks from portable code," in *Applied Cryptography and Network Security*, 2018.

[54] M. Schwarz, M. Lipp, D. Moghimi, J. V. Bulck, J. Stecklina, T. Prescher, and D. Gruss, "Zombieload: Cross-privilege-boundary data sampling," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[55] S. Van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Ridl: Rogue in-flight data load," in *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[56] Y. Yarom and N. Benger, "Recovering openssl ecdsa nonces using the flush+ reload cache side-channel attack.," *IACR Cryptol. ePrint Arch. 2014/140*, 2014.

[57] A. Shusterman, A. Agarwal, S. O'Connell, D. Genkin, Y. Oren, and Y. Yarom, "Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses," in *USENIX Security Symposium*, 2021.

[58] L. Domnitser, A. Jaleel, J. Loew, N. Abu-Ghazaleh, and D. Ponomarev, "Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks," *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[59] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[60] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *USENIX Security Symposium*, 2016.

[61] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *USENIX Security Symposium*, 2020.

[62] R. Bahmani, F. Brasser, G. Dessouky, P. Jauernig, M. Klimmek, A.-R. Sadeghi, and E. Stapf, "{CURE}: A security architecture with customizable and resilient enclaves," in *USENIX Security Symposium*, 2021.

[63] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *International Symposium on Computer Architecture (ISCA)*, 2007.

[64] F. Liu and R. B. Lee, "Random fill cache architecture," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2014.

[65] M. K. Qureshi, "Ceaser: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[66] M. K. Qureshi, "New attacks and defense for encrypted-address cache," in *International Symposium on Computer Architecture (ISCA)*, 2019.

[67] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "Scattercache: Thwarting cache attacks via cache set randomization," in *USENIX Security Symposium*, 2019.

[68] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," in *Network and Distributed System Security Symposium (NDSS)*, 2020.

[69] G. Saileshwar and M. Qureshi, "Mirage: Mitigating conflict-based cache attacks with a practical fully-associative design," in *USENIX Security Symposium*, 2021.

Chapter 7

Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies

Publication data

Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede, "Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies". *USENIX Security Symposium*, 2022, pp. 3647–3664

For compactness, the appendices are not included. Please refer to [155].

Contributions

Main author together with Furkan Turan, who developed the hardware implementation.

Double Trouble: Combined Heterogeneous Attacks on Non-Inclusive Cache Hierarchies

Antoon Purnal, Furkan Turan and Ingrid Verbauwhede

imec-COSIC, KU Leuven

Abstract. As the performance of general-purpose processors faces diminishing improvements, computing systems are increasingly equipped with domain-specific accelerators. Today's high-end servers tightly integrate such accelerators with the CPU, e.g., giving them direct access to the CPU's last-level cache (LLC).
Caches are an important source of information leakage across security domains. This work explores *combined* cache attacks, complementing traditional co-tenancy with control over one or more accelerators. The constraints imposed on these accelerators, originally perceived as limitations, turn out to be advantageous to an attacker. We develop a novel approach for accelerators to find eviction sets, and leverage precise double-sided control over cache lines to expose undocumented behavior in non-inclusive Intel cache hierarchies.
We develop a compact and extensible FPGA hardware accelerator to demonstrate our findings. It constructs eviction sets at unprecedented speeds ($< 200\,\mu s$), outperforming existing techniques with one to three orders of magnitude. It maintains excellent performance, even under high noise pressure. We also use the accelerator to set up a covert channel with fine spatial granularity, encoding more than 3 bits per cache set. Furthermore, it can efficiently evict shared targets with tiny eviction sets, refuting the common assumption that eviction sets must be as large as the cache associativity.

1 Introduction

Heterogeneous computing yields great increases in performance and energy efficiency with specific processing capabilities for certain tasks. Recently, FPGAs have emerged in datacenters for providing such capabilities. They can be used to accelerate wide-scale data center services, such as machine learning applications. More interestingly, cloud service providers give control of FPGAs to customers, who can implement custom accelerators and integrate them into

Table 1: Positioning of threat models considered in this work

Attacker \ Victim	CPU	Secondary
CPU	Last-Level Cache [70, 33, 21] Coherence Directory [66] **This work**	Packet Chasing [55]
Secondary	Grand Pwning Unit [11] JackHammer [62]	NetCAT [27]
Combined	**This work**	

their applications. After initial steps by Amazon's AWS, which already allows users to rent FPGA-supported instances, AMD and Intel acquired the two main FPGA manufacturers (resp. Xilinx and Altera). The aim is to aid customers in their transition through easy and low-overhead integration of software and hardware. As of yet, their security is not fully mature [57]. At the same time, economic incentives attract infrastructure providers to multi-tenancy, i.e., hosting multiple (distrusting) entities on the same physical machine. This work evaluates the impact of heterogeneous multi-tenancy on the quintessential shared hardware component: the cache hierarchy.

Caches play a fundamental role in high-performance computing. By serving the majority of memory requests from fast levels of storage close to the processor (CPU), they overcome the bottleneck caused by comparatively slow memory. Equally fundamental, however, is the timing side channel they introduce, as access latencies depend on access patterns of co-located processes. While some attack techniques rely on cache flushes and shared memory [14, 70], others work with contention [33, 21]. To determine the access patterns of the victim, contention-based attacks employ so-called *eviction sets*, i.e., sets of addresses that contend for cache resources.

Over time, the cache side channel was proven effective to extract keys from cryptographic implementations [41, 3, 17, 70], retrieve user input [50, 40, 13], or infer kernel secrets [16, 23, 12]. Caches have also been used to establish covert channels [33, 37] and are a key enabler of recent transient execution attacks [31, 25]. The ongoing switch to non-inclusive cache hierarchies for high-end CPUs was believed to thwart several attack classes, but this belief has been disproven [66]. However, non-inclusive hierarchies remain relatively unexplored compared to their inclusive counterparts.

The lion's share of the cache attack literature considers CPU processes targeting other processes. Recent studies introduced some heterogeneity, whether it be peripheral devices attacking CPU processes [11, 27, 62], or CPU processes attacking peripherals [55]. This work identifies combined microarchitectural attacks as a threat deserving further examination (cf. Table 1). It is becoming

increasingly common to control multiple entities (i.e., devices), which differ in computational capabilities and access to the memory subsystem.

Intel's Data-Direct IO (DDIO) [18] is a prominent technology that gives PCIe devices direct access to the CPU's last-level cache (LLC). For instance, existing FPGA-accelerated cloud platforms make use of PCIe-based FPGA accelerator cards. DDIO provokes an interesting dynamic in the cache hierarchy, which CPU processes observe through the lens of their private caches, whereas PCIe (DDIO) devices, from now on referred to as *secondary devices*, observe it directly through the LLC. This double view is especially interesting in non-inclusive cache hierarchies, where the LLC is less amenable to direct interaction [66]. This work investigates the collusion of microarchitectural attackers in heterogeneous systems, applied to the widely-used DDIO technology, which already has been shown to bear security implications [27, 62, 55].

In light of the growing interest in heterogeneous computing, this paper seeks to answer the following questions:

How precisely can combined attackers control shared cache state? Can the constraints imposed on DDIO devices turn out to be advantageous? Do common assumptions remain valid in the face of combined attackers?

In this paper, we study combined heterogeneous attacks on emerging non-inclusive Intel cache hierarchies. We identify a set of properties governing the interaction between the CPU and DDIO devices. Attacks originating from secondary devices, e.g., network cards [27] or FPGAs [62], perceive these properties as limitations. For combined attackers, who can dispatch between CPU and secondary device, they enable otherwise infeasible techniques. Ultimately, this leads us to challenge common assumptions, as summarized in Table 2. When relevant, we instantiate secondary devices with FPGAs.

Contributions. Summarized, our main contributions are:

- We explore key primitives for combined cache attacks and discover a new DDIO-related structure in the LLC.
- We develop a fast and reliable procedure for secondary devices to find eviction sets in non-inclusive Intel caches.
- We leverage precise LLC manipulation for reliable eviction with fewer congruent addresses than there are ways.
- We design an FPGA accelerator that implements the aforementioned techniques and make it openly available:

https://github.com/KULeuven-COSIC/Double-Trouble

Table 2: Challenging common understanding

Common Understanding	Our Finding
Eviction set construction reached speed limits [60]	**Accelerating Eviction Set Construction (Section 4)**
Secondary devices only allocate to DDIO region in LLC [18, 27, 62]	**Discover undocumented DDIO+ region (Section 5)**
Non-inclusive LLC: needs directory conflicts [66]	**Eviction from private cache through LLC (Section 6.1)**
Cannot evict from remote socket without `flush` *[22, 68]*	**Flushless cache attacks across sockets (Section 6.1)**
Minimal eviction set is as large as associativity [33, 60]	**Reliable Eviction with Tiny Eviction Sets (Section 6.2)**
Amplitude-based encoding precluded by self-eviction [37]	**Modulation and Multi-bit Symbols (Section 8.2)**

2 Background

2.1 Heterogeneous Computing

As the limits of general-purpose computers are pushed, domain-specific computation gains importance. Companies have started playing games with custom chips, combining CPUs with accelerators, e.g., for machine learning or networking. Instead of inefficiently increasing CPU core counts, these architectures complement CPUs with custom accelerators, offering high-performance computation, often at low power. Popular examples are Google's TPU and Apple's M1.

On the server side, FPGA-attached CPUs serve the grounds to play this game. With their hardware programmability, FPGAs allow users to implement custom accelerators for their specific needs. Some cloud providers (e.g., AWS) already provide homemade accelerators to customers, or a marketplace where accelerators can be sold or rented. CPU giants have acquired FPGA manufacturers (Intel-Altera, AMD-Xilinx), and are working on tight integration of CPUs and FPGAs.

Today, these platforms attach FPGAs to CPUs as PCIe accelerator cards. On the CPU side, kernel drivers and APIs enable applications to communicate with their hardware accelerators. On the FPGA side, Control and Status Registers (CSRs) offer basic data transfers, and Direct Memory Access (DMA) allows the FPGA to access system memory. In this paper, we focus on the latter, as it interacts with the CPU memory subsystem. Although we focus on FPGA-specific terminology, many conclusions carry over to other PCIe-connected devices, e.g., Network Interface Cards (NICs) or Thunderbolt.

2.2 Cache Organization

Modern cache hierarchies comprise multiple levels. Lower cache levels are closer to the CPU, and are usually smaller and faster than higher levels. Typical Intel processors have three cache levels, with L1 and L2 caches private to each core, and the last-level cache (L3, or LLC) shared between cores.

Caches are organized as arrays of *cache lines* of, typically, 64 bytes. Most caches are *set-associative*, meaning that they are partitioned in *sets*. Each cache line maps to exactly one set, based on an indexing function applied to their address. The *associativity* refers to the number of lines that can reside simultaneously in the same set, i.e., the number of *ways*, *W*.

When a requested line is not present in a cache (i.e., a *cache miss*), it is usually installed after propagating the request to the next level. In the absence of empty ways, the *cache replacement policy* determines the line to *evict* in favor of the incoming one. Lines mapped to the same set are *congruent*. Contending for the same resource, they can evict each other.

The *inclusion* invariants of the cache hierarchy determine whether cache lines can reside simultaneously in multiple levels. A cache is *inclusive* w.r.t. another (lower-level) cache if every line in the latter must also be present in the former. *Exclusive* caches cannot have lines in common. Caches that do not satisfy either invariant are *non-inclusive*. Historically, Intel LLCs are inclusive, but to keep up with increasing core counts, non-inclusive LLCs are becoming commonplace [38].

Contemporary LLCs are partitioned in *slices*, with an undocumented and architecture-dependent mapping [36]. For large core counts, the slices are interconnected with a mesh architecture [38]. High-end systems can have multiple CPU sockets, connected with a coherent memory hierarchy.

2.3 Data-Direct IO (DDIO)

Direct Cache Access (DCA) [15] is a mechanism developed for the fast exchange of Ethernet frames between CPUs and Network Interface Cards (NICs). Instead of accessing main memory, NICs interact directly with the CPU's LLC to alleviate memory bottlenecks and cache thrashing, improving I/O performance [19, 29, 34, 9, 10]. DDIO [18] is Intel's implementation of DCA, available on server-grade CPUs. It is enabled on such CPUs by default, and PCIe devices (NICs, FPGAs, etc.) transparently interact with the LLC instead of memory.

Unfortunately, specific DDIO behavior is largely undocumented, especially for non-inclusive cache hierarchies. Some works partially reverse-engineer it [62, 27]. Section 3.2 covers known and yet unknown DDIO behavior in detail.

2.4 Cache Attacks

The observation that the execution time of a program depends on its control flow and interaction with the cache hierarchy characterized the first generation of cache attacks [26, 43, 3]. Later, the case was made for co-located attackers, i.e., those running code on the same physical platform as potential victims. By manipulating and observing the cache state, they observe much more fine-grained access patterns [44, 41].

Arguably the strongest technique is FLUSH+RELOAD, where the attacker flushes a target line from the cache (e.g., using `clflush` on `x86`), and later reads it to determine whether the victim accessed it in the meantime. In the absence of `clflush` [30], an attacker can evict the shared line instead, which is referred to as EVICT+RELOAD [13]. Both techniques require shared memory with the victim (e.g., KSM [2]). In contrast, PRIME+PROBE only relies on cache contention. In particular, the attacker occupies an entire cache set with her own lines, waits, and afterwards loads these lines again. If another process has accessed lines congruent to those of the attacker, this will be reflected in the attacker's access latency. Due to its low requirements, PRIME+PROBE has been mounted from restricted environments [40, 27].

The target cache needs to be shared between attacker and victim. Initial attacks considered same-core attackers and targeted the L1 cache [44, 41]. Later attacks managed to target the LLC [70, 33, 21], enlarging the threat to cross-core attacks.

Until recently, cross-core EVICT+RELOAD and PRIME+PROBE relied explicitly on the inclusive nature of the LLC. In this case, eviction from the LLC implies invalidation in all lower-level caches to preserve the inclusion invariant [33, 21]. This allows to evict lines from other cores' private caches.

Non-inclusive Caches. In non-inclusive caches, lines in L2 are not necessarily present in the LLC. In fact, they rarely are, since loads from memory are installed in L1/L2, skipping the LLC. Conversely, contention on the LLC alone does not invalidate lines in the lower-level caches of other cores. To overcome this problem for non-inclusive Intel CPUs, Yan et al. [66] propose contention on the *coherence directory* (CD), also referred to as the *snoop filter*. The CD tracks lines present in lower-level caches, and is inclusive to accelerate coherence transactions with other cores [72]. On Intel CPUs, the LLC and CD share the same slice and set mapping.

This paper considers cross-core attacks in non-inclusive Intel caches. The targets are the LLC and CD, and lines are *congruent* when they share the same LLC/CD set and slice.

2.5 Eviction Set Construction

Prior to a Prime+Probe or Evict+Reload attack, the attacker constructs *eviction sets*, i.e., sets of congruent addresses. If physical addresses and their mapping to LLC/CD sets and slices are known, finding congruent addresses is trivial.

In practice, however, the attacker is limited on both fronts. First, unprivileged processes observe virtual addresses, organized in 4 KiB or 2 MiB pages, and do not know the virtual-to-physical address translation. Therefore, physical address control is limited to the page frame bits, which are unaffected by translation (cf. Figure 1). Second, the slicing function is undocumented and architecture-dependent [36, 66].

Liu et al. [33] construct eviction sets for inclusive LLCs. Vila et al. [60] accelerate it by improving the time complexity from quadratic to linear. Yan et al. [66] find LLC/CD eviction sets in non-inclusive Intel caches. To that end, they introduce helper sets that are congruent in L2 but not the LLC. Techniques for non-inclusive caches are currently underdeveloped w.r.t. inclusive caches. Moreover, because of the indirect interaction with the LLC, their noise-resilience is unclear.

2.6 Experimental Setup

We work remotely on Intel Labs (IL) Academic Compute Environment (ACE), with dual-socket Xeon Platinum CPUs (28 cores/slices per socket). We also use a local lab setup, with dual-socket Xeon Silver CPUs (8 cores/slices per socket). All platforms have non-inclusive LLCs. The platforms utilize Intel's PCIe-based FPGA accelerator cards called Programmable Acceleration Cards (PACs), either with Arria 10 (A10) or Stratix 10 (S10) family FPGAs. Table 3 summarizes the platforms and their cache hierarchy.

A basic FPGA design can transparently interact with the memory subsystem over DDIO. At a high level, it can read and write to memory, and distinguish between access latencies (L2/LLC/RAM) based on immutable timing sources. A detailed description of our implementation is deferred to Section 7.

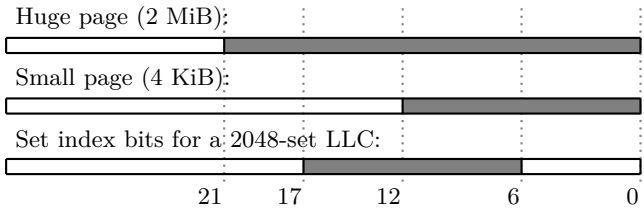


Figure 1: Control over cache set index depends on page sizes

Table 3: Platforms Used for Experimentation

Platform	CPU *Intel Xeon*	Arch.	Core Count	FPGA *Intel PAC*
ACE 1	Plat. 8180	SKL-SP	28	A10
ACE 2	Plat. 8280	CLX-SP	28	S10 (x2)
Local	Silver 4208	CLX-SP	8	A10 (x2)

SKL: Skylake, CLX: Cascade Lake

Cache	Info	Ways	Size *per Core*
L1	Core-Private	8	32 KB
L2	Core-Private	16	1 MB
LLC	Shared, Non-Inclusive	11	1.375 MB

3 Double Trouble: Combined Cache Attacks

3.1 Threat Model

The main threat model in this work is the combined attacker ($\mathcal{A}_{\texttt{CMB}}$). As indicated in Figure 2, she controls at least one CPU core and a secondary device connected over DDIO. In our case, an FPGA is used as the secondary device. The attacker can dispatch operations to software and hardware, and share memory between them. For completeness, we also consider traditional attackers ($\mathcal{A}_{\texttt{STD}}$) without a secondary device.

The attacker has no privileges and does not know the slice mapping. We do not assume the availability of a `clflush` instruction (in accordance to, e.g., [66]). Although our findings do not strictly require huge memory pages, we

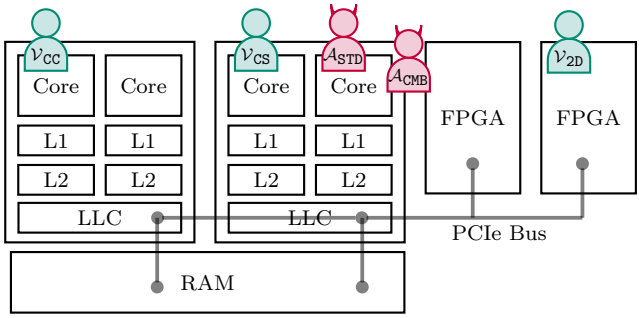


Figure 2: Combined attackers control a CPU process and secondary device. We consider three victim types ($\mathcal{V}_{\texttt{CC}}$,$\mathcal{V}_{\texttt{CS}}$,$\mathcal{V}_{\texttt{2D}}$).

assume them to be available, as they are enabled by default on server-grade platforms with FPGA acceleration (e.g., OPAE [20]).

To navigate the heterogeneity in attacker and victim properties, Table 4 summarizes our main results and indicates the configurations to which they apply. We distinguish between the degree of co-location: attacker and victim running on different cores ($\mathcal{V}_{\texttt{CC}}$), on different sockets ($\mathcal{V}_{\texttt{CS}}$), or a victim secondary device attached to the attacker socket ($\mathcal{V}_{\texttt{2D}}$).

Section 4 introduces a new algorithm for swift and reliable eviction set construction, overturning the limitations of secondary devices. Section 5 exposes undocumented behavior in the LLC, which we use to obtain an intra-cache-set granularity. Section 6 shows how standard attackers can evict shared lines without CD contention, and how combined attackers can do so with tiny eviction sets. These results require shared memory between the attacker and victim. Section 7 describes our implementation of an FPGA hardware accelerator, which is evaluated in Section 8.

3.2 Key Properties

3.2.1 Spatially Limited Interaction With LLC

To prevent thrashing, DDIO devices only interact with a fraction of the last-level cache (LLC) [18]. Lines *read* by the secondary device are not allocated in the LLC [27, 62], but they are served from the LLC if already present. Lines *written* by the secondary device are allocated, but only to a limited number of ways (two by default) in every set [10, 27].

In contrast to previous suggestions [18, 27], we find that the replacement policy is not LRU (cf. rationale in Appendix C).

Non-Default DDIO Configurations. The number of LLC ways to which DDIO can write-allocate is, by default, two [10, 27]. However, this can be configured in the `IIO_LLC_WAYS` Model Specific Register (MSR) [32, 10], with a bitmask that represents the cache ways used by DDIO. The minimal and default setting is `0x600`. More ways can be activated by setting more bits, provided that the selected ways are consecutive. In this paper, we denote the number of DDIO ways as D.

Table 4: Applicability of the main results of this work

Capabilities	Attacker		Victim			Shared Memory
	$\mathcal{A}_{\texttt{STD}}$	$\mathcal{A}_{\texttt{CMB}}$	$\mathcal{V}_{\texttt{CC}}$	$\mathcal{V}_{\texttt{CS}}$	$\mathcal{V}_{\texttt{2D}}$	
Eviction Set Finding		✓	✓	✓	✓	
Intra-set Granularity		✓	✓		✓	
Eviction without CD	✓	✓	✓	✓		✓
Reduced Eviction		✓	✓	✓		✓

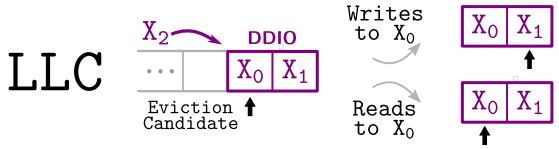


Figure 3: Secondary reads do not change eviction candidates.

The default configuration ($D=2$) is, arguably, the most important to study from a security point of view. In accordance with prior studies [27, 62], we focus our attention on this configuration, and assume it unless otherwise indicated. However, when relevant, we generalize our findings to $2 \leq D \leq W$.

Property #1: Spatially limited LLC interaction.
Secondary devices interact directly with the LLC. Only writes trigger cache line placement, which is statically constrained to a limited number of ways (two by default).

3.2.2 Reading Without Consequences

The property that secondary devices do not read-allocate in the cache has an underappreciated corollary: it allows attackers to *query the cache state without disturbing it*. We illustrate the implications of this power with two relevant examples.

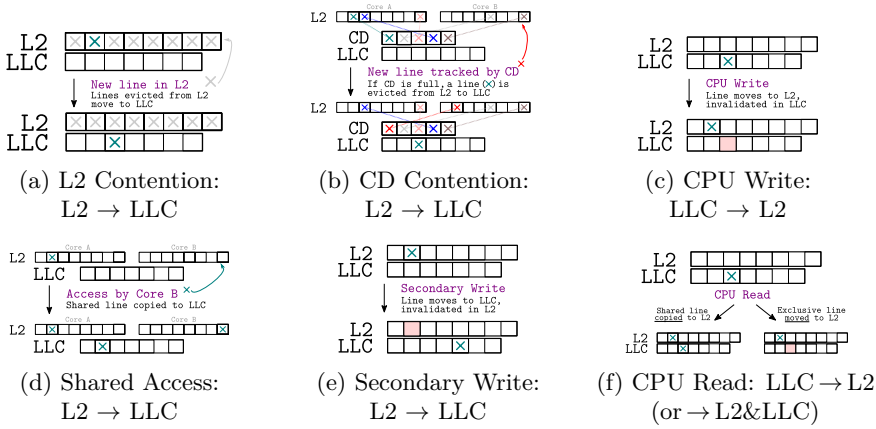
Example: Counting LLC Entries. Consider how to infer how many (out of N) lines are cached in the LLC. This can be done by measuring the access latency of all N lines, counting those within the predetermined LLC timing range.

CPU-only attackers are limited in their accuracy. Consider the case where at least one line is not cached. Measuring the access latency of this line allocates it in the cache. In this process, it may evict the other lines, perturbing the measurement.

The combined attacker, in contrast, can measure the cache state *reliably*, i.e., a partial measurement does not endanger the validity of the full measurement, and *repetitively*, i.e., several measurements of the same state can be combined.

Example: Eviction Candidate. For lines already in the LLC, do secondary reads influence the *eviction candidate*, i.e., the line to be evicted upon installation of a new congruent line?

Consider Figure 3, where X_0, X_1 and X_2 are congruent lines. X_0 and X_1 are placed in the DDIO region with secondary writes. X_1 is written repeatedly to ensure that X_0 is the eviction candidate. Indeed, we observe that writing X_2 evicts X_0.



(a) L2 Contention: L2 → LLC

(b) CD Contention: L2 → LLC

(c) CPU Write: LLC → L2

(d) Shared Access: L2 → LLC

(e) Secondary Write: L2 → LLC

(f) CPU Read: LLC → L2 (or → L2&LLC)

Figure 4: Techniques for combined attackers to manipulate the cache hierarchy.

In a first experiment, the secondary device *writes* a few times to X_0. We observe that placement of X_2 now evicts X_1, so the earlier write to X_0 changed the eviction candidate to X_1. The second experiment performs several *reads* of X_0. If the replacement policy records these reads, the eviction candidate should change to X_1. Still, we find that placing X_2 always evicts X_0. We conclude that secondary reads, in contrast to writes, do not influence the LLC replacement policy state.

Finding #2: Non-destructive secondary reads.
Secondary reads are non-destructive. Reading uncached lines does not trigger cache allocation, and reading LLC lines does not influence their replacement policy state.

3.2.3 Two-sided Cache Hierarchy Manipulation

Combined attackers can approach non-inclusive cache hierarchies from two sides. We now cover known and novel primitives to trigger movement between L2[1] and the LLC. Moving or copying lines to the LLC is useful, as lines evicted from the LLC are invalidated in all cache levels [66]. Moving lines from the LLC to L2 is useful to invalidate specific LLC ways. Our repository supports these findings with experiments.

L2 to LLC. The most straightforward way to move an L2 line to the LLC is to access sufficient addresses mapped to the same L2 set (cf. Figure 4a). This

[1]Since L2 is inclusive w.r.t. L1, for our purposes they can be consolidated.

technique can only be used by processes running on the core tied to the specific L2 cache.

Yan et al. [66] demonstrate the eviction of lines from remote private caches, i.e., L2 caches associated with other cores. They do this by generating contention on the coherence directory CD (cf. Figure 4b), the inclusive structure co-located with the LLC that keeps track of lines in L2 caches.

On our test platforms, we observe that when a line is accessed by two processes on different cores, it is copied to the LLC. The line then co-exists in the LLC and both private L2 caches (cf. Figure 4d). We find this novel technique to be accurate and simple to move a target line to the LLC. However, note that this primitive is limited to attacker-readable lines.

Specifically for the combined attacker, there is yet another primitive available (cf. Figure 4e). The secondary device writes the line, which moves it to the LLC. Now dirty, the line is invalidated from all other caches to maintain coherence. Note that this primitive is limited to attacker-writable lines.

LLC to L2. A line can be moved from the LLC to L2 by read or write requests from CPU cores. Lines that are written (Figure 4c) move from the LLC to L2, invalidating the LLC line for coherence purposes [67]. If the line is read (Figure 4f), we find that lines that only exist in the LLC (e.g., in *modified* or *exclusive* state) move back to L2 and are invalidated in the LLC. In contrast, lines present in multiple caches (e.g., in *shared* or *forward* state) are *copied* to L2, i.e., they remain allocated in the LLC. As these observations are partially inconsistent with prior work [66, 67], we share our rationale in Appendix B.

Replacement policies prefer to install incoming lines in empty ways (if present) to avoid unnecessary eviction of useful lines. Some primitives are able to produce empty ways by invalidation. We refer to these ways as *magnet ways*.

Finding #3: Precise manipulation of cache hierarchy.
Combined attackers can accurately migrate lines between cache levels, and invalidate them from selected caches.

4 Fast Eviction Set Finding using DDIO

The eviction set construction problem is the following: given a target cache line, find EV congruent lines in the designated cache (of associativity W). We review existing algorithms for the LLC, and expose a new method for combined attackers, enabled by limitations of secondary devices (**#1**, **#2**).

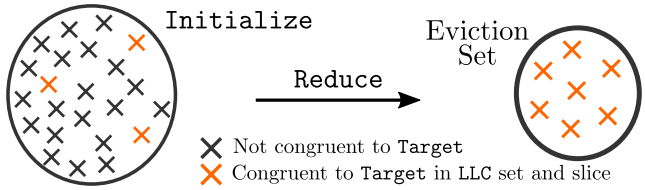


Figure 5: Traditional reduction-based algorithms

4.1 Reduction Algorithms

Figure 5 depicts the structure of traditional eviction set construction algorithms. First, an initial set of candidate addresses is constructed [33], for which the size depends on attacker control over physical address bits [60]. Then, it is reduced to a *minimal* eviction set, i.e., a set, typically of size EV $\geq W$, that no longer contains any non-congruent addresses.

The reduction is an iterative procedure based on a *congruence test* that removes a portion of the current set and tests whether the remainder still evicts the target. If so, the portion is not necessary for eviction and can be discarded. Initial algorithms for inclusive caches [33, 21, 40] remove one element at a time, leading to quadratic complexity in the initial set size. Vila et al. [60] propose to perform the congruence test on groups of addresses instead, achieving linear complexity.

Yan et al. [66] develop a reduction algorithm for non-inclusive Intel LLCs with a custom congruence test (cf. Section 2.4). Their algorithm has quadratic complexity, but may be amenable to similar improvements [60]. However, it appears to need several eviction and measurement iterations to cope with the complications of non-inclusive caches.

4.2 Acceleration with `Discover-Expand`

Secondary devices generally perceive a smaller LLC associativity (**#1**), with $D=2$ by default. As a result, D+1 congruent addresses are sufficient to manifest contention. In addition, the non-destructive reads (**#2**) enable a congruence test that determines whether a single address *is* congruent with a target, instead of whether a pool of addresses *contains* some that are congruent. These properties lead to an effective search algorithm, based on expansion rather than reduction (cf. Figure 6).

The algorithm takes as input a target address (TARGET), desired eviction set size (EV), and stride representing attacker-controllable address bits (STRIDE). The output is an eviction set, i.e., EV addresses that are mapped to the same LLC set and slice as TARGET. The algorithm does not build an initial set, and comprises two phases: `Discover` obtains the first $D-1$ congruent addresses,

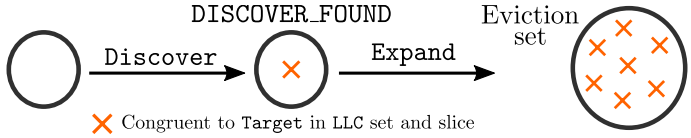


Figure 6: Expanding eviction set construction. For $D=2$, `Discover` only needs to find a single congruent address.

and `Expand` completes the eviction set.

Algorithm 1 Eviction Set Construction: `Discover` finds the first D–1 congruent addresses, `Expand` finds the others

Input: `TARGET`: an address for the eviction set, `EV`: desired eviction set size, `STRIDE`: indicates attacker-controlled bits
Output: `es` eviction set

```
1:  es[] ← empty list
2:  do                                                  ┐ Discover
3:      secondary_write(TARGET)                         │
4:      secondary_write(es[0,1,.,len(es)-1])            │
5:      do                                              │
6:          TEST_ADDRESS ← new candidate                │
7:          secondary_write(TEST_ADDRESS)               │
8:      while secondary_read(TARGET) is fast            │
9:      append TEST_ADDRESS to es                       │
10: while len(es) < D-1                                 ┘
11: do                                                  ┐ Expand
12:     secondary_write(TARGET)                         │
13:     secondary_write(es[0,1,.,D-2])                  │
14:     do                                              │
15:         TEST_ADDRESS ← new candidate                │
16:         secondary_write(TEST_ADDRESS)               │
17:     while secondary_read(TARGET) is fast            │
18:     append TEST_ADDRESS to es                       │
19: while len(es) < EV                                  ┘
```

a new address from search space with `STRIDE`

`DISCOVER_FOUND`: addresses found in `Discover`, `es[0,1,.,D-2]`

4.2.1 `Discover`: Finding the First $D-1$ Addresses

First, the `Discover` phase writes TARGET, installing it in the LLC's DDIO ways. Then, it iteratively writes a new candidate address and afterwards measures the access latency of TARGET. If the latter has been evicted (from LLC to RAM), the candidate is determined to be congruent with it.

As the DDIO associativity is D, it is expected that $D-1$ congruent test

addresses are overlooked for every congruent candidate that is actually added to the eviction set. However, as soon as $D-1$ elements have been found (a set denoted as DISCOVER_FOUND), the `Discover` phase can terminate in favor of the more efficient `Expand` phase.

4.2.2 `Expand`: Finding More Addresses

The `Expand` phase writes TARGET and DISCOVER_FOUND to occupy all D DDIO ways. Then, similar to `Discover`, new candidates are written, each time measuring the access latency of TARGET to determine whether the candidate is congruent. This step is repeated to obtain the full eviction set.

4.2.3 Algorithmic Complexity

The stride of the search depends on control over physical address bits (cf. Section 2.5) and the number of LLC/CD slices (cf. Section 2.2). As the slice mapping is unknown, candidates can, at best, be selected based on their LLC set index bits.

Our test LLCs have 2048 sets with 8 or 28 slices (SLICES). Huge pages allow to configure all set index bits, so the congruence probability for candidate addresses is SLICES^{-1}, assuming lines are distributed among slices uniformly at random.

The congruence probability directly relates to the expected number of candidates to test in the `Expand` phase. In the `Discover` phase, however, the first $D-1$ congruent candidates may be missed. In first order, the expected number of candidates to test is $(D-1)\cdot[D\cdot\mathsf{SLICES}]+(W-D+1)\cdot[\mathsf{SLICES}]$. Section 8.1 evaluates accuracy and speed in practice.

Algorithm 2 Verification algorithm to check congruence

Input: `TARGET`, `DISCOVER_FOUND`, `CANDIDATE`
Output: boolean: are the inputs congruent?

1: secondary_write(`TARGET`);
2: secondary_write(`DISCOVER_FOUND`);
3: secondary_write(`CANDIDATE`);
4: **return** true **if** secondary_read(`TARGET`) is fast

`TARGET` and `CANDIDATE` are interchanged in consecutive runs

4.2.4 Low-level Aspects

Replacement Policy. The simplified algorithm assumes LRU replacement. However, we find that repeated writes influence the eviction candidate (cf. Appendix C). To ensure that TARGET is the eviction candidate, the accelerator performs the `secondary_write` on lines 4/13 of Algorithm 1 twice.

Table 5: Access sequences to cache line `L` to prepare its state, with the resulting cache level for `L`, whether `L` is modified, and whether a secondary write allocates `L` to the DDIO region

Access Sequence	Level	Modified	DDIO
① `sw_flush(L)`	RAM		✓
② `sw_write(L)`	L2	✓	
③ `sw_flush(L); sw_read(L)`	L2		✓
④ `hw_write(L); sw_read(L)`	L2	✓	

(`sw_*` and `hw_*` denote actions by CPU and secondary device, resp.)

Verification. To increase robustness against false positive errors (e.g., due to noise), Algorithm 2 optionally performs additional checks. It is called with TARGET, DISCOVER_FOUND and the address to be tested, and performs two permuted verification tests. This way, a false positive candidate will only pass if there is noise in two different cache sets.

As the verification needs $D+1$ addresses (including TARGET), the accuracy of the DISCOVER_FOUND set itself is confirmed together with the first address of the `Expand` phase. If it fails, both addresses are discarded and `Discover` is restarted. As soon as DISCOVER_FOUND is verified, all subsequent addresses can be verified in isolation. To increase confidence, multiple verification repetitions can be performed.

Write Limitation. The routine requires write access to TARGET. However, an LLC-congruent address to the intended target can be used as input to the algorithm (cf. Section 8.3).

5 Structure of the LLC Set

This section revisits the interaction between DDIO devices and the LLC (**#1**). Secondary writes to uncached lines are known to allocate in the LLC, specifically to one of the DDIO ways [27, 62, 10]. It is worth investigating whether this behavior holds for lines that are already cached, e.g., in L2. Contrary to expectation, it appears to depend on the state of the line.

Another Region in the LLC Set. Consider a test set of N LLC-congruent addresses ($N \geq 2$). To fix their cache level and state, the elements of the test set first undergo the access sequences (① - ④) as given by Table 5. Then, they are written by the secondary device. Finally, the secondary device counts how many remain in the LLC (cf. Section 3.2.2). All read and write operations are repeated to remove the influence of the replacement policy, and we consider 10 000 measurements.

For all access sequences, we observe that subsequent secondary writes allocate to the LLC, consistent with current understanding. However, consider what

happens as we now apply contention to the DDIO region. This contention is achieved with secondary writes to another uncached contention set (access sequence ①, which is known to lead to DDIO allocation).

For sequences ① and ③, no test lines remain after DDIO contention (i.e., they were allocated to the DDIO region). However, for sequences ② and ④, both addresses remain in the LLC (i.e., they must have been placed *elsewhere*). In conclusion, cache lines in specific states are secondary write-allocated to the LLC, but *outside of the DDIO region.*

In the remainder of this work, we refer to this unknown region as the DDIO$^+$ region. We can only speculate on the condition to be placed in this region, but we observe modified lines to be assigned to it (e.g., sequences ② and ④). The rationale for this behavior is not immediately obvious and we assume it to be an undocumented performance optimization.

Associativity. We now infer the structure of the LLC set, starting with the associativity of the DDIO and DDIO$^+$ subregions.

For the DDIO region, the secondary device writes N congruent addresses and counts how many remain in the LLC. The associativity is the largest N for which none are consistently evicted. As expected, we observe an associativity of D.

For the DDIO$^+$ region, we perform a similar experiment, but with access sequence ④ preceding the secondary write. We observe a DDIO$^+$ associativity of 2, irrespective of D.

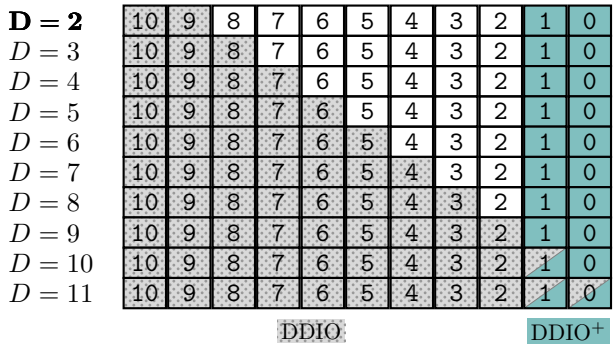
The LLC set. To learn which ways belong to the DDIO/DDIO$^+$ region on the Xeon Silver, we extend the mapping technique of Farshin et al. [10]. In particular, it uses Intel CAT [19] to let a software process evict specific LLC ways as specified by a bitmask. The correlation between the bitmask and evictions of DDIO/DDIO$^+$ lines reveals the composition of these regions.

For the DDIO region, we use a CAT mask that spans D ways, i.e., the DDIO associativity. First, the secondary device writes D test lines. Then, software generates contention in the ways specified by the mask. Finally, the secondary device counts the test lines in the LLC. The mask corresponding to the D leftmost ways results in all test lines being evicted; shifting it one to the right evicts $D-1$ lines, etc. No lines are evicted if the mask does not overlap with any of the D leftmost ways.

For the DDIO$^+$ region, we use a similar methodology with a 2-way CAT mask. Access sequence ④ is used to produce lines in the DDIO$^+$ state. The CAT mask corresponding to the 2 rightmost ways results in both DDIO$^+$ lines being evicted.

LLC Model. Section 5 summarizes the inferred LLC structure for our local platform (Xeon Silver). CPU memory traffic allocates to all ways in the set. Secondary devices only write-allocate to the DDIO or DDIO$^+$ regions (depending on the state of the written line). The DDIO region is contiguous and has

associativity D, growing from the most-significant ways. The DDIO$^+$ region is contiguous and has associativity 2, and covers the least-significant ways. In the event that the DDIO and DDIO$^+$ regions overlap (i.e., for $D \geq 10$), the least-significant ways accommodate both lines in the DDIO and DDIO$^+$ state.



Finding #1 (rev.): Spatially limited LLC interaction.
Another portion of the LLC set is malleable by secondary devices. This region (DDIO$^+$) has associativity two.

6 Revisiting Cache Eviction

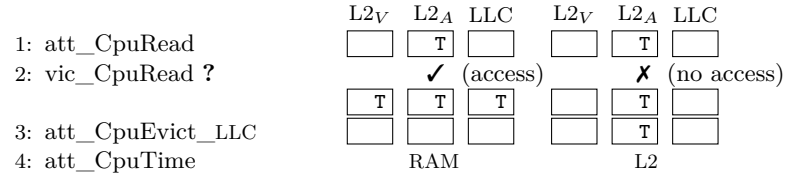
With magnet ways (**#3**), this section challenges the concept of *minimal* eviction sets through efficient eviction with fewer elements than the cache associativity (**#1**). First, as a stepping stone of independent interest, we evict shared lines from a victim's private caches without directory contention. For simplicity, we consider the default DDIO configuration ($D = 2$).

6.1 Eviction without Coherence Directory

Algorithm 3 evicts a shared line from remote victim caches with the shared access method. First, the attacker prepares the target in L2, and waits. If the victim accesses the target, it resides in attacker and victim L2, *and* the LLC (cf. Figure 4d). Second, the attacker evicts it from the LLC which, if it was indeed there, invalidates the copies in all L1/L2 caches. Otherwise, the target remains in the attacker's L2. This invalidation is not strictly required for non-inclusive LLCs, but happens in practice [67]. LLC eviction (line 3) is implemented by accessing W lines with threads on different cores (Figure 4d).
Algorithm 3 can also be inverted, making it slightly more complex (details in Appendix D). Surprisingly, the inverted version can evict lines from L2 caches

Algorithm 3 Eviction without Coherence Directory (CD)

On the right, the LLC and L2 states of victim ($L2_V$) and attacker ($L2_A$) are shown for each operation on the target address (`T`).



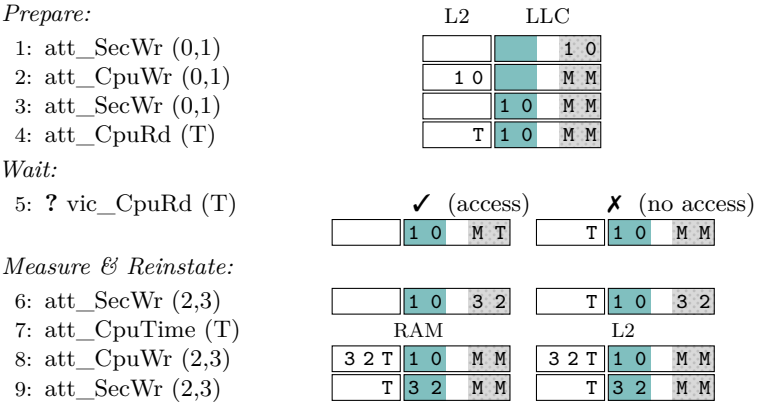
in remote sockets.

6.2 Reduced Eviction

Combined attackers can produce magnet ways, i.e., empty ways to attract incoming lines (cf. Section 3.2.3). Assuming a magnet way in the DDIO region, a combined attacker can evict a target line from a victim cache by, first, triggering its LLC allocation (where the magnet will attract it) and second, evicting it from the DDIO region with only two congruent addresses. Without magnet ways, the target may be installed anywhere in the LLC set, and W addresses are needed to reliably evict it.

Algorithm 4 Eviction with Reduced Eviction Set

On the right, the attacker's L2 and LLC (with DDIO$^+$ and DDIO) are shown, with lines 0-3, LLC magnet ways (`M`), and target (`T`) .

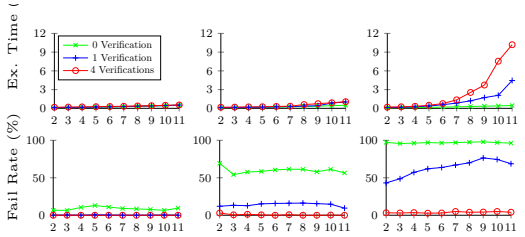


Reiterate from line 5, swapping the roles of `0-1` *and* `2-3`.

Algorithm 4 shows how to produce and exploit DDIO magnet ways using a

Stress Level	# of Verf.	Ex. Time (ms)	Fail Rate (%)
No	0	0.10	6.72
	1	0.12	0.00
	4	0.17	0.00
-m 1	0	0.09	69.27
	1	0.12	11.99
	4	0.17	3.00
-m 8	0	0.08	97.25
	1	0.13	43.15
	4	0.19	3.24

(a) Detailed evaluation for $D = 2$ (default)



(b) The effect of D (on X axis) on the performance and accuracy.

Figure 7: Performance of HW-accelerated eviction set construction on our Xeon Silver setup (EV $= W$, avg. of 1000 runs).

tiny eviction set of four addresses (`0-3`). Strictly speaking, two congruent lines suffice. However, to make eviction repeatable, we suffer from DDIO$^+$ behavior and rely on our model LLC structure to overcome it (cf. Section 5).

First, the secondary device writes lines `0-1`, followed by CPU writes. Afterwards, the secondary device writes them again, while the CPU reads the target line. This terminates the preparation phase, with the target in the attacker's L2, two magnet ways in the DDIO region, and lines `0-1` in DDIO$^+$.

Then, the attacker waits. If the victim accesses TARGET, it moves to the LLC DDIO region, attracted by the magnet ways. Afterwards, secondary writes to `2-3` evict the DDIO region (`0-1` cannot serve this purpose, as they are modified and allocate to DDIO$^+$ instead). A CPU timing measurement of TARGET reveals it to reside either in L2 (no victim access) or in RAM (victim access). Finally, `2-3` are written by the CPU and the secondary device, recreating the magnet ways in the DDIO region and placing `2-3` in DDIO$^+$ instead of `0-1`. The next iteration can now start, and `0-1` swap roles with `2-3`.

The reduced eviction can also be inverted, making it work across CPU sockets on our platforms (cf. Appendix D).

7 Implementation

We work with the FPGA-accelerated cloud platforms of Section 2.6 and implement a hardware (HW) module to demonstrate our findings. This section explains its functionality.

Read-Write Primitives. As we work with Intel FPGAs, we use Open Programmable Acceleration Engine (OPAE) to integrate FPGA acceleration into software applications. OPAE divides the FPGA's programmable fabric into two parts; a blue-bitstream pre-programmed by Intel, and a green-bitstream

that implements the user's hardware accelerators. The blue-bitstream acts as a bridge between accelerators and software, and provides them with Direct Memory Access (DMA).

DMA and a timing source are essential components for cache-timing experiments. With OPAE, a DMA operation consists of two transactions; one for asserting memory read and write requests, and another to monitor the completion of this request. To measure latency, we create a counter-based timing source on the FPGA, similar to Weissman et al. [62]. It counts the cycles that expire between requests and replies (at 400 MHz), allowing to distinguish the memory level that serves the request (L2, LLC or RAM), as given in Appendix A. The FPGA counter is not synchronized to the CPU counter.

The hardware design also features a set of software-configurable registers to instruct the actions of the accelerator, e.g., to perform a timed read or write at a given address. More sophisticated instructions are described next.

Fast Eviction Set Finding. We extend the hardware module with an advanced state machine that implements the fast eviction-set finding algorithm introduced in Algorithm 1.

The algorithm is fully encapsulated in hardware and proceeds without additional interaction. Moreover, a few settings are exposed to users. Essential settings are the target for which to find the eviction set, the desired size, the number of verification repetitions, and the timing threshold (LLC vs. RAM accesses). The delay between consecutive memory operations can also be configured to ensure their in-order execution.

Access Sequences. The hardware module supports encapsulating commonly used sequences of read and write operations to reduce HW-SW interaction overhead, e.g., for fast reduced eviction (cf. Section 6.2). Again, the configuration of these sequences happens with software-accessible registers.

8 Evaluation and Discussion

This section evaluates three applications of our accelerator, demonstrating: (1) the speed and accuracy of eviction set construction, (2) a covert channel encoding information in the number of evicted lines, and (3) reduced eviction in practice.

8.1 HW-Accelerated Eviction Set Finding

8.1.1 Performance

Figure 7 presents the performance and accuracy of hardware-accelerated eviction set construction, measured on our local Xeon Silver platform (8 slices, cf. Section 2.6). We consider sets of size $\text{EV} = W = 11$, and conduct the measure-

ments on both idle and noisy systems. For the latter, we use `stress` to emulate moderate and high noise, resp. with `-m 1` and `8`.

We report *end-to-end* execution time, i.e., we do not exclude any preparation steps or interactions with hardware. The existence of even a single false positive in the set classifies it as *failed*, even if the remainder of the set is correct. The candidate verification (cf. Algorithm 2) significantly reduces such false positives at the cost of slightly increased runtime.

Hardware-accelerated eviction set construction is very effective. On our local platform, the accelerator can find an LLC/CD eviction set in around 120 µs on idle systems, or 200 µs under stress. For the default DDIO configuration, the throughput of one accelerator (in EVS/s, eviction sets per second) reaches more than 8000 EVS/s for noise-free systems, and around 5000 EVS/s for very high noise. Such speeds allow to map out the entire 11 MB LLC in ≈2 s. For reduced eviction sets (EV = 4), the average throughput is around 16 kEVS/s. On the ACE platforms, the accelerator is roughly 3 times slower, since the slice count increases from 8 to 28.

Influence of D. For non-default DDIO configurations (i.e., $D > 2$), the performance of eviction set construction decreases with D. As D grows, the associativity perceived by the accelerator increases (**#1**), which increases the relative weight of the `Discover` phase. Especially for high noise pressure, this impacts the performance for two main reasons (cf. Section 4.2.3). First, during discovery, more guesses are required to detect a congruence, as detection may only occur for every D–1 congruent guesses. Second, false positives have adverse effects, as all DISCOVER_FOUND are discarded upon a failed verification test, in contrast to just a single discarded address during `Expand`. All in all, our eviction set construction is still accurate and faster than related work. Moreover, $D = 2$ is the default configuration. To our knowledge, no benchmarking tools exist to help users decide when to change it.

8.1.2 Comparison

Expansion-based Methods. As covered in Section 4.2, expansion-based methods work well when the congruence test is non-disturbing. To our knowledge, the only other expansion-based method[2] is our work on Prime+Scope [47], which obtains a repeatable congruence test by exploiting the properties of lines that are cached in multiple levels simultaneously.

Although the algorithm based on Prime+Scope bears a similar structure to Algorithm 1, it differs in several important aspects. First, it is concerned with a different threat model. Second, the non-destructive measurement relies on fundamentally different properties. Third, the Prime+Scope version for

[2] One exception is Prime+Prune+Probe [46], which applies to a new class of randomized protected caches that are not in use today.

non-inclusive Intel LLCs needs to orchestrate two attacker threads on different cores to allocate in the LLC (cf. Figure 4d). Finally, CPU-based attackers perceive the full LLC associativity, in contrast to the accelerator (e.g., $D=2$).

Reduction-based Methods. For reference, we also compare the accelerator to reduction-based methods. Yan et al. [66] proposed the state-of-the-art reduction algorithm for non-inclusive caches. As their code is not available, we are unable to fairly compare to them. Moreover, their work may be amenable to similar optimizations as those shown for inclusive caches [60]. To have a meaningful data point for comparison, we compare with the highly optimized implementations for *inclusive caches* by Vila et al. [60]. Arguably, this serves as an upper bound for performance in non-inclusive caches, because of the obstacles identified in prior work [66]. On the other hand, our accelerator is agnostic to LLC inclusion, so we expect it to apply to inclusive LLCs with similar performance.

Results. Like the accelerator, we measure the performance of the Prime+Scope [47] code on the local Xeon Silver platform. For the Vila et al. [60] code, we match their CPU and configuration. Importantly, the number of slices is the same for both platforms under consideration (cf. Section 4.2.3).

As Table 6 shows, our HW accelerator achieves good end-to-end performance. Compared to the algorithm based on Prime+Scope (P+S), it is another order of magnitude faster. Depending on the noise level, it is between two and three orders of magnitude faster than reduction-based methods, at the cost of a small decrease in accuracy.

Note, however, that the threat model for the hardware accelerator is different. It assumes a secondary device but no code execution, whereas CPU-only methods assume code execution (native or otherwise [40, 60]) but no secondary device.

8.1.3 Robustness, Simplicity and Stealth

Robustness. The accelerator maintains good performance for high noise (cf. Figure 7), with several contributing factors. The timing source is a noise-free HW counter, and interference with other processes is limited to the DDIO region. The algorithm itself is also robust. False-negative errors only increase the execution time, and most false-positive errors are detected by Algorithm 2. Provided that the `Discover` phase is successful, remaining false positives do not affect the other addresses in the `Expand` phase, as every address is individually tested for congruence. In contrast, reduction algorithms may have to implement backtracking [60] to avoid getting stuck, as false positives trigger the removal of congruent addresses.

Simplicity. Fully described by a few lines of pseudocode, the accelerator is simple to understand. It does not use hierarchy-specific techniques, e.g., directory contention or helper eviction sets [66]. The accelerator interacts with a D-way set-associative view of the LLC which, at least for small D, is a great

Table 6: End-to-end performance (1000 runs) of our accelerator compared to PRIME+SCOPE [47] on our local setup, and optimized reduction-based algorithms [60].

Impl.	CPU & Cache	Stress Level	Error Rate (%)	Exec. Time (msec)
Ours*	Skylake-SP Xeon Slv. 4208 11-way LLC Non-Inclusive	no	0.0	0.17
		-m 1	0.5	0.17
		-m 8	3.5	0.17
P+S [47]		no	0.0	1.11
		-m 1	0.0	1.24
		-m 8	0.3	3.56
[60]†	Skylake Core i5-6500 12-way LLC Inclusive	no	0.0	19
		-m 1	0.0	35
		-m 3	0.0	206

∗ for the default DDIO configuration with $D = 2$.

† with initial set size 120, while other works do not use an initial set.

simplification. Furthermore, it is oblivious to associativity, replacement policy and noise in low-level caches. The same FPGA bitstream is used on all our test platforms.

Stealth. Due to its speed, eviction set construction is hard to detect at runtime. Moreover, the timing source is implemented on the FPGA fabric, so accesses to it are invisible to the CPU or blue-bitstream. Finally, it is essentially a state machine, requesting read and write operations to the blue-bitstream and timing them. Its resource utilization is very low, making it a hard-to-notice attachment. Appendix F covers detailed utilization numbers for the accelerator, showing that it barely increases compared to Intel's Hello World baseline.

8.2 Amplitude-Based Covert Channel

We implement a covert channel between combined attackers, demonstrating the precise control over the cache hierarchy with only a few congruent addresses. It transfers information by manipulating DDIO ($D=2$) and DDIO$^+$ ways simultaneously and independently. Moreover, due to non-perturbing reads (**#2**), it reliably encodes amplitude information in the signal (i.e., the *number* of evicted ways), which performs poorly for traditional attackers due to self-eviction [37]. It does not use shared memory; the parties use their own eviction sets (agreed upon in advance, e.g., using HW-acceleration).

Figure 8 shows two versions; one over the DDIO region, and one that combines DDIO/DDIO$^+$ regions. Both consist of three stages. First, the receiver primes

	Receiver	Sender

PR	`CpuWr(C,D)`
	`SecWr(C,D)`
	`SecWr(A,B)`
		`CpuWr(3,4)`

TX		`SecWr(3/4)` `SecWr(1/2)`
PB	`SecTime(C,D)` `SecTime(A,B)`	

PR: Receiver puts `A,B` into DDIO, and `C,D` into DDIO$^+$.

TX: Sender may replace `A,B,C,D` with `1,2,3,4`.

PB: Receiver checks if `A,B` are still in DDIO, and `C,D` in DDIO$^+$.

Figure 8: Covert channel with prime (PR), transmit (TX) and probe (PB). Symbols are always transferred over DDIO, and optionally over DDIO$^+$ (optional operations underlined). Parties use their own eviction sets, resp. `A,B,`$\cdots$ and `1,2,`$\cdots$

(PR) the DDIO lines (if applicable also in DDIO$^+$). Second, the sender transmits (TX) a symbol by overwriting zero, one or two of the receiver's DDIO (and DDIO$^+$) lines. Third, the receiver probes (PB) its lines to determine how many have been evicted.

The DDIO channel encodes a ternary symbol, i.e., $\log_2 3 = 1.58$-bits per cache set. For transferring 512 packets of 256 symbols, we achieve 264 Kbps bandwidth (BW) with 2.26 % Symbol Error Rate (SER). The DDIO/DDIO$^+$ channel encodes two ternary symbols, i.e., $\log_2(3{\cdot}3) = 3.17$ bits per cache set. In this case, the BW is slightly lower at 211 Kbps with 2.20 %SER, because of extra interfacing with hardware.

Comparison. We use a shared timestamp counter to synchronize transmitter and receiver, as well as a known preamble. Our implementation is open to optimizations, e.g., common engineering practices like synchronization or error correction, or transmission over multiple sets.

Although the covert channel only serves to illustrate the fine-grained spatial capabilities of combined attackers, we briefly compare it with closely related implementations. Weissman et al. [62] build a covert channel from FPGA to CPU. It achieves 95 Kbps and, like ours, can be improved. Yan et al. [66] establish a cross-core channel using CD contention on a non-inclusive LLC, and achieve 0.2 Mbps. To our knowledge, the fastest cross-core covert channels with unshared memory achieve 2–4 Mbps [45, 42, 47]. Ours is an order of magnitude slower, mostly because of hardware interfacing overhead, but is open to improvements.

Table 7: Eviction patterns for shared lines, their eviction set size, eviction rate, number of accesses and attacker model

Pattern	EV	Ev. Rate	Accesses	$\mathcal{V}_{CC}$	$\mathcal{V}_{CS}$	$\mathcal{A}$
CD-11-9 [66]	11	≈ 25%	216	✓		$\mathcal{A}_{STD}$
CD-13-9 [66]	13	≈ 95%	234	✓		$\mathcal{A}_{STD}$
CD-14-9 [66]	14	≈100%	252	✓		$\mathcal{A}_{STD}$
L2-16-9/LLC-11 [66]	16+11	≈100%	144+22	✓		$\mathcal{A}_{STD}$
No CD (Alg. 3)	11	≈100%	22	✓	✓	$\mathcal{A}_{STD}$
Reduced (Alg. 4)	4	≈100%	6	✓	✓	$\mathcal{A}_{CMB}$

8.3 Reduced Eviction

Eviction Rate. Table 7 compares the eviction rates of our new patterns with those reported by Yan et al. [66] for shared lines. For all patterns, the goal is to evict a shared line, currently in another core's L2, from the hierarchy. We achieve a near-perfect eviction rate, with fewer accesses and the bonus of working across sockets (cf. Appendix D). Our CD-less eviction (Algorithm 3) implements LLC eviction with two threads each writing EV=11 addresses once, generating direct LLC contention. The reduced eviction implements Algorithm 4.

Yan et al. observe that using unshared lines to evict shared lines from remote L2 caches through the CD has unsatisfactory results. To overcome this, they use shared lines, instantiating two threads that repeatedly access a CD eviction set; e.g., two threads iterating 9 times over a set of size EV = 13 (CD-13-9 in Table 7) yield an eviction rate of ≈ 95% and 234 accesses. They also propose a pattern with contention on L2 (EV = 16) and CD (EV = 11) simultaneously, and attribute its success to bypassing the CD replacement policy. Our work suggests that the underlying mechanism is actually an instance of Algorithm 3; replacing shared access (Figure 4d) with L2 contention (Figure 4a) to transfer the line to LLC. Hence, what appeared to be CD contention is actually LLC contention.

End-to-end Example: AES. We demonstrate the feasibility of reduced and cross-socket eviction with the OpenSSL 1.0.1e AES T-Tables implementation, a now-standard target for side-channel research. We do not claim algorithmic improvements and simply refer to the illustrative synchronous first-round attack [41, 56] to show the feasibility of our techniques (cf. Appendix E for attack details). In short, the attacker evicts the cache lines containing the T-Tables, triggers encryptions with known plaintext bytes, and monitors access patterns to the tables. Through statistical differences between table accesses, the attacker learns the upper half of every secret key byte.

We use the hardware accelerator to construct eviction sets and assume the victim binary to reside in small read-only pages. To showcase reduced eviction, we early-abort the accelerator for EV = 4. To overcome the writing limitation, we construct the eviction sets indirectly for addresses with the same small page

offset, and then test whether they contend with the tables in the LLC/CD. This test can work with several mechanisms (cf. Figure 4). We select Algorithm 4 for its speed and reliability. To construct all necessary sets on the Xeon Platinum 8180 (ACE1, 28 slices), we observe a median runtime of 194 ms and perfect accuracy over 100 runs.

Limitations. For reduced eviction, if both magnet ways are occupied before TARGET is installed, e.g., due to noise, the target ends up outside of the DDIO region. Though this rarely happens on our setup, normal behavior can be reinstated by evicting the full set once. For cross-socket reduced eviction, the target can become stuck in the victim socket if the victim reads it while still in the attacker socket's LLC.

Results. In the absence of noise, reduced eviction consistently reveals the subkeys within 300 traces, both in same- and cross-socket attacks (cf. Appendix E). However, robustness in the latter case is significantly lower, and we recommend using an eviction set with full associativity (cf. Algorithm 5).

9 Related Work

9.1 Cache-based Side Channel Attacks

Table 8: Non-inclusive Cache Attacks (Shared Memory)

Contribution	Flushless	Cross-Socket	Single-Thread	Reduced Eviction
Lipp [30]	✓	✗	✓	✗
Irazoqui [22]	✗	✓	✓	✗
Yan (F+R) [66]	✗	✗	✓	✗
Yan (E+R) [66]	✓	✗	✗	✗
Ours ($\mathcal{A}_{\texttt{STD}}$)	✓	✓	✓	✗
Ours ($\mathcal{A}_{\texttt{CMB}}$)	✓	✓	✗	✓

Non-inclusive Cache Attacks. Lipp et al. [30] mount FLUSH+RELOAD and EVICT+RELOAD on small non-inclusive ARM caches. Irazoqui et al. [22] illustrate that FLUSH+RELOAD applies to all caches in the same coherence domain, even across sockets. These attacks bypass non-inclusive LLCs by relying on self-eviction [30], or using `clflush` [22]. Yan et al. [66], in contrast, propose a multi-threaded EVICT+RELOAD to reliably evict the shared target (cf. Section 8.3). We show an EVICT+RELOAD without CD manipulation ($\mathcal{A}_{\texttt{STD}}$), and one with four addresses ($\mathcal{A}_{\texttt{CMB}}$), refuting that eviction sets must cover the full cache associativity. Table 8 positions our work within this subset of related work.

In the absence of shared memory, the attacker can mount PRIME+PROBE [66] or PRIME+SCOPE [47] on the coherence directory, leveraging our eviction set

construction.

Cross-CPU. Yao et al. [69, 68] present a flushless cross-socket covert channel based on non-uniform memory access and cache coherence. They rely on a cooperating transmitter to evict the target (i.e., covert channel). Our cross-socket channel does not have this requirement, showing that shared memory is a security risk even when `clflush` is disabled and the victim is the only tenant on a CPU socket. A noteworthy non-cache cross-socket side-channel attack is DRAMA [45].

Secondary devices. Frigo et al. [11] accelerate microarchitectural attacks with the GPU, which is also connected to the cache hierarchy (though differently than DDIO devices). Their focus is Rowhammer-based fault injection [24, 53, 58, 39].

Weissman et al. [62] instantiate the secondary device as an FPGA. They leverage non-destructive reads (**#2**) to accelerate Rowhammer, and study cache attacks from CPU to FPGA and vice-versa. However, the statically constrained DDIO region provides challenges for FPGA-based attacks.

Kurth et al. [27] also construct eviction sets with a secondary device, i.e., a NIC. Their network-based threat model faces more challenges: it takes about five minutes to produce 64 eviction sets over the network. However, since properties **#1** and **#2** hold, our eviction set algorithm may accelerate it.

Taram et al. [55] describe a CPU process that infers network memory access patterns by the NIC (based on DDIO).

9.2 Countermeasures

Constant-time programming successfully thwarts the combined attacker explored in this paper, as it removes vulnerable code patterns. It is now common practice to harden cryptographic implementations, and several techniques have been proposed, e.g., [28, 8, 1, 49, 64]. However, access patterns can reveal other secrets, such as user input [50, 13], browsing behavior [40, 54], or model parameters [65]. Additionally, capturing all side-channel leaks remains difficult in practice [51].

Hardware-based countermeasures have attracted attention in recent years, and are generally based on, e.g., partitioning the cache [7, 32], randomizing the address-to-index mapping [61, 48, 63], or approximating fully associative caches [6, 52]. For non-inclusive cache hierarchies, SECDIR [67] hardens the coherence directory explicitly. However, existing hardware-based proposals non-trivially interact with DDIO/DDIO$^+$ regions. Additionally, such countermeasures must make explicit all potential transfers between cache levels, as undocumented transfers (cf. Section 6.1) might endanger their security.

Runtime detection using on-die counters [5, 71, 4] could be generalized to combined attackers. It should be investigated whether they sufficiently capture accelerator activity. For FPGAs, they can be embedded in the blue bitstream.

Some works propose limiting access to high-resolution timers [59, 35]. Such countermeasures do not generally thwart combined attacks, as they can bring

their own timing source.

Invalidating the findings of Section 3.2 counteracts the results in this work. The accelerator would suffer if writes occupy the full set (**#1**), or reads alter LLC state (**#2**). However, the performance implications are significant, and increasing DDIO access to the cache improves attacks from accelerators alone [27, 62]. The precise manipulation of the cache hierarchy (**#3**) seems to be fundamental to DDIO and is non-trivial to disable. An exception is the unexpected cross-socket transfer (cf. Section 6.1). This transaction is not essential to maintain coherence. We believe it to be a performance heuristic.

10 Conclusion

Heterogeneous multi-tenancy is a dangerous trend, providing attackers with evermore expressive primitives to manipulate shared microarchitectural state. This work exposed undocumented behavior in non-inclusive Intel caches and DDIO. Leveraging these insights, we developed a proof-of-concept FPGA hardware accelerator to shatter speed records for eviction set construction, build covert channels with multi-bit symbols, and evict lines from the cache with tiny sets.

Acknowledgments

We thank the anonymous USENIX Security reviewers for their insightful feedback. This research is partially funded by the European Research Council (ERC `#695305`) and the Flemish Government (FWO project `TRAPS`). It is also supported by CyberSecurity Research Flanders (`#VR20192203`) and a generous gift from Intel. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] José Bacelar Almeida, Manuel Barbosa, Gilles Barthe, François Dupressoir, and Michael Emmi. Verifying Constant-time Implementations. In *USENIX Security Symposium*, 2016.

[2] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing Memory Density by using KSM. In *Proceedings of the Linux Symposium*, 2009.

[3] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[4] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting Cache Attacks Through Self-observation. In *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2018.

[5] John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. On the Feasibility of Online Malware Detection with Performance Counters. *ACM SIGARCH Computer Architecture News*, 2013.

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

[7] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Non-Monopolizable Caches: Low-Complexity Mitigation of Cache Side Channel Attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 2012.

[8] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. CacheAudit: A Tool for the Static Analysis of Cache Side Channels. In *USENIX Security Symposium*, 2013.

[9] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Make the Most Out of Last Level Cache in Intel Processors. In *EuroSys Conference*, 2019.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q Maguire Jr, and Dejan Kostić. Reexamining Direct Cache Access to Optimize I/O Intensive Applications for Multi-hundred-gigabit Networks. In *USENIX Annual Technical Conference (ATC)*, 2020.

[11] Pietro Frigo, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *IEEE Symposium on Security and Privacy (S&P)*, 2018.

[12] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[13] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *USENIX Security Symposium*, 2015.

[14] 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 (S&P)*, 2011.

[15] Ram Huggahalli, Ravi R. Iyer, and Scott Tetrick. Direct Cache Access for High Bandwidth Network I/O. In *32st International Symposium on Computer Architecture (ISCA)*, 2005. doi:10.1109/ISCA.2005.23.

[16] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[17] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[18] Intel. Intel Data Direct I/O Technology Overview. https://www.intel.co.jp/content/dam/www/public/us/en/documents/white-papers/data-direct-i-o-technology-overview-paper.pdf, 2012.

[19] Intel. Intel CAT: Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html, 2015.

[20] Intel. Open Programmable Acceleration Engine: Libraries. https://github.com/OPAE/opae-libs/blob/master/include/opae/buffer.h#L28, 2021.

[21] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[22] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross Processor Cache Attacks. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016.

[23] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking Kernel Address Space Layout Randomization with Intel TSX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[24] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. *ACM SIGARCH Computer Architecture News*, 2014.

[25] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[26] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology - CRYPTO*, 1996.

[27] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical Cache Attacks From the Network. In *IEEE Symposium on Security and Privacy (S&P)*, 2020.

[28] Adam Langley. ctgrind—checking that functions are constant time with Valgrind, 2010. *URL https://github.com/agl/ctgrind*, 2010.

[29] Sheng Li, Hyeontaek Lim, Victor W Lee, Jung Ho Ahn, Anuj Kalia, Michael Kaminsky, David G Andersen, O Seongil, Sukhan Lee, and Pradeep Dubey. Architecting to Achieve a Billion Requests per Second Throughput on a Single Key-value Store Server Platform. In *International Symposium on Computer Architecture (ISCA)*, 2015.

[30] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *USENIX Security Symposium*, 2016.

[31] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

[32] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B Lee. Catalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[33] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks Are Practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[34] Ilias Marinos, Robert NM Watson, and Mark Handley. Network Stack Specialization For Performance. *ACM SIGCOMM Computer Communication Review*, 2014.

[35] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking Timekeeping and Performance Monitoring Mechanisms to Mitigate Side-channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[36] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2015.

[37] Clémentine Maurice, Manuel Weber, Michael Schwarz, Lukas Giner, Daniel Gruss, Carlo Alberto Boano, Stefan Mangard, and Kay Römer. Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[38] David Mulnix. Intel® Xeon® Processor Scalable Family Technical Overview. http://web.archive.org/web/20080207010024/http://www.808multimedia.com/winnt/kernel.htm, 2017. Accessed: 2020-08-13.

[39] Onur Mutlu. The RowHammer Problem and Other Issues we may Face as Memory Becomes Denser. In *Design, Automation & Test in Europe (DATE)*, 2017.

[40] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

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

[42] 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*, 2021.

[43] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. *IACR Cryptol. ePrint Arch. 2002/169*, 2002.

[44] Colin Percival. Cache Missing for Fun and Profit. In *BSDCan*, 2005.

[45] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-cpu Attacks. In *USENIX Security Symposium*, 2016.

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

[47] 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.

[48] Moinuddin K. Qureshi. CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[49] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *Design, Automation & Test in Europe (DATE)*, 2017.

[50] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 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.

[51] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache ATtacks on TLS Implementations. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[52] Gururaj Saileshwar and Moinuddin Qureshi. MIRAGE: Mitigating Conflict-Based Cache Attacks with a Practical Fully-Associative Design. In *USENIX Security Symposium*, 2021.

[53] Mark Seaborn and Thomas Dullien. Exploiting the DRAM Rowhammer Bug to Gain Kernel Privileges. *Black Hat*, 2015.

[54] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting Through the Cache Occupancy Channel. In *USENIX Security Symposium*, 2019.

[55] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet Chasing: Spying on Network Packets over a Cache Side-channel. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[56] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *Journal of Cryptology*, 2010.

[57] Furkan Turan and Ingrid Verbauwhede. Trust in FPGA-Accelerated Cloud Computing. *ACM Computing Surveys*, 2020.

[58] Victor Van Der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clémentine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer attacks on mobile platforms. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[59] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating Fine Grained Timers in Xen. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2011.

[60] Pepe Vila, Boris Köpf, and José F. Morales. Theory and Practice of Finding Eviction Sets. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[62] Zane Weissman, Thore Tiemann, Daniel Moghimi, Evan Custodio, Thomas Eisenbarth, and Berk Sunar. JackHammer: Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2020.

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

[64] Jan Wichelmann, Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MicroWalk: A Framework for Finding Side Channels in Binaries. In *Annual Computer Security Applications Conference (ACSAC)*, 2018.

[65] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *USENIX Security Symposium*, 2020.

[66] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[67] Mengjia Yan, Jen-Yang Wen, Christopher W Fletcher, and Josep Torrellas. SecDir: a secure directory to defeat directory side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[68] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are Coherence Protocol States Vulnerable to Information Leakage? In *IEEE Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[69] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. Covert timing channels exploiting non-uniform memory access based architectures. In *Proceedings of the on Great Lakes Symposium on VLSI*, 2017.

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

[71] Tianwei Zhang, Yinqian Zhang, and Ruby B Lee. Cloudradar: A Real-time Side-channel Attack Detection System in Clouds. In *Research in Attacks, Intrusions, and Defenses (RAID)*, 2016.

[72] Li Zhao, Ravi R. Iyer, Srihari Makineni, Don Newell, and Liqun Cheng. NCID: a Non-inclusive cache, Inclusive Directory Architecture for Flexible and Efficient Cache Hierarchies. In *Conference on Computing Frontiers*, 2010.

Chapter 8

Systematic Analysis of Randomization-based Protected Caches

Alea iacta est.

Julius Caesar

Publication data

Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede, "Systematic Analysis of Randomization-based Protected Cache Architectures". *IEEE Symposium on Security & Privacy*, 2021, pp. 987–1002

For compactness, the appendices are not included. Please refer to the full version of the paper [154].

Contributions

Principal author.

Systematic Analysis of Randomization-based Protected Cache Architectures

Antoon Purnal[1], Lukas Giner[2], Daniel Gruss[2] and Ingrid Verbauwhede[1]

[1]imec-COSIC, KU Leuven [2]Graz University of Technology

Abstract. Recent secure cache designs aim to mitigate side-channel attacks by randomizing the mapping from memory addresses to cache sets. As vendors investigate deployment of these caches, it is crucial to understand their actual security. In this paper, we consolidate existing randomization-based secure caches into a generic cache model. We then comprehensively analyze the security of existing designs, including CEASER-S and SCATTERCACHE, by mapping them to instances of this model. We tailor cache attacks for randomized caches using a novel PRIME+PRUNE+PROBE technique, and optimize it using burst accesses, bootstrapping, and multi-step profiling. PRIME+PRUNE+PROBE constructs probabilistic but reliable eviction sets, enabling attacks previously assumed to be computationally infeasible. We also simulate an end-to-end attack, leaking secrets from a vulnerable AES implementation. Finally, a case study of CEASER-S reveals that cryptographic weaknesses in the randomization algorithm can lead to a complete security subversion.

Our systematic analysis yields more realistic and comparable security levels for randomized caches. As we quantify how design parameters influence the security level, our work leads to important conclusions for future work on secure cache designs.

1 Introduction

Caches reduce the latency for memory accesses with high locality. This is crucial for performance but also an inherent side channel that has been exploited in many microarchitectural attacks, e.g., on cryptographic implementations [3, 35, 56, 15], user input [41, 34, 13, 33], system secrets [14, 17, 10], covert channels [28, 12, 31], and transient-execution attacks like Spectre [20, 6, 4] and Meltdown [24, 49].

Figure 1: Security argument for randomized caches.

Due to the limited size of the cache, some addresses are bound to be allocated to the same cache set, *i.e.*, they are *congruent* and contend for the same resources. While some attacks are enabled by the attacker's capability to flush cache lines, others work purely with this *cache contention*. The basic building block for measuring cache contention is the *eviction set*, a set of congruent addresses. Accessing the addresses in this eviction set brings the cache into a known state. Measuring how long this takes, tells the attacker whether some process worked on congruent addresses since the last eviction.

To mitigate contention-based attacks, cache hardware can be augmented to so-called protected cache architectures. Some designs reduce interference through better isolation [42, 18, 52, 57, 58, 25, 19, 43], partial isolation (e.g., locking cache lines) [53, 9], or fully associative subcaches [8]. Another promising line of work is *randomized* cache architectures [53, 54, 22, 26, 27, 47, 39, 40, 55], which randomize the otherwise predictable mapping of memory addresses to cache sets. Several recently proposed randomized caches [47, 39, 40, 55] evaluate a dedicated hardware mapping to perform the randomization on the fly. Consequently, these designs only slightly change the interface to the outside, and can maintain efficient and scalable sharing of caches. However, even if the mapping is (cryptographically) unpredictable, there are cache collisions due to the limited size of the cache. Hence, existing proposals incorporate some notion of *rekeying*, *i.e.*, renewing randomization at runtime. This limits the temporal window in which eviction sets can be used for an attack.

While randomized cache architectures show promise to thwart eviction-based cache attacks with reasonable overhead, supporting them with quantified security claims (a default for cryptographic algorithms) is challenging. Figure 1 depicts the established security argument. The randomized mapping is used as a trust anchor for security in ideal attack conditions, yielding a (conservative) estimate for the rekeying condition. Currently, the security transfer from the randomization mapping to ideal-case security is not well-understood, which we highlight by improving state-of-the-art attacks by orders of magnitude. Assuming that system activity increases the attack complexity, ideal-case security implies real-world security. However, it is unclear to which extent the rekeying condition can be relaxed.

The high interest in these novel cache designs and their seeming relevance to mitigate a growing list of attacks motivates the following fundamental questions of this paper:

Can we accurately compare security levels for randomized caches? How realistic are security levels reported for secure randomized caches? Do secure randomized caches provide substantially higher security levels than regular caches?

In this paper, we systematically cover the attack surface of randomization-based protected caches. We consolidate existing proposals into a generic randomized cache model, and identify attacker objectives in such caches. We then analyze this model, resulting in a comprehensive and parametrized analysis, serving as a baseline for future secure caches and their analysis.

We present Prime+Prune+Probe (PPP), a technique to find probabilistic but reliable eviction sets in randomized caches. Improving the approach by Werner et al. [55], PPP dramatically outperforms traditional eviction, turning infeasible attacks (e.g., $>10^{30}$ accesses) into feasible ones (e.g., $<10^{7}$ accesses).

We also analyze security under complicating system effects, e.g., noise and multiple victim accesses, culminating with successful key recovery from a vulnerable AES implementation.

Latency constraints associated with the cache hierarchy have inspired designers to invent new [39, 47] or repurpose existing [47] low-latency structures for the randomization mapping. Security arguments then rely on their alleged unpredictability. We falsify this assumption for Ceaser-S, and propose that future designs use mappings that resist extensive cryptanalysis.

Contributions. In summary, our main contributions are:

- We consolidate existing proposals into a generic randomized cache architecture model.
- We derive a comprehensive and parametrized analysis of all computation-based randomized cache architectures. We improve noise-free attacks by several orders of magnitude.
- We analyze non-ideal effects in profiling on randomized caches, and demonstrate the first end-to-end attack.
- We study the security requirements of the core randomized mapping and show that the security of Ceaser-S can be completely subverted, even with frequent rekeying.

Outline. Section 2 provides background. Section 3 presents our generalized cache model. Section 4 generalizes contention-based attacks for randomized caches. Section 5 presents ideal-case eviction set construction, Section 6 describes optimizations, and Section 7 considers aggravating system effects. Section 8 shows how exploiting internals can completely subvert security guarantees. Section 9 discusses results and compares existing proposals. Section 10 concludes.

2 Background

2.1 Caches and Cache Hierarchies

CPUs hide memory latency using caches to buffer data expected to be used in the near future. Caches are organized in cache lines. In a directly mapped cache, each memory address can be cached by exactly one of the cache lines, determined by a fixed address-based mapping. If a memory address can be cached in any cache line, the cache is called fully-associative. If a memory address can only be cached in a (fixed) subset of cache lines, the cache is called set-associative. Addresses mapping to the same set are called *congruent*. Upon a cache line fill request, a replacement policy determines which cache line in the set is replaced. The so-called cache line tag uniquely identifies a cached address. CPU caches can be virtually or physically indexed and tagged, *i.e.*, cache (set) index and the cache line tag are derived from the virtual or physical address.

CPUs have multiple cache levels, with the lower levels being faster and smaller than the higher levels. If all cache lines from a cache A are required to be also present in a cache B, cache B is called *inclusive* with respect to cache A. If a cache line can only reside in one of two cache levels at the same time, the caches are called *exclusive*. If the cache is neither inclusive nor exclusive, it is called *non-inclusive*. The last-level cache (LLC) is often inclusive to lower-level caches and shared across cores to enhance the performance upon transitioning threads between cores and to simplify cache coherency and lookups.

The L1 cache is often considered the lowest level cache. It is usually virtually indexed and physically tagged. All higher-level caches are usually physically indexed and physically tagged.

Again for performance, the last-level cache today is typically composed of multiple independent *slices*, e.g., one slice per physical or logical core. Each (physical) address maps to one of the slices. After selecting the slice, the cache (set) index is selected as described before. The slices are interconnected, e.g., by a ring bus, allowing all cores to access all last-level cache lines. The mapping from physical addresses to slices has been reverse-engineered for certain microarchitectures [29]. In this work, we focus on the complete mapping function which combines the mapping from addresses to slices, sets, and lines.

2.2 Cache Attacks

Caches reduce the latency of memory accesses with temporal or spatial locality, e.g., recent memory accesses. An attacker can observe the latency and make deductions, e.g., on other recent memory accesses. The first cache attacks deduced cryptographic secrets by observing the execution time [21, 36, 48, 3]. The best techniques today are Flush+Reload [56] and Prime+Probe [35]. Flush+Reload flushes an address, then waits, and by reloading determines whether

the victim accessed it in the meantime. While Flush+Reload requires a flush instruction to remove a cache line from all cache levels, Evict+Reload [13] uses cache contention. Both Flush+Reload and Evict+Reload only work on (read-only) memory shared between attacker and victim. Prime+Probe [35] overcomes this limitation. Prime+Probe measures cache contention instead of memory latency. The attacker fills (primes) a subset of the cache (e.g., a slice, a set, a line) and measures (probes) how long it takes. The time to fill the subset is higher if a victim replaces an attacker cache line with a congruent address.

Mounting Prime+Probe requires information about how addresses map to cache lines, which can be gained implicitly in certain scenarios. This is trivial for the L1 cache and, hence, the first Prime+Probe attacks targeted the L1 cache [37, 35]. More recently, Prime+Probe attacks were mounted on last-level caches [28, 34, 30, 23, 31].

Cache attacks based on cache contention generally consist of two phases. In the *profiling phase*, the attacker finds a so-called *eviction set*, a set of addresses with a high degree of contention in a subset of the cache. In the *exploitation phase*, the attacker accesses this eviction set to bring the cache into a known state. For Evict+Reload, the attacker uses it to evict an entire cache set (including a target address) and to later on reload the target address to determine whether it has been accessed in the meantime. Prime+Probe works similarly, except that it does not reload the target address but accesses the eviction set again to measure contention caused by victim memory accesses.

Early approaches for *finding* eviction sets were based on knowing addresses and their congruence, and simply collected a set of such addresses. With address information unavailable, the attacker instead starts with a set of addresses, large enough to be a superset of an eviction set with high probability. Elements are removed from this set until it has minimal size. Recently, this eviction set reduction has been improved from quadratic to linear complexity in the size of the initial set [51, 40].

2.3 Randomized Cache Architectures

State-of-the-art randomized cache architectures replace predictable address-to-index mappings with deterministic but *random-looking* mappings. The original proposals consider a software-managed look-up table, whereas newer designs compute the randomized mapping on-the-fly in hardware.

2.3.1 Table-based architectures

RPCache [53] uses a permutation table to randomize the mapping from memory addresses to cache lines. Occasionally updating the permutation aims to mitigate statistical attacks. Random-fill cache [26] issues cache fill requests to

random addresses in spatial proximity instead of the accessed ones. Table-based architectures face scalability issues, which are especially prohibitive for last-level caches.

2.3.2 Computation-based architectures

Recent designs (TIME-SECURE CACHE [47], CEASER [39, 40], SCATTERCACHE [55]) cope with this scalability problem by computing the mapping in hardware instead of storing it. This computation should have very low latency. Given their flexibility and scalability, computation-based designs are proposed for last-level caches, which have the largest latency budget and are important to protect as they are usually shared across cores.

2.3.3 Cache partitions

Algorithmic advances in eviction set construction [51, 40] have shown that *only* randomizing the memory address is insufficient to protect against contention-based cache attacks. As a key insight, CEASER-S and SCATTERCACHE partition the cache and use the randomized mapping to derive a different cache-set index in each of these partitions. Not only does this significantly raise the bar for *finding* eviction sets, but it also hinders *using* them in the exploitation phase.

2.3.4 Rekeying

Even if the mapping from address to cache set in each partition is unpredictable, the attacker can, over time, still identify sets of addresses contending in the cache. Thus, randomized caches rely on *rekeying*, *i.e.*, sampling a new key to refresh the randomization. Selecting an appropriate rekeying condition marks an important security-performance trade-off.

2.3.5 Security analysis

Computation-based randomized caches show promise to mitigate cache-based side-channel attacks. Although all proposals come with first-party security analyses, they currently lack a systematic and complete analysis (that we rely on and know, e.g., for cryptographic schemes).

3 Generic Randomized Cache Model

In this section, we present a generic randomized cache model that covers all proposed computation-based randomized caches to this date. We use it to cover the attack surface of randomized caches systematically. In later sections, we will quantify the influence of each parameter on the residual attack complexity.

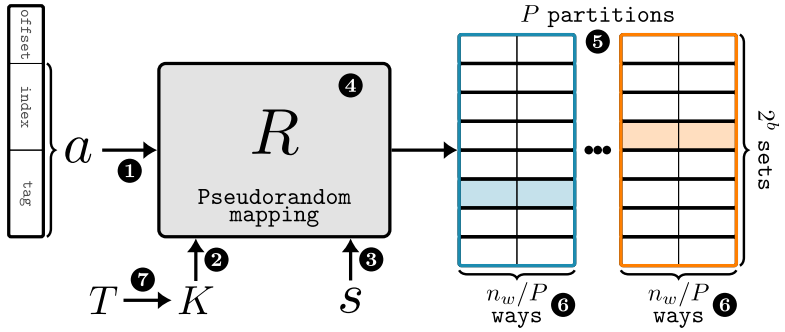


Figure 2: Computation-based randomized cache model

3.1 Randomization-based Protected Cache Model

Although some protected cache designs fix the cache configuration, we consider a generic n_w-way set-associative cache with 2^b sets (*i.e.*, b index bits). Then, let $N = n_w \cdot 2^b$ denote the number of cache lines. As with traditional caches, the atomic unit of the mapping from addresses to cache sets is the cache line, for which we assume a generic size of 2^o bytes (*i.e.*, o line offset bits). The model makes abstraction of the line offset bits, as they do not contribute to the randomization.

In accordance with traditional caches, processes cannot monitor the data in the cache directly, nor can they infer to which cache way a certain memory address is allocated. The only interface available is the access latency when reading specific addresses, *i.e.*, it is *low* in case of a cache hit and *high* in case of a miss. In some practical cases, an attacker might also have access to flush semantics. However, our attacks do not rely on it and we thus assume it to be disabled architecturally.

3.1.1 Generic model

Figure 2 depicts our generic computation-based randomized cache, featuring the following components:

❶ The **memory address** $\boldsymbol{a}$ is the primary input to the randomization design. a is either a physical or virtual address, impacting the degree of control an attacker has over a.

❷ The **key** $\boldsymbol{K}$ captures the design's entropy (unpredictability).

❸ The **security domain separator** s optionally differentiates the randomization for processes in different threat domains.

❹ The **randomized mapping** $R_K(a, s)$ is the core of the architecture. It is a *pseudorandom* mapping, *i.e.*, deterministic but random-looking, for which the algorithmic description is publicly known, but the key K is not (Kerckhoff's

Table 1: Instantiating the generic model for existing cache designs.

Design	K	s	P	R
Unprotected	$\varnothing$	$\varnothing$	1	slice + bits
Tsc [47]	keys / select	$R_{K_s}(a)$	1	HashRP / RM
Ceaser [39]	key	$\varnothing$	1	Llbc
Ceaser-S [40]	key	?	2-4	Llbc
ScatterCache [55]	key	$R_K(a,s)$	n_w	Qarma [1]

principle). The LLC slicing function can be encapsulated in R (*i.e.*, one randomized cache), or not (*i.e.*, per-slice randomized caches).

❺ The randomized cache is divided into ***P* partitions**, where $1 \leq P \leq n_w$. An input address a has, in general, a different index in each of these partitions. To accommodate this, R has to supply $P \cdot b$ pseudorandom bits. We assume P divides n_w.

❻ When caching a, one of the partitions is *truly randomly* selected, and the corresponding cache-set index in this partition is determined based on the *pseudorandom* output of R. Then, one of the cache lines in this set is replaced by a, adhering to the **replacement policy within the partition**. We consider random replacement (`RAND`) and least-recently used (`LRU`). Under attack, several stateful policies can degenerate to `LRU` [11].

❼ The **rekeying period** T denotes the condition for entropy renewal. It should be strict enough to maintain high security, and loose enough to maintain high performance.

3.1.2 Instantiating Caches

Table 1 shows how existing designs instantiate this model. The key K can be a cryptographic key (Ceaser-S, ScatterCache), a set of cryptographic keys, or selection of a random permutation (Time-Secure Cache). Time-Secure Cache (Tsc) implements domain separation with a per-process key, ScatterCache via additional input to the mapping, and Ceaser-S mentions it without implementation details. Traditional unprotected caches, Ceaser, and Tsc all have one single partition. In ScatterCache, $P = n_w$ (the maximum), whereas Ceaser-S recommends $2 \leq P \leq 4$. The rekeying condition T can use, e.g., the wall-clock time, the number of accesses to the cache, or more complex policies.

3.1.3 Software Simulator

We implement our model as a C++ randomized LLC simulator, which we parametrize and use to obtain all experimental results in this work. For simulation purposes, many well-analyzed cryptographic primitives can be used for R_K. We use AES because of its hardware support.

3.2 Attacker Models

We now systematically cover the attack surface of randomized caches and define relevant attacker models in such caches.

Leveraging a provable security methodology from cryptography, we propose to analyze the randomized mapping R (❹) separately from how it is used. On the one hand, we consider *black-box attacks*, which assume that R behaves ideally. In this case, processes cannot efficiently recover K, find inputs to R_K that produce output collisions, or infer any information about cache set indices in one partition based on observations in another. On the other hand, we also consider *shortcut attacks* that exploit R directly. Physical side-channel attacks on R (e.g., using power consumption) are out of scope for this work but can be addressed orthogonally with established approaches [7].

We further assume full attacker control over input address a (❶) as the mapping R should dissolve any attacker control regardless of the input. The key K (❷) is considered full entropy (e.g., generated by a TRNG). If security domains (❸) are supported, we assume that an attacker cannot obtain the same identifier s as the victim. The attacker cannot observe the output of R (❺) directly, but only gather metadata about it by measuring cache contention. Finally, the attacker cannot modify the rekeying condition (❼) (e.g., it is enforced in hardware).

In line with Figure 1, we consider the following three attacks:

$\mathcal{A}_{ideal}$ In an **ideal black-box attack**, the mapping R is considered to behave ideally, and the system is completely noise-free. The victim performs only a single memory access, exactly the one the attacker wants to observe later (cf. Sections 5 and 6).
$\mathcal{A}_{nonid}$ In a **non-ideal black-box attack**, $\mathcal{A}_{ideal}$ is extended with aggravating system assumptions, and serves to study the increase in attack complexity with respect to $\mathcal{A}_{ideal}$, e.g., noise and multiple victim accesses (cf. Section 7).
$\mathcal{A}_{short}$ In a **shortcut attack**, internals of the mapping R are exploited to find eviction sets much faster than in the black-box case, *i.e.*, a shortcut is found (cf. Section 8).

Existing analyses [40, 55] study attacker $\mathcal{A}_{ideal}$, as it describes the transfer of security properties from the mapping R_K to the cache architecture (cf. Figure 1). It allows selecting a conservative rekeying condition for a specific design. Besides its general applicability, it also covers some practical settings. For instance, trusted execution environments like Intel SGX are subject to precise control over victim execution [32], *i.e.*, precisely stepping to a single instruction (e.g., a memory access) and even repeating it an arbitrary number of times [50, 44].

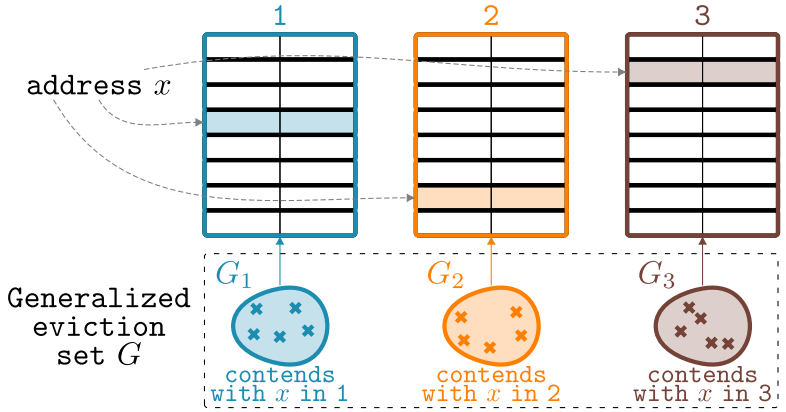


Figure 3: Generalized eviction sets are based on partial congruence ($n_w = 6$, $P = 3$, $b = \log_2 8 = 3$)

4 Exploiting Contention on Randomized Caches

This section introduces generalized eviction to overcome the challenges introduced by randomized caches. Next, it generalizes traditional attacker objectives to randomized caches.

4.1 Generalizing Eviction

4.1.1 Full congruence

In an eviction set for a traditional cache, every address a_i in this set is *fully congruent* with x. Hence, if x is currently cached, each a_i has the potential to evict it.

In a randomized cache, an attacker can theoretically still find a set of addresses that collide with the target address x in *every partition*. However, the probability for a randomly selected address to be fully congruent with x is 2^{-bP}, *i.e.*, it plummets exponentially with P. Already for $P \geq 2$, relying on full congruence to construct eviction sets is highly impractical.

4.1.2 Partial congruence

To overcome the full congruence problem, one can also try to evict a target address x based on *partial congruence*. This approach, introduced by Werner et al. [55] for special case $P = n_w$, constructs an eviction set using addresses congruent with the target in *one partition only*.

To understand eviction with partial congruence in general, consider Figure 3, where the attacker wants to evict a target x in a toy randomized cache with 6 ways ($n_w = 6$), 8 sets ($b = 3$) and 3 partitions ($P = 3$). Assume the attacker has found sets of addresses G_1, G_2, G_3, satisfying that all elements in G_i are congruent with x in partition i but not in the other partitions.

Eviction based on partial congruence is probabilistic. If x is allocated to partition i, it *could* be evicted by G_i. An element in G_i can only contribute to evicting x when it is also assigned to partition i; this assignment is truly random (*i.e.*, not pseudorandom). In what follows, we let a *generalized* eviction set G for a target address x denote the superset of addresses that collide with x in one partition: $G = \bigcup_{i=1}^{P} G_i$.

4.1.3 Eviction probability

Given a target x to evict, we now derive the eviction probability p_e as a function of the size $|G|$ of the generalized eviction set G. We assume that G contains an equal share for every partition, *i.e.*, $|G_i| = \frac{|G|}{P}$, $(1 \leq i \leq P)$. This assumption holds probabilistically in practice, and we will show how it can be met deterministically in Section 6.2.

For replacement policy `RAND`, the eviction probability generalizes the expression by Werner et al. [55]. Regardless of the partition in which x resides, $\frac{|G|}{P}$ addresses in G could evict it, each with probability n_w^{-1}. Consequently, we have:

$$p_{e,\texttt{RAND}}(|G|) = 1 - \left(1 - \frac{1}{n_w}\right)^{\frac{|G|}{P}}$$

For `LRU`, evicting x requires the attacker to evict the full set in the partition in which x currently resides. This corresponds to the event that at least $\frac{n_w}{P}$ out of the $\frac{|G|}{P}$ addresses for the designated partition are actually mapped to this partition. It is described by the complement of the cumulative binomial with $\frac{|G|}{p}$ trials, $\frac{n_w}{P} - 1$ successes and success probability $\frac{1}{P}$:

$$\begin{aligned} p_{e,\texttt{LRU}}(|G|) &= 1 - \texttt{binom}\left(\frac{|G|}{P}, \frac{n_w}{P} - 1, \frac{1}{P}\right) \\ &= 1 - \sum_{i=0}^{\frac{n_w}{P}-1} \binom{\frac{|G|}{P}}{i} \left(\frac{1}{P}\right)^i \cdot \left(1 - \frac{1}{P}\right)^{\frac{|G|}{P}-i} \end{aligned}$$

Conversely, selecting the eviction probability p_e fixes $|G|$, presented in Table 2 for different cache configurations and p_e.

4.2 Generalizing Attacker Objectives

We now generalize eviction set objectives from traditional to randomized caches and evaluate their utility.

Table 2: Generalized eviction set size for several instances.

RP	p_e [%]	$n_w=4$		$n_w=8$			$n_w=16$		
		$P=2$	$P=4$	$P=2$	$P=4$	$P=8$	$P=2$	$P=4$	$P=16$
RAND	50	6	12	12	24	48	22	44	176
	90	18	36	36	72	144	72	144	576
	95	22	44	46	92	184	94	188	752
LRU	50	6	12	14	28	48	30	60	176
	90	14	36	24	60	144	42	100	576
	95	16	44	26	72	184	46	116	752

A *targeted eviction set* for an address x is a set of addresses that, when accessed, evicts x from the cache with high probability. The complexity and utility of this objective depends on the capability of the attacker to access the target address x.

The attacker can access x if it is an in-process address or resides in memory shared between attacker and victim. By accessing x directly, the attacker can measure its access latency. This objective is useful even in randomized caches, e.g., for Evict+Reload side- and covert channels or to trigger direct DRAM accesses for eviction-based Rowhammer [2, 11].

In the other case, the attacker does not learn the access latency of x, and victim accesses to x are needed for constructing eviction sets. It is the primary attack vector for randomized caches, as it represents the general scenario where x is not accessible by the attacker (unshared memory), *or* accessible to the attacker but decoupled in the cache for different security domains. In addition to the previous objectives, generalized eviction sets in this setting are useful, e.g., for Prime+Probe side- and covert channels, or to extend transient execution windows by evicting branch condition values from the cache.

An *arbitrary eviction set* (normally the easiest to construct [51]) is a set of memory addresses that, when accessed, has a high probability that at least one of its elements is evicted from the cache. Although this objective has proven to be useful in traditional caches, e.g., for covert channels [31], its generalization to randomized caches with $P > 1$ and security domains does not seem to map to any known adversarial goals.

Takeaway: Generalize eviction to avoid full congruence.
Rely on partially congruent addresses to efficiently (but probabilistically) measure contention in randomized caches.

5 Constructing Generalized Eviction Sets

The generalized eviction set G is the primitive at hand for attacking randomized caches. Once G has been constructed, contention-based attacks like Prime+

PROBE are also possible in randomized caches, although with a larger set and lower success probability (cf. Section 4.1). The major hurdle is the *profiling* attack stage, *i.e.*, constructing G itself. Purnal and Verbauwhede [38] performed an initial study of this problem.

This section is concerned with the construction of G for a target address x, using the capabilities of the black-box attack $\mathcal{A}_{ideal}$ (cf. Section 3.2). We focus on the general case of a target x that is not attacker-accessible (cf. Section 4.2), as the security domain separator s lifts most attack objectives from the accessible to the non-accessible case. We will later show the optimizations that can be applied *should* they be accessible. Our novel profiling approach is generically applicable and efficient, improving state-of-the-art methods by orders of magnitude.

Conventionally, eviction sets are constructed by reducing a large set of addresses to a smaller set while maintaining a high eviction rate. This traditional *top-down* approach is highly effective for $P = 1$, but both the size of its initial set *and* its reduction step are strongly hindered by partitioning the cache ($P > 1$). We cope with the sheer infeasibility of reducing the initial set by adopting a new *bottom-up* approach: The attacker starts from an empty set and incrementally adds addresses for which cache contention with the target address was observed.

When measuring contention with a target x that is not attacker-accessible, the only available procedure is to prepare the cache state, wait for victim execution, and observe changes in the cache state. Finding a generalized eviction set G then comprises several iterations of this procedure. A successful iteration is one that *catches* an access to x, and the success probability of an iteration is the *catching probability* p_c.

Takeaway: Adopt a bottom-up strategy to construct G.
In partitioned randomized caches, detecting contention is much more efficient than detecting *absence* of contention.

5.1 Generic Prime+Prune+Probe

To maximize the probability of catching a victim access to x in a given iteration, we develop a specialized PRIME+PROBE, tailored for finding eviction sets in randomized caches.

5.1.1 Prime+Prune+Probe

An iteration begins with a **`prime`** step, where the attacker accesses a set of k addresses, loading them into the cache. For $k > 1$, there can be cache contention *within* this set. Thus, as a key step to eliminate false positives, the **`prune`** step iteratively re-accesses the set. This forces all self-evicted addresses to be cached again, at a potentially different location than before. The **`prune`** step terminates

Table 3: Catching probability p_c as a function of cache and attack instance, and whether the target address is cached or not.

RP	**Catching probability**	
	$p_{c,n}$ (not cached)	$p_{c,c}$ (cached)
RAND	$\frac{k'}{N}$	$\sum_{i=1}^{n_w} \binom{n_w}{i} \frac{i^2 \cdot k'^i \cdot (N-k')^{n_w-i}}{n_w^2 \cdot N^{n_w}}$
LRU	$\approx 1 - \texttt{binom}(k, \frac{n_w}{P} - 1, \frac{1}{P \cdot 2^b})$	$\approx p_{c,n} \cdot \frac{p_{c,n}(P-1)+1}{P}$

as soon as no more self-evictions occur when accessing the set.

If there are still self-evictions after a few iterations, pruning becomes more aggressive and additionally discards all addresses with high access latency (*i.e.*, those evicted by another attacker address). Upon termination of the **prune** step, the attacker has a set of $k' \leq k$ known addresses guaranteed to reside in the cache. Let m_{pr} denote the total number of pruning iterations.

Now, the attacker triggers the victim to perform the access of interest (*i.e.*, access x, as in conventional Prime+Probe). This memory access evicts one attacker address with probability p_c, which depends both on the attack parameter k and the randomized cache parameters (cf. Section 5.2). In the **probe** step, the attacker accesses the set of k' addresses again, adding addresses with high latency to G (*i.e.*, victim evicted them).

In Prime+Prune+Probe (PPP), the **prune** step is crucial and noise-absorbing. Without it, the attacker cannot distinguish evictions due to victim accesses from those by the priming set. By pruning, the attacker completely removes these false positives. Appendix A experimentally relates pruning parameters k, k' and m_{pr} for different cache configurations.

The Prime+Prune+Probe procedure is repeated until enough accesses are *caught* and added to G. This constitutes the bottom-up approach; G is not the result of shrinking a large initial set. Instead, it is built from the ground up.

5.1.2 Penalty for being cached

In case the target address x is already cached, a single PPP iteration must both *evict* x and *catch* the access to x when it is reloaded into the cache.

The attacker can either (1) first evict x probabilistically, by accessing many different addresses or other techniques; (2) apply PPP as-is, tolerating a suboptimal catching probability p_c. These strategies trade off the success probability of one iteration (p_c) with its execution time (number of accesses). In what follows, we consider both a cached and uncached x. Any profiling strategy then has higher p_c than when the target is *always* cached, and lower p_c than when it is *never* cached.

5.2 Catching Probability p_c

The catching probability p_c is the success rate of one PRIME+PRUNE+PROBE iteration and depends on the randomized cache (n_w, b, P, policy `RP`) and attack parameter k'. Table 3 establishes p_c for several configurations. We distinguish whether x is cached (denoted $p_{c,c}$), vs. not cached (denoted $p_{c,n}$).

5.2.1 Target is not cached ($p_{c,n}$)

After **`prime`** and **`prune`**, the victim access to x caches it in a random partition, and R_K pseudorandomly determines the cache set *within* this partition.

For `RAND`, x evicts an attacker address with probability equal to the coverage of the cache after pruning (*i.e.*, $p_{c,n} = k'/N$).

For `LRU`, x evicts an attacker address if there are at least $\frac{n_w}{P}$ addresses in the attacker set that were mapped to the same cache partition *and* set of x during **`prime`** and **`prune`**.

It can be approximated (and lower-bounded) by the complement of the cumulative binomial with k trials, $\frac{n_w}{P} - 1$ successes and binomial success probability $(P \cdot 2^b)^{-1}$, *i.e.*, $p_{c,n} = 1 - \texttt{binom}(k, \frac{n_w}{P} - 1, \frac{1}{P \cdot 2^b})$. In practice, due to self-evictions during pruning, the actual number of binomial trials is slightly higher than k, resulting in increased $p_{c,n}$.

5.2.2 Target is cached ($p_{c,c}$)

Catching an access to a cached target x requires *both* evicting x and detecting its reintroduction in the cache, resulting in a penalty on p_c. The probabilities $p_{c,c}$ (exact for `RAND`, approximate for `LRU`) are derived in Appendix B and collected in Table 3. The penalty is maximal for $k' = 1$, being n_w (`RAND`) or P (`LRU`), and decreases with k' as **`prime`**/**`prune`** implicitly evict an increasing cache portion.

Appendix C complements the theoretical analysis with empirical validation. It also explores the relation between p_c and k', and the penalty on p_c for a cached target.

Takeaway: Add pruning to Prime+Probe profiling.
Pruning enables testing more than one guess per iteration. It improves profiling for `RAND` and is essential for `LRU`.

6 Optimizations for Prime+Prune+Probe

This section describes optimizations of PRIME+PRUNE+PROBE for (A) total cache accesses and (B) victim invocations. We then evaluate PPP strategies on a range of cache instances.

6.1 Optimizing for total cache accesses

6.1.1 Burst Accesses

As derived, the catching probability $p_{c,c}$ (target already cached) holds at the start of constructing the generalized eviction set G. As the elements of G have explicitly been observed to collide with x, they can be accessed in burst before the PPP iteration, essentially implementing a *targeted* eviction of x. As profiling progresses and G grows, the burst becomes more successful, and the penalty for a cached target shrinks, hence $p_{c,c} \to p_{c,n}$ asymptotically. The burst access optimization thus hides the caching penalty. It applies to both `RAND` and `LRU`, but the latter can be accelerated even more.

6.1.2 Bootstrapping

A PPP iteration for `LRU` succeeds if **`prime`**/**`prune`** fill the full set for x in the designated partition. As G contends with x, we add G as bootstrapping elements to the PPP set. Thus, filling the full set becomes more likely.

However, if a victim access to x evicts a bootstrapping element instead of a PPP guess, the iteration is wasted: G was already known to contend with x. This issue can be resolved by relying on `LRU` statefulness. Adding G at the *end* of the PPP set ensures that PPP evictions precede bootstrapping evictions.

Bootstrapping implicitly implements burst accesses, and works very well for `LRU`. However, it is unattractive for `RAND`.

Takeaway: Use elements in G to accelerate finding more.
Burst accesses hide caching penalty effectively as G grows.
Bootstrapping increases p_c by helping to fill the `LRU` set.

6.2 Optimizing for victim invocations

We now explicitly minimize the required victim accesses A_v. This is relevant, e.g., for long victim programs or cases where victim runs are limited. We decouple it as $A_v = \frac{c}{p_c}$, relating it to accesses c needed to be *caught* (*i.e.*, successful iterations), and to p_c (*i.e.*, success probability of one iteration).

Section 5 already maximized denominator p_c with Prime+Prune+Probe. We now independently minimize numerator c, forming a flexible profiling framework to globally optimize A_v. It first preselects candidate addresses that have higher catching probabilities. The framework comprises three steps:

`Step 1.` Use Prime+Prune+Probe to find, for every partition i, *one* address a_i that collides with x in that partition.

`Step 2.` For each a_i, construct a candidate pool with addresses that collide with it in at least one partition.

`Step 3.` Resume PRIME+PRUNE+PROBE with the obtained candidate pools instead of randomly selected addresses.

The first step simply constructs a smaller G with PPP. Assume it needs to continue until G contains at least one element for every partition. The expected accesses to catch is then given by the coupon collector problem in statistics, with one set of P coupons: $E[c] = P(1 + 1/2 + \cdots + 1/P)$.

The second step finds addresses that contend with the a_i obtained in `Step 1`, instead of profiling x directly. As the a_i are attacker-accessible, their access latency can be measured, and no victim accesses are required. Addresses that contend with a_i also contend with x with probability $\geq P^{-1}$, which is much more likely than a randomly selected address ($\approx 2^{-b}$).

The third step resumes PPP for target x with candidate pools for the a_i. Every iteration accesses the pools, prunes, triggers access to x, and probes. For sufficiently large candidate pools, $p_c \approx 1$, significantly reducing A_v as compared to `Step 1`.

Conceptually, the first and third step are similar in nature. They can also be independently accelerated, as in Section 6.1.

We now explore the complexity and acceleration opportunities of `Step 2`. As the access latency of the *targets* a_i can be measured, catching probabilities can increase, and there is no penalty if the a_i are already cached. We again distinguish between replacement policies, and measure the complexity in attacker accesses A_a (as there are no victim accesses).

6.2.1 Optimizing `Step 2` for `RAND`

We propose to construct the candidate pool through *reverse* PRIME+PRUNE+PROBE. Let $\mathrm{S} = \{a_1, a_2, \ldots, a_c\}$ be the starting set obtained in `Step 1`. The elements of S are now the *targets* instead of the victim address x. Every iteration tries *one* random address guess g.

PRIME+PRUNE+PROBE (PPP) primes the cache with k guesses and observes eviction by the target. REVERSE PPP instead primes the cache with the *targets* S, prunes, accesses the *guess* g, then probes S. If accessing an element of S is slow, say a_k, we add g to the candidate pool for a_k. Every iteration has $p_c = \frac{c}{N}$, and there are $\approx c + 1$ attacker accesses per iteration, (*i.e.*, very little pruning, and **`probe`** overlaps with the next **`prime`**). The expected number of attacker accesses to obtain one element for the candidate pool hence is $E[A_a] \approx N$.

6.2.2 Optimizing `Step 2` for `LRU`

For `LRU`, reverse PPP is even more effective. Again, let $\mathrm{S} = \{a_1, a_2, \ldots, a_c\}$ be the set from `Step 1`, and let g denote a random address guess.

Assume the attacker primes the cache with S, prunes it, and observes self-evictions. For LRU, this implies that S filled a full cache set ($\frac{n_w}{P}$ lines). In this case, the attacker does reverse PRIME+PRUNE+PROBE, where one iteration consists of **`prime`** and **`prune`** with S, accessing g, and **`probe`** with S. If accessing an element of S is slow in the **`probe`** step, say a_k, we add g to the candidate pool for a_k. This approach has $p_c = \frac{1}{P \cdot 2^b}$, and there are $\approx c+1$ accesses per iteration, resulting in expected number of attacker accesses $E[A_a] \approx (c+1) \cdot P \cdot 2^b$.

Importantly, as g collides with multiple a_k in S, it very likely collides with x and can directly be added to S. Thus, it immediately grows eviction set G without accesses by the victim, bypassing `Step 3`. However, it can only be started if priming S has observed self-evictions. Interleaving it with `Step 1` implicitly generates new attempts at this precondition.

6.2.3 Flexibility of the Framework

The three-step framework flexibly instantiates randomized caches and attack scenarios. If the victim program is tiny and executes continuously, all profiling time is spent in `Step 1`. The shares of `Step 2` and `Step 3` grow as soon as the victim program becomes the bottleneck in any way. Finally, if x is attacker-accessible, reverse PPP from `Step 2` is used immediately. The framework also enables splitting G based on the partition of contention with x, making the eviction probabilities (Section 4.1) exact.

Takeaway: Use elements in G to reduce victim accesses.
Filtering candidate addresses based on contention with G allows to (partially) refrain from victim invocations.

6.3 Evaluation of profiling strategies

Figure 4 depicts victim and total cache accesses for the presented profiling strategies, obtained from simulated profiling runs (cf. Section 3.1.3). We observe a mostly linear progression in constructing G. One exception is reverse PPP, where the construction of the candidate pools does not grow G immediately (jump), but accelerates the profiling that follows.

Optimizations like burst accesses and bootstrapping improve both total and victim accesses. In contrast, probabilistic full cache evictions and three-step profiling incur a trade-off between total accesses and victim invocations. Of course, one can freely interpolate between these extreme strategies.

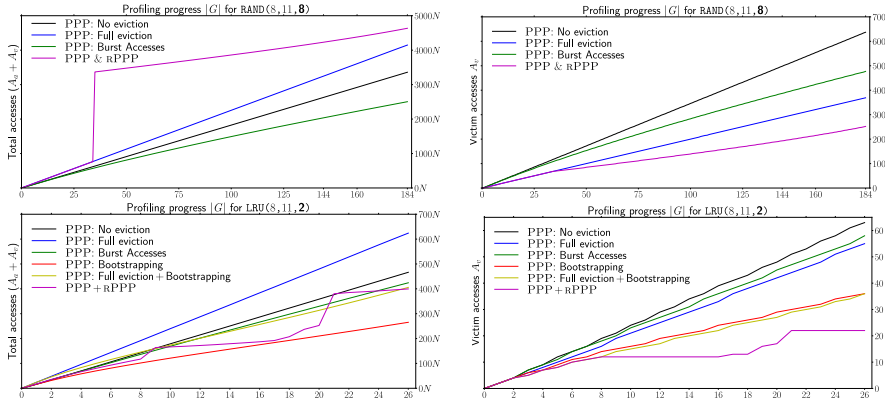


Figure 4: Effort of profiling strategies for `RAND` **(top)** and `LRU` **(bottom)**, measured as total **(left)** and victim **(right)** cache accesses, averaged for 10^4 simulated profiling runs. Cache instances are denoted $\texttt{RP}(n_w, b, \boldsymbol{P})$. k is fixed to $\frac{N}{2}$ (`RAND`) and $\frac{3N}{4}$ (`LRU`) to isolate the influence of the strategy. Pruning becomes aggressive from the sixth iteration, if not already terminated. Full evictions between PPP iterations, if performed, use $2N$ addresses for `LRU` and $3N$ for `RAND`.

6.4 Influence of randomized cache instance

6.4.1 Sets, ways and partitions

Both profiling and exploitation in randomized caches are influenced by the parameters of the instance. We investigate the effectiveness of PPP on several instances for `RAND` and `LRU`. Figure 5 captures our findings, again based on simulation (cf. Section 3.1.3).

Larger caches resist better against PPP. Increasing cache ways (n_w) seems to compare favorably to increasing sets (2^b). While the latter only proportionally prolongs profiling and does not affect exploitation, the former inhibits both profiling and exploitation. In particular, $|G|$ increases for the same exploitation p_e, and profiling is prolonged as $|G|$ increases while the accesses per element of G stay roughly the same.

Similarly, for the same cache dimensions (n_w, b), both PPP profiling and exploitation suffer from increased partitioning P. Especially for `RAND`, there is no indication from our ideal-case analysis why one should not opt for maximal partitioning. In general, we find that PPP can be hindered by tuning cache sets, ways, and partitions, but not to the point where it becomes infeasible. What really works is limiting the cache access budget for the attacker (*i.e.*, a strict rekeying condition).

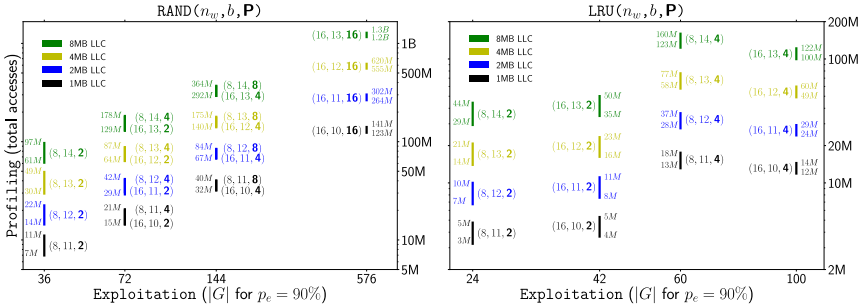


Figure 5: Influence of randomized cache parameters, for `RAND` **(left)** and `LRU` **(right)**. To isolate the influence of the instance, profiling strategies are fixed to burst accesses and $k = \frac{N}{2}$ for `RAND`, and bootstrapping and $k = \frac{3N}{4}$ for `LRU`. Instances are indicated as $(n_w, b, \boldsymbol{P})$, and positioned for mean profiling effort (y-axis, log scale), and eviction set size for exploitation (x-axis, log scale). Vertical lines span the 5-95th percentiles (ranges indicated) over 10^3 simulated runs.

6.4.2 Rekeying period

The difference between the profiling state of the art and rekeying period T is the design's security margin. Although tempting, setting T just low enough to thwart known techniques does not account for potential improvements.

As an example to obtain (very) conservative rekeying periods, we now leverage the security of R_K to derive minimal complexities to construct generalized eviction sets *of certain quality*, *i.e.*, with a lower bound on eviction probability p_e (e.g., $p_e \geq 90\%$). We use the following central assumptions:

A R_K is indistinguishable from a random function.
B Victim addresses of interest are not attacker-accessible.
C The eviction probability p_e for G is lower-bounded.

As the target is not accessible to the attacker (**B**), she can only infer accesses with PPP (cf. Section 5.1.1): bring cache in known state, wait for victim execution, and probe.

To achieve an eviction rate p_e (**C**), the profiling needs to catch at the very least $m \geq p_e P$ victim accesses to different partitions. Indeed, an attack with $p_e > \frac{m}{P}$ has inferred information about partitions for which no memory access has been caught. By contradiction with **A**, it cannot exist.

Beyond $\mathcal{A}_{\boldsymbol{ideal}}$, we further contrive the setting in favor of the attacker. We consider strongly idealized pruning (*i.e.*, $k' = k$ and $m_{pr} = 1$), and a permanently uncached target (*i.e.*, $p_c = p_{c,n}$). Furthermore, we scope the algorithm as catching a single access in m partitions, neglecting the necessary expansion to full G.

Table 4: Rekeying periods T to ensure that the success rate to construct G with $p_e \geq 95\%$ is upper-bounded by $1/2^{\{8,12,16,24,32\}}$. The cache instance is `RAND(16, 13, 16)` and all accesses are counted as cache hits (e.g., 10 ns)

	T for profiling		$T/2$ for profiling	
Success Rate	T	time	T	time
2^{-8}	$40N$	$\approx$ 10 sec	$80N$	$\approx$ 20 sec
2^{-12}	$29N$	$\approx$ 2.5 min	$58N$	$\approx$ 5 min
2^{-16}	$22N$	$\approx$ 30 min	$44N$	$\approx$ 60 min
2^{-32}	$9N$	$\approx$ 2 years	$18N$	$\approx$ 4 years

A perfectly ideal PRIME+PRUNE+PROBE iteration then requires k accesses for **`prime`**, k for **`prune`**, 1 for the victim access, and k to **`probe`**. Assuming the attacker somehow manages to combine **`probe`** of one iteration with **`prime`** of the next, we use $2k+1$ accesses per iteration as lower bound.

We outline the idea for a randomized cache with random replacement. The only degree of freedom in the idealized PPP is the number of addresses k in the **`prime`** step. Indeed, their order or frequency does not impact the cache coverage $\frac{k}{N}$.

Given a rekeying period of T cache accesses, the probability of observing at least one access in at least m distinct partitions is (using a generalization of the birthday problem in statistics):

$$\max_k \left[\sum_{i=m}^{\frac{T}{2k+1}} \binom{\frac{T}{2k+1}}{i} \frac{k^i (N-k)^{\frac{T}{2k+1}-i}}{N^{\frac{T}{2k+1}}} \sum_{l=m}^{P} \binom{P}{l} \sum_{r=0}^{l} (-1)^r \binom{l}{r} \left(\frac{l-r}{P}\right)^i \right]$$

Conversely, Table 4 captures rekeying periods T that upper bound the fraction of successful rekeying periods. Pessimistically assuming that every memory access is a cache hit, it gives an expected continuous profiling time of having *one* successful construction of G within the rekeying period. As the obtained G is only useful for one period, Table 4 also includes the case where half of it is used for exploitation. Note that these minimal efforts strongly depend on m, and hence on the quality of G that can be tolerated for exploitation (C).

7 Lifting Idealizing Assumptions

In this section, we explore for the first time the more challenging attack $\mathcal{A}_{nonid}$ with complicating system activity (cf. Section 4), as opposed to the commonly assumed $\mathcal{A}_{ideal}$.

We start with a victim program performing more memory accesses than of interest to the attacker and present an end-to-end attack on a vulnerable

AES implementation. We then quantify the influence of random noise on Prime+Prune+Probe.

The central assumption is that the attacker wants to profile specific addresses of the victim and that the access probability of said addresses can be changed via inputs to the victim.

7.1 Multiple Victim Accesses

In the profiling phase, the attacker identifies addresses of interest in a victim program and distinguishes between them if there are multiple, requiring disjoint eviction sets for each target. From this perspective, we model the execution of victim code as a set of *static* and *dynamic* memory accesses. Static accesses are performed regardless of the attacker's input, *i.e.*, code and data accesses performed in all victim executions. Dynamic accesses do not always occur, e.g., state- or input-dependent code or data accesses.

The attack targets are one or more addresses that are accessed upon a certain *event* the attacker wants to spy on [13]. Like Gruss et al. [13], we cannot distinguish addresses in the static set, as the cause-effect relationship is the same for all of them. Hence, for our attack, all targets are in the dynamic set.

To profile the cache addresses of interest, we propose a two-phase approach. First, we collect a superset of addresses containing colliding addresses for all static and dynamic cache lines. Second, we obtain disjoint sets of addresses from the superset, each with colliding addresses for one target.

The attacker distinguishes static and dynamic accesses by the property that dynamic accesses are statistically performed *less often* than static accesses, which are *always* performed. With the assumption from the beginning of this section, we consider a scenario where an attacker controls, e.g., via input, which dynamic accesses the victim performs in any given execution. This control can be exerted positively (*i.e.*, a dynamic access is always performed for a specific input) as well as negatively (*i.e.*, a dynamic access is never performed for a specific input). The latter scenario repeatedly calls the victim with inputs that cause it to access all but one address. Thus, it can be separated from the superset, as all other addresses in it are accessed eventually. In general, any manipulation of access probabilities in the victim can be observed. This approach describes a stronger attacker, as targeted addresses can be distinguished from others in both the dynamic and static set in the same step.

7.1.1 Implementation

In the following, we focus on maximum partitioning $P = n_w$, as non-random replacement policies like LRU generally require special treatment but behave predictably. We employ catching with intermediate full eviction. The analysis of Section 5.2.1 applies. To generate distinct and large eviction sets for our

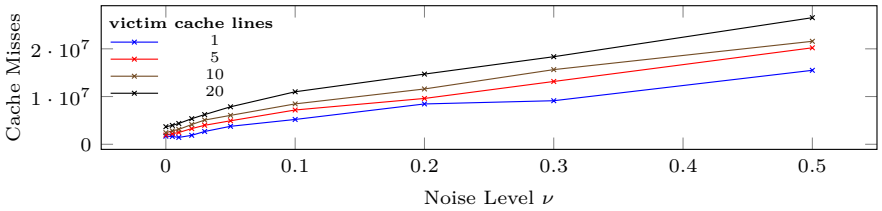


Figure 6: Cache misses for creating a superset with $3 \cdot n_w$ addresses per victim cache line, as a function of noise ν for different numbers of total victim cache lines, $k = 1000$ (avg. over 100 runs). Instance is `RAND(8, 9, 8)`

n_{target} target addresses, we slightly modify the three-step approach described in Section 6.2. All experiments are obtained in simulation (cf. Section 3.1.3).

To find sets of addresses a_i, in `Step 1`, we first construct the previously described superset using PRIME+PRUNE+PROBE (Section 6.2). Instead of only one victim memory access, all n_{stat} static and n_{dyn} dynamic victim accesses are now observed by the attacker. To identify a enough colliding addresses for all targets, we construct a superset of at least $n_w \cdot (n_{stat} + n_{dyn})$ addresses. The expected amount of memory required to find a collision in a specific way is $\frac{cachesize}{n_w}$, though higher confidence requires more. We can apply the coupon collector's problem (cf. Section 6.2) for an estimated factor of $\frac{coupon(n_w)}{n_w}$, but as more addresses need to be profiled, the probability to catch enough addresses for all targets decreases. Consequently, this step requires a number of repetitions, depending on the **`prime`** parameter k.

Next, we separate unwanted addresses from target addresses within the superset. To this end, we call the victim with inputs that exclude exactly one of the n_{target} cache lines. By repeatedly evicting the cache, calling the victim with the required parameters, and measuring accesses in the superset, we generate a histogram for all target addresses. After a certain number of repetitions, addresses that are *never* evicted by the victim are very likely to collide with the targeted address.

Repeating this process n_{target} times, we get disjoint sets of addresses for each target cache line. `Step 2` and `Step 3` can be applied to these sets of addresses (a_i) to construct the final generalized eviction sets like in the single-access case.

From our experiments (cf. Figure 6), we estimate that the number of cache misses (the largest factor of the execution time) increases sub-linearly with the total amount of accesses by the victim $(n_{stat} + n_{dyn})$. This is because the catching probability p_c increases with n_{target}. The superset's separation depends linearly on n_{target} and the overall size of the superset.

7.1.2 End-to-end Attack on AES T-Tables

We choose the 10 round T-tables implementation of AES in OpenSSL 1.1.0g as an example, as it is a well-known target for cache attacks [3, 35, 45, 13]. We perform the One-Round Attack, described by Osvik et al. [35], and thus recover 64 bits in the 16 upper nibbles of the 16-byte key (see Appendix D).

The parameters for this attack are $n_{stat} = 27$ and $n_{dyn} = 65$. With $n_{target} = 64$, the 4 T-tables are a difficult attack target, as the profiling time scales linearly with n_{target}.

For profiling, we require AES runs that access all but the target address, for each target. We can prepare 64 such key/plaintext pairs offline. All AES runs are recorded as memory access traces with the Intel PIN Tool [16] and injected into the simulator (cf. Section 3.1.3) at the appropriate times. Lacking more efficient eviction methods, we rely on probabilistic full cache eviction. In total, eviction accounts for $\approx 90\%$ of all accesses during the attack, which in turn makes the superset-splitting step of the profiling the largest contributor to the overall runtime. Because we assume no restriction on the number of encryptions, we do not perform `Step 2` for this attack, as pruning the generated candidate pools would also require the costly splitting phase. Instead, we see that using fewer colliding addresses for each target (cf. Table 5) still performs well. We can compensate for the lower detection probability by increasing the number of encryptions during the exploitation phase.

We use cache parameters from modern Intel processors: 8 slices (with a slicing function [29]) of 1 MB each, so each slice is a randomized cache with $n_w = 8/16$ and $b = 11$. We run the same attack for $P = n_w = 8/16$ and $P = 2$, with replacement policies random and `LRU`. As seen in Table 5, the attack is generic enough for all configurations, without special considerations for `LRU`. The variance in the number of addresses found per target increases for $P = 2$, especially for `LRU`, but since this specific attack sums over the hits on different addresses, this effect is mitigated for the end result (see Appendix D). For $P = 2$, we speed up the attack by reducing the cache lines used for full cache eviction from $2N$ to $1.5N$, as well as reducing the superset size (cf. Section 6.4.1).

This end-to-end implementation is not optimal, as there are many parameters that could be optimized. Nonetheless, we can see that cache attacks can still be executed in a reasonable time frame. If we model the attack as a mixture of sequential accesses for full cache evictions and timed random accesses for the sets, we can calculate the average attack times shown in Table 5. For this rough estimate, we use access times measured on a real system with the same miss rates (i7-8700K @ 3.60GHz, sequential access: $\approx$11.4c, timed (`rdtsc`) random miss: $\approx$235c, hit: $\approx$222c).

Table 5: End-to-end attack on T-table AES for different configurations (means over 100 runs). $n_{slices}=8$, $b=11$. Where not shown, standard deviations are $< 0.5\%$ of the mean.

n_w	P	policy	misses [10^9]	hits [10^9]	#AES	∅ collisions/addr.	correct nibbles	est. t [min]
8	8	n/a	12.03	3.59	56663	20.47 ± 3.61	15.90 ± 0.33	1.58
8	2	RAND	2.78	2.25	23682	15.52 ± 3.37	16.00 ± 0.00	0.63
8	2	LRU	3.21	1.75	26060	17.74 ± 7.53	15.94 ± 0.28	0.78
16	16	n/a	46.27	9.25	157072	37.91 ± 5.65	15.77 ± 0.45	6.89
16	2	RAND	4.69	3.91	39192	26.62 ± 5.85	15.93 ± 0.26	1.32
16	2	LRU	7.85	2.63	66640	26.99 ± 11.66	15.75 ± 0.51	2.60

Takeaway: Unpredictability requires key agility.
Frequent rekeying is essential to maintain the benefits of randomization, even in non-ideal conditions.

7.2 Influence of Noise

In the ideal case ($\mathcal{A}_{ideal}$), there is no noise from memory accesses by the attacker process itself, nor the victim, or any other process in the system (including the operating system). Section 7.1 already implicitly includes noise generated by the victim's code execution. We now additionally consider noise introduced by other system activity. We make the simplifying assumption that noise accesses are random and occur at a rate of ν random accesses for every attacker access.

Multiple steps of the (unmodified) profiling algorithm from Section 6.2 are affected by noise. Spurious memory accesses during the **`prune`** step increase the number of pruning iterations m_{pr} significantly and reduce the size k' of the resulting set. The **`probe`** step samples noise in addition to the collisions with the targeted victim cache line.

Figure 7 and Figure 8 show both effects. Though the unmodified **`prune`** step terminates, the resulting set size k' can be seen to decrease quickly with ν, while the number of pruning iterations m_{pr} increases. Hence, with noise, the attacker could explore the PPP parameter space in favor of a smaller k. Figure 8 also shows a faster decrease in correct collisions for higher k, while the cost of pruning grows.

Alternatively, the attacker can consider *early-aborting pruning*, *i.e.*, terminating **`prune`** before it is *entirely* free of misses. Indeed, a large part of the pruning iterations are no longer due to self-evictions, but due to sampling noise. The false positives introduced by the early-abort are then removed in a later stage.

The separation phase from Section 7.1 is effective at filtering false positives caused by noise during PPP, since static victim accesses, dynamic victim

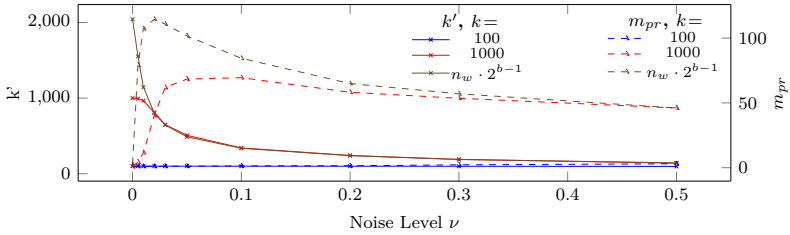


Figure 7: Pruning m_{pr} and k' as a function of noise ν, for various k (average over 100 runs). Instance is $\texttt{RAND}(8, 9, \mathbf{8})$

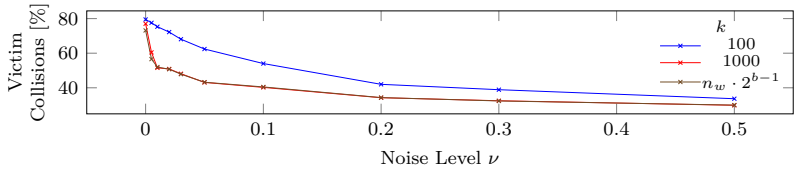


Figure 8: Percentage of caught addresses in the superset that genuinely collide with victim addresses in exactly one way, as a function of ν for various k (avg. over 100 runs). Instance is $\texttt{RAND}(8, 9, \mathbf{8})$

accesses, and false positives exhibit different behavior in the separation phase. In contrast to static or dynamic accesses, false positives occur only in some runs, leading to multiple runs with 0 accesses. Hence, they appear in more than one set in the end and can be removed.

Figure 6 shows the total number of cache misses for the generation of supersets for victims of different total sizes ($n_{stat}+n_{dyn}$). These supersets contain exactly $3 \cdot n_w \cdot (n_{stat}+n_{dyn})$ addresses that collide with victim cache lines in exactly one way. They additionally contain non-colliding addresses introduced by noise and self-eviction in the proportion shown in Figure 8, which is removed during separation. We can see that the number of cache misses (and by extension, the runtime) for this step grows approximately linearly with noise.

7.3 Infrequent victim events

In the case where an event in the victim happens only *once* or a limited number of times (e.g., user input), the probe set G needs to be large enough to achieve a very high detection probability, which places more weight on accurate profiling compared to 7.1.2. On the other hand, when events trigger accesses to multiple cache lines, all of them can be used for detection. Attacks will mostly need to be asynchronous, which necessitates some form of continuous monitoring. We

leave an investigation of practical implementations for future work.

8 Shortcut Attacks

In this section, we consider attack $\mathcal{A}_{short}$ and draw attention to the soundness of the randomized mapping by achieving *shortcuts* during a case study on CEASER and CEASER-S.

In particular, we demonstrate how weaknesses in their common randomized mapping allow us to reliably construct eviction sets without any memory accesses. We first describe Low-Latency Block Cipher (LLBC), the CEASER-specific implementation of the mapping R_K. Drawing inspiration from differential cryptanalysis, we show how input differences propagate through the LLBC, and we derive an expression for precomputing address differences that systematically yield cache set collisions, *independent of key, partition, and address.*

We describe the attack first for CEASER [39] before tackling the generalized and improved CEASER-S [40].

8.1 Low-Latency Block Cipher in Ceaser(-S)

CEASER instantiates R_K by encrypting the input address a with a custom LLBC with 40-bit blocks and 80-bit key. In particular, it divides the input address in two equally sized (left-right) chunks $a = (\texttt{L} \mid\mid \texttt{R})$ and produces an output encrypted address $R_K(a) = (\texttt{L'} \mid\mid \texttt{R'})$. From this output, the lowermost b bits determine the cache set index: $s = \lfloor R_K(a) \rfloor_b = \lfloor \texttt{R'} \rfloor_b$.

The encryption proceeds as a keyed four-stage Feistel network (depicted in Figure 9). Each stage instantiates a round function $F(X, K)$, taking 40-bit input (20 bit X and 20 bit K) and producing 20-bit output (Y). In each round function, 20 intermediary bits W_i are first computed as $W_i = S_i(X, K)$, where S_i defines exclusive or (`xor`) of 20 input bits (out of 40). The W_i are shifted with a bit-permutation P to obtain Y.

In CEASER, the round functions are randomly sampled, fixed at design time, and explicitly different in every stage. Let $F^{[r]}$ denote the round function for stage r, and $K^{[r]}$ the 20-bit subkey for this stage. Describe the bit-permutation with $i \leftarrow P(i)$, *i.e.*, a bit at position $P(i)$ moves to position i. Next, let $\mathcal{X}_i$ and $\mathcal{K}_i$ denote the indices from resp. X and $K^{[r]}$ that are `xor`ed to obtain intermediary bit $W_i = S_i(X, K)$. The round function output is $Y = (Y_0||Y_1||...||Y_{19}) = (W_{P(0)}||W_{P(1)}||...||W_{P(19)})$. The round function $F^{[r]}$ thus comprises 20 functions $F_i^{[r]}(X, K^{[r]})$ each computing one Y_i:

$$Y_i = F_i^{[r]}(X, K^{[r]}) = \sum_{j \in \mathcal{X}_{P(i)}} X_j + \sum_{k \in \mathcal{K}_{P(i)}} K_k^{[r]} \tag{1}$$

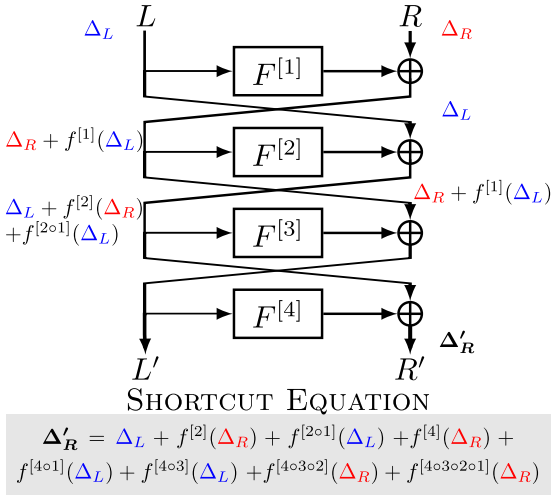


Figure 9: Differential propagation through CEASER's LLBC. For brevity, we introduce $f^{[j\circ i]}(\cdot)$ as shorthand for $f^{[j]}(f^{[i]}(\cdot))$.

Observing the linearity in the entire cipher (particularly in the SBoxes S_i, supposed to be non-linear), we draw inspiration from differential cryptanalysis to bypass R_K altogether.

8.2 Constructing and Using the Shortcut

The outcome of the shortcut is a set of addresses a_i that collides in the cache with a target address a, *i.e.*, $R_K(a_i) = R_K(a)$. The attacker could attempt this shortcut by recovering the mapping key K, granting the shortcut for the lifetime of the key. Our approach, in contrast, is fully key-independent. It is a restricted take on *chosen-plaintext attacks*, where the restriction stems from being embedded in a cache. Specifically, the adversary can *choose* a set of plaintexts to R_K (*i.e.*, input addresses a_i), but does not observe any cryptographic output.

We rephrase the shortcut as a differential problem, *i.e.*, to finding a set of Δa satisfying $R_K(a+\Delta a) = R_K(a)$. Matching with the Feistel topology, we denote the input difference $\Delta a = (\Delta_L||\Delta_R)$ and the output difference $(\Delta_{L'}||\Delta_{R'})$. Achieving the shortcut is then equivalent to finding pairs Δ_L and Δ_R, not both zero, that result in the same set index bits: $\lfloor \Delta_{R'} \rfloor_b = 0^b$.

8.2.1 Δ−Propagation

We derive the propagation first through the round function $F^{[r]}$, then the full LLBC. Let $+$ denote $GF(2)$ addition (bitwise `xor`). As a well-known cryptanalytic fact, differences propagate unaffected through addition. Let Δ_X and Δ_Y denote differences at the input and output of $F^{[r]}$. Stated differently, if $Y = F^{[r]}(X, K^{[r]})$ and $Y' = F^{[r]}(X + \Delta_X, K^{[r]})$, then $\Delta_Y = Y' + Y$. Now compute the i-th output bit $\Delta_{Y,i}$:

$$\begin{aligned}\Delta_{Y,i} &= Y'_i + Y_i = F_i^{[r]}(X + \Delta_X, K^{[r]}) + F_i^{[r]}(X, K^{[r]}) \\ &= \sum_{j \in \mathcal{X}_{P(i)}} (X_j + \Delta_{X,j} + X_j) + \sum_{k \in \mathcal{K}_{P(i)}} (K_k^{[r]} + K_k^{[r]}) \\ &= \sum_{j \in \mathcal{X}_{P(i)}} \Delta_{X,j} = f_i^{[r]}(\Delta_X)\end{aligned}$$

If we let $\Delta_Y = f^{[r]}(\Delta_X)$, then $f^{[r]}$ captures the effect of round function $F^{[r]}$ on an input difference Δ_X. Similar to $F^{[r]}$ before, $f^{[r]}$ is an umbrella for 20 functions:

$$\Delta_Y = f^{[r]}(\Delta_X) = (f_0^{[r]}(\Delta_X) \;||\; f_1^{[r]}(\Delta_X) \;||\; ... \;||\; f_{19}^{[r]}(\Delta_X))$$

Note that $f^{[r]}$ only depends on the input difference Δ_X. Crucially, it is *independent of both X itself and the key K*.

8.2.2 Shortcut Equation

Armed with the Δ-propagation through round functions $F^{[r]}$, Figure 9 shows our probability 1 differential trail through Ceaser's full LLBC, yielding an expression for output difference Δ'_R. This expression, which we dub the Shortcut Equation, describes $\Delta a = (\Delta_L || \Delta_R)$ satisfying output collision: $\lfloor \Delta'_R \rfloor_b = 0^b \Rightarrow R_K(a) = R_K(a + \Delta a)$.

A straightforward way to find solutions to this equation fixes (say) Δ_L and tests variable Δ_R for equality. The expected *offline* complexity for each $\Delta a = (\Delta_L, \Delta_R)$ is 2^{b-1} evaluations of the shortcut equation. Since very often $b < 20$, the naïve computation is very practical. As the shortcut equation describes twenty linear equations over $GF(2)$, one could also algebraically determine a compact expression for Δ_L and Δ_R.

8.2.3 Implications

The shortcut does not require *knowledge* of key K, and is even completely *independent* of K. Furthermore, it is also independent of the input a. Although in general $R_K(a) \neq R_{K'}(a)$, eviction sets constructed for key K' and input a' are still eviction sets for *any other* (K, a) pair:

$$R_K(a) = R_K(a + \Delta a) \Rightarrow R_{K'}(a' + \Delta a) = R_{K'}(a') \tag{2}$$

This follows from the key-independence of the SHORTCUT EQUATION. Hence, rekeying does not invalidate eviction sets constructed using the shortcut. This has the devastating consequence that, as soon as the Δa have been precomputed offline, the attacker can construct arbitrary eviction sets for any target a with zero cache accesses, completely bypassing R_K.

8.2.4 Extension to Ceaser-S

CEASER-S implements partitions with P parallel LLBC instances with different keys. By Equation (2), collision in one partition implies collision in all partitions. Thus, our shortcut equally impacts CEASER-S, allowing easy construction of *fully congruent* eviction sets.

8.2.5 Mitigation

At the very least, the LLBC rounds should incorporate non-linear SBox layers. This spot mitigation thwarts the presented shortcut, but more subtle attacks could remain.

Takeaway: Do not overestimate the mapping's security.
Shortcut attacks can be fundamentally eliminated by a randomization mapping that resists formal cryptanalysis.

9 Discussion

In this section, we relate and compare the contributions in this paper to the most closely related work, as well as provide specific recommendations and directions for future work.

9.1 Prime+Prune+Probe on specific designs

Our generic model for computation-based randomized caches permits to instantiate existing designs, extend their security analysis, and compare them in terms of profiling effort.

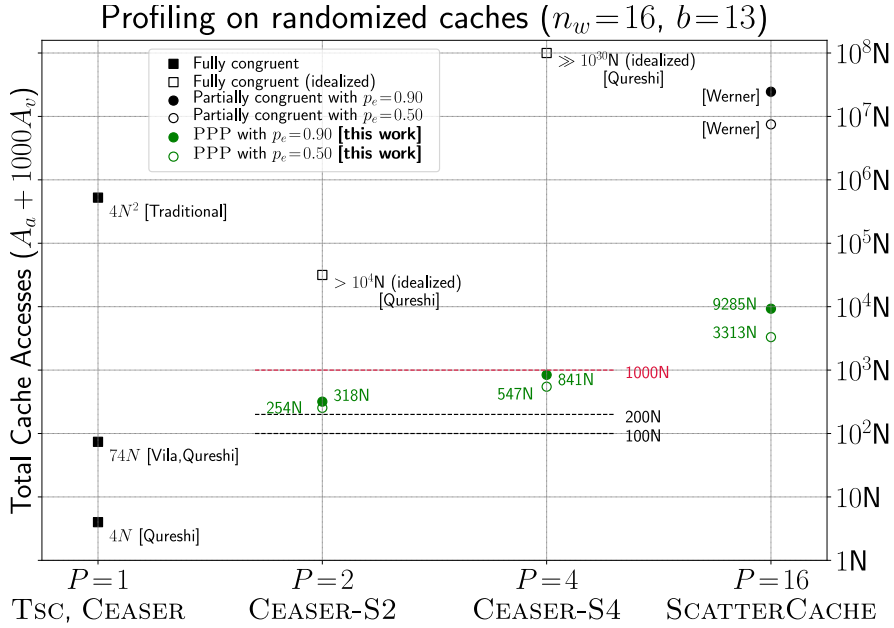


Figure 10: Complexity ($\mathcal{A}_{ideal}$) to construct a fully congruent or generalized ($p_e = 90\%$ or $p_e = 50\%$) eviction set. Randomized caches are monolithic 8 MB ($n_w = 16$, $b = 13$, $N = 131\,072$ lines). Cost metric is total cache accesses; victim runtime is modeled with 1 000 accesses. Fully congruent eviction assumes initial set (before reduction) of $2N$. PPP uses the best-performing strategy (cf. Section 6.3), with $k = \frac{N}{2}$ (`RAND`) or $k = \frac{3N}{4}$ (`LRU`).

We consider an 8 MB cache with 16 ways (n_w) and 13 index bits (b) (*i.e.*, $N = 131\,072$). We assume a non-accessible target address (e.g., by enabling security domain separation s). Although we consider $\mathcal{A}_{ideal}$ (cf. Section 3.2), *i.e.*, we are able to pinpoint one target access of interest, victim execution time cannot be neglected. Therefore, we assume a modestly-sized victim program, performing 1 000 accesses per invocation.

Figure 10 shows total cache accesses to profile a generalized eviction set G with $p_e = 90\%$. For each instance we use Prime+Prune+Probe and optimize for total cache accesses.

9.1.1 Single-partition caches

Randomized caches with $P = 1$ (Ceaser, Tsc) can be treated as traditional caches without adversary control over physical addresses. They require extremely frequent rekeying as fully congruent eviction sets can be obtained with the

efficient top-down approach [51, 40].

9.1.2 Ceaser-S

First-party CEASER-S analysis [40] only considers fully congruent eviction. As fully congruent addresses are extremely scarce, it is completely infeasible for larger P.

We instantiate the model to CEASER-S2 (resp. CEASER-S4) by setting $P=2$ (resp. $P=4$) and replacement policy `LRU`. While CEASER-S could accomodate several policies (e.g., `LRU, RRIP, ...`) [40], we believe `LRU` leads to an accurate security assessment. Indeed, many stateful replacement policies can be degraded to `LRU` with some repeated accesses [11].

In what follows, we assume the problems from Section 8 to be fixed. There are three proposed CEASER-S instances, with rekeying periods resp. $100N$, $200N$ and $1000N$. We observe that PRIME+PRUNE+PROBE consistently obtains high-quality generalized eviction sets within the rekeying period of the $1000N$-instance. While prior profiling techniques succeed on average once every 68 years [40], PPP on CEASER-S2 has average complexity of $\approx 320N$, leaving on average 68% of *every* rekeying period available for exploitation. The more conservative designs $(100N, 200N)$ resist PPP for the majority of rekeying periods, though with considerably reduced security margin. We observe an extreme gap between PPP and previous idealized estimates, easily exceeding 20 orders of magnitude for $P=4$ and 50 orders of magnitude for $P=8$ (not displayed).

9.1.3 ScatterCache

First-party analysis [55] already considers generalized eviction. Their approach can be seen as a corner case of PPP, *i.e.*, using $k=1$ (cf. Section 5).

We instantiate SCATTERCACHE by setting $P = n_w = 16$, implicitly with replacement `RAND`. Optimized for total accesses, PPP improves profiling with three orders of magnitude for the considered configuration. The main contribution of PPP is that it requires much fewer victim invocations, as it permits to test many addresses in parallel ($k \gg 1$). While SCATTERCACHE does not specify a rekeying frequency, our results indicate that it should be determined more conservatively than expected.

9.1.4 Shortcuts

With a case study on CEASER-S, we show with devastating consequences that the security of the randomization should not be taken for granted, even if its output is not directly observable. A similar study was conducted in concurrent work [5]. Instantiating a sound cryptographic algorithm thwarts all shortcuts

but affects performance. Though not investigated, TSC risks shortcuts due to absence of cryptographic structure. Shortcuts in SCATTERCACHE are only possible by significant cryptanalytic advances for QARMA [1].

9.2 Future Work

Our work provides a baseline to compare future secure caches and their analysis. Future work should investigate how our techniques can be applied to concurrent work [46]. This paper also shows the importance of cryptanalytic resistance of the core randomization mapping. Stringent latency constraints could inspire new designs in the space of low-latency cryptography.

The rekeying period may be varied for different security levels. This can be transparently implemented through frequently and unpredictably updating s for high-security processes (e.g., enclaves), while refreshing K in larger intervals for regular processes. We also propose heuristic-based rekeying, invalidating eviction sets upon observation of certain microarchitectural events (e.g., many LLC cache misses or PPP signatures). It should be noted that rapid rekeying only mitigates attacks in scope for randomized caches, *i.e.*, potential cache-contention channels that do not target set contention might remain.

The gap between our conservative rekeying periods (Section 6.4.2) and PPP profiling in practice is quite large. Future work could explore closing this gap by improving profiling, relaxing theoretical bounds, or a combination of both.

10 Conclusion

Analyzing the residual attack surface of randomized cache architectures is a complex undertaking. In this work, we have established a generic framework to jointly analyze all existing computation-based randomized caches. We showed that, similar to cryptanalysis, randomized cache designs must be subjected to systematic analysis to gain confidence in their security. In this effort, we have contributed on three main fronts.

First, we have advanced the profiling state of the art for randomization-based secure caches. We developed novel attack techniques for such caches, including PRIME+PRUNE+PROBE and optimizations like bootstrapping and multi-step profiling.

Second, we have started bridging the gap between the usually assumed ideal attack and complicating effects like noise and multiple victim accesses. We have simulated an end-to-end attack, leaking AES keys from a vulnerable implementation.

Finally, we have falsified the implicit assumption that any randomized mapping successfully results in a secure cache.

Acknowledgments

We would like to thank the anonymous reviewers and our shepherd, David Kohlbrenner, for their valuable feedback. This work was supported in part by the European Research Council (ERC) under the EU Horizon 2020 research and innovation programme (grant agreements No 681402 and No 695305). It was also supported by the CyberSecurity Research Flanders VR20192203 and the Research Council KU Leuven C16/15/058. Antoon Purnal is funded by an FWO fellowship. Additional funding was provided by a generous gift from Intel. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] R. Avanzi, "The QARMA block cipher family," in *IACR ToSC*, 2017.

[2] Z. B. Aweke, S. F. Yitbarek, R. Qiao, R. Das, M. Hicks, Y. Oren, and T. Austin, "ANVIL: Software-based protection against next-generation Rowhammer attacks," *ACM SIGPLAN Notices*, 2016.

[3] D. J. Bernstein, "Cache-timing attacks on AES," 2005.

[4] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[5] R. Bodduna, V. Ganesan, P. SLPSK, K. Veezhinathan, and C. Rebeiro, "Brutus: Refuting the security claims of the cache timing randomization countermeasure proposed in CEASER," in *IEEE CA Letters*, 2020.

[6] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. Von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019.

[7] S. Chari, C. S. Jutla, J. R. Rao, and P. Rohatgi, "Towards sound approaches to counteract power-analysis attacks," in *CRYPTO*, 1999.

[8] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "HybCache: Hybrid Side-Channel-Resilient Caches for Trusted Execution Environments," in *USENIX Security Symposium*, 2020.

[9] M. Green, L. Rodrigues-Lima, A. Zankl, G. Irazoqui, J. Heyszl, and T. Eisenbarth, "AutoLock: Why Cache Attacks on ARM Are Harder Than You Think," in *USENIX Security Symposium*, 2017.

[10] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR," in *CCS*, 2016.

[11] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript," in *DIMVA*, 2016.

[12] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+Flush: A Fast and Stealthy Cache Attack," in *DIMVA*, 2016.

[13] D. Gruss, R. Spreitzer, and S. Mangard, "Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches," in *USENIX Security Symposium*, 2015.

[14] R. Hund, C. Willems, and T. Holz, "Practical Timing Side Channel Attacks against Kernel Space ASLR," in *S&P*, 2013.

[15] M. S. Inci, B. Gulmezoglu, G. Irazoqui, T. Eisenbarth, and B. Sunar, "Cache Attacks Enable Bulk Key Recovery on the Cloud," in *CHES*, 2016.

[16] Intel Corporation, "Pin - A Dynamic Binary Instrumentation Tool," https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool.

[17] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *CCS*, 2016.

[18] T. Kim, M. Peinado, and G. Mainar-Ruiz, "StealthMem: system-level protection against cache-based side channel attacks in the cloud," in *USENIX Security Symposium*, 2012.

[19] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "DAWG: A Defense Against Cache Timing Attacks in Speculative Execution Processors," *MICRO*, 2018.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[21] P. C. Kocher, "Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems," in *CRYPTO*, 1996.

[22] J. Kong, O. Acıiçmez, J.-P. Seifert, and H. Zhou, "Hardware-software integrated approaches to defend against software cache-based side channel attacks," in *HPCA*, 2009.

[23] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "ARMageddon: Cache Attacks on Mobile Devices," in *USENIX Security Symposium*, 2016.

[24] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[25] F. Liu, Q. Ge, Y. Yarom, F. Mckeen, C. Rozas, G. Heiser, and R. B. Lee, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *HPCA*, 2016.

[26] F. Liu and R. B. Lee, "Random Fill Cache Architecture," in *MICRO*, 2014.

[27] F. Liu, H. Wu, K. Mai, and R. B. Lee, "Newcache: Secure cache architecture thwarting cache side-channel attacks," *IEEE Micro*, vol. 36, no. 5, pp. 8–16, Sep. 2016.

[28] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks Are Practical," in *S&P*, 2015.

[29] C. Maurice, N. Le Scouarnec, C. Neumann, O. Heen, and A. Francillon, "Reverse Engineering Intel Complex Addressing Using Performance Counters," in *RAID*, 2015.

[30] C. Maurice, C. Neumann, O. Heen, and A. Francillon, "C5: Cross-Cores Cache Covert Channel," in *DIMVA*, 2015.

[31] C. Maurice, M. Weber, M. Schwarz, L. Giner, D. Gruss, C. A. Boano, S. Mangard, and K. Römer, "Hello from the Other Side: SSH over Robust Cache Covert Channels in the Cloud," in *NDSS*, 2017.

[32] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[33] J. Monaco, "SoK: Keylogging Side Channels," in *S&P*, 2018.

[34] Y. Oren, V. P. Kemerlis, S. Sethumadhavan, and A. D. Keromytis, "The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications," in *CCS*, 2015.

[35] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *CT-RSA*, 2006.

[36] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," *Cryptology ePrint Archive, Report 2002/169*, 2002.

[37] C. Percival, "Cache missing for fun and profit," in *BSDCan*, 2005.

[38] A. Purnal and I. Verbauwhede, "Advanced Profiling for Probabilistic Prime+Probe Attacks and Covert Channels in ScatterCache," in *arXiv 1908.03383*, 2019.

[39] M. K. Qureshi, "CEASER: Mitigating Conflict-based Cache Attacks via Encrypted-address and Remapping," in *MICRO*, 2018.

[40] ——, "New Attacks and Defense for Encrypted-address Cache," in *ISCA*, 2019.

[41] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, You, Get off of My Cloud: Exploring Information Leakage in Third-party Compute Clouds," in *CCS*, 2009.

[42] D. Sanchez and C. Kozyrakis, "Vantage: scalable and efficient fine-grain cache partitioning," in *ISCA*, 2011.

[43] S. Sari, O. Demir, and G. Kucuk, "FairSDP: Fair and secure dynamic cache partitioning," in *International Conference on Computer Science and Engineering (UBMK)*, 2019.

[44] D. Skarlatos, M. Yan, B. Gopireddy, R. Sprabery, J. Torrellas, and C. W. Fletcher, "MicroScope: Enabling Microarchitectural Replay Attacks," in *ISCA*, 2019.

[45] R. Spreitzer and T. Plos, "Cache-access pattern attack on disaligned aes t-tables," in *COSADE*, 2013.

[46] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "PhantomCache: Obfuscating Cache Conflicts with Localized Randomization," in *NDSS*, 2020.

[47] D. Trilla, C. Hernandez, J. Abella, and F. J. Cazorla, "Cache Side-channel Attacks and Time-predictability in High-performance Critical Real-time Systems," in *DAC*, 2018.

[48] Y. Tsunoo, T. Saito, and T. Suzaki, "Cryptanalysis of DES implemented on computers with cache," in *CHES*, 2003.

[49] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-order Execution," in *USENIX Security Symposium*, 2018.

[50] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *SysTEX*, 2017.

[51] P. Vila, B. Köpf, and J. F. Morales, "Theory and Practice of Finding Eviction Sets," in *S&P*, 2019.

[52] R. Wang and L. Chen, "Futility scaling: High-associativity cache partitioning," in *MICRO*, 2014.

[53] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *ISCA*, 2007.

[54] ——, "A novel cache architecture with enhanced performance and security," in *MICRO*, 2008.

[55] M. Werner, T. Unterluggauer, L. Giner, M. Schwarz, D. Gruss, and S. Mangard, "SCATTERCACHE: Thwarting Cache Attacks via Cache Set Randomization," in *USENIX Security Symposium*, 2019.

[56] Y. Yarom and K. Falkner, "FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-channel Attack," in *USENIX Security Symposium*, 2014.

[57] D. Zhang, Y. Wang, G. E. Suh, and A. C. Myers, "A hardware design language for timing-sensitive information-flow security," *ACM Sigplan Notices*, vol. 50, no. 4, pp. 503–516, 2015.

[58] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *CCS*, 2016.

Chapter 9

ShowTime: Amplifying Arbitrary CPU Timing Side Channels

Time is a river of passing events, and strong is its current; no sooner is a thing brought to sight than it is swept by and another takes its place, and this too will be swept away.

Marcus Aurelius

Publication data

Antoon Purnal, Marton Bognar, Frank Piessens and Ingrid Verbauwhede, "ShowTime: Amplifying Arbitrary CPU Timing Side Channels". *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2023

Contributions

Principal author.

ShowTime: Amplifying Arbitrary CPU Timing Side Channels

Antoon Purnal[1], Marton Bognar[2], Frank Piessens[2] and Ingrid Verbauwhede[1]

[1]imec-COSIC, KU Leuven [2]imec-DistriNet, KU Leuven

Abstract.
Microarchitectural attacks typically rely on precise timing sources to uncover short-lived secret-dependent activity in the processor. In response, many browsers and even CPU vendors restrict access to fine-grained timers. While some attacks are still possible, several state-of-the-art microarchitectural attack vectors are actively hindered or even eliminated by these restrictions.
This paper proposes ShowTime, a general framework to expose arbitrary microarchitectural timing channels to coarse-grained timers. ShowTime consists of CONVERT routines, transforming microarchitectural leakage from one type to another, and AMPLIFY routines, inflating the timing difference of a *single* microarchitectural event to make it distinguishable with crude sources of time.
We contribute several CONVERT and AMPLIFY routines and show how to combine them into powerful attack primitives. We demonstrate how a single cache event can be amplified so that even the human eye can classify it with 98% accuracy and how stateless time differences as minuscule as 20 ns can be captured, converted, and amplified in a single observation. Additionally, we generate cache eviction sets, both in real-world restricted browser environments and natively using timers with precisions ranging from microseconds to *seconds*. Our findings imply that timer restrictions alone, even when ruthlessly implemented beyond practical limits, provide insufficient protection against CPU timing attacks.

1 Introduction

In modern computing systems, programs may affect the execution of other programs through incidental interference in shared hardware components (e.g., caches or computational units). Such interference, predictably, affects the performance of software on multi-tenant systems. However, sharing the processor

hardware (i.e., the processor *microarchitecture*) also carries security implications. Indeed, by measuring how long it takes to execute specific actions (i.e., a *timing side channel*), malicious programs can determine the usage patterns of specific microarchitectural components. Therefore, any program with secret-dependent resource utilization unintentionally encodes its secrets in the microarchitecture, exposing it to co-located adversaries. Several attacks manage to exploit this behavior, revealing cryptographic keys [1, 2, 3, 4], operating system secrets [5, 6, 7, 8], or user input [9, 10, 11].

Microarchitectural leakage is often categorized into *stateful* channels, whose effect on the microarchitecture endures for some time after the secret-dependent execution, and *stateless* channels, for which the influence on the microarchitecture disappears as the secret-dependent instructions finish executing. Initially, stateful attacks (e.g., [1, 4, 12, 13, 14]) attracted more attention as they allow the side-channel measurement to happen sometime after the secret-dependent activity. Recently, however, stateless side-channel attacks were also proven to be powerful [15, 16, 17, 18, 19, 20, 21].

Microarchitectural leakage by itself, whether it be stateful or stateless, produces only minuscule timing differences (e.g., 10-100 ns). Therefore, the lion's share of timing side-channel attacks rely on high-precision sources of time, either by consulting existing timer interfaces (e.g., [4, 12, 10, 22, 14, 23]) or by producing fine-grained and monotonically increasing values that correlate with time (e.g., [24, 25, 8]). In response, some platforms disable unprivileged access to high-precision timers [26]. Even stronger, there have been academic proposals [27, 28, 29] to orchestrate computing environments that eliminate all high-precision timing sources. Similar measures are currently deployed in modern browsers [30, 31, 32, 33, 34].

Without fine-grained timers, one option is to repeatedly trigger the leak (*multi-shot amplification*) [35, 36, 37, 38]. However, this requires a deterministic repetition of the leak, which is not generally possible. Moreover, timing differences typically accumulate slowly. In contrast, the website `leaky.page` [39] performs a sequence of memory accesses that, conditioned on the presence of a single target memory access (*single-shot amplification*), accrue measurable timing differences (e.g., 100 µs). While powerful, this technique can only expose the presence of same-process memory accesses.

At this time, it is unclear whether several state-of-the-art microarchitectural attacks, e.g., those that obtain privilege escalation [40, 41, 42] or extract secrets across processor cores [4, 12, 10], still pose a threat without high-precision timers. A key unsolved problem is finding *eviction sets*, i.e., sets of memory addresses contending for capacity in the last-level cache (LLC) [12, 43, 44, 45, 46]. Moreover, stateless channels appear challenging to amplify, as they are inherently short-lived. Therefore, we ask:

Can cross-core side-channel attacks be mounted with low-precision timers? Can stateless side channels be amplified at all? What are the limits to single-shot microarchitectural amplification?

In this paper, we show that microarchitectural attacks are not foundationally thwarted by restricted timing sources. We present ShowTime, a generic framework to produce considerable timing differences from fine-grained leaks, regardless of their source.

ShowTime composes two phases of independent interest. First, the CONVERT phase transforms an initial microarchitectural leak to make it amenable to enter the second phase, AMPLIFY, which produces a large timing difference depending on its input state.

For the AMPLIFY phase of ShowTime, we develop novel instruction sequences that exhibit considerable differences in runtime, depending on a single microarchitectural state difference. First, we generalize the `leaky.page` amplifier [39] for use in ShowTime, permitting the single-shot amplification of additional events, such as differences in load order or cache line invalidations. We show how to increase its *amplification ratio*, i.e., the ratio between slow and fast executions of the amplifier, from 1.3 to 2.3. Additionally, we develop robustness measures to increase the maximal time difference it can reliably produce, from 500 µs to 5 ms. Second, we discover a powerful single-shot amplifier for adversaries with native code execution. Its amplification ratio exceeds 10, meaning that it takes, e.g., less than 1.1 ms to produce a difference of 1 ms. Due to its unprecedented robustness, it can produce timing differences far beyond any timer coarseness imposable in practice.

For the CONVERT phase of ShowTime, we contribute several conversion techniques, relying on well-known CPU behavior like out-of-order execution, cache line back-invalidation, and thread-level parallelism. They make it possible to convert (single-shot, potentially cross-core, potentially stateless) microarchitectural leaks to make them attacker-visible, attacker-amplifiable, and both.

To show how ShowTime fares with stateless leaks, we develop a proof-of-concept attack to expose port contention on *another processor core* through its delaying effect on following instructions. Though this secret-dependent delay does not exceed 20 ns, ShowTime can capture, convert and amplify its presence or absence. We also demonstrate that ShowTime can be used to reveal information on the CPU frequency at a given point in time, which is an inherently stateless microarchitectural context that has recently been shown to produce severe leakage [47, 48].

Exploring the limits of microarchitectural amplification, we find that our strongest amplifier can generate time differences so enormous that the human eye can classify a single initial cache hit or miss with more than 98% accuracy. In addition, we find eviction sets for the LLC using the Unix Epoch, an excessively crude "timer" reflecting the number of seconds elapsed since January 1, 1970.

We also construct LLC eviction sets in JavaScript with a 100 µs timer, which is the most restricted scenario in the latest Chrome release. The median execution time is 25 s, for an accuracy of 70%. These results show that the ShowTime convert-and-amplify strategy is also successful from the browser, which is a restricted execution context where timers are already limited in practice.

Contributions. Our main contributions are the following:

- We provide a framework to expose fine-grained timing leaks of arbitrary type to coarse-grained timing sources.
- We develop robust amplifiers capable of producing large time differences from unique microarchitectural events.
- We show how to reliably convert activity in one microarchitectural component into controlled activity in another.
- We evaluate ShowTime for cross-core stateless attacks, frequency measurements, and eviction set construction.

We disclosed our findings to Intel and Google.

Availability. To facilitate the reproduction of our research, artifacts are available at

https://github.com/KULeuven-COSIC/ShowTime

2 Background

Cache Hierarchy. Modern processors consume and produce data faster than main memory technology can provide and accept it. To overcome this issue, processors feature a *cache hierarchy*; a series of successively smaller and faster pieces of on-chip memory. Typically, caches are implemented as a two-dimensional array of *cache lines*. This array is indexed into *sets*, to which cache lines are mapped based on their memory address. Lines mapped to the same set are called *congruent*, and the number of congruent lines mapped to the same set is the cache's associativity (or its number of *ways*).

The cache hierarchy on Intel processors comprises three levels. Each core has its own L1 and L2 cache, the two fastest and smallest levels. The L3 or *last-level cache* (LLC) is shared between all CPU cores. Most Intel LLCs abide by an *inclusive* policy, stating that all cache lines in L1/L2 necessarily have a copy in the LLC.

In the event of a *cache miss*, i.e., the cache does not contain the requested memory address, the next level cache is consulted, cascading all the way to main memory in case the request triggers a cache miss in all levels. To install the new line in the cache set, one of its existing entries is selected to be *replaced*

(or *evicted*), and the state machine that governs this selection is the *replacement policy*.

Sometimes, the programmer or compiler may want to instruct the processor to fetch specific data before it is used. On Intel processors, several so-called *prefetch* instructions exist for this purpose.

Cache Attacks. The presence of a shared cache hierarchy implies that processes affect each other's runtime through competitive use of the cache space. This introduces a *timing side channel*. For instance, a malicious process occupying an entire cache set can determine, by measuring the access latency of its own cache lines, whether one of them was evicted by the activity of another process. Such an attack is known as Prime+Probe [1, 12]. Recently, a more precise version, Prime+Scope [46], was proposed, which concentrates the contention with the victim into a single cache line. A key prerequisite for Prime+Probe-style techniques is to find *eviction sets*, i.e., addresses that map to the same set in the target cache.

Other Microarchitectural Leakage. While the cache hierarchy has been the most prominent target for timing attacks, CPU microarchitectures feature several other components that expose processes to the metadata leakage of other processes. Any component competitively shared between potential attacker and victim processes can be the target of a timing attack. Some of these components are core-private, e.g., L1 caches [1], execution ports [16, 17], TLBs [14], and fetch/decode units [49], implying that an attacker needs to obtain core-level co-location with their victim to mount an attack. Other components, e.g., DRAM row buffers [50], and on-chip interconnects [18, 19, 20], are competitively shared across cores, relaxing the co-location requirements for the attacker.

Out-of-Order Execution. To maximally make use of available hardware resources, modern processors implement *out-of-order execution*. This feature exploits instruction-level parallelism, allowing independent instructions to execute as soon as their operands are available, instead of strictly adhering to the order specified by the program's (inherently serial) software description. Out-of-order execution itself may be a source of hardware vulnerabilities [51].

3 ShowTime

3.1 Threat Model

We consider an attacker with unprivileged code execution. The only timing sources available to the attacker are *coarse-grained timers*, i.e., their granularity (e.g., 5 µs, 100 µs, or 100 ms) is several orders of magnitude larger than the timing variations the attacker intends to measure (e.g., 10 ns). We explicitly do *not* assume that the attacker runs on the same CPU core as the victim, that huge memory pages are available, or that the CPU frequency is fixed.

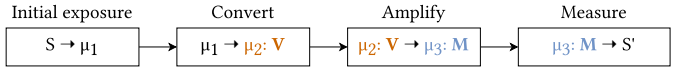


Figure 1: ShowTime framework.

3.2 General Framework

Figure 1 shows the ShowTime cascade for a general microarchitectural side-channel attack. The data flow starts from the initial exposure of secret architectural data (S) to the microarchitectural context (μ_1). At the end, the attacker reconstructs the secret data by measuring a transformed context. For this reconstruction to be possible, the final context needs to have two properties:

- Visible (**V**): the leakage is present in a component shared with the adversary or observable through an explicit interface [52].
- Measurable (**M**): the leakage is strong enough to be picked up by the measurement source. Being measurable implies being visible.

ShowTime aims to achieve measurability for arbitrary initial exposure types with Convert and Amplify phases, which we now describe briefly. In practice, some steps may be repeated, skipped, or reordered.

3.2.1 Initial Exposure

The initial encoding of secret data into the microarchitecture can be categorized according to different criteria:

- Visible (**V**) or invisible: the latter may be the case for leaks in core-private resources such as execution ports [16, 17].
- Unintentional (in a side-channel attack), or intentional (in a covert channel or a transient execution attack [51, 53]).
- Stateful or stateless, i.e., with persisting (stateful) or ephemeral (stateless) interference in the microarchitectural context.

3.2.2 Convert

The Convert step translates exposure in one microarchitectural component into exposure in another. This can be required for multiple reasons. If the initial exposure is not visible (**V**), it can be transformed to a visible encoding in the microarchitectural context. If the initial leakage is not measurable (**M**) and cannot be directly amplified (e.g., it is stateless), it can first be transformed into an amplifiable (e.g., more persistent [54, 55]) state.

Similar to the initial exposure, conversions can be unintentional or intentional. Unintentional conversions can occur through gadgets in the victim

code or implicitly through processor hardware features (e.g., dynamic voltage and frequency scaling (DVFS) converts power differences into frequency differences [47]).

3.2.3 Amplify

Leakage that is visible (**V**) but not measurable (**M**) requires amplification before it can be decoded. An amplifier is a piece of code whose execution time is deliberately made sensitive to a specific difference in the microarchitectural context.

Prior work mostly focuses on constructing *multi-shot* amplifiers, which repeatedly trigger an identical initial exposure [35, 36, 37, 38]. However, attackers cannot always force the victim to repeatedly execute with the same inputs. Moreover, while existing amplification techniques are theoretically capable of achieving arbitrary time differences, it is unclear whether they remain applicable in practice as timing sources get restricted even further, e.g., to 100 ms [56].

In this work, we focus on single-shot amplification. From here on, *amplifier* and *amplification* refer to single-shot methods.

3.2.4 Measure

The final step in ShowTime is to read out the side-channel information in the architectural domain to reconstruct (S') the secret (S). For timing side channels, the architectural value is typically obtained by reading a monotonically increasing value before and after executing the target code. This value can either be readily accessible (e.g., `rdtsc` in x86 or `performance.now()` in JavaScript), or implemented by the attacker [24, 25, 57, 58]. The difference can then be thresholded to recover the initial secret.

Alternatively, the thresholding can also be a part of the measurement, e.g., by testing whether the target executes slower or faster compared to an action with a known execution time [59]. Other software-accessible measurement interfaces include hardware transactional memory [52, 60], on-chip power consumption monitors [61], and the CPU frequency [47] manager. However, such direct interfaces can be or have been disabled [62] or weakened [61]. In this work, we focus on timing side channels.

Restrictions in Browsers. In response to Spectre [53], browsers limit JavaScript features that can be correlated to the precise passage of time, especially in combination with interacting with other websites [31]. Websites can opt into these features by explicitly setting two HTTP response headers, enabling cross-origin isolation.

In current versions of Chrome ($\geq$ 92) [31] and Firefox ($\geq$ 79) [63], `SharedArrayBuffer` is one of these restricted features, as it can be used for constructing a precise timer [24]. In Chrome, the granularity of `performance.now()` is limited to 5 µs

on isolated and 100 µs on non-isolated sites [30, 31, 32]. In Firefox [34] and Safari (WebKit) [33] `performance.now()` is further degraded to a precision of 1 ms. Even before Spectre, the Tor Browser limited its precision to 100 ms [56].

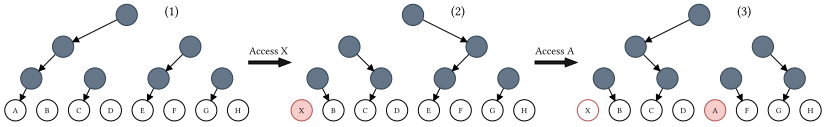


Figure 2: Amplifier based on the L1 replacement policy state.

4 Single-Shot Amplification

This paper studies *single-shot amplifiers*, i.e., unprivileged programs whose execution time depends on a single difference in a microarchitectural context. Not relying on multiple victim code invocations makes these amplifiers applicable in more attack scenarios.

Amplifier Quality. The capabilities of a single-shot amplifier can be quantified by different metrics. Its *amplification ratio* ($A \geq 1$) defines the ratio between the amplifier's slow and fast execution times. The *maximal output timing difference* Δ is the absolute difference between the slow and fast times the amplifier can reliably produce. As we will see, although some amplifiers are theoretically capable of producing arbitrary timing differences, in practice they seem to degrade when a specific timing difference is reached. Finally, the initial microarchitectural contexts the amplifier can capture determines how widely *applicable* the amplifier is.

4.1 Amplification Using the L1 PLRU

PLRU Replacement Policy. Modern Intel processors have an 8-way set-associative L1 data cache, which implements an approximation of the least-recently-used (LRU) replacement policy, dubbed PLRU (for pseudo-LRU). Conceptually, cache lines are organized as the leaves of a balanced binary tree structure (cf. Figure 2). The state of each node of the tree is carried by one state bit, which can be thought of as an arrow, pointing to one of its two children.

On a cache *hit*, i.e., when the requested line can be served directly from the L1d cache, the arrows at each node along the path from the root to the cache line are set to point away from this line. On a cache *miss*, the requested line is loaded into the cache. The line to be replaced (or *evicted*) is selected by following the direction of the arrows from the root of the tree to one of the leaves. Then, similarly to a cache hit, the direction of all traversed arrows is set

to point away from the newly inserted line. The line that is currently cached, but is next to be evicted, is said to be the *eviction candidate*.

Basic PLRU Amplification. `leaky.page` [39] proposes a single-shot amplification technique that captures an *L1 Eviction* event, e.g., the secret-dependent eviction of an attacker line A by a line X that maps to the same L1 set. In particular, their PLRU amplifier repeatedly traverses a sequence of memory loads mapping to specific cache lines (which, to prevent the secret-dependent state from being destroyed, does not include the line X itself). This traversal exhibits a different ratio of L1 hits and misses conditioned on the secret-dependent eviction of A (the *L1 Eviction* event). Given that L1 cache misses take longer to resolve than L1 cache hits, the different hit/miss pattern gives rise to *fast* and *slow* instances. By repeating the traversal until the time difference between the fast and the slow pattern is larger than the timer granularity Δ, the occurrence of the *L1 Eviction* event can be revealed with a low-resolution timer.

Concretely, consider cache lines `A-H`, which all map to the same L1d set. Accessing the preparation pattern `load(AECGBFDH)` (i.e., a load to A, then to E, etc.) produces the initial state of the PLRU tree as in Figure 2, or one that is equivalent to it, up to permuting the two children of each node. Note that this is only guaranteed as long as none of the lines `A-H` are cached prior to the pattern, which is a prerequisite that can be fulfilled by evicting the relevant L1 set. To ensure that the processor does not reorder the loads of line `A-H`, they are serialized through a data dependency [39].

To describe the traversal pattern, we adopt a compact notation. The format is $\texttt{traverse(*)}_{B^n}$, where `*` is one iteration of the base pattern, and B^n implies line B is accessed once every n accesses of the base pattern. Therefore, $\texttt{traverse(AECGFDH)}_{B^4}$ is short for the repeated traversal of `load(AECBGFDBHAEBCGFBDHABECGBFDHB..)`.

Due to the PLRU replacement policy in the L1d cache, all accesses are L1 hits if the *L1 Eviction* event did not occur, and only 25% of them are hits if it did occur. Figure 2 explains why. In case the *L1 Eviction* did not occur, the cache remains in state (1). Naturally, as every element of the traversal pattern is still in L1, all accesses will be hits. If the event did occur, A was evicted from the cache and replaced by X (2). At the same time, E became the next eviction candidate. In the first step of the traversal, we access A, which, since it was evicted, results in a cache miss. Since E is the new eviction candidate, E will be replaced by A, and C becomes the eviction candidate (3). In the next step, we access E, but since it was just evicted, it will result in another miss, etc. The repeated access to B serves to prevent X from being evicted, without accessing X itself.

Improving the Amplification Ratio. To enhance the power of the PLRU amplifier, we propose to perform the traversal with addresses that are congruent in L2. This automatically implies congruence in L1 as well. The traversal

Table 1: Traversal and refresh patterns (novel in bold), along with the sequence of hits (H) and misses (M) they generate. Accesses corresponding to line B are underlined.

Traversal	Hit/Miss (1)	Hit/Miss (2)	Type
`traverse(AECGFDH)`$_{B^4}$	HHHH̲HHHH̲..	MMMH̲MMMH̲..	Distance 1
`traverse(AECGFDH)`$_{B^3}$	HHH̲HHH̲..	MMH̲MMH̲..	Distance 2
`traverse(AECGFDH)`$_{B^2}$	HH̲HH̲..	MH̲MH̲..	Distance 3

Refresh	Type
`load(012B̲345B̲ECGB̲FDHB̲)`	Distance 1
`load(01B̲23B̲4B̲5B̲ECB̲GDB̲FB̲HB̲)`	Distance 2
`load(0B̲1B̲2B̲3B̲EB̲GB̲FB̲HB̲)`	Distance 3

patterns remain the same. However, the penalty for the slow pattern becomes larger, as some of the cache misses need to be served from the LLC instead of L2. We also considered traversing LLC-congruent lines but did not observe an additional penalty compared to L2-congruent lines.

Increasing Robustness. Consider when a competing L1d access to the same set occurs, e.g., by another process running on the same physical core. If, at any point, line X or B is evicted, the amplifier no longer works. The original `traverse(AECGFDH)`$_{B^4}$ sequence is not very robust against this; if another access to the L1 set occurs before any of the accesses to B, X is evicted (i.e., once every four accesses, X is the *first* in line to be evicted in case of a cache miss). Therefore, Table 1 contains more robust sequences where X is at worst *two* (distance-2) or *three* (distance-3) cache misses away from being evicted. This is obtained by accessing B more frequently and comes at the cost of (slightly) decreasing the amplification ratio.

As an optional robustness measure against degradation of the L1 state due to noise, we also propose to *refresh* it periodically. That is, we periodically evict the L1 set with additional lines 0-5 that map to the same set, without affecting the presence or absence of lines X and B. This can occur with the refresh patterns in Table 1. Note that refreshes should only be repeated once every so many traversals, e.g., 128, and hence are negligible for the execution time.

Expanding Measurable Events. We now discuss how the PLRU single-shot amplifier can be generalized to be conditioned on other initial microarchitectural contexts relating to the L1 data cache, i.e., *L1 Reordering* and *L1 Back-Invalidation* (cf. Table 2).

L1 Reordering is captured in the following manner. The L1 PLRU state is prepared as before (i.e., `load(AECGBFDH)`). Recall that line A is the eviction candidate after the preparation. We aim to capture the load order of lines D

Table 2: Amplifying other events in the L1d cache. The adaptor modifies the state difference to match the one in Figure 2, such that identical traversal patterns can be used.

Amplify	Initialize	Event (option 1/2)	Adaptor
L1 Eviction	`load(AECGBFDH)`	`load(X)` / ⊥	⊥
L1 Reordering	`load(AECGBFDH)`	`load(DH)` / `load(HD)`	`load(XFHB)`
L1 Back-Invalidation	`load(AECGBFDH)`	`invalidate(E)` / ⊥	`load(XFHB)`

and H. If D is accessed before H, A remains the eviction candidate. If, instead, H is accessed before D, E becomes the eviction candidate. Now consider another access to line X, serialized to happen after both loads. It evicts either line A or E, depending on the load order of D and H. Then, after accessing a short adaptor sequence (Table 2), traversing the original *L1 Eviction* pattern exhibits the same hit/miss pattern as the *L1 Eviction* event.

L1 Back-Invalidation is captured as follows. The state is prepared as before (i.e., `load(AECGBFDH)`). Line A is the eviction candidate. Consider the event where line E is evicted from the LLC. To satisfy the LLC inclusion property, this triggers a back-invalidation of line E in L1. Now consider another access, to line X, happening after this potential back-invalidation. If the invalidation has occurred, X takes the place of E, since the L1 replacement policy favors filling empty ways. If it did not occur (⊥), X evicts A. Again, a short adaptor sequence makes it behave like the original *L1 Eviction* pattern.

Amplify: L1 PLRU.
PLRU can capture reorderings and invalidations. L2-congruent lines increase the amplification ratio, and distance-2/3 sequences boost robustness.

4.2 Amplification Using prefetchNTA

Non-Temporal Prefetch on Intel x86. `prefetchNTA` is a software prefetch instruction with a *non-temporal hint*, communicating to the processor that this data will not be used multiple times. Its microarchitectural behavior on Intel CPUs was previously studied by Guo et al. [64]. Importantly, lines cached using this instruction are treated differently by the LLC replacement policy. For details on this replacement policy, we refer the reader to prior work [65, 66, 67, 64].

For our purposes, three generic properties are relevant. First, for lines that are not cached, `prefetchNTA` performs a cache line fill in the LLC, but with the highest age, making it very likely to become the eviction candidate. Second, prefetching lines that are already cached in the LLC does *not* affect their LLC

```
.rept 1000  ; repeat at will
    mfence
    prefetchnta(A)
    mfence
    prefetchnta(B)
.endr
```

Listing 1: Prefetch-based amplifier.

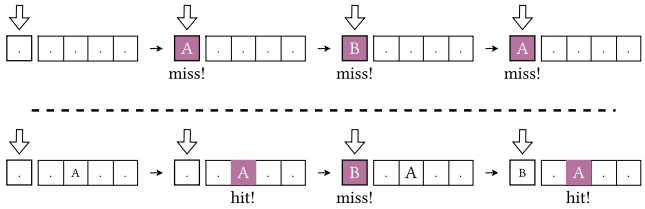


Figure 3: Working principle of the prefetch amplifier (LLC).

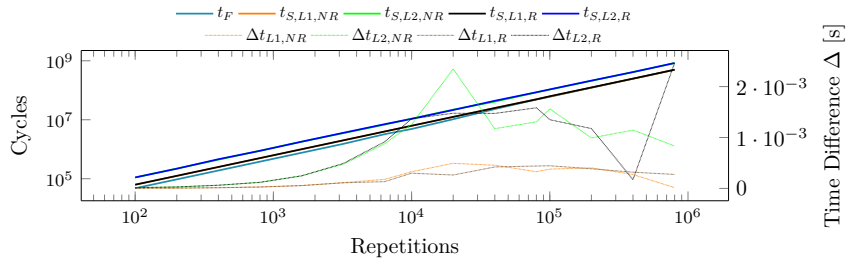
replacement policy state. Third, `prefetchNTA` takes a (much) longer time to execute for lines in memory than for those in the cache.

Technique. Figure 3 shows the working principle of the prefetch-based amplifier. The initial microarchitectural context that conditions the amplifier is the LLC caching state of an attacker line A. Assume that the attacker also has access to a line B, which is not cached but maps to the same LLC set as A. The amplifier is a repeated alternating `prefetchNTA` of lines A and B, serialized with `mfence` instructions to maintain their execution order (Listing 1).

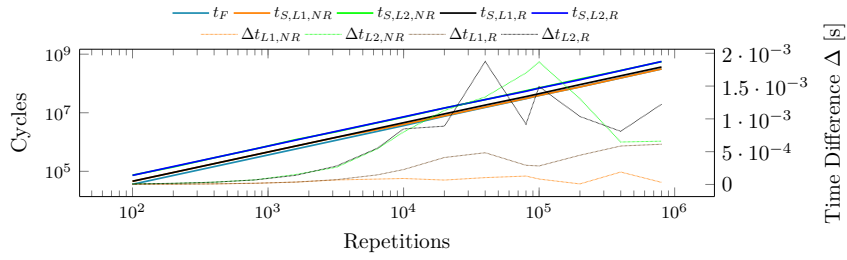
Consider the case where line A is not cached, shown on the top half of Figure 3. The `prefetchNTA` of line A caches it in the LLC as the eviction candidate (indicated by the empty arrow). The `prefetchNTA` of line B evicts A, and installs B as the new eviction candidate. As the pattern is repeated, every prefetch is served from memory, slowing down the execution.

Now, consider the case where line A is cached (but is not the eviction candidate). The first `prefetchNTA` of A is fast and does not affect its replacement state. Although the first `prefetchNTA` of B is slow, it caches B in the LLC as the eviction candidate and, importantly, does not evict A. All future `prefetchNTA`s of A and B are fast, as both lines remain cached without evicting each other.

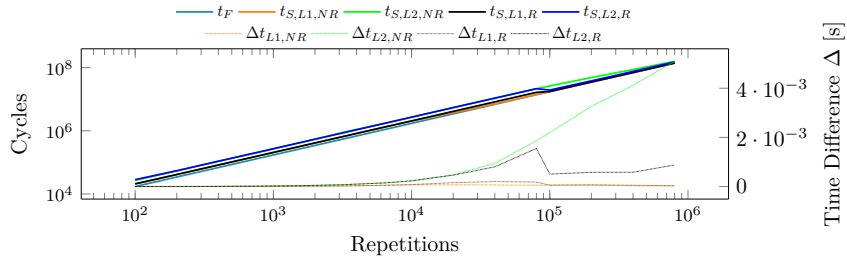
Robustness. The fast and slow instances of the prefetch amplifier share the invariant that there is always a prefetched attacker-chosen line in the cache. Therefore, if there are spurious cache line fills (i.e., *noise*) in the LLC set, it is likely that a prefetched line is evicted. In neither of the fast or slow instances does this destroy the state difference needed to keep the amplifier functional. In the fast case, the spurious access will evict B from the LLC, which will make it be loaded from memory *once*, after which the pattern can continue, as A is



(a) Distance-1 PLRU amplifiers



(b) Distance-2 PLRU amplifiers



(c) Distance-3 PLRU amplifiers

Figure 4: Performance of L1 PLRU amplifiers. The subscripts in the legend denote whether L1- or L2-congruent addresses are used and whether there is a refresh (R) or not (NR). On some subfigures, $t_{s,L*,NR}$ and $t_{s,L*,R}$ may overlap.

still cached normally. In the slow case, the spurious access will evict either A or B from the LLC but, regardless of which one, the next prefetch would have been slow anyway.

Amplify: Non-Temporal Prefetch.
Quick LLC eviction enables robust single-shot amplification.

4.3 Evaluation

PLRU Amplifiers. Figure 4 depicts the fast (t_F) and slow (t_S) traversal times of the L1 amplification patterns as a function of the number of repetitions, along with the time difference they produce. We do not evaluate the *L1 Eviction*, *L1 Reordering*, and *L1 Back-Invalidation* amplifiers separately, since they have identical performance. For each data point, we consider 100 runs of 100 iterations and take the median over all runs. When refresh patterns are enabled, they are accessed once every 1024 traversals (with distance 2). The amplification ratio is constant for small Δ, but as Δ increases, noise accumulates and the amplification ratio degrades until supposedly fast and slow traversals are no longer distinguishable. However, amplifiers vary in their resilience to degradation.

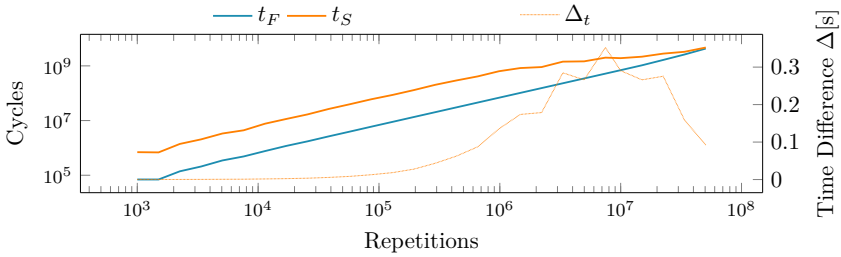


Figure 5: Performance of the prefetch-based amplifier (median of 10 batches of 100 runs per data point).

Prefetch Amplifier. Figure 5 shows the median traversal times using the prefetch amplification method on the Intel Core i7-7700K. The initial amplification ratio exceeds one order of magnitude, which it maintains until roughly 1 billion cycles, after which it declines. Due to its robustness, the amplifier is able to produce time differences of several hundreds of ms from a single initial difference.

Comparison. Table 3 collects the best amplifier instances of each type, along with their amplification ratio A and maximal output difference Δ. It confirms that traversing L2-congruent lines, instead of L1-congruent lines, produces

Table 3: Comparison of single-shot amplifiers.

Source	Amplifier	Single-Shot	A	Max. Δ
`leaky.page` [39]	L1 PLRU (L1-congr., dist-1)	✓	1.3	$\approx 500\,\mu s$
This Work	L1 PLRU (L2-congr., dist-1)	✓	2.3	2.4 ms
This Work	L1 PLRU (L2-congr., dist-2)	✓	2.0	1.8 ms
This Work	L1 PLRU (L2-congr., dist-3)	✓	1.6	5.1 ms
This Work	prefetchNTA	✓	10.1	350 ms

a larger time difference. For the L1-congruent distance-1 amplifier [39], our best implementation achieves a maximal output difference of 500 µs. For the distance-3 sequences, we observe output differences up to 1.5 ms for L1-congruent addresses, and 5 ms for L2-congruent addresses.

In Figure 4, periodic refreshes appear to increase the robustness of the L1 sequences, but no such effect is visible for the L2 sequences.

Measurement Rate. The rate at which timing measurements can be performed for a timing source of granularity Δ is determined by the amplification ratio A of the amplifier. With A defined as the ratio $\frac{t_S}{t_F}$, and $t_F - t_S = \Delta$, this implies that $t_F = \frac{\Delta}{A-1}$ and $t_S = \frac{A \cdot \Delta}{A-1}$. As an example, to produce a timing difference of $\Delta = 100\,\mu s$, a slow measurement for the `leaky.page` PLRU amplifier ($A = 1.3$) takes $\approx 4.3\Delta = 430\,\mu s$. It takes $\approx 1.8\Delta = 180\,\mu s$ for our best PLRU amplifier, and $\approx 1.1\Delta = 110\,\mu s$ for our prefetch-based amplifier. Note that these estimates are only valid for the regimes in which A is constant (and hence independent of Δ). If amplifiers are used beyond their robustness capabilities, they may not even produce any meaningful timing difference anymore (cf. Figure 4).

Practical Considerations. The prefetch-based amplifier relies on the x86 `prefetchNTA` instruction and on Intel's implementation choice of marking prefetched lines for quick eviction from the inclusive LLC [64]. A similar amplifier may be devised to exploit the LLC replacement policy (cf. [66, 54, 68]) without a prefetch instruction, at the cost of a lower amplification ratio. The L1-based amplifiers do not require the exposure of specific instructions and can hence be used in restricted environments (cf. [39] and Section 6.3).

The prerequisites for our single-shot amplifiers are met for a wide range of Intel processors [39, 64]. However, other CPU families are not guaranteed to satisfy them. Still, the existence of single-shot amplification demonstrates that innocuous implementation decisions invalidate high-level security properties that are, at the surface, completely unrelated. We leave an exploration of single-shot amplifiers in other processor families to future work.

5 Converting CPU Side Channels

In this section, our objective is to convert side channels of interest to state differences that are amenable to single-shot amplification.

5.1 Back-Invalidation

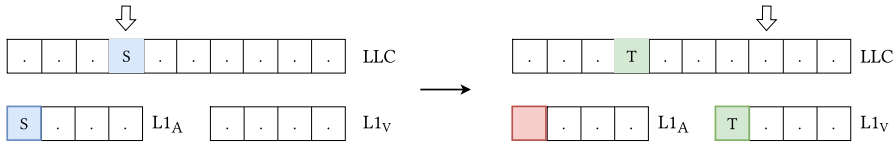


Figure 6: Conversion based on CPU back-invalidation logic. If a line (S) gets evicted from the LLC (by T), all copies of S in the core-private caches get invalidated.

The first technique uses back-invalidation, a deterministic microarchitectural behavior on processors with inclusive LLCs. When a cache line is evicted from the LLC, it is automatically invalidated in the L1 and L2 caches to preserve the inclusiveness invariant. Therefore, the CPU back-invalidation logic produces an implicit conversion from the LLC caching status to an *L1 Back-Invalidation* event, which is amenable to single-shot amplification (cf. Section 4).

With this technique, memory accesses to addresses that map to the monitored LLC set produce the invalidation of a fixed and predictable line in L1. Note that this conversion immediately implies the single-shot amplification of the cross-core Prime+Scope [46] cache attack, which infers LLC activity through the invalidation of a specific line (i.e., *the scope line*) from the L1 cache.

Convert: CPU Back-Invalidation.
LLC evictions automatically produce *L1 Back-Invalidation* events.

5.2 Time to Order

The second conversion technique, Time to Order, exploits the out-of-order execution of instructions on modern processors. Instructions that do not have *data hazards*, i.e., data dependencies on architecturally older instructions, may be executed ahead of these older instructions. Therefore, in an out-of-order processor, the execution order of instructions depends on the time it takes for their dependencies to resolve. As a result, well-designed instruction sequences can encode the latency of specific instruction paths into the execution order of instructions that depend on these paths.

```
dep = prepare-uarch()

// first leg                  // second leg
dep1 = secret-delay(dep)      dep2 = fixed-delay(dep)
dep1 = instr-1(dep1)          dep2 = instr-2(dep2)

race-end(dep1, dep2)
```

Listing 2: Time to Order conversion.

Concretely, as in Listing 2, consider an execution race between two independent legs, which are orchestrated to start at the same time, i.e., through a shared data dependency on another instruction (or the preparation step `prepare-uarch`). One of the legs has a secret-dependent latency, i.e., it contains an operation for which we want to expose the execution time. The other leg has a fixed latency, implemented as an instruction sequence with a fixed execution time (e.g., a sequence of data-dependent multiplications). The length of the fixed-delay sequence is chosen such that the relative execution order of the final instructions of the two legs (resp. `instr-1` and `instr-2`) depends on the latency of the secret-dependent operation. If, in a later stage, the execution order of `instr-1` and `instr-2` can be exposed, it reveals whether the secret-dependent latency is above or below the threshold determined by the fixed-delay leg.

In short, Time to Order turns a timing difference into a difference in the execution order through a microarchitectural race condition. In principle, the timing difference at the *input* (i.e., the event that determines the length of the variable-time leg) can be of arbitrary type. We now show how an instruction ordering at the *output* is amenable to single-shot amplification. Depending on the preparation and the choice of `instr-1` and `instr-2`, time differences can be cascaded to L1 (cf. Section 4.1) or the LLC (cf. Section 4.2).

5.2.1 Conversion to L1 Caching Status.

Table 4 shows how Time to Order can convert a time difference into an *L1 Reordering* event. The microarchitectural state is prepared by filling the PLRU tree as in Section 4.1, and the instructions at the end of each leg are simply the loads as described for the *L1 Reordering* amplifier.

There is no explicit restriction on the type of `secret-delay` that can be converted with Time to Order. Therefore, it is more generally applicable than the back-invalidation conversion (Section 5.1), which has the benefit of being deterministic. A relevant event to capture and convert into the L1 PLRU state is the presence of an LLC hit or miss. Indeed, properly wielding this conversion (see Section 6) reinstates the capability of constructing LLC eviction sets in the browser [10, 43] using the L1 PLRU amplifier, as well as cross-core [12, 10, 46]

Table 4: Time to Order for our single-shot amplifiers.

Amplifier	`prepare-uarch`	`secret-delay`	`instr-1`	`instr-2`
L1 Reordering	`load(AECGBFDH)`	any	`load(D)`	`load(H)`
Prefetch	`evict(A)`	any	`load(A)`	`prefetchNTA(A)`

and cross-process microarchitectural attacks.

Convert: Time to Order (L1).
Encodes a time difference into the L1 PLRU replacement policy.

5.2.2 Conversion to LLC Caching Status.

As a conversion to LLC caching status, Table 4 shows a simple Time to Order instance. The instructions at the end of the legs are, respectively, a `prefetchNTA` and a regular load for the same cache line. If the prefetch comes first, the line becomes the eviction candidate. If the load comes first, it (generally) does not. The resulting LLC state difference can directly be amplified using the `prefetchNTA` sequence (cf. Listing 1).

Convert: Time to Order (LLC).
Encodes a time difference into a line's LLC caching status.

The Time to Order primitive is versatile. With some profiling, seemingly unrelated input side channels (stateless or otherwise) can be converted into LLC/L1 state changes, provided that a race can be found for which the outcome reliably depends on the initial leakage type (e.g., cache line status). Other initial leakage types include time-varying instructions, port contention [16, 17, 57], branch predictor state [23], ROB contention [69, 55], DRAM contention [50] and LLC interconnect contention [18, 19, 20]. We cover some of these examples as case studies in Section 6 but leave a full exploration of all amplifiable CPU side channels to future work.

5.3 Architectural Reordering

The final contribution of this section is *architectural reordering*, a novel integrated conversion and measurement routine (cf. Figure 7). To preserve the semantics of a given instruction stream, the processor always executes store operations to the same address in program order. However, no such guarantees exist for stores in *parallel threads*. By conditioning the order of store instructions in different threads on a timing difference, an attacker can directly produce an

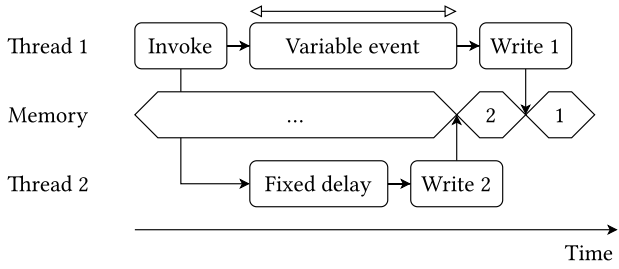


Figure 7: Architectural reordering.

architectural state change without any timing source.

Accuracy. We use architectural reordering to infer the cache state of a line (causing a hit or a miss in L1). We perform 100 runs of 10,000 iterations for a random initial state. The median accuracy is 100% for hits and 99.98% for misses.

Limitations. This technique relies on multiple threads and a mechanism for these to modify the same memory location. The easiest way of accomplishing this in the browser is with Web Workers and shared memory, which already implies the availability of known high-resolution timers [24, 57]. However, Architectural Reordering has an atypical behavioral footprint. Instead of continuously increasing a value and then querying it, which is signature behavior for a timing attack, the attacker thread performs two writes and a read to a single memory location per timing measurement, while the helper thread only performs a single write per measurement.

Convert + Measure: Architectural Reordering.
Eliminates explicit timing measurements by converting time differences into differences in an architecturally visible value.

6 Case Studies

6.1 Extreme Amplification

6.1.1 Human Timers

As discussed in Section 4.3, our prefetch-based amplifier can produce comparatively huge timing differences based on a single initial cache hit or miss. It is worth asking whether the complete elimination of *all* sources of time from the execution environment would thwart CPU timing attacks at a fundamental

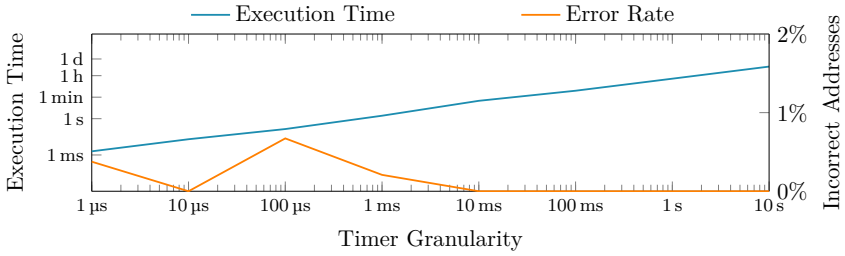


Figure 8: Constructing LLC eviction sets with the prefetch-based amplifier for extremely coarse-grained timers (200 runs for 1 µs-100 ms, 25 runs for 1 s, 1 run for 10 s).

level. To answer this question convincingly, we explore an artificially restrictive setting where there are no timers on the attacker's end, leaving them to rely only on their human perception.

We perform a study on fifteen human participants, aged 20-30. Each participant classifies 100 random single-shot side-channel observations as either fast or slow. The machine is an Intel Core i7-7700K, with which the participants interact over SSH. Participants are exposed to the measurement by a command line tool that prints `Start` (and `Stop`) when the traversal pattern starts (and ends), and are tasked to classify the runs corresponding to a single initial cache hit or cache miss. Based on some manual calibration, we parametrize the amplifier such that it produces a timing difference of 150 ms; the fast traversal takes 16 ms, whereas the slow traversal takes 166 ms. Like in other timing attacks, the participants first calibrate on a small number of practice observations (although, of course, the threshold is perceptual and not numerically quantified).

Together, the participants achieve an average accuracy of 98.4% (median 99%). Several participants achieve a perfect score. Note that the evaluation includes all error sources, such as human error, jitter due to SSH and I/O, and amplifier degradation due to noise.

6.1.2 Eviction Set Construction

As a relevant application, we also implement a routine to construct LLC eviction sets with arbitrarily coarse-grained timers. We do not rely on the availability of huge pages (2 MB or 1 GB), i.e., we only assume attacker control over the lower 12 bits of the physical address (4 KB pages).

We use the eviction set construction method due to Purnal et al. [46], together with the prefetch-optimization due to Guo et al. [64]. The routine tests individual cache lines for congruence in the LLC. To detect congruence, we

```
victim_preamble();
x = calculate(_);        // <--- contention source ---+
load(f(x));              // load that depends on x    |
if (secret) {            //                           |
    _ = calculate(_);    // <--- contention source ---+
}
```

Listing 3: Cross-core port contention leakage.

use the prefetch amplifier. If addresses A and B are congruent, they constantly evict each other in a prefetch loop; otherwise, they do not. As A and B never enter the cache as anything other than the eviction candidate, amplifying this event is even more robust than for a generic side-channel setting (cf. Section 4.3). Lines that demonstrate congruence are accumulated in an eviction set until the desired number of addresses is obtained.

Figure 8 shows the execution time and error rate of the routine for varying timer precisions, on an Intel Core i7-7700K (16-way LLC). The error rate is the fraction of addresses that are not congruent with the randomly generated target. To emulate timers of arbitrary coarseness, we instrument calls to the `rdtsc` hardware counter. For the 1 s-granular timer, we use the Unix Epoch instead, i.e., the number of seconds elapsed since midnight on January 1, 1970.

We even attempt to construct an eviction set using a 10-second granular timer, representing an amplification of 8 orders of magnitude w.r.t. to the timing difference between a cache hit and a cache miss (e.g., 100 ns). With a runtime of less than 6 hours, the attempt is successful.

Amplify: Single-shot amplification up to seconds.
It is possible to amplify microarchitectural timing differences beyond any timer restriction that can be practically imposed.

6.2 Amplifying Stateless Leakage

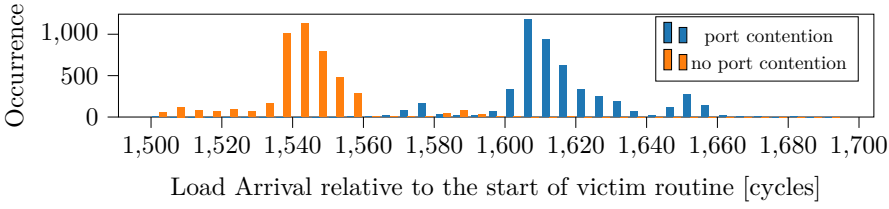


Figure 9: Fine-grained cross-core port contention attack.

```
1 load(BCGAB);  // Reinstate A; makes pattern repeatable
2
3 // first leg                  // second leg
4 x = load(SCOPE);              y = fixed_delay();
5                               y = load(G ^ y);
6
7 load(X ^ x ^ y);              // Evict A or E
```

Listing 4: Repeatable Time to Order conversion.

6.2.1 Cross-Core Port Contention

Consider the code pattern in Listing 3, which leaks the boolean value of `secret` through port contention [16, 17, 38]. If the operations on lines 2 and 5 use the same execution ports, they interfere, delaying each other's execution. As ports are core-private resources, this stateless leakage is not directly visible to processes running on other cores. However, there is an implicit CONVERT performed by the victim code that still transmits this information; the presence or absence of contention introduces a secret-dependent delay on the `load` on line 3.

Fine-Grained Timer. We first expose the secret-dependent time of the memory access using a high-precision timer. Later, we apply ShowTime to decode the same information with a low-precision timer. We instantiate the contention sequence `calculate()` as 16 `vsqrtpd` (floating point square root) instructions. Figure 9 shows how the secret-dependent delay of the memory access, relative to the start of the victim program, can be picked up across cores by the high-precision Prime+Scope [46] attack. Note that the presence of the load itself does not encode any side-channel information, i.e., it happens independent of the secret. The time variation of the LLC eviction is roughly 70 cycles (i.e., less than 20 ns).

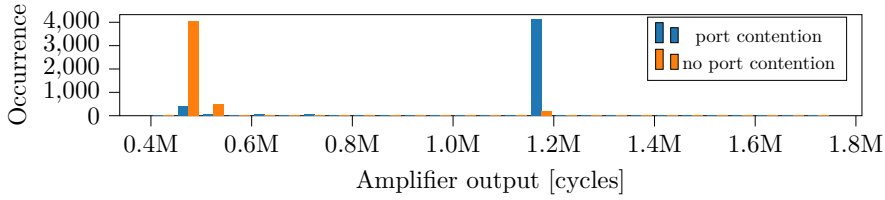


Figure 10: Coarse-grained cross-core port contention.

Coarse-Grained Timer. There are several challenges to exposing these fine-grained time variations to a low-precision timer. First, the CONVERT stage needs to implement an implicit threshold between the histograms in Figure 9, and the result of this threshold should be encoded in a stateful microarchitectural

component from which it can be amplified. Second, the conversion requires a high timing sensitivity, comparable to accessing the scope line, which is already just sufficient enough to reveal the contention (cf. Figure 9).

Our solution is the conversion pattern in Listing 4. The cache state is first prepared as follows. The LLC is prepared as in Prime+Scope, so the victim load will evict a designated cache line, i.e., the scope line. The L1 is prepared with the basic PLRU preparation pattern (cf. Section 4.1). The conversion is made repeatable with `load(BCGAB)`, which evicts line X if it took the place of A (first leg won), but not if it took the place of E (second leg won). Therefore, if the monitored load evicts the scope line during any of the iterations of Listing 4, it is encoded in the L1 state. Before running the attack, we calibrate how often it needs to be repeated, relative to the start of the victim routine, to implicitly implement the threshold. As this conversion pattern takes only 30 cycles on average, it is precise enough to implement the necessary implicit threshold in Figure 9.

The complete ShowTime cascade is as follows. First, there is an unintentional conversion from port contention into an LLC eviction that occurs at a secret-dependent time. This secret-dependent time is converted into an *L1 Reordering* event, where the order depends on whether the LLC eviction occurs during the time that the attacker repeats the conversion pattern. Finally, the *L1 Reordering* event is amplified. Even though this cascade has several moving parts, the results are satisfying, as can be observed in Figure 10.

Discussion. Behnia et al. [54] exploit a code pattern similar to Listing 3. However, they question whether such a minor difference in load timing can be captured by a cache attack on the LLC. Therefore, they require all conversions to take place in the victim code, i.e., the victim itself should encode the time difference in the LLC replacement policy. In our work, we show that this requirement can be relaxed; high-precision LLC cache attacks can exfiltrate minute time differences directly, with and without fine-grained timers.

If an attacker can co-locate a process on the victim's CPU core, the contention may be exposed with a direct Time to Order conversion. We leave an exploration of this setting to future work.

6.2.2 Instantaneous CPU Frequency

Recently, attacks exploiting dynamic voltage and frequency scaling (DVFS) have been proposed [47, 48]. With DVFS, the instantaneous frequency of a CPU changes based on its power consumption which, in turn, may depend on the data being processed. We now explore whether information on the instantaneous CPU frequency can be exposed in the absence of direct interfaces (e.g., `cpufreq`) and fine-grained timers.

Listing 5 (cf. Appendix A) contains the proof-of-concept code pattern. We observe that Time to Order races can be orchestrated to be sensitive to the

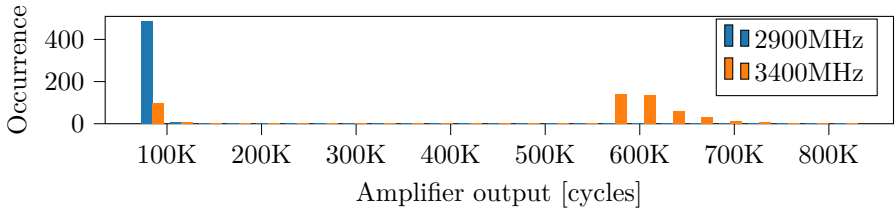


Figure 11: Exposing CPU frequency with crude timers.

instantaneous CPU frequency. We fix the CPU frequency using `sudo cpupower frequency-set` on an Intel Core i5-7500, running Rocky Linux 8.7. We set it to either 3400 MHz (the base frequency of the CPU) or 2900MHz, representing a 15% frequency adjustment. Figure 11 shows that the resulting histograms are clearly distinguishable. To our knowledge, we are the first to remark that the CPU frequency, an inherently stateless microarchitectural property, can be captured, converted, and amplified. We defer a comprehensive study of this phenomenon, as well as the achievable frequency granularity, to future work.

Convert + Amplify: Stateless Side Channels.
Stateless timing leaks can be exposed with coarse-grained timers.

6.3 ShowTime in Restricted Environments

With ShowTime, we can construct LLC eviction sets in JavaScript, which is a key prerequisite for several browser-based attacks, e.g., cross-core Prime+Probe [10], Rowhammer [40, 41], and Spook.js [70]. In addition, finding addresses that are L2/LLC-congruent permits the use of stronger PLRU amplifiers for the remainder of the attack.

With coarse-grained timers, the number of measurements replaces the number of memory references as the bottleneck for the execution time. Therefore, we use the group elimination method by Vila et al. [43] rather than the Prime+Scope method [46]. We modify Vila's JavaScript code [71] to use ShowTime as the measurement. In particular, we use Time to Order to translate the LLC eviction signal to the L1 cache and use the distance-3 PLRU amplifier for robustness (cf. Listing 6 in Appendix A for details.)

We start with an initial set that is a superset of an eviction set with 95% probability (cf. [43]) and exclude the runs where this is not the case. To obtain the ground truth, we verify the correctness of the eviction set using `/proc/pagemap` [71]. On a non-isolated website in Chrome 108, with a `performance.now()` precision of 100 µs, we construct a fully correct eviction

Table 5: Converting CPU Side Channels.

Source	Method	Input Channel	Output Channel
Behnia et al. [54]	OoO Execution	Port/MSHR Cont.	LLC
Aimoniotis et al. [55]	ROB Size	ROB Pressure	L1/LLC
Wang et al. [47]	DVFS	Power	Freq. / Time
Back-Invalidation	**LLC Inclusiveness**	**LLC**	**L1**
Time to Order	**OoO Execution**	**Any**	**L1/LLC**
Arch. Reordering	**Thread Parallelism**	**Any**	**Data**

set in 176 out of 250 runs, with a median runtime of 25 seconds. For this proof-of-concept implementation, we did not exhaust all possible optimization opportunities. As the objective of this work is to study microarchitectural attacks in the face of coarse-grained timers, we also made no attempts to increase the effective precision of the timing sources themselves.

Convert + Amplify: ShowTime in the browser.
ShowTime applies to restricted browser settings. It can be used to construct LLC eviction sets with coarse-grained timers.

7 Related Work

Multi-Shot Amplification. Mcilroy et al. [35] provide a theoretical argument for the availability of arbitrary multi-shot timing amplification on processors implementing optimizations. Wikner et al. [37] and Schwarzl et al. [36] consider multi-shot amplification for mounting Spectre attacks from JavaScript. Some other works cope with low-resolution timers by aggregating the latency of many memory accesses [72, 59], with the drawback of losing all spatial information of the side channel. Rokicki et al. [38] amplify (stateless) port contention from JavaScript in a covert channel setting, where multi-shot measurements are possible.

Multi-shot amplification is also used for the software-based exploitation of physical side-channels (e.g., power consumption [61] and CPU frequency [47, 48]). Future work should investigate the feasibility of single-shot amplification for these attack vectors.

Existing Conversions. Table 5 summarizes the prior work and our contributions in the space of converting CPU side channels. Behnia et al. [54] convert several sources of core-private contention to LLC caching status by exploiting specifics of the LLC replacement policy. Aimoniotis et al. [55] exploit that incorrectly speculated loads only get executed if they fit in the reorder buffer (ROB) [69], converting ROB contention into caching status. Our work contributes simple conversions of several side channels into state differences

that are amenable to single-shot amplification.

Disabling Timing Sources. Browsers already cripple timers [34, 33, 30, 31, 32, 56], but this is also proposed for native (mobile/desktop/cloud) code [28, 27]. Prior work demonstrates that, in some cases, attackers can build [25, 24, 8, 57] or simply bring [73] their own timing sources. However, our work practically demonstrates that even when attackers cannot use these methods, restricted timers are not a holistic countermeasure against timing attacks.

Note that our findings do not threaten the validity of other side-channel countermeasure classes, such as constant-time programming [74], isolation [75, 76] or randomization [77].

Concurrent Work. In concurrent work, Xiao et al. [78] leverage out-of-order execution ("race gadgets") to convert microarchitectural state changes, similar to one of our conversion routines (Time to Order, cf. Section 5.2). However, they do not consider amplifying stateless channels. They also contribute single-shot amplifiers ("magnification gadgets"), including the *L1 Reordering* PLRU amplifier, and others that are not cache-based. Though they suggest that arbitrary amplification can be achieved, they do not demonstrate amplifying timing differences beyond 100 µs. In our experiments, we overcome several practical challenges to obtain timing differences that are larger by one to four orders of magnitude.

Another concurrent work [79] uses transient execution to encode the caching status of one cache line into many cache lines. In this manner, they obtain single-shot amplification of cross-core cache events. Similar to our work, they also construct LLC eviction sets in a browser environment using a 100 µs timer.

8 Conclusion

In this paper, we contributed the ShowTime framework to expose arbitrary microarchitectural timing leaks in a single shot to coarse-grained timers. Our techniques can capture cross-core and stateless microarchitectural leaks, bypass currently imposed timer restrictions, and even amplify nanosecond-range timing differences such that they are detectable by humans.

Acknowledgments

We thank the anonymous AsiaCCS reviewers for their feedback and the humans for participating in the timer study. This research is partially funded by the European Research Council (ERC #101020005) and the Flemish Government through the FWO project TRAPS. It was also supported by the CyberSecurity Research Flanders (#VR20192203), Horizon Europe (#101070008) and the

Research Fund KU Leuven. Antoon Purnal is supported by a grant of the Research Foundation - Flanders (FWO).

References

[1] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at the RSA Conference on Topics in Cryptology (CT-RSA)*, 2006.

[2] Daniel J Bernstein. Cache-timing attacks on AES, 2005.

[3] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *Cryptographic Hardware and Embedded Systems (CHES)*, 2016.

[4] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *USENIX Security Symposium*, 2014.

[5] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space aslr. In *IEEE Symposium on Security and Privacy (S&P)*, 2013.

[6] Yeongjin Jang, Sangho Lee, and Taesoo Kim. Breaking kernel address space layout randomization with intel tsx. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[7] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing smap and kernel aslr. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[8] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. Aslr on the line: Practical cache attacks on the mmu. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[9] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. 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.

[10] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The spy in the sandbox: Practical cache attacks in javascript and their implications. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[11] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*, 2015.

[12] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[13] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A shared cache attack that works across cores and defies VM sandboxing – and its application to AES. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[14] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security Symposium*, 2018.

[15] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 2017.

[16] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[17] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. Smotherspectre: Exploiting speculative execution through port contention. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[18] 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*, 2021.

[19] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. Meshup: Stateless cache side-channel attack on cpu mesh. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[20] Miles Dai, Riccardo Paccagnella, Miguel Gomez-Garcia, John McCalpin, and Mengjia Yan. Don't mesh around: Side-channel attacks and mitigations on mesh interconnects. In *USENIX Security Symposium*, 2022.

[21] Zirui Neil Zhao, Adam Morrison, Christopher W Fletcher, and Josep Torrellas. Binoculars: Contention-based side-channel attacks exploiting the page walker. In *USENIX Security Symposium*, 2022.

[22] Erik Bosman, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Dedup est machina: Memory deduplication as an advanced exploitation vector. In *IEEE Symposium on Security and Privacy (S&P)*, 2016.

[23] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Notices*, 2018.

[24] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in javascript. In *Financial Cryptography and Data Security*, 2017.

[25] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2017.

[26] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *USENIX Security Symposium*, 2016.

[27] Bhanu C Vattikonda, Sambit Das, and Hovav Shacham. Eliminating fine grained timers in xen. In *ACM Workshop on Cloud Computing Security (CCSW)*, 2011.

[28] Robert Martin, John Demme, and Simha Sethumadhavan. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2012.

[29] David Kohlbrenner and Hovav Shacham. Trusted browsers for uncertain times. In *USENIX Security Symposium*, 2016.

[30] W3C. High resolution time. https://www.w3.org/TR/hr-time-3/, 2022.

[31] Google. Making your website "cross-origin isolated" using COOP and COEP. https://web.dev/coop-coep/, 2020.

[32] Google. Align performance api timer resolution to cross-origin isolated capability - chrome platform status. https://chromestatus.com/feature/6497206758539264, 2021.

[33] WebKit. What spectre and meltdown mean for webkit. https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/, 2018.

[34] MDN. performance.now() - web apis | mdn. https://developer.mozilla.org/en-US/docs/Web/API/Performance/now, 2022.

[35] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.

[36] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Robust and scalable process isolation against spectre in the cloud. In *European Symposium on Computer Security (ESORICS)*, 2022.

[37] Johannes Wikner, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Spring: Spectre returning in the browser with speculative load queuing and deep stacks. In *Workshop On Offensive Technologies (WOOT)*, 2022.

[38] Thomas Rokicki, Clémentine Maurice, Marina Botvinnik, and Yossi Oren. Port contention goes portable: Port contention side channels in web browsers. In *ACM SIGSAC Asia Conference on Computer and Communications Security (AsiaCCS)*, 2022.

[39] Stephen Röttger and Artur Janc. A spectre proof-of-concept for a spectre-proof web. https://security.googleblog.com/2021/03/a-spectre-proof-of-concept-for-spectre.html, 2021.

[40] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A remote software-induced fault attack in javascript. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[41] Finn de Ridder, Pietro Frigo, Emanuele Vannacci, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. Smash: Synchronized many-sided rowhammer attacks from javascript. In *USENIX Security Symposium*, 2021.

[42] Andreas Kogler, Jonas Juffinger, Salman Qazi, Yoongu Kim, Moritz Lipp, Nicolas Boichat, Eric Shiu, Mattias Nissler, and Daniel Gruss. Half-double: Hammering from the next row over. In *USENIX Security Symposium*, 2022.

[43] Pepe Vila, Boris Köpf, and José F. Morales. Theory and practice of finding eviction sets. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[44] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher W. Fletcher, Roy H. Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[45] Antoon Purnal, Lukas Giner, Daniel Gruss, and Ingrid Verbauwhede. Systematic analysis of randomization-based protected cache architectures. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[46] 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.

[47] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W Fletcher, and David Kohlbrenner. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *USENIX Security Symposium*, 2022.

[48] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2022.

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

[50] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. Drama: Exploiting dram addressing for cross-cpu attacks. In *USENIX Security Symposium*, 2016.

[51] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security Symposium*, 2018.

[52] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean M. Tullsen. Prime+abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security Symposium*, 2017.

[53] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[54] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS*, 2021.

[55] Pavlos Aimoniotis, Christos Sakalis, Magnus Själander, and Stefanos Kaxiras. Reorder buffer contention: A forward speculative interference attack for speculation invariant instructions. In *IEEE Computer Architecture Letters*, 2021.

[56] The Tor Project. Commit: Bug 1517: Reduce precision of time for Javascript. . https://gitlab.torproject.org/tpo/applications/tor-browser/-/commit/dcd5fcc102a3eb19c20013542fa3ca399db66da4, 2015.

[57] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. Sok: In search of lost time: A review of javascript timers in browsers. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[58] Andreas Kogler, Daniel Weber, Martin Haubenwallner, Moritz Lipp, Daniel Gruss, and Michael Schwarz. Finding and exploiting cpu features using msr templating. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[59] Anatoly Shusterman, Ayush Agarwal, Sioli O'Connell, Daniel Genkin, Yossi Oren, and Yuval Yarom. Prime+probe 1, javascript 0: Overcoming browser-based side-channel defenses. In *USENIX Security Symposium*, 2021.

[60] Daniel Gruss, Julian Lettner, Felix Schuster, Olga Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security Symposium*, 2017.

[61] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *IEEE Symposium on Security and Privacy (S&P)*, 2021.

[62] Intel. Intel Transactional Synchronization Extensions (Intel TSX) Asynchronous Abort. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-transactional-synchronization-extensions-intel-tsx-asynchronous-abort, 2019.

[63] MDN. Firefox 79 release notes for developers. https://developer.mozilla.org/en-US/docs/Mozilla/Firefox/Releases/79#javascript, 2020.

[64] Yanan Guo, Xin Xin, Youtao Zhang, and Jun Yang. Leaky way: A conflict-based cache covert channel bypassing set associativity. In *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[65] Andreas Abel and Jan Reineke. nanobench: a low-overhead tool for running microbenchmarks on x86 systems. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2020.

[66] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. RELOAD+REFRESH: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security Symposium*, 2020.

[67] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. Cachequery: Learning replacement policies from hardware caches. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020.

[68] Samira Briongos, Ida Bruhns, Pedro Malagón, Thomas Eisenbarth, and José M. Moya. Aim, wait, shoot: How the cachesniper technique improves unprivileged cache attacks. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, 2021.

[69] Henry Wong. Measuring Reorder Buffer Capacity, May 2013.

[70] Ayush Agarwal, Sioli O'Connell, Jason Kim, Shaked Yehezkel, Daniel Genkin, Eyal Ronen, and Yuval Yarom. Spook.js: Attacking chrome strict site isolation via speculative execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2022.

[71] Pepe Vila. Tool for testing and finding minimal eviction sets. https://github.com/cgvwzq/evsets, 2019.

[72] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security Symposium*, 2019.

[73] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Double trouble: Combined heterogeneous attacks on non-inclusive cache hierarchies. In *USENIX Security Symposium*, 2022.

[74] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology - CRYPTO*, 1996.

[75] Dan Page. Partitioned cache architecture as a side-channel defence mechanism. In *IACR Cryptol. ePrint Arch. 2005/280*, 2005.

[76] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*, 2016.

[77] Zhenghong Wang and Ruby B. Lee. New cache designs for thwarting software cache-based side channel attacks. In *International Symposium on Computer Architecture (ISCA)*, 2007.

[78] Haocheng Xiao and Sam Ainsworth. Hacky racers: Exploiting instruction-level parallelism to generate stealthy fine-grained timers. In *ASPLOS*, 2023.

[79] Daniel Katzman, William Kosasih, Chitchanok Chuengsatiansup, Eyal Ronen, and Yuval Yarom. The gates of time: Improving cache attacks with transient execution. In *USENIX Security Symposium*, 2023.

Appendix

A Extra Conversion Patterns

```
// Prepare prefetch amplifier
dep = prefetchNTA(A);

// Shared Dependency
dep = load(X + dep);   // Cache miss

// Compute chain races against memory loads,
//  the outcome of this race is frequency-dependent

// First leg               // Second leg
dep1 = load(Y + dep);      dep2 = COMPUTE_CHAIN(dep);
dep1 = load(A + dep1);     dep2 = load(B + dep2);

// evicts A if the first leg won
dep = load(C + dep1 + dep2);

// Amplify time difference
prefetch_amplifier(D, A); // Fast if A is cached
```

Listing 5: Instantaneous frequency measurement. The loads of X and Y are cache misses, and only A-D map to the same LLC set.

```
// Test whether group still evicts victim
dep = load(victim);
dep = evict(candidate_set ^ dep);

// Start timer
start = performance.now();

// Prepare PLRU amplifier
dep = prepare_PLRU(dep);

// first leg                  // second leg
dep1 = load(victim ^ dep);  dep2 = delay(dep);
dep1 = load(D ^ dep1);      dep2 = load(H ^ dep2);

// amplify the difference
amplify_L1(dep1, dep2);

// End timer
end = performance.now();
```

Listing 6: Constructing LLC eviction sets.

Curriculum Vitae

Antoon (Toon) Purnal obtained a MSc degree in electrical engineering at KU Leuven in 2018. In September 2018, he joined COSIC as a PhD candidate. His research was generously funded by a personal grant from the Fund for Scientific Research Flanders (FWO). During the summer of 2022, he interned at Intel Labs to work on microarchitectural security.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Celestijnenlaan 200A box 2402
B-3001 Leuven
antoon.purnal@esat.kuleuven.be
https://www.esat.kuleuven.be/cosic/