Cache Side Channel Attacks on Modern Processors

by

Yanan Guo

M.S., University of Pittsburgh, 2020

Submitted to the Graduate Faculty of

the Swanson School of Engineering in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2024

UNIVERSITY OF PITTSBURGH

SWANSON SCHOOL OF ENGINEERING

This dissertation was presented

by

Yanan Guo

It was defended on

May 31st 2024

and approved by

Samuel Dickerson, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Jingtong Hu, Ph.D., Associate Professor, Department of Electrical and Computer Engineering

Peipei Zhou, Ph.D., Assistant Professor, Department of Electrical and Computer Engineering

Youtao Zhang, Ph.D., Professor, Department of Computer Science, School of Computing and Information

Wenjie Xiong, Ph.D., Assistant Professor, Bradley Department of Electrical and Computer Engineering, Virginia Tech

Dissertation Director: Jun Yang, Ph.D., Professor, Department of Electrical and Computer Engineering

Cache Side Channel Attacks on Modern Processors

Yanan Guo, PhD

University of Pittsburgh, 2024

Modern CPUs feature many microarchitectural structures shared among users. Although such resource sharing offers performance benefits, it also creates opportunities for side channel attacks. Attackers capable of manipulating microarchitectural states can bring these structures into specific states, and then monitor any unintended state changes induced by the victim. Cache timing covert channels and side channel attacks, or cache attacks for short, are extremely potent. Attackers can exploit changes in cache states to leak sensitive information from another user. For performance and efficiency purposes, modern CPUs often include instructions and designs that allow users to directly influence cache states. This inadvertently makes it easier for attackers to manipulate these states, potentially resulting in new and more efficient cache attacks. This dissertation analyzes how these instructions and designs can be exploited for powerful cache attacks and develops mitigation strategies against these attacks.

First, we reverse engineer the prefetch-for-write instruction (`PREFETCHW`) on Intel CPUs and uncover a severe vulnerability on them. Based on this vulnerability, we develop two new cache attacks. These attacks significantly outperform arguably the most prevalent cache attack, Flush+Reload, in both bandwidth and temporal resolution.

Second, we study the non-temporal prefetch instruction (`PREFETCHNTA`) on Intel processors and uncover its unique behavior within the cache hierarchy. This behavior enables a fast route to trigger cache conflicts. We demonstrate that applying this instruction in conflict-based cache attacks can significantly improve the attack performance.

Third, the CPU uncore has been a frequent target for side channel attacks, as it is shared among all users. Many studies suggest using uncore resource partitioning as a countermeasure, given that most uncore attacks stem from resource contention. However, we show that such partitioning is not foolproof. Specifically, we reverse engineer the details of the uncore frequency scaling technique on Intel processors and discover that this technique creates a robust side channel that cannot be stopped by traditional defense designs based on partitioning.

Finally, we study the potential countermeasures against these new attacks and propose defense mechanisms to mitigate each of these attacks with minimal impact on performance.

Table of Contents

List of Tables

List of Figures

Preface

I would like to take this opportunity to thank everyone who has helped me during my Ph.D. journey. This dissertation would not have been possible without their support.

First and foremost, I want to express my deepest gratitude to my advisor, Dr. Jun Yang. Over the past six years, she has provided constant support and guidance to me. I really struggled to conduct research and publish papers for almost the first three years of my Ph.D.. However, during that time, she always believed in me and encouraged me. Dr. Yang is an outstanding scholar and advisor from whom I have learned a lot about both research and life. I really appreciate her for providing me with the opportunity to do my Ph.D. with her. I believe that this will be a memory to cherish forever.

I also wish to thank my committee members, Dr. Youtao Zhang, Dr. Wenjie Xiong, Dr. Peipei Zhou, Dr. Jingtong Hu, and Dr. Samuel Dickerson. I am grateful for their suggestions and for serving on my committee. In addition, I want to thank Dr. Xulong Tang, who started his assistant professor journey around the same time I began my Ph.D. study. He provided extensive help, especially during the challenging time when I struggled to publish papers. His support meant a lot to me.

I am deeply grateful to my parents, Lanfu Liu and Yuanzhang Guo, for their constant love and support. I regret not being able to visit them during my entire Ph.D. journey and feel sad seeing them age over video calls. I really hope to see them in person soon.

I also want to extend my thanks to Andrew Zigerelli. Andrew is the person who got me into this field. He has a deep passion for computer security and loves sharing what he knows about it. Over the last six years, he has greatly enriched my knowledge of computer security. Andrew also provided immense mental support throughout this journey. I often doubt whether I am a qualified researcher, but Andrew constantly and sincerely reassures me that I am doing great in my field. Andrew also took care of many things in my life so that I could focus more on research. I cannot imagine completing this Ph.D. without him.

I also want to sincerely thank Yubo Du. Yubo is a very happy person, and my life became much happier after she joined our group. I truly enjoy spending time with her and greatly appreciate that she stayed with me every day during my last days in Pittsburgh. She is the nicest person I know. Whenever I had complaints or felt disappointed about something, Yubo was always there to listen and understand. During my job search, she made food for me, took care of Lucky, and did everything she could to help. I am so glad to see her doing great in her research. She always says it is because I helped her a lot, but I know it's really because she is so brilliant and hardworking. I have no doubt that she will become an amazing researcher one day, and I wish her all the best on this journey.

I also want to thank Dr. Xin Xin. Xin and I shared the same office for five years, and during this time, I learned a lot from him. Observing how he conducts himself and his research greatly influenced me. When I reflect on the changes in myself from before my Ph.D. study to now, I realize that one of the significant changes is that I have become more like Xin. Xin helped me a lot during these years, especially when I was struggling to get my papers published. He involved me in his project, which provided me with valuable learning experiences. Furthermore, he is an excellent chef; without him, I would have starved during my Ph.D. His support and friendship have been invaluable.

Next, I want to thank Liang. Liang is a unique person in our group. He brought a lot of laughter to our lab, which was one reason I always wanted to work in the lab instead of home. Additionally, I want to thank my other labmates, Aditya Pawar, Congming Gao, Chen Li, and Tyler Garrett. They have helped me a lot throughout my Ph.D.

I want to use this special paragraph to thank Ruobing Han. Ruobing significantly changed my perspective on life. I used to worry a lot about research and publication, but he helped me understand that life is more than just research. There are many other great things we can do. He also provided valuable suggestions when I faced a really tough career choice. I wish him a bright future.

In addition, I want to thank Lucky Guo, my lovely dog who always stayed with me during the tough days. I always worked late at night, which unfortunately kept him up as well. Sometimes I focused too much on research and forgot to feed him on time. But as far as I know, he never complained about it.

Finally, I want to thank Yanan Guo. I was tough and brave throughout this journey, and I am glad that I never gave up and made it to this point. I am truly proud of myself.

1.0 Introduction

Computer security has become very critical today, since a significant amount of personal and sensitive information is stored on computers and other digital devices. Attackers often break into these systems, stealing valuable information and leading to serious privacy breaches, financial losses, and other severe consequences.

One important category of computer threats is side channel attacks. These attacks exploit indirect information flows from a system's physical or logical behavior to gain access to sensitive data. Among these attacks, cache timing side channel attacks (and covert channels), or cache attacks for short, are particularly noteworthy: they are not only easy to carry out but also very difficult to detect. Cache attacks exploit the CPU cache's behavior: different cache behaviors, such as hits and misses, create significant timing differences to the execution of an instruction. Attackers can use these timing variances to infer secrets from a victim, such as cryptographic keys and personal data (e.g., [134, 55, 136, 81, 130, 28, 133, 56, 57, 67, 77, 89, 88, 97, 129, 138, 139, 65, 15, 17, 106, 29, 107, 41, 64]). Since the first successful cache attack in 2005, which leaked an AES encryption key [91], there has been a continuous increase in the number and sophistication of these attacks.

For example, early cache attacks were demonstrated in the local private cache within a CPU core (e.g., [91, 136]), and researchers later found that cross-core attacks on the last-level cache (LLC) are practical as well (e.g., [81, 66]). In addition, early cache attacks only exploit cache hits and misses while the later ones also utilize other cache states, such as ages (e.g., [130, 28]) and dirtiness (e.g., [36]), to leak information. Recently, cache attacks have also become powerful primitives used in transient execution attacks (e.g., [74, 78, 102, 118, 108, 117, 103, 114, 119, 26, 24]). In addition, the growing threat of cache attacks has prompted researchers to develop numerous defense mechanisms against them (e.g., [32, 27, 95, 96, 128, 52]).

Given the prominence of cache attacks, it is crucial to thoroughly understand their landscape. In this dissertation, we aim to uncover previously unknown cache attacks on modern computing systems. We will also analyze whether existing countermeasures can effectively prevent these attacks and whether new countermeasure designs are required. By doing so, we provide valuable insights into the landscape of cache attacks and help enhance the overall security of digital systems, protecting sensitive information from malicious attackers and safeguarding user security and privacy.

1.1 Problem Statement

In many cache attacks, the attacker needs to manipulate the cache and bring it into a known state, and then monitors how the victim's execution changes this state. For performance and efficiency, modern CPUs often incorporate advanced instructions and designs that allow users to *directly influence* microarchitectural states, including cache states. These enhancements enable more efficient processing and improved performance for legitimate applications. However, it also makes it much easier for the attacker to manipulate these hardware states. By exploiting these vulnerabilities, the attacker can potentially launch new and more effective side channel attacks. This issue gets even worse when these new features are implemented with vulnerabilities such as insufficient access controls. This dissertation focuses on analyzing the new cache attacks that arise from the advanced features present in modern CPUs and developing effective mitigation strategies. Specifically, in this dissertation, we seek to answer the following two questions:

- *Whether and how do the advanced designs and instructions on modern CPUs enable new and more powerful cache attacks?*
- *If new attacks exist, what strategies can be employed to prevent them?*

1.2 Research Overview

We focus on the security issues facilitated by two special-purpose prefetch instructions, `PREFETCHW` and `PREFETCHNTA`, as well as the frequency scaling technique on Intel processors. Specifically, we reverse engineer the implementation details of these instructions and this technique, identify the new attacks they cause, and propose countermeasures to address these vulnerabilities. The details of these studies are as follows:

Coherence-based cache attacks. In most cache attacks, the attacker repeatedly evicts the victim's data from one cache level and waits for the victim to load it back. Cross-core cache attacks usually target the last-level cache, evicting the data from it to memory. As a result, there is frequent data transfer between the on-chip cache hierarchy and off-chip memory during the attack. This data transfer is slow, and thus significantly limits the speed and resolution of the attack. In contrast, our study enables two new cross-core cache attacks, in which the data is only moved within the cache hierarchy, markedly enhancing the attack performance. Specifically, we discover a serious security flaw in Intel CPUs related to the prefetch-for-write instruction, `PREFETCHW`. With this vulnerability, the attacker can use `PREFETCHW` to change the cache coherence state of the data. This change intrinsically moves the data between the private caches of different cores, enabling a novel cross-core private cache eviction method. Based on this method, we develop two cache attacks named Prefetch+Prefetch and Prefetch+Reload, respectively. These attacks significantly outperform arguably the most prevalent cache attack, Flush+Reload, showing a 3× increase in bandwidth and a 43× improvement in resolution on certain CPUs.

Cache attacks bypassing set associativity. In conflict-based cache attacks, the attacker deliberately causes cache conflicts to evict the victim data and learn the victim's access pattern. Conflict-based attacks are considered more practical than others, since they usually do not assume any system features. However, they are typically much slower because building conflicts in a cache set requires the attacker to prime the set, i.e., filling the set with the attacker's cache lines, which involves extensive cache operations and takes very long to finish.

We found that on recent Intel CPUs, it is actually possible to build set conflicts in the LLC without priming the LLC set. Specifically, `PREFETCHNTA` is an x86 prefetch instruction for non-temporal data. According to the x86 vendors [3, 8], using this instruction can minimize the LLC pollution caused by data prefetching. We reverse engineer the implementation of `PREFETCHNTA` on Intel CPUs and found that with this instruction, the attacker can cause conflicts in a certain way of an LLC set, without filling the entire set. Consequently, we build the first conflict-based cache attack without priming the cache set. The speed of this attack is over 3× higher than the speed of prior conflict-based cache attacks. This instruction also enables a new eviction set construction algorithm which is w times more efficient than the state-of-the-art, where w is the LLC set associativity (e.g., 16 on some Intel CPUs).

Frequency scaling attacks. A CPU consists of multiple cores and an uncore. The uncore comprises all the on-chip components excluding the cores, such as the last-level cache and the on-chip interconnect. Modern Intel CPUs incorporate uncore frequency scaling, a power management technique that dynamically adjusts the uncore frequency when the CPU operates in low-power mode. Our study shows that while this frequency scaling improves power efficiency, it also introduces new side channel attacks. Specifically, we demonstrate that this technique causes the uncore frequency to vary depending on the workloads running on the cores. An attacker can use the uncore frequency variations to deduce information about other running workloads. This attack has crucial implications for defensive designs. Many research proposals suggest building a secure uncore through uncore resource partitioning, since most existing uncore attacks, such as attacks on the last-level cache, stem from resource contention. However, our study proves that uncore attacks are not just about contention. Even with future partitioning mechanisms in place, the inherent fact that the entire uncore operates at the same frequency could still pose security threats. This underscores the need for more comprehensive secure uncore designs.

Countermeasures. We conduct a thorough analysis of the potential countermeasure mechanisms for the attacks proposed above. We found that the attacks based on the special-purpose prefetch instructions can be prevented using one or more existing countermeasure designs. However, as explained above, the attack based on uncore frequency scaling cannot be mitigated by any existing countermeasures. Therefore, we propose several software-level mitigation solutions that can be immediately implemented to protect users.

1.3 Dissertation Organization

The rest of this dissertation is organized as follows. Chapter 2 introduces CPU cache structures, existing cache attacks and defenses, as well as the objective of this dissertation.

Chapter 3 introduces two new coherence-based cache attacks utilizing the `PREFETCHW` instruction. Chapter 4 introduces a new conflict-based cache attack utilizing the `PREFETCHNTA` instruction. Chapter 5 introduces a new cache attack based on uncore frequency scaling. Chapter 6 discusses the countermeasures against these new attacks. Chapter 7 discusses the related work. Chapter 8 summarizes this dissertation and discusses future work.

2.0 Background

2.1 CPU Cache Architecture

Most CPU caches on modern x86 processors are divided into L1, L2, and L3. The L1 and L2 caches are very fast but relatively small. Typically, they are organized separately for each CPU core, and are thus often referred to as private caches. In contrast, the L3 cache, also known as last-level cache (LLC), is a larger but slower cache, shared among CPU cores.

Caches operate on fixed-size (e.g., 64 bytes) data blocks called cache lines. Additionally, caches are usually set-associative: a cache is organized into multiple cache sets. Every cache set consists of multiple equivalent cache ways, and each of them can store one cache line. The address bits of a cache line determine which cache set that this line is mapped to. Most LLCs in Intel processors are inclusive, meaning that data present in private caches are necessarily also present in the LLC (and conversely, data not in the LLC are not in the private caches). However, recent Intel server processors (e.g., Intel Xeon Scalable processors [14, 132]) use non-inclusive LLCs, i.e., cache lines in private caches may not be present in the LLC. For non-inclusive LLCs, a separate directory structure is required for tracking the cache lines that are in the private caches but not in the LLC.

When a CPU core performs a memory access request, it first checks whether the target cache line is present in its L1 or L2 cache. If present, the request results in a private cache hit; if not, it is a private cache miss and the core must further check the LLC (and the directory for a non-inclusive LLC). If the cache line is found, the request finishes and the data is sent to the CPU. If not, the cache forwards the request to the memory controller, which can read data from DRAM.

2.2 Cache Covert Channels and Side Channel Attacks (Cache Attacks)

Covert channels and side channels are methods used to extract or communicate sensitive information in unintended ways. A covert channel enables the transmission of information between two entities (the sender and the receiver) that are not supposed to communicate directly, bypassing traditional security mechanisms. Side channels, on the other hand, involve the extraction of information by observing and measuring unintended information flow in a system. A covert channel becomes a side channel (a.k.a., side channel attack) when the victim (sender) is unintentionally sending secrets to the attacker (receiver) through the covert channel.

A cache covert channel is a method by which the sender can covertly transmit information to the receiver by manipulating the state of the CPU cache. For example, the sender intentionally brings a certain cache line into cache (or not) to encode 1-bit data, and the receiver decodes the data by measuring the access time of this cache line. Similarly, if a victim application's execution causes cache state changes that can be observed by the attacker (e.g., through timing information), then there exists a cache side channel attack.

In this section, we give an overview of the prior studies on cache covert channels and side channel attacks, as well as the mitigations against them. Note that we refer to "cache covert channels and side channel attacks" as "cache attacks" in the rest of this dissertation.

2.2.1 Existing Cache Attacks

There are typically two types of cache attacks. The first type utilizes the contention on certain cache hardware (e.g., the ring interconnect [90] or L1 cache ports [87, 135]): the attacker passively monitors the latency of accessing this hardware resource to infer the victim's usage of it. Such attacks are usually referred to as contention-based attacks or stateless attacks. The other type utilizes cache states: the attacker actively brings the cache line/cache set to a certain state, then lets the victim execute (which potentially changes the state), and later checks the state again to infer the victim's behavior. Such attacks are often referred to as eviction-based attacks or stateful attacks. In this overview, we focus on stateful attacks because they are more related to the attacks proposed in this dissertation. We can further divide stateful attacks into private cache attacks and LLC attacks, based on the cache level from which the attacker evicts the victim's data.

Private cache attacks. In private cache attacks, the attacker learns the victim's cache behavior by monitoring the state of the victim's data in the private cache. For example, in L1 Evict+Reload, the attacker evicts the victim's data (which is shared with the attacker) from the L1 cache to the L2 cache by building set conflicts, and waits for the victim's execution. Later the attacker accesses this data and times the access to determine it is in the L1 or L2 cache: if it is in the L1 cache, it means the victim accessed the data and brought it back to the L1 cache, otherwise the victim did not access. Private cache attacks could have high-bandwidth since they do not create slow DRAM accesses. However, most private cache attacks require the attacker to be on the same physical core with the victim[1] (e.g., [130, 89]), and many of them further require Simultaneous multithreading (SMT). This significantly limits the attacks, as cloud providers may allocate users to different cores and may disable SMT for security [20, 33, 83].

LLC attacks. In LLC attacks (e.g., [134, 69, 81]), the attacker monitors the state of the victim's data in the LLC. The LLC is usually shared among CPU cores. Thus, different than private cache attacks, LLC attacks do not require the attacker to be on the same core as the victim. These attacks are considered more practical. However, DRAM accesses are usually involved in LLC attacks. To monitor the victim's access on the LLC data, the attacker needs to first evict the victim's data from the LLC to memory. For example, in Flush+Reload [134], the attacker flushes the victim's cache line (which is shared with the attacker) from the LLC (and also the private caches), and waits for the victim's execution. Later the attacker accesses this cache line and times the access to determine it is in the LLC or not: if it is in the LLC (i.e., faster to access), it means the victim accessed this cache line and brought it back to the LLC, otherwise the victim did not access. The bandwidths of LLC attacks are usually bottlenecked by DRAM latencies.

[1] The directory Prime+Probe attack [132] and its optimization, the directory Prime+Scope attack [93], are an exception: they are cross-core private cache attacks. The details of these attacks are discussed later in this section.

Orthogonal to private cache attacks and LLC attacks, we can also divide stateful attacks into two categories based on whether they rely on the existence of shared data (between the attacker and victim).

Attacks with shared data. The required data sharing for these attacks is usually achieved with page deduplication or shared libraries. Eviction+Reload and Flush+Reload mentioned above are two typical attacks that rely on data sharing. In addition, Gruss et al. proposed a variant of Flush+Reload, named Flush+Flush [55], that also requires shared data. Instead of reloading the victim's cache line, it flushes this cache line again and times the flushing to learn the victim's behavior. This attack is stealthier than Flush+Reload because it does not generate any accesses (to the victim's cache line) and is then hard to detect using performance counters.

Instead of checking whether the victim brought the target cache line to cache, some attacks work by checking whether the victim changed the cache state of the target cache line. For example, in prior work [133, 67], the attacker monitors the changes to the coherence state of the target cache line. In Reload+Refresh [28], the attacker instead monitors the changes to the age of the target cache line.

Attacks without shared data. Attacks that do not assume data sharing are arguably more practical: security-conscious operating systems/hypervisors may disable implicit data sharing across processes/virtual machines. Prime+Probe [81, 66] is one of such attacks. The attacker first primes the cache set that the victim's cache line is mapped into by filling the set with her own cache lines. This evicts the victim's cache line. Then after waiting for a period of time, the attack probes this set (by re-accessing the cache lines in the priming stage) and measures the probing latency: if the victim accessed her cache line and brought it back to this cache set, one of the attacker's cache lines was evicted and it now takes longer to probe; if the victim did not access then it is faster to probe. Prime+Probe can work both on the private cache and the LLC. But the LLC Prime+Probe is much more powerful since it can work across CPU cores. *In the rest of this dissertation, we refer to "LLC Prime+Probe" as "Prime+Probe".*

The prerequisite of Prime+Probe is having an inclusive LLC: when the victim's cache line is evicted from the LLC, it is also evicted from the private caches (if present). Yan et al.

later proposed a directory Prime+Probe attack which builds set conflicts in the coherence directory [132], enabling Prime+Probe on platforms with non-inclusive LLCs. Very recently, Purnal et al. presented an optimization of Prime+Probe, named Prime+Scope. In this attack, the attacker primes the target LLC set in a special way such that the eviction candidate in this set is known to the attacker. By doing this, this attack achieves a much higher resolution than Prime+Probe.

Note that there are other ways to classify cache attacks. For example, based on how the attacker evicts the victim's data (from a certain cache level), stateful cache attacks can be divided into flush-based attacks (e.g., [134, 55, 133, 67]) and conflict-based attacks (e.g., [81, 93, 56]).

2.2.2 Existing Countermeasures against Cache Attacks

Researchers have proposed many secure cache designs to mitigate cache attacks over the past decade. Many prior studies suggest defending attacks that require data sharing by simply disabling data sharing across security domains [95, 96, 128]. For attacks that do not require data sharing, we discuss the two most popular countermeasure strategies.

Cache partitioning is considered to be one of the most effective methods for defending cache attacks. It divides the cache into multiple partitions and assigns them to different security domains. This prevents the data from different domains from interfering with each other in the cache, and thus prevents the information leakage across domains. Early solutions (e.g., [72]) divide the cache among its ways: each security domain is assigned a certain number of ways in each cache set, and different domains do not share their ways. Although these designs offer strong security guarantees, they have very poor scalability due to the limited number of ways in each cache set. More recent solutions divide the cache among its sets: each security domain is assigned a group of cache sets (e.g., [39, 113]).

Another approach to defend cache attacks is to randomize the mapping from a memory address (cache line) to a cache set [96, 52, 128, 95]. This approach can also be combined with other techniques (e.g., skewed cache) to further improve the security guarantee. For example, ScatterCache [128] uses a randomized cache mapping and adopts skewed associativity to partition the cache into multiple skews; each skew uses a different (and randomized) set mapping. In addition, PhantomCache [111] uses restricted randomization which limits the randomized mapping of a memory address within only a certain number of sets. Upon a load or store operation, this design requires many cache accesses to check whether the target cache line is in cache, significantly increasing the power consumption.

2.3 Goal of This Dissertation

To protect user security and privacy, it is crucial to prevent cache attacks on CPUs. However, recent advancements in modern CPU design pose significant challenges to this: for performance and efficiency reasons, modern CPUs include features (e.g., instructions) that allow software-level users to *more directly* influence cache states. Unfortunately, this also makes it easier for attackers to manipulate and observe cache states. As a result, new and potentially more powerful cache attacks may emerge on modern CPUs. Therefore, in this dissertation, we aim to first study the new attacks enabled by these advancements in CPU design. Then, we will analyze whether these attacks can be defended using existing secure cache designs and, if not, explore how we can mitigate these attacks.

3.0 New Coherence-Based Cache Attacks with The `PREFETCHW` Instruction

3.1 Overview

Based on how the attacker evicts the victim's data, most cache attacks can be classified into *flush-based* attacks(e.g., [134, 55, 133, 67]) and *conflict-based* attacks(e.g., [81, 93, 56]). Flush-based attacks usually assume data sharing between the attacker and victim. Thus, the attacker directly performs `CLFLUSH` on the victim's data to evict it from all cache levels. For conflict-based attacks, the attacker instead achieves the eviction by constructing set conflicts, i.e., the attacker fills the cache set (that the victim's data occupies) with her own data. Many secure cache designs have been proposed to defend cache attacks. For example, flush-based attacks can be prevented by modifying `CLFLUSH` (to make it a privileged instruction), as suggested in prior work [134, 96, 128, 131]. Conflict-based attacks can be defended by stopping/limiting attackers from discovering congruent addresses [96, 128]. Thus, in this dissertation we present new cache eviction methods to enable practical cache attacks.

`PREFETCHW` is an x86 prefetch instruction introduced in 2000. It is now available on all Intel Xeon Scalable processors and recent Core processors (since Broadwell). According to the technology manual [3], the function of this instruction is to prepare data for *future writes*. It is different from other prefetch instructions (e.g., `PREFETCHT0`) which only move the target cache line closer to the CPU core (i.e., to a higher cache level) to get ready for future accesses. `PREFETCHW` moves the cache line to the requesting core's L1 data cache (L1D cache), as well as sets the coherence state of the cache line to be *Modified*. This can accelerate future write operations from this requesting core, because a cache line in Modified state indicates that 1) the current private cache has exclusive ownership of this cache line, meaning a write operation on this cache line can be directly served by the private cache, and 2) this cache line is already marked as dirty, so the flag (i.e., the dirty bit) does not need to be changed when serving a write operation. For correctness, setting the coherence state of a cache line to Modified causes all copies of this cache line in other cores' private caches to be invalidated [34, 13].

In this dissertation, we make two important observations regarding PREFETCHW on Intel processors. First, although its purpose is to accelerate future writes, PREFETCHW works on data with *read-only* permission. Second, the execution time of PREFETCHW is related to the current coherence state of the target data. With the first observation, an attacker on a different core than the victim can use PREFETCHW on the shared data between the attacker and the victim (which is usually read-only [134]), to evict this data from the victim's private cache. In addition, the second observation means that the attacker can time the execution of PREFETCHW on the shared data between the attacker and victim to learn the coherence state changes of this data, which could be related to the victim's cache accesses.

Based on these two observations, we first propose two new covert channels, named Prefetch+Load and Prefetch+Prefetch. In Prefetch+Load, the sender transmits a bit by prefetching (with PREFETCHW) the shared data between the sender and receiver (for "1") or not prefetching (for "0"). The receiver (on a different core) receives the bit by loading this data and timing the load to determine if it is a local private cache hit (for "0") or a remote private cache hit (for "1"). In Prefetch+Prefetch, the sender transmits a bit by loading (or not) the shared data, and the receiver receives the bit by prefetching the data and timing the prefetch instruction to determine whether the sender loaded. We show that our prefetch-based channels have very high capacities: on our Kaby Lake processor, when only using one shared cache line between the sender and receiver, the capacities are 840KB/s for Prefetch+Load, and 822KB/s for Prefetch+Prefetch, which are *the highest single-line capacities among all existing CPU cache covert channels.*

We then modify the covert channels and build the Prefetch+**Re**load and Prefetch+Prefetch side channel attacks, which can be used to leak the victim's access patterns, similar to previous cache attacks (e.g., [134, 55, 81, 130, 28, 133, 136]). Prefetch+Prefetch can be directly used as a side channel attack by letting the victim be the sender, and the attacker be the receiver, since in this attack the sender transmits signals by accessing (or not) the shared data. However, in Prefetch+Load, the sender is sending signals by prefetching (or not), which is unlikely a side channel. Thus, we modify it and build Prefetch+Reload, where the attacker owns two threads running on different cores. The attacker first uses one thread to prefetch and waits for the victim's possible access, and then reloads using the other thread. When the attacker reloads, she will get a last level cache (LLC) hit if the victim accessed this data; otherwise she will get a remote private cache hit. Then, the attacker can determine the victim's behavior using timing information: a remote private cache hit and an LLC hit take different amounts of time to finish. We show that our attacks can be deployed on Intel processors to leak secrets from real-world applications, and that they can be used in transient execution attacks, making those attacks faster (and more potent) than before. To the best of our knowledge, our prefetch-based attacks are the first cross-core *private cache* side channel attacks that can work with both inclusive and non-inclusive LLCs: in our attacks, the victim's data is only evicted from the private cache but never the LLC.

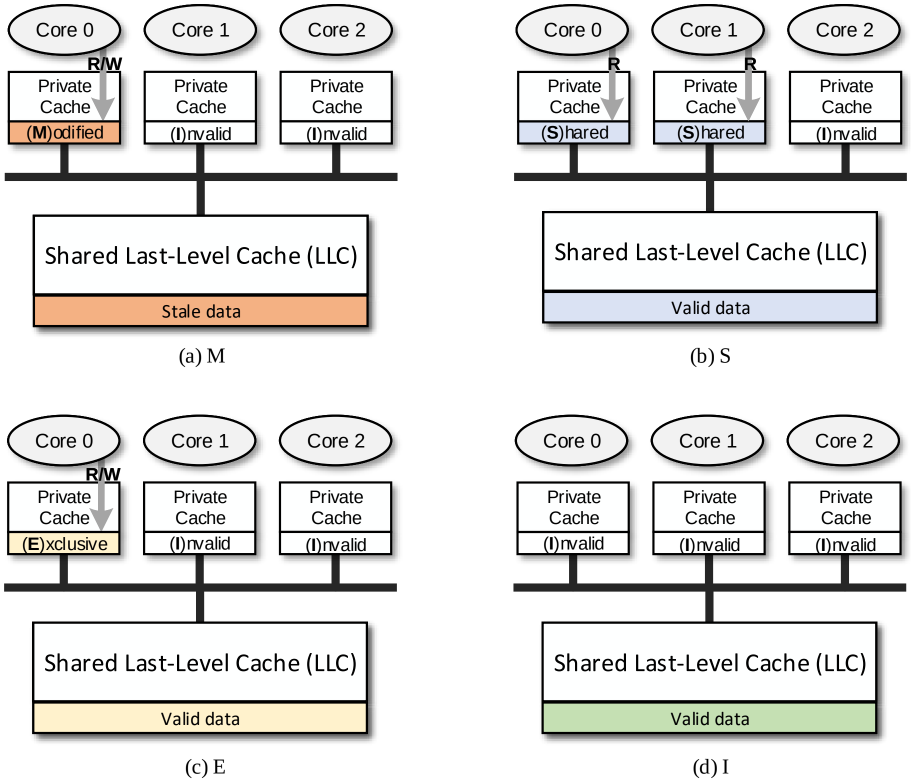
In this chapter, we refer to "prefetch using `PREFETCHW`" as "prefetch".

3.2 Cache Coherence

In multi-core systems, a cache line can be present in multiple private caches, due to data sharing. A cache coherence protocol[1] is required for maintaining data consistency among the copies of a cache line in different private caches: each private cache line is assigned a coherence state, and the LLC needs to track this state to prevent the use of stale data. For inclusive LLCs, the coherence states of private cache lines are stored together with the tag array in the LLC since all the private cache lines are also in the LLC. For non-inclusive LLCs, the directory structure mentioned earlier is used for storing the coherence states of cache lines that are in the private caches but not the LLC.

Most modern x86 processors use variants of the MESI coherence protocol [13, 34]. In the rest of this section, we use inclusive cache as an example to introduce MESI. For non-inclusive caches, the protocol is essentially the same, except that a cache line in a private cache might not be present in the LLC. With MESI, there are four possible states of a private cache line:

[1] In this dissertation, we only focus on the cache coherence inside a processor; this should not be confused with the coherence among sockets (processors) [67].



(a) M (b) S (c) E (d) I

Figure 1: The four possible states of a private cache line, when using the MESI protocol.

- ***Modified (M)***, in which the cache line is only present in one private cache and is dirty, i.e., the copy of this cache line in the LLC contains stale data (Figure 1(a)). Additionally, when a private cache line is in M state, the current owner core has read/write permission for it.
- ***Shared (S)***, in which the cache line is present in one or more private caches and is clean, i.e., the data of this cache line matches all other copies (both in other private caches and the LLC). The current core can only read this cache line (Figure 1(b)).
- ***Exclusive (E)***, in which the cache line is only present in one private cache, and is clean (Figure 1(c)). The current core can read/write this cache line; however, a write operation will change the state of this cache line to M.
- ***Invalid (I)***, in which the cache line is invalid, and thus the current core has neither read nor write permission for it (Figure 1(d)).

With MESI, a memory request from a CPU core will sometimes 1) change the coherence state of the target cache line, and 2) take different amounts of time to finish, depending on the coherence state of the target cache line.

State transitions. There are many different coherence state transitions, we only discuss the two scenarios related to our attacks. First, as shown in Figure 2(a), when a CPU core (core 1) is reading a cache line that is present in the LLC and the private cache of another core (core 0) in M state, this read request will first miss in its private cache and then search the LLC. Although this target cache line can be found in the LLC, its content is potentially stale. Thus, the LLC will fetch the data from the owner private cache (in core 0) that contains the up-to-date data of this cache line, change the coherence state of this cache line in that private cache (in core 0) to S, update the content of this cache line in the LLC, and then return the updated cache line to the requesting core (core 1) as well as fill its private cache with a copy of this cache line in S state. Thus, after serving this read request, the target cache line is present in two private caches, and is in S state in both caches, as shown in Figure 2(b). This case is usually referred to as *remote private cache hit*.

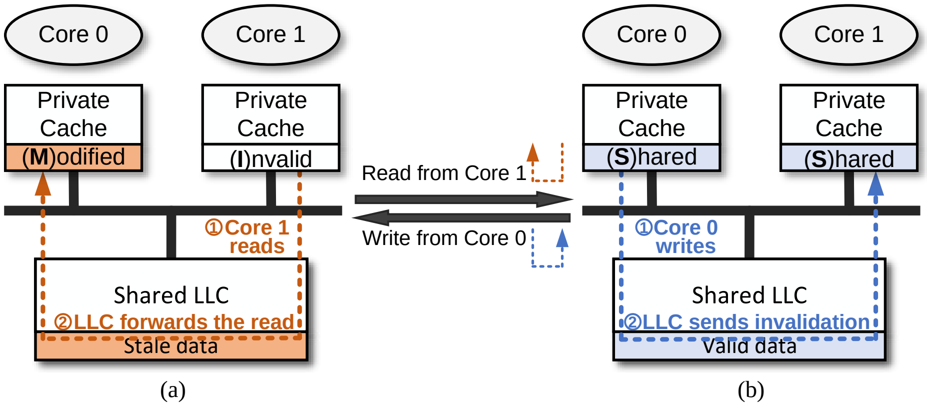


Figure 2: The illustration of cache coherence state changes. The state of a line changes from M (shown in (a)) to S (shown in (b)) when a CPU core is loading it; conversely, the state changes from S to M when a CPU core is writing it. Dashed lines shows the request path of the read/write operation.

Second, as shown in Figure 2(b), when a CPU core (core 0) is trying to write a cache line that is in S state in its own private cache, this private cache (in core 0) needs to send request to the LLC to acquire write permission before it can serve this write operation. As a result, the LLC will send invalidation signal(s) to the other private cache(s) that the cache line is present in (in core 1), and then change the state of the cache line in the private cache of the requesting core (core 0) to M so that the requesting core can write this cache line in its private cache. Thus, after this write operation, the target cache line is only present in the requesting core's private cache, and is in M state, as shown in Figure 2(a).

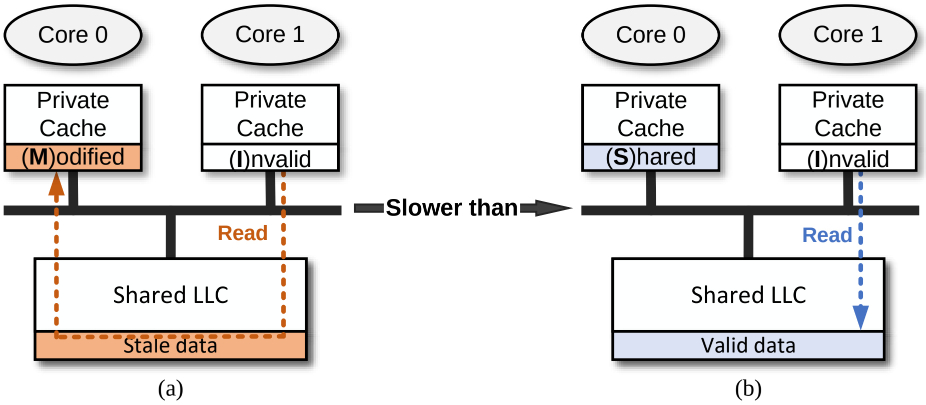


Figure 3: The illustration of an LLC access with the target cache line in M state (a), and S state (b).

Timing difference. As one can observe in Figure 3, if a CPU core is reading a cache line that is not present in its private cache but is present in the LLC, the total latency it takes to finish this read request can be different when this cache line has different coherence states: a remote private cache hit is much slower than an LLC hit. When another core has a copy of this cache line in M state in its private cache, this request results in a remote private cache hit. As explained earlier, serving this request will require fetching data from the owner private cache. In contrast, when all the private cache copies of this cache line are in S state, the data of this cache line in the LLC is up-to-date. This means the LLC can serve this read request directly, resulting in an LLC hit. Due to these different execution paths, an LLC hit is much faster than a remote private cache hit. This has been observed by previous work [133] and has been verified in our experiments. On an Intel Core i7-6700 processor, an LLC hit takes less than 60 cycles to finish and a remote private cache hit takes about 90 cycles.

3.3 Prefetch

Prefetch is a technique to boost performance by fetching data and placing them closer to the CPU core (e.g., from the LLC to L1 cache) before they are needed. Prefetch can be performed in two ways: 1) hardware prefetch, which is implemented in cache hardware and is transparent to users (e.g., the adjacent cache line prefetcher); 2) software prefetch, which needs to be explicitly done by the programmer/compiler. Recent x86 CPUs offer many different instructions for software prefetch, such as `PREFETCHT0`, `PREFETCHT1`, `PREFETCHT2`, `PREFETCHNTA`, and `PREFETCHW`.[2] These instructions are used to hint the processor that a memory location is very likely to be accessed soon [3], then the processor will prefetch the corresponding data into certain level of cache, thereby accelerating future accesses to this data. Software prefetch is an important way to improve performance. For example, compilers sometimes inject prefetch instructions to accelerate for loops.

3.4 Characterizing The Prefetch-For-Write Instruction

Among the prefetch instructions discussed in Section 3.3, `PREFETCHW` (or `PREFETCHWT1` on some CPU models) works slightly differently than the others. It not only brings the data close to the CPU core, but also changes the coherence state of the data: `PREFETCHW` places the target data cache line into the L1D cache[3] and sets the coherence state of this cache line to M. According to the technology manual [12, 3], the role of `PREFETCHW` is to *accelerate future writes on the target cache line.* As explained in Section 2.1, the CPU core

[2] Some CPU models (e.g., Intel Xeon Phi Processor 7200) use `PREFETCHWT1` instead of `PREFETCHW`.

[3] `PREFETCHW` can only be used on data but not instructions [12, 3]. Thus, the cache line will be brought into L1D cache.

can directly write a cache line in its local L1 cache iff the state of this cache line is E/M. Thus, `PREFETCHW` pre-sets the coherence state of the target cache line to M so that future writes on this cache line will likely have an L1 hit. `PREFETCHW` sets the cache line state to M instead of E because writing a cache line in E state results in changing the state to M, and thus has higher latency than writing a cache line that is already in M state.

Most of the recent Intel desktop and server processors (since the Broadwell microarchitecture) support `PREFETCHW`. When used appropriately, it can significantly improve performance. However, we make two observations about `PREFETCHW` on Intel processors, which can be leveraged to create security vulnerabilities.

Observation 1 *`PREFETCHW` successfully executes on data with read-only permission.*

We observe this by monitoring the coherence state changes of the data, using timing information. Specifically, as shown in Listing 3.1, we run a program with two threads ($thread_0$ and $thread_1$, both in one process), and pin them on different physical cores. We use `mmap` [6] to map part of a system file (e.g., glibc) as a read-only data block (in cache line size) in this program and name it d_0: both threads can only read d_0. If any thread tries to write d_0, it will trigger a segmentation fault. $thread_0$ and $thread_1$ both consist of a for loop with the same amount of iterations. In each iteration, $thread_0$ first executes, then waits for $thread_1$ to execute. After $thread_1$ finishes this iteration, they both move to the next iteration and repeat this procedure again. We use pthread mutex locking [7] to ensure that in each iteration $thread_0$ and $thread_1$ execute sequentially (the implementation details of locking is omitted in Listing 3.1).

We run the code in Listing 3.1 twice: in the first experiment (i.e., expt_idx = 0 in line 3), in each iteration of the for loop, $thread_0$ performs `PREFETCHW` on d_0, and then $thread_1$ loads d_0 as well as times the load. In the second experiment (i.e., expt_idx = 1 in line 3), in each iteration $thread_0$ stays idle and then $thread_1$ still loads d_0 and times the load.

Figure 4 shows a segment of the timing results from $thread_1$ (in line 15) in both of the above experiments on an Intel Core i7-6700 processor. Note that we observe similar results on other Intel processors that support `PREFETCHW`. In experiment 0, $thread_0$ prefetches d_0 in each iteration, which causes $thread_1$ to take around 90 cycles to load d_0 after the prefetch.

```
void* thread0 (void* addr_d0, int expt_idx){
    for(int i = 0; i < 1000000; i++){
        /* check the experiment index*/
        if(expt_idx == 0){
            /* execute prefetchw on d0*/
            prefetchw(addr_d0);}
        /*let thread1 execute 1 iteration*/
        wait_for_thread1();
    }}

void* thread1 (void* addr_d0){
    for(int i = 0; i < 1000000; i++){
        /*let thread0 execute 1 iteration*/
        wait_for_thread0();
        int result = read_and_time(addr_d0);
    }}


int main() {
    /* open and map a file as read-only*/
    int fd = open(FILE_NAME, O_RDONLY);
    int* addr_d0 = mmap(fd, PROT_READ, ...);

    /*pin thread0 on core0 and launch it*/
    /*pin thread1 on core1 and launch it*/
    ...
```

Listing 3.1: The code snippet to verify Observation 1.

In contrast, in experiment 1, $thread_0$ stays idle, which causes $thread_1$ to take only around 30 cycles to load d_0. This timing difference infers that d_0 is in different states in the above two experiments. In experiment 0, every time when $thread_0$ prefetches, it will load d_0 to its private cache and set the coherence state of it to be M. According to MESI, explained in Section 2.1, this will invalidate the copy of d_0 in the private cache of $thread_1$ (if it exists). Therefore, when $thread_1$ later loads d_0, it will have a *remote private cache hit* (see Figure 2). This load also changes the state of d_0 from M to S and fills a copy of it in the private cache of $thread_1$. Thus, the same cache behavior (i.e., invalidating the copy of d_0 in the private

```
void* thread0 (void* addr_d0, int expt_idx){
    for(int i = 0; i < 1000000; i++){
        /* check the experiment index*/
        if(expt_idx == 0){
            read(addr_d0);}
        /*let thread1 execute 1 iteration*/
        wait_for_thread1()
    }}

void* thread1 (void* addr_d0){
    for(int i = 0; i < 1000000; i++){
        /*let thread0 execute 1 iteration*/
        wait_for_thread0();
        int t1 = rdtsc(); /* get time stamp*/
        prefetchw(addr_d0);
        int result = rdtsc()-t1;
    }}

int main() {
    /* open and map a file as read-only*/
    int fd = open(FILE_NAME, O_RDONLY);
    int* addr_d0 = mmap(fd, PROT_READ, ...);

    /*pin thread0 on core0 and launch it*/
    /*pin thread1 on core1 and launch it*/
    ...
```

Listing 3.2: The code snippet to verify Observation 2.

cache of *thread*$_1$) will happen when *thread*$_0$ prefetches in the next iteration. However, in experiment 1, since *thread*$_0$ is not prefetching, when *thread*$_1$ loads d_0, it will very likely have a *local private cache hit*, which is much faster than a remote private cache hit (30 cycles[4] vs. 90 cycles on the tested processor).

Rationale. We observe reliable cache state changes on read-only data when executing `PREFETCHW`, with a F-score of 1.0 (n = 1000000). This indicates that Intel processors very unlikely perform a *write permission check* when executing `PREFETCHW`. This does not cause

[4] Due to the granularity of time stamp counters, this measured latency is in fact longer than the real private cache access latency.

any error in the architecture level, because PREFETCHW only has microarchitectural effects: although it can get a cache line ready for future writes, if later the program without write permission for this cache line actually tries to write it, it will still trigger a fault and likely terminate the process. However, later we will show that allowing coherence-based cache invalidation (which should only happen upon writes) on read-only data cause significant security problems. This is because in cache attacks based on shared memory, the attacker can manipulate the coherence state of the shared data (which is usually read-only) between the victim and attacker to monitor the victim's access to this data.

Observation 2 *The execution time of* ***PREFETCHW*** *is related to the coherence state of the target cache line.*

We observe this with the program shown in Listing 3.2. We still use two threads pinned on different physical cores, and let them execute sequentially in each iteration of the for loop. Again, we run the program twice: in experiment 0 (expt_idx = 0 in line 3), in each iteration, $thread_0$ loads d_0, and then $thread_1$ performs PREFETCHW on d_0 and times the prefetch. In experiment 1, $thread_0$ stays idle and $thread_1$ still prefetches and times the prefetch in each iteration.

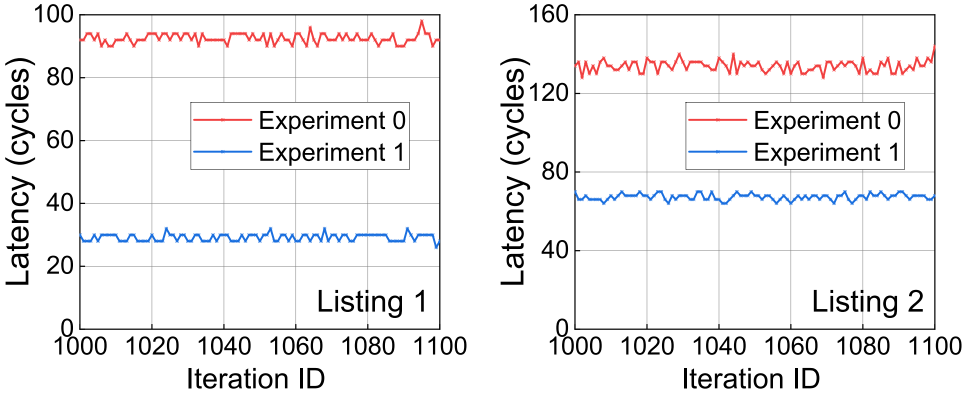


Figure 4: The timing measurement results in $thread_1$ of Listing 3.1 and Listing 3.2.

Figure 4 shows the execution time of PREFETCHW observed by $thread_1$ (in line 16) on our Intel Core i7-6700 processor in both experiments. In the first experiment, it always takes around 130 cycles for `PREFETCHW` to finish; however, in the second experiment it only takes around 70 cycles. This is because in the first experiment, after $thread_0$ loads d_0, the state of d_0 becomes S, and a copy of d_0 is filled into the private cache of $thread_0$ (see Figure 2). Then when $thread_1$ prefetches, it needs to change the state from S to M, which means it has to inform the LLC to invalidate the copy of d_0 in the private cache of $thread_0$. However, in the second experiment, since $thread_0$ stays idle, when $thread_1$ prefetches, d_0 is already in M state. Thus, in this case `PREFETCHW` does not cause any state change and finishes much faster.

Table 1: The evaluated processors for the two observations.

Processor	Microarch.	LLC Type	Observ.1	Observ.2
Intel Core i7-6700	Skylake	Inclusive	✓	✓
Intel Core i7-6800K	Skylake	Inclusive	✓	✓
Intel Core i7-7700K	Kaby Lake	Inclusive	✓	✓
Intel Core i9-10900X	Cascade Lake	Non-incl.	✓	✓
Intel Xeon Silver 4114	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8151	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8124M	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8175M	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8259CL	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8275CL	Skylake-SP	Non-incl.	✓	✓
Intel Xeon Platinum 8375C	Ice Lake	Non-incl.	✓	✗

Affected processors. We have tested these two observations on many Intel processors including the available 1st/2nd/3rd Generation Intel Xeon Salable Processors on AWS EC2, and five Intel desktop/server processors we own. As shown in Table 1, Observation 1 is valid on all the tested processors, and Observation 2 is valid on most, excluding the Intel Xeon Platinum 8375C processor. On this processor, there is no difference on the execution time of PREFETCHW when the target data is different coherence states: PREFETCHW always takes 70 to 80 cycles to finish, even when the target data is not already in M state.

In general, we believe that Observation 1 should be valid on all Intel processors that support PREFETCHW, and Observation 2 should be valid on most of them. Note that all 1st/2nd/3rd Generation Intel Xeon Scalable processors and most Intel Core i7/i9 processors (other than the early generations before Broadwell) support PREFETCHW.

3.5 Covert Channels Based on PREFETCHW

Based on the observations in Section 3.4, we build two new cache covert channels: Prefetch+Load and Prefetch+Prefetch. In this section, we first introduce the threat model, then discuss the details of each attack.

3.5.1 Threat Model

We assume that the two essential parties in the attack, the sender and receiver, are two unprivileged processes that are running on the same processor with multiple CPU cores. The sender and receiver can launch themselves on different physical cores (e.g., using `taskset` [11]). We also assume that the sender and receiver can share data; however, the shared data can be read-only (e.g., via shared library or page deduplication),[5] similar to previous attacks [133, 130, 55, 134, 28]. In addition, the sender and receiver should agree on pre-defined channel protocols, including the synchronization, core allocation, data encoding, and error correction protocols. We do not have requirement on the LLC inclusivity; our attacks work with both inclusive and non-inclusive LLCs. We also do not require SMT; SMT can be turned off for security.

[5] Page deduplication (a.k.a kernel same-page merging [19]) was originally created for virtual environments but is now included in OSs. Although many cloud providers no longer use it, it is usually still available in OSs [28].

3.5.2 Prefetch+Load

We build the first covert channel, Prefetch+Load, following Observation 1. Algorithm 1 shows the details of it. In this attack, the sender and receiver first agree on the shared cache line used to transmit information. Then in each iteration of the attack, the sender transmits a bit "1" by performing `PREFETCHW` on the shared cache line, or a bit "0" by idling. The receiver loads the same cache line and times the load to determine if it is a remote private cache hit or local private cache hit: the receiver receives a bit "1" when having a remote private cache hit, and otherwise receives a bit "0".

Note that different than the experiments in Section 3.4, the sender and receiver cannot synchronize using pthread mutex locking, since they do not belong to the same process. Thus, we let the sender and receiver synchronize the transmission using time stamp counters (TSCs), as done in prior covert channels (e.g., [130, 133, 134, 55, 93]).

Algorithm 1: Prefetch+Load Covert Channel

```
line0: the shared cache line between the sender and receiver
message[n]: the n-bit long message to transfer on the channel
Th0: the timing threshold for distinguishing local and remote private cache hit

Sender Algorithm

// Send 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    sync_with_receiver();
    if $message[i] == 1$ then
        Prefetch line0;
    else
        Do not prefetch;

Receiver Algorithm

// Detect 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    sync_with_sender();
    Access line0 and time the access;
    if $access\_time > Th0$ then
        Received a bit "1";
    else
        Received a bit "0";
```

3.5.3 Prefetch+Prefetch

Our second attack, Prefetch+Prefetch, is based on Observation 2. As shown in Algorithm 2, in each iteration of the attack, the sender transmits “1” by loading the shared cache line, or transmits “0” by idling. After this, the receiver performs PREFETCHW on the shared cache line and times the prefetch to decode the bit: when the sender sends “1”, it takes longer for the receiver to prefetch than when the sender sends “0”. Prefetch+Prefetch follows the same synchronization method with Prefetch+Load.

Algorithm 2: Prefetch+Prefetch Covert Channel

```
line0: the shared cache line between the sender and receiver
message[n]: the n-bit long message to transfer on the channel
Th0: the timing threshold on PREFETCHW to distinguish M and S states

Sender Algorithm

// Send 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    sync_with_receiver();
    if $message[i] == 1$ then
        Load line0;
    else
        Do not load;

Receiver Algorithm

// Detect 1 bit in each iteration.
for $i = 0$; $i < n$; $i++$ do
    sync_with_sender();
    Prefetch line0 and time the prefetch;
    if $prefetch\_time > Th0$ then
        Received a bit “1”;
    else
        Received a bit “0”;
```

3.6 Side Channel Attacks Based on `PREFETCHW`

3.6.1 Basic Idea and Assumptions

In the Prefetch+Prefetch covert channel, the sender is sending the signal by "accessing (or not) the shared data". Thus, this attack can be directly applied as a side channel attack to leak a victim's *access pattern* on the shared data: the victim is the sender, and the attacker is the receiver. This leakage (the victim's access pattern) is same as the one in previous cache attacks (e.g., [55, 134, 133, 56]).

However, Prefetch+Load cannot be directly used as a side channel, because the sender is transmitting the signal by "prefetching (or not) the shared data". In other words, the attacker (receiver) can only detect the victim's (sender's) prefetch patterns on the shared data. Since software prefetch is not as common as memory accesses in real-world applications, the attack opportunities are limited. However, we can modify the attack slightly to make it work more generally.

We term the new attack *Prefetch+**Re**load*. The attacker prefetches the shared data to pre-set the coherence state, and then waits for the victim to possibly access this data. Later the attacker reloads the data (using a different thread on a different core, explained later) and uses the timing information to learn the current coherence state of the data, which leaks whether the victim has loaded this data (thus changing the coherence state). Different than Prefetch+Load, in Prefetch+Reload, the attacker needs to have two threads running on different cores.

Threat model. We assume a similar threat model as the one for the covert channels. First, the attacker is an unprivileged process that can 1) run on the same processor with the victim and 2) share data with the victim (e.g., through a shared library). The attacker aims at leaking the victim's access pattern on a shared data block, as in [134, 55]. In addition, the attacker can launch her thread(s) on different core(s) than the victim's.

For Prefetch+Reload, the attacker needs to have two threads running on different physical cores; but for Prefetch+Prefetch, there is still only one thread required in the attacker's process, which is the same setup as the covert channels.

3.6.2 Prefetch+Reload

In this attack, we assume that the attacker controls two threads named *Trojan* and *Spy*. Trojan and Spy should be located on different cores, which are also both different than the victim's core, i.e., Trojan, Spy, and the victim all run on different cores. As mentioned in Section 2.1, the execution times of a remote private cache hit and an LLC hit are different. The Prefetch+Reload attacker uses this timing difference to observe cache state changes caused by the victim's accesses. Specifically, before the victim accesses the target shared cache line, Trojan executes `PREFETCHW` on this cache line, which invalidates the copies of this cache line in the victim's and Spy's private caches (if they exist), and places a copy of this cache line (in M state) in Trojan's private cache, as shown in Step 1 of Figure 5. Then, if the victim accesses this cache line, according to MESI, the coherence state changes from M to S, and the copy of this cache line in the LLC is updated (although the content did not change, see Section 2.1) and is now valid (Step 2 in the left path of Figure 5).

Unfortunately, Trojan cannot observe this state change caused by the victim's access: if Trojan accesses (reloads) this cache line, she will get a private cache hit, no matter if the victim accessed this line or not. This is because the victim's read does not invalidate the copy in Trojan's private cache (Step 2 in the left path of Figure 5). However, Spy is able to distinguish whether the victim accessed this cache line. Trojan's original `PREFETCHW` invalidated the copy in Spy's private cache. Thus, if Spy now accesses this cache line, she will get an LLC hit if the victim has accessed this cache line after Trojan's prefetch (Step 3 in the left path of Figure 5); otherwise, she will get a remote private cache hit (Step 3 in the right path of Figure 5). We recall that Spy can distinguish these two situations by timing the access (the difference is over 30 cycles on our desktop processor). Based on this, we build Prefetch+Reload. Similar to previous cache attacks, each iteration in this attack contains three steps, as shown in Figure 5:

Step 1: Trojan performs `PREFETCHW` on the target cache line and becomes the exclusive owner of this cache line.

Step 2: The attacker waits for the victim's behavior: if the victim accesses this cache line, its coherence state will become S, meaning the copy in the LLC is now valid.

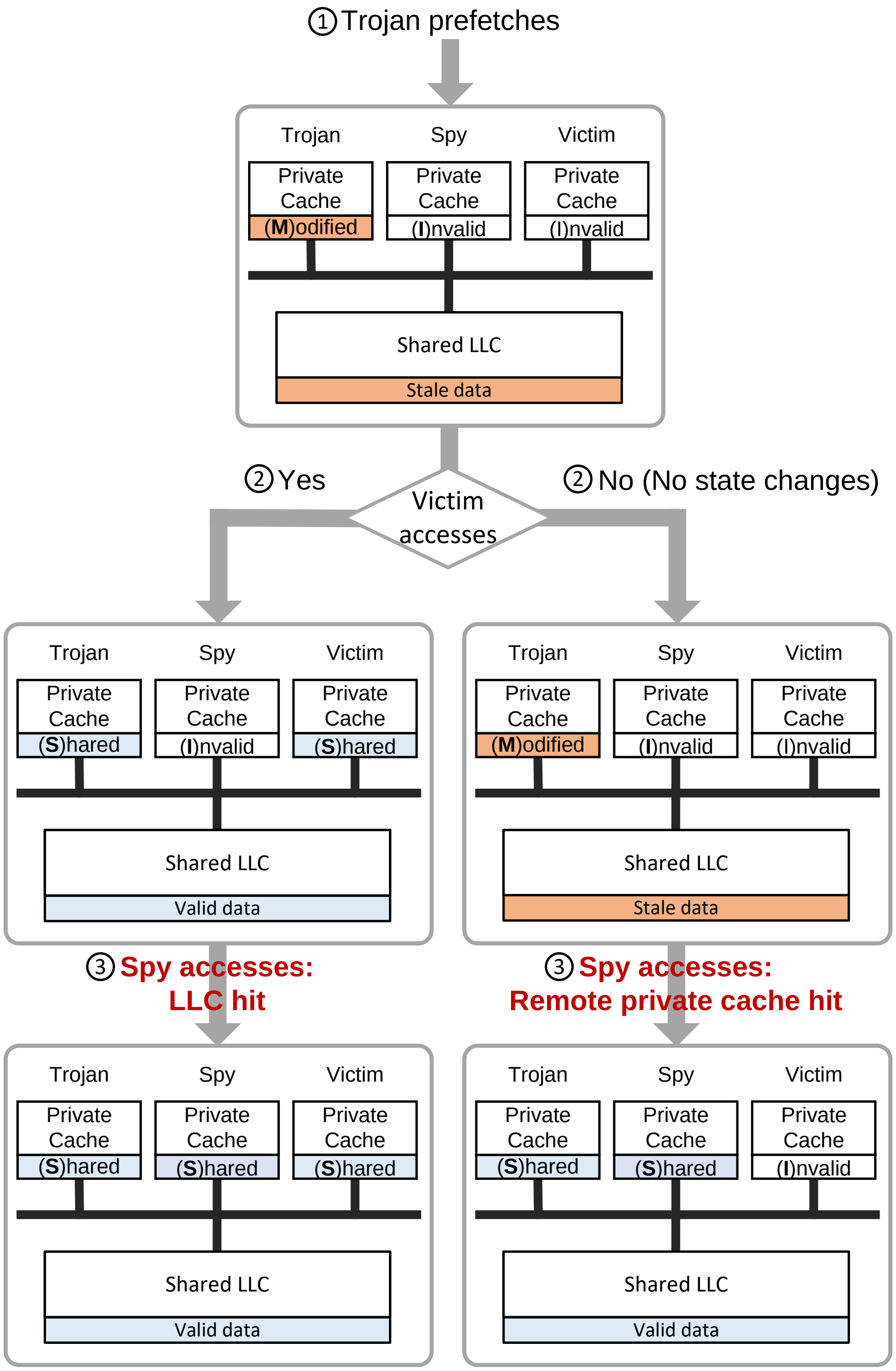


Figure 5: The details of the three steps in Prefetch+Reload.

Step 3: Spy accesses this cache line and times the access to determine it was a remote private cache hit or an LLC hit. If it was a remote private cache hit, then the victim did not access this cache line; otherwise the victim did access.

LLC presence. Prefetch+Reload requires that the target shared cache line is present in the LLC, so that Spy can get an LLC hit in Step 3, if the victim has accessed this cache line. This is naturally true for inclusive LLCs, since all the cache lines in the private cache are also present in the LLC. However, it is not guaranteed for non-inclusive LLCs. In those caches, a cache line is directly brought into the private cache when loaded from DRAM, bypassing the LLC; it usually goes to the LLC when evicted from the private cache due to cache replacement [14, 132]. Thus, strictly speaking, it is the attacker's responsibility to ensure the presence of this cache line in the LLC, if it is non-inclusive. For example, before the attacker starts the attack loop, she can first build set conflicts in her private cache to evict this cache line to the LLC.

In fact, empirically we found that in Step 1, when `PREFETCHW` invalidates the copies of the target cache line in Spy and the victim's private caches, this cache line will be placed in the LLC if it does not already exist. Therefore, in practice the attacker does not need to explicitly place this cache line in the LLC.

3.6.3 Prefetch+Prefetch

Following the Prefetch+Prefetch covert channel, we build the Prefetch+Prefetch side channel attack. The attacker learns if the victim accessed the shared cache line by timing `PREFETCHW`. In contrast to the Prefetch+Reload side channel attack, each iteration in this attack only has two steps:

Step 1: The attacker prefetches the target shared cache line using `PREFETCHW`, and times this operation to learn whether the victim accessed this cache line in the last iteration.

Step 2: The attacker waits for the victim's behavior.

As explained earlier in Section 3.4, in Step 1 above, if the victim accessed this cache line, `PREFETCHW` executes slower; if the victim did not access, `PREFETCHW` executes faster.

In contrast to most previous cross-core cache attacks, which can only work on the LLC and require repeatedly evicting the target cache line to DRAM (e.g., Flush+Reload), the proposed prefetch-based attacks work on the private cache. Thus, the target cache line is always kept in the on-chip cache hierarchy. Compared to cross-core LLC attacks, cross-core private cache attacks have two benefits. First, higher bandwidth, since cache accesses are fast and are usually much faster than DRAM accesses. This is especially important when the attacks are used as covert channels. Second, stealthier, since there are less cache misses, especially LLC misses involved in the attacks [23]. *To the best of our knowledge, the proposed prefetch-based attacks are the first cross-core private cache side channel attacks that can work regardless of the LLC inclusivity.*

3.7 Evaluation

We evaluate the proposed covert channels and side channel attacks on modern Intel processors. For covert channels, we evaluate the channel capacities, comparing them to previous cache covert channels on CPU. For side channel attacks, we demonstrate how they can be used to leak information from common applications. For side channel attacks, we demonstrate how they can be used to leak information from common applications. We also collect the cache miss rates of our attacks and compare them to SPEC 2017 [9] workloads to show the attack stealthiness. In addition, we also show how our attacks strengthen transient execution attacks.

3.7.1 Covert Channel Evaluation

We implement Prefetch+Load, Prefetch+Prefetch, and Prefetch+Reload on four Intel processors, including two desktop processors and two server processors. Note that although Prefetch+Reload is introduced as a side channel attack in Section 3.6, it can be a covert channel as well. Table 2 lists the specifications of the four tested processors. The two desktop processors have inclusive LLCs, and the server processors have non-inclusive LLCs.

We use one shared cache line between the sender and receiver to transmit secrets. Although using more shared cache lines or using channel coding techniques (e.g., [55]) may further improve the channel capacity [55, 133]; here we do not include them since we aim at showing the conservative results (i.e., the lower bounds).

Table 2: The specifications of the tested processors.

	Desktop processors		Server processors	
Platform	Core i7-6700	Core i7-7700K	Xeon Platinum 8124M	Xeon Platinum 8151
Microarchitecture	Skylake	Kaby Lake	Skylake-SP	Skylake-SP
Num of cores	4	4	N/A[6]	N/A
Frequency	3.4 GHz	4.2 GHz	3.0 GHz	3.4 GHz
LLC type	Inclusive	Inclusive	Non-inclusive	Non-inclusive

We measure the channel capacity and bit error rate of each attack, under different transmission intervals. Although the raw transmission rate increases when decreasing the transmission interval, the bit error rate may also increase, especially when the interval is too short. To find the best transmission rate, we use the channel capacity metric (as in [92, 90]). This metric is computed by multiplying the raw transmission rate with $1 - H(e)$, where e is the bit error rate and H is the binary entropy function. The results are shown in Figure 6. The bit error rates of all three attacks stay low (lower than 0.6%) and are almost constant, when the raw transmission rate is under a threshold (e.g., 660 KB/s for Prefetch+Reload in Figure 6(a)). Thus, the channel capacity increases proportionally to the raw transmission rate. It reaches the peak when the raw transmission rate is around this threshold. Beyond this threshold, the increasing error rate causes a decrease in the channel capacity. The peak channel capacities of the three attacks are summarized in Table 3. Prefetch+Reload always

[6] We use Intel Xeon Scalable processors on Amazon AWS EC2 platform, and we leased four physical cores on the tested processors for our experiments.

has lower capacity than the other two attacks because more cache operations are involved in each iteration of Prefetch+Reload.

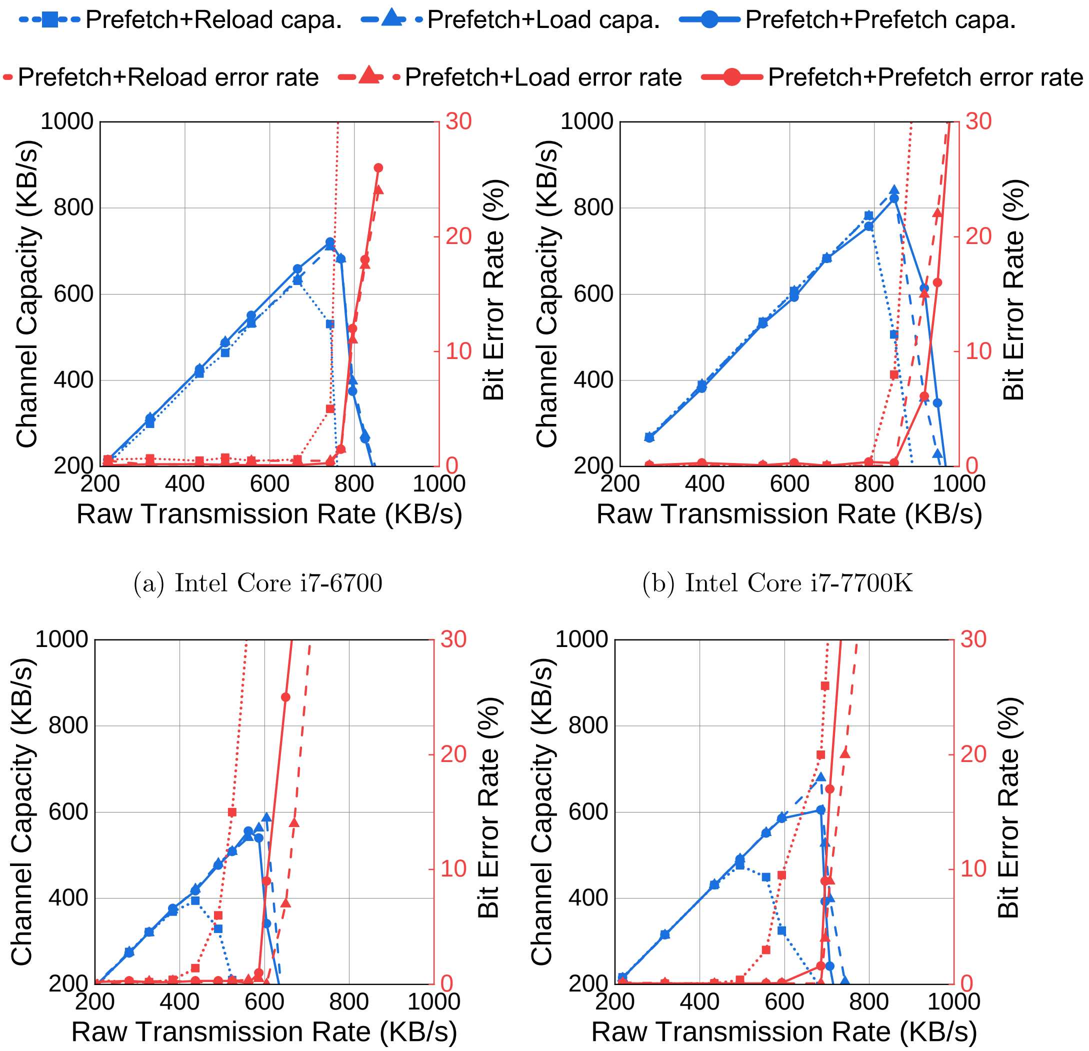
Our prefetch-based attacks are faster than almost all existing cache attacks on x86 CPUs. First, for attacks tested on desktop processors, the ring interconnect contention-based attack [90] is reported with a very high capacity which is 518 KB/s on a 4.0 GHz desktop processor. Flush+Reload and Flush+Flush have capacities of 298 KB/s and 496 KB/s on a 3.6 GHz desktop processor [55], respectively. Prime+Scope [93], the optimized attack for Prime+Probe, achieves 438 KB/s on a 3.5 GHz desktop processor. Second, most of the attacks that were tested on server processors, including the L1 LRU attack [130], the directory Prime+Probe attack [132], and the Flush+Coherence attack [133] have capacities of less than 200 KB/s. The directory version of Prime+Scope achieves 387 KB/s.

To the best of our knowledge, our attacks are only slower than Streamline [100]. This attack claims to achieve a capacity of 1801 KB/s. However, it has such a high channel capacity because the sender and receiver use 64 MB shared data to transmit secrets; our results are based on one shared cache line (64 B).

3.7.2 Side Channel Evaluation

3.7.2.1 Side Channel Attack on Cryptographic Code

Our first attack targets cryptographic libraries, where the access patterns to some instructions are related to the value of the cryptographic key. More specifically, we target the square-and-multiply algorithm [53] which is used in GnuPG 1.4.13 for ciphers such as RSA [98] and ElGamal [45]: leaking the exponent e of this algorithm leaks the private key. As shown in Algorithm 3, in each loop iteration, it first executes a `sqr` and a `mod` instruction. Then, if the exponent bit is “1”, a `mul` and another `mod` instruction are executed; otherwise they are skipped. Thus, by monitoring the access pattern to the cache lines that contain `sqr` and `mul`, the attacker can leak each bit of the exponent e and therefore the decryption key.
Implementation. As done in the Flush+Reload attack on GnuPG [134], we use `mmap` to map the pages that contain `sqr` and `mul` into the attacker's address space. Note that during the execution of the victim (GnuPG), the cache lines containing those instructions



(a) Intel Core i7-6700

(b) Intel Core i7-7700K

(c) Intel Xeon Platinum 8124M

(d) Intel Xeon Platinum 8151

Figure 6: The capacities and bit-error-rates of the prefetch-based channels on various Intel processors.

are brought into the victim's L1 instruction cache (L1I cache). However, since we map the instruction pages as data blocks in the attacker's address space, the same cache lines containing those instructions are brought to the attacker's L1D cache. Thus, although `PREFETCHW` can only prefetch cache lines into L1D cache, it can still leak the victim's access patterns to instructions.

Table 3: The maximum capacities of the prefetch-based channels.

Platform	Desktop processors		Server processors	
	Core i7-6700 (3.4 GHz)	Core i7-7700K (4.2 GHz)	Xeon Platinum 8124M (3.0 GHz)	Xeon Platinum 8151 (3.4 GHz)
Prefetch+Reload	631 KB/s	**782 KB/s**	394 KB/s	476 KB/s
Prefetch+Load	709 KB/s	**840 KB/s**	586 KB/s	680 KB/s
Prefetch+Prefetch	721 KB/s	**822 KB/s**	556 KB/s	605 KB/s

Algorithm 3: Square-and-multiply Exponentiation

```
Input: base $b$, modulo $m$, exponent $e = (e_{n1}...e_0)_2$
Output: $b^e \mod m$
$r \leftarrow 1$
for $i = n - 1; i >= 0; i - -$ do
    $r \leftarrow r^2 \mod n$
    if $e_i == 1$ then
        $r \leftarrow r * b \mod n$
```

Results. For simplicity, we only show the attack results of Prefetch+Prefetch on the Intel Xeon Platinum 8151 processor. However, we have performed this attack on other processors listed in Table 2 too, using both Prefetch+Prefetch and Prefetch+Reload. Here we use a waiting latency of 500 cycles in each iteration of Prefetch+Prefetch. Figure 7 shows the timing measurement results from the attacker for 200 samples: a lower prefetch latency (less than 100 cycles) indicates that the victim did not access the target cache line during the last iteration; a higher prefetch latency (around 200 cycles) means the victim did access. As explained above, an access to `sqr` followed by an access to `mul` indicates a bit "1", and

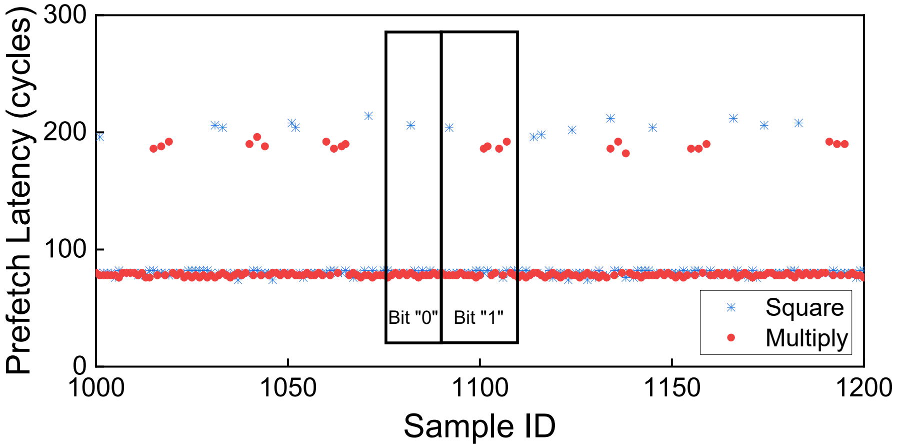


Figure 7: A segment of the prefetch latencies measured in Prefetch+Prefetch while attacking GnuPG; part of the the exponent e shown here is “111001011001”.

two consecutive accesses to `sqr` (one from the current iteration, one from the next iteration) indicate a bit “0” (in the current iteration). Thus, part of the exponent e shown in Figure 7 is “111001011001”. The average attack accuracy is 96.2%.

3.7.2.2 Side Channel Attack on Keystroke Timing

Our second attack focuses on leaking the precise timing information of keystrokes, i.e., detecting when a keyboard input occurs. This leakage is important since it can assist reconstructing typed words from users [137, 110, 75]. Previous work shows that certain functions in graphics libraries are called when a keystroke happens (e.g., [123, 55]). Thus, we can monitor the accesses to the cache lines containing these functions to detect keystrokes.

Implementation. We attack an address in the shared GDK library which is invoked when processing keystrokes. Specifically, we launch gedit as the victim, and input keystrokes in it. At the same time, we run the prefetch-based attacks to monitor accesses to the address selected in the GDK library, and record the timing measurement results. The attacker process raises an alarm when a keystroke is detected.

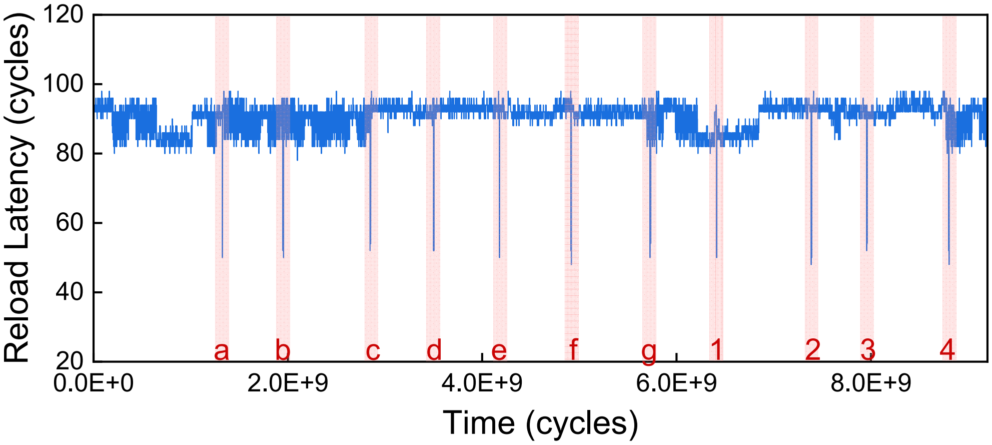


Figure 8: The access latencies measured in Step 3 of Prefetch+Reload when a user types "abcdefg1234" in gedit; we monitor address 0x7b980 of libgdk.so.[7]

Results. Figure 8 shows the timing trace collected by Prefetch+Reload when the user is typing "abcdefg1234" in gedit, on our Intel Core i7-6700 processor. Again, the attack has been done on the other desktop processor too (but not on the server processors since EC2 instances do not come with GUI). As one can observe, when a keystroke occurs, the reload operation (in Step 3 of Prefetch+Reload) takes around 50 cycles to finish; it takes over 80 cycles to reload when there is no keystroke. This significant timing difference makes keystrokes very detectable. During the attack, we observe zero false positives and zero false negatives.

3.7.2.3 Attack Stealthiness

Since most previous cross-core attacks have a large amount of cache misses, especially LLC misses, prior work (e.g., [23]) has suggested detecting the attacker process by monitoring the cache miss rates of each process. In this section, we show that our prefetch-based attacks cannot be detected in this way. We use performance counters to collect the attacker's miss rates to L1D cache, L2 cache and LLC, while attacking GnuPG and user inputs

[7] We found the appropriate library and address to monitor following the method in prior work [56].

(keystrokes), respectively. The results are in Figure 9. As one can observe, the LLC miss rates of Prefetch+Reload are less than 1% in both attack scenarios. For Prefetch+Prefetch, the LLC miss rates are a little higher (about 10%) because there are fewer LLC accesses.

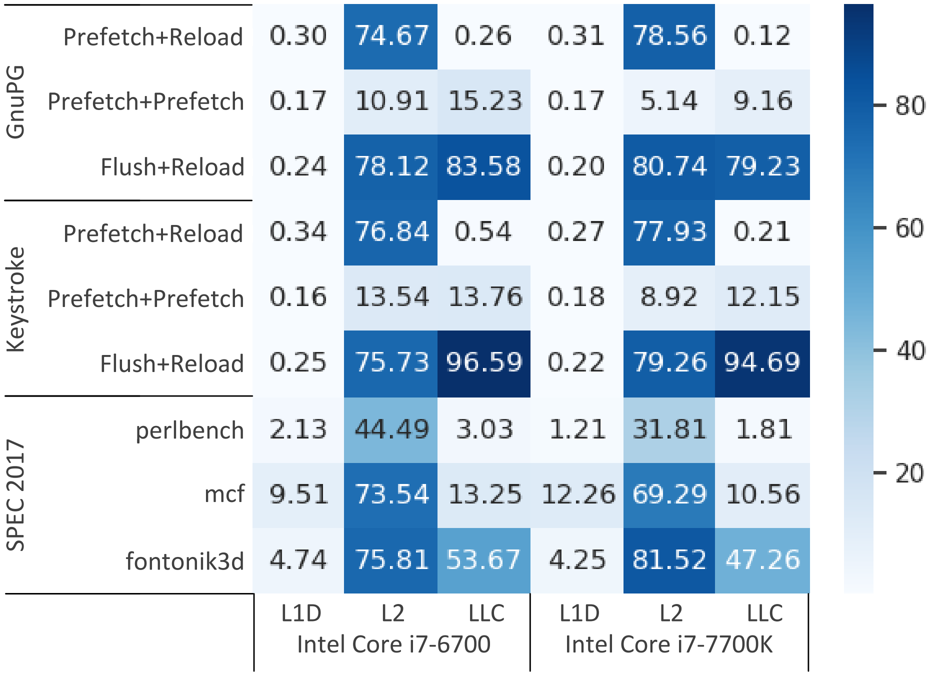


Figure 9: The cache miss rates of 1) the attacker processes in various cache attacks and 2) three workloads in SPEC 2017.

To compare, in Figure 9 we also list the attacker's cache miss rates of Flush+Reload as well as the cache miss rates of three typical SPEC 2017 workloads that have high, medium, and low memory access locality, respectively. From this figure, the LLC miss rates of our prefetch-based attacks are lower than or similar to the ones of SPEC workloads. In contrast, in Flush+Reload, the attacker has very high LLC miss rates: they are about 70% to 80% when attacking GnuPG, and are over 90% when attacking user inputs. These are much higher than the LLC miss rates of all the three SPEC workloads. In fact, from our experiments, there are only two workloads (out of 23) in SPEC that have an LLC miss rate higher than 30%.

Although in the prefetch-based attacks, the target cache line is always evicted from L1D cache, the miss rates to L1D cache are still less than 1% because other data accesses (to the cache lines which are not used for leaking secrets) cause a lot of L1 hits. This means we cannot detect the attacks by monitoring the L1 miss rate. In addition, Prefetch+Reload has high miss rates to L2 cache (similar to Flush+Reload). However, as shown in Figure 9, SPEC 2017 workloads can also have high miss rates to L2 cache, making it hard to detect those attacks through L2 miss rates.

Our attacks may be detected using performance counters related to software prefetch. For example, `ls_pref_instr_disp.store_prefetch_w` counts the total amount of dispatched `PREFETCHW` instructions; `ls_inef_sw_pref.data_pipe_sw_pf_dc_hit` counts the amount of prefetch instructions that did not fetch data outside of the private cache. These two performance counters together may identify a process that repeatedly uses `PREFETCHW` on data that is not in the local private cache. However, such a process is not necessarily an attacker, because a program using `PREFETCHW` to accelerate a loop could also have this behavior: in each iteration, the data required for the next iteration is prefetched from the LLC. Thus, using these two performance counters may result in false positives. Other finer-grained performance counters such as the ones related to read for ownership (RFO) requests may give a better attacker signature. We leave finding effective combinations of performance counters as future work.

3.7.2.4 Windowless Prefetch+Prefetch

Using the terminology in prior work [93], `PREFETCHW` has two important properties: 1) `PREFETCHW` is preserving, meaning the measurement (prefetching and timing the prefetch) does not change the state in the absence of the victim's event; 2) `PREFETCHW` is also concurrent, meaning it detects the events that temporally overlap with it. With these two features, Prefetch+Prefetch can be used in a windowless way (no waiting window between two consecutive prefetches is necessary). We verify this using the following experiment.

We use two processes, the victim and attacker. The victim process first waits a random amount of time, and then triggers an event (accessing the target shared cache line). This

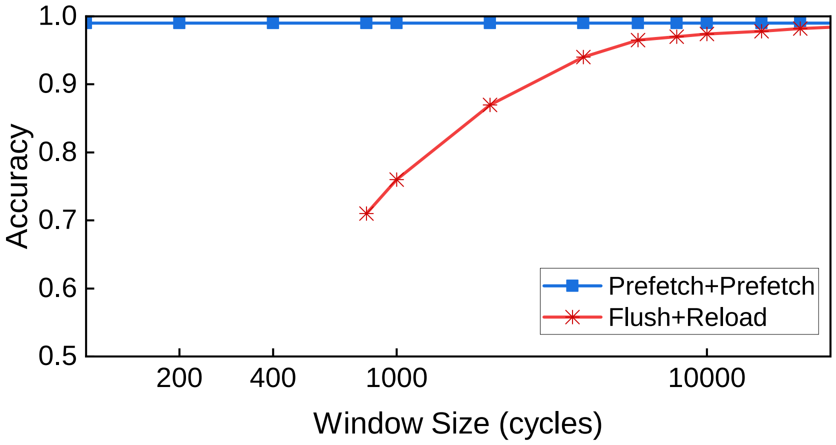


Figure 10: The accuracy of Prefetch+Prefetch and Flush+Reload on our Intel Core i7-6700 processor, with different waiting window sizes.

process terminates after triggering the event. The attacker process runs Prefetch+Prefetch with a waiting window in each attack iteration to detect the victim's event. The attacker process terminates after detecting the event or after the victim process terminates. We run this experiment with different window sizes and repeat the experiment for 1000 times for each window size. Figure 10 shows the attacker's detection accuracy on our Intel Core i7-6700 processor. Note that the results on other processors in Table 2 are similar. For comparison, we also show the accuracy of Flush+Reload on the same processor. For Prefetch+Prefetch, the attacker's detection accuracy does not change when the window size varies; the attacker always has a very high detection accuracy which is around 1. This indicates that Prefetch+Prefetch, unlike prior attacks such as Flush+Reload, can always be used as a windowless attack. Such a windowless attack has much higher temporal resolution than a windowed attack since the latter's resolution is bounded by the window size. For example, to reach 95% detection accuracy, Flush+Reload needs a waiting window with over 4000 cycles.

3.7.3 Prefetch-Based Channels in Transient Execution Attacks

Transient execution attacks such as Spectre [74] and Meltdown [78] usually require a microarchitectural covert channel to transfer the secrets to the attacker. Currently, most transient execution attacks (e.g., [74, 78, 118, 30, 102]) use the Flush+Reload channel because it is simple, reliable, and ubiquitous. Here we demonstrate that prefetch-based channels can also work with transient execution attacks to leak secrets, and may even work better than Flush+Reload. We use Spectre v1 as an example to show the details and benefits of using prefetch-based channels in transient execution attacks.

Higher bandwidth. When using Flush+Reload, the sender operation in Spectre is a memory access where the address depends on the secret value. Since Prefetch+Reload and Prefetch+Prefetch use the same sender function as Flush+Reload, a victim program vulnerable to Spectre with Flush+Reload is also vulnerable to Spectre with Prefetch+Reload and Prefetch+Prefetch. We have verified this using the Spectre v1 PoC code [10]. We modify it accordingly such that Prefetch+Reload or Prefetch+Prefetch is used in the attacker code; the victim remains the same. In addition, as observed in prior work [78], the leakage rate of a transient execution attack is significantly affected by the capacity of the covert channel used in the attack. Since Prefetch+Reload and Prefetch+Prefetch have much higher capacities than Flush+Reload, Spectre works faster with these two channels. For example, on our Intel Core i7-6700 processor, when leaking an 8-bit secret in each transmission, the leakage rate of Spectre is 3.02 times and 1.61 times as fast when using Prefetch+Prefetch and Prefetch+Reload, respectively, as compared to Flush+Reload.

More leakage in the transient window. When using Spectre with Flush+Reload, the data access for sending (encoding) the secret in the transient window is a *slow DRAM access*, since this data was flushed by the attacker. In contrast, the data access for secret encoding is a *remote private cache hit* when using Prefetch+Reload or Prefetch+Prefetch, which is usually faster than a DRAM access. This indicates that within the same transient window, more encoding operations can be performed using the two prefetch-based channels than Flush+Reload, and thus more secrets may be leaked. An example Spectre v1 gadget that can benefit from this is shown in Listing 3.3. There are n operations in the branch, where each operation accesses a secret and encodes it to a cache index. The secrets are array1[x] to array1[x+n] (when x is out of bounds); each of the secrets is encoded to an index of a sub-array in array2. The more of these n operations we can perform in the transient window, the more secrets we can leak out at once.

```
if (x+n < array1_size)
{
    y0 = array2[0][array1[x] * 4096];
    y2 = array2[1][array1[x+1] * 4096];
    ...
    yn = array2[2][array1[x+n] * 4096];
}
```

Listing 3.3: The Spectre v1 code example when a bounds check is followed by multiple secret accessing and encoding operations. This code is essentially a for loop in a conditional branch; we show the unrolled version for clarity.

This gadget might be found in a victim; it is essentially the original Spectre v1 gadget with multiple secrets accessed and encoded in the branch (instead of one). Additionally, in the scenario where the attacker has control over the gadget (e.g., spectre-type-meltdown),[8] the attacker can build such a gadget to leak multiple secrets in one transient window and thus accelerate the attack. We still prove this with the Spectre v1 PoC code and modify the attacker code to use Prefetch+Reload or Prefetch+Prefetch. We also modify the victim code to simulate the gadget in Listing 3.3 where n secrets are accessed and encoded in the branch. We run this code and collect the amount of these n secrets the victim can transmit within one transient window, and draw the histograms in Figure 11. We omit the results when leaking by Prefetch+Reload since its encoding stage is same as the one of Prefetch+Prefetch.

On the desktop processors, the victim can transmit up to 17 8-bit secrets speculatively when using Prefetch+Reload or Prefetch+Prefetch, while the victim can transmit at most 8 secrets when using Flush+Reload. However, on server processors, the amount of transmitted

[8] Spectre can be used for exception suppression in Meltdown.

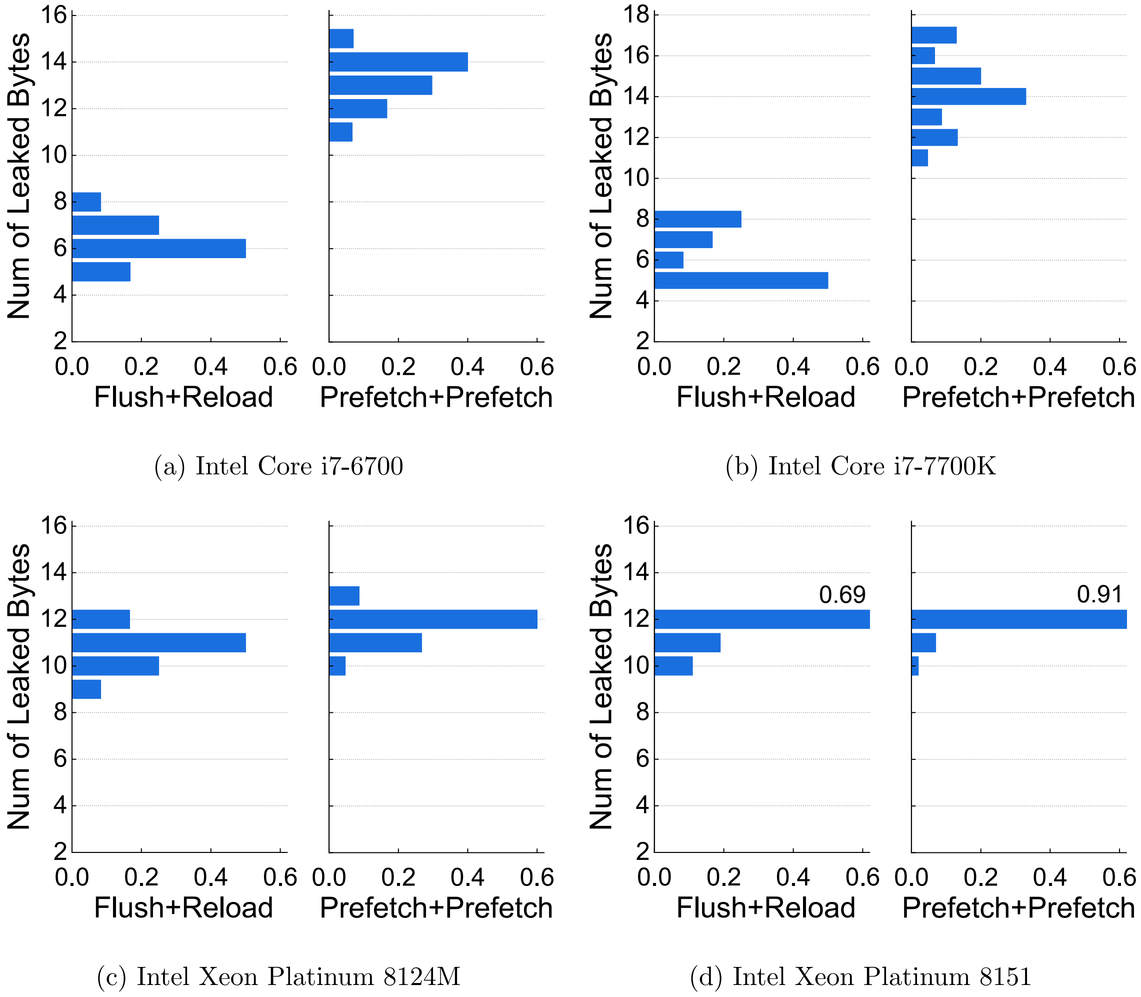
secrets when using prefetch-based channels is only slightly larger than the one when using Flush+Reload. This is because on these processors, the latency of a remote private cache hit is much longer, compared to desktop processors (160 cycles vs. 90 cycles). Note that although same-core private cache attacks, such as the L1 LRU attack [130], can also achieve more secret encodings in a transient window than Flush+Reload, these attacks are less practical, because they are limited by the number of private cache sets. In these attacks, secret values are encoded into the cache set index instead of cache line index.

Other transient execution attacks. All of the three prefetch-based channels can be used in transient execution attacks when the attacker has full control of the gadget (e.g., Meltdown). As shown above, Prefetch+Reload and Prefetch+Prefetch has faster encoding operations than Flush+Reload, enabling more leakage in a transient window. The same is true for Prefetch+Load, since a remote private cache hit for `PREFETCHW` is usually faster than a DRAM access. In a Meltdown PoC with the three prefetch-based channels, we can reliably leak 8 bytes in the transient window on Our Intel Core i7-6700 processor; we can only leak 6.1 bytes on average when using Flush+Reload.

3.8 Discussion

3.8.1 Attack Reliability

According to Intel [3], a prefetch instruction will not fetch any data when the request buffer between the L1 and L2 cache is full. This may reduce the performance of the prefetch-based attacks, when SMT is available and a memory-intensive thread is located on the same core as the attacker thread. We verified this by running `stress -m 1` in a co-located thread (i.e., the hyperthread sibling) of the attacker thread: this causes many prefetch instructions from the attacker to be ignored, which significantly reduces the attack performance. For example, on our Intel Core i7-6700 processor, the channel capacity of Prefetch+Prefetch is reduced to 56 KB/s. However, SMT enables many security vulnerabilities (e.g., [116]) and thus is often suggested to be disabled. In fact, if SMT is available, the attacker can always



(a) Intel Core i7-6700

(b) Intel Core i7-7700K

(c) Intel Xeon Platinum 8124M

(d) Intel Xeon Platinum 8151

Figure 11: The distributions of the amount of secret bytes that can be accessed and encoded in a transient window, when leaking by Flush+Reload and Prefetch+Prefetch, respectively.

launch same-core private cache attacks instead. Our cross-core private cache attacks target the scenarios where same-core attacks are impractical or impossible.

3.8.2 `PREFETCHW` on AMD processors

Modern AMD processors also support `PREFETCHW`; this instruction was originally invented by AMD [12], and was later adopted by Intel. We performed the same experiments as the ones in Section 3.4 on AMD desktop and server processors. However, from our experiments, PREFETCHW does not cause any coherence state changes on data with read-only permission; it only works on data with write permission. Thus, we believe that AMD processors actually have permission checks for PREFETCHW.

3.9 Chapter Summary

In this chapter, we proposed the first two cross-core private cache side channel attacks that work with both inclusive and non-inclusive LLCs. One of the prefetch instructions on x86 processors, PREFETCHW, prepares the data for future writes by modifying the coherence state of the data. We made two important microarchitectural observations on PREFETCHW. First, it works on data with read-only permission. Second, its execution time is related to the coherence state of the target data. Given these observations, the coherence state modifications by PREFETCHW enable significant security vulnerabilities. Using PREFETCHW, we first built two covert channels that have very high capacities. We also demonstrated that these high-capacity covert channels enable more powerful transient execution attacks. We then slightly modified the covert channels to build two side channel attacks and showed that these attacks leak information from real-world applications.

4.0 New Conflict-Based Cache Attacks with The `PREFETCHNTA` Instruction

4.1 Overview

Conflict-based cache attacks are an important class of cache attacks where the attacker (receiver) deliberately causes cache conflicts to learn the victim's (sender's) access pattern. For example, in Prime+Probe [81, 66], the attacker first primes a cache set by filling it with the attacker's cache lines, and then waits for the victim's execution: if the victim accesses her own cache line that is mapped to this set, one of the attacker's lines is evicted. Later the attacker probes the cache set and times the probing to learn whether the victim accessed her cache line. Conflict-based attacks are very practical since they usually do not assume any system features (such as page deduplication). However, when used as covert channels, their bandwidths are limited. This is because priming a cache set requires many accesses and takes very long to finish. For example, to build Prime+Probe on a 16-way associative LLC, priming the set needs about 16 cache accesses. In this dissertation, we seek to answer the following question:

Is it possible to cause cache conflicts without encountering the effort in priming the cache set?

x86 processors feature many prefetch instructions. Developers or compilers can use these instructions to inform the processor that a memory location will be accessed or modified soon. Then, the processor preloads the data (usually in cache line level) and places it closer to the CPU core, in order to accelerate future requests. `PREFETCHNTA` is one of the x86 prefetch instructions. When executing `PREFETCHNTA` on Intel processors with an inclusive LLC, the target data is brought into the requesting core's local L1 cache as well as the LLC [3, 4]. However, to avoid LLC pollution, the prefetched data is not placed into the most recently used position in the LLC set, and will be chosen for LLC replacement sooner than a regular cache fill. In this dissertation, we reverse-engineer the detailed cache behavior of `PREFETCHNTA` on Intel processors and make three important observations. First, PREFETCHNTA installs the target cache line into the LLC set as the eviction candidate. Second, when PREFETCHNTA hits in the LLC, it does not update the age of this cache line in the LLC. Third, the execution time of PREFETCHNTA is related to the location of the target cache line in the memory hierarchy.

When filling a cache line into the LLC using PREFETCHNTA, it replaces the current eviction candidate in the set and then the prefetched line becomes the new eviction candidate. This means, when two processes both prefetch their own cache lines into the same LLC set, they will compete for the eviction candidate position (cache way), causing conflicts in one way of the LLC set. Based on this, we propose a new conflict-based cache covert channel, named NTP+NTP (Non-Temporal Prefetch). In this channel, the sender and receiver first agree on the LLC set for transmitting secrets. Then in each iteration of the transmission, the sender sends one bit by prefetching her cache line into the target LLC set (for "1") or not prefetching (for "0"). The receiver receives the bit by prefetching the receiver's cache line (which is also mapped into this target LLC set), and times the prefetch to determine if it is an LLC miss (for "1") or not (for "0"). If the sender prefetches her cache line into the LLC, it evicts the receiver's cache line that was prefetched into the same set; later the receiver's prefetch will miss in the LLC. We show that NTP+NTP has very high capacity as a conflict-based covert channel: on our Skylake processor, the capacity of NTP+NTP is 302 KB/s which is over 3× than the capacity of Prime+Probe. *To the best of our knowledge, NTP+NTP is the first conflict-based LLC covert channel that does not require priming the cache set.*

Although NTP+NTP is unlikely a side channel, we found that PREFETCHNTA can be used in many cache side channel attacks that are based on replacement state changes to make the attacks more efficient. This is because PREFETCHNTA makes it easier for users to manipulate cache replacement states. For example, Prime+Scope [93] is a cache attack proposed very recently. Prime+Scope achieves the highest-to-date temporal resolution for cache attacks, and is thus very powerful. However, this attack has strict requirements on the replacement state of the target LLC set. To satisfy the requirements, it uses a very long access sequence to prime the LLC set. On our Skylake processor, the priming comprises 192 cache references and takes about 1900 cycles to finish. This long priming step limits the attack from detecting frequent victim events. In contrast, when using PREFETCHNTA, the priming only needs 33 cache references and takes about 1000 cycles to finish, resulting in a much faster attack. In addition, using `PREFETCHNTA` makes cache conflicts occur more often and thus makes eviction set construction faster. In this dissertation, we propose a new eviction set construction algorithm which significantly outperforms the state-of-the-art.

In this chapter, we refer to "prefetch using `PREFETCHNTA`" as "prefetch".

4.2 Cache Replacement Policy

When the CPU core loads a cache line that is not present in a cache level (i.e., cache miss), the cache line is usually filled to this cache level (into a certain set). If the set this cache line is mapped into is already full, one of the lines that are currently cached in this set will be evicted to make space for this new cache line. The cache replacement policy decides which line should be evicted, i.e., the *eviction candidate.*

LRU. LRU is one of the most widely used replacement policies as it provides high cache utilization and thus good performance. LRU always selects the least recently used cache line in a set as the eviction candidate. Thus, when using LRU, we need to track the age of each cache line in a set. For a w-way associative cache, $\log w$ bits are necessary to record the age of each way (cache line) in a set, for a total of $w \log w$ for each set. This makes tracking and updating the ages of cache lines very expensive in terms of storage and latency.

Pseudo LRU. Recent x86 CPUs use pseudo LRU algorithms to achieve high cache hit rate as well as maintain low age updating/tracking overhead. Typical Pseudo LRU algorithms include Tree-LRU [109] and Bit-LRU [82]. Prior work [28] has reverse engineered that recent Intel Core processors use *Quad-age LRU* for their LLCs. With this policy, each cache line in an LLC set is assigned with two bits to represent its age. Thus, the maximum (oldest) age for a cache line is 3, and the minimum (youngest) age is 0. The details of this policy are shown as follows:

- **Insertion policy.** When a cache line is filled into the LLC, its age is initialized as 2.[1]
- **Replacement policy.** When replacement is necessary, Quad-age LRU searches all the ways in the target LLC set in order, and evicts the cache line that is stored in the first way with age 3; if such a way does not exist, it increases the age of every way by 1 and searches again.
- **Updating policy.** When an access request from the CPU core hits in the LLC, the age of the target cache line is reduced by 1 (if the age is 0 then it will not be changed).

Figure 12 shows an example of how the state of an LLC set changes with a sequence of CPU requests. In this figure, the replacement policy checks the cache lines in the set from the left to right, when looking for the eviction candidate.

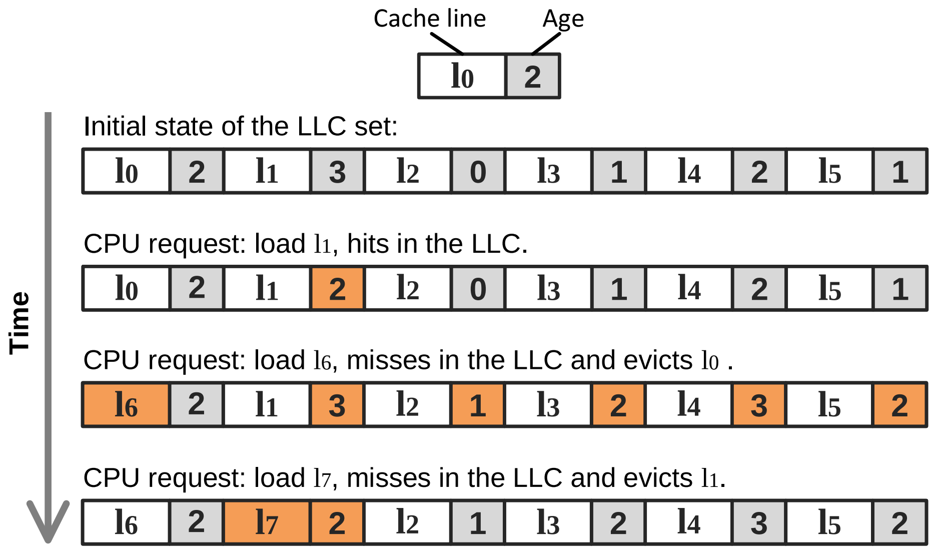

Cache line
Age
l0 2
Time
Initial state of the LLC set:
l0 2 l1 3 l2 0 l3 1 l4 2 l5 1
CPU request: load l1, hits in the LLC.
l0 2 l1 2 l2 0 l3 1 l4 2 l5 1
CPU request: load l6, misses in the LLC and evicts l0 .
l6 2 l1 3 l2 1 l3 2 l4 3 l5 2
CPU request: load l7, misses in the LLC and evicts l1.
l6 2 l7 2 l2 1 l3 2 l4 3 l5 2

Figure 12: The state change details of an LLC set upon CPU requests; changes after each request are highlighted.

[1] On early Intel processors (before Skylake), sometimes cache lines are inserted into the LLC with the age initialized as 3.

4.3 Characterizing The Non-Temporal Prefetch Instruction

4.3.1 Non-Temporal Prefetch

Among the prefetch instructions discussed in Section 3.3, PREFETCHNTA works slightly different than the others: it minimizes the LLC pollution when fetching data into the cache hierarchy. To accelerate future accesses from the requesting core, PREFETCHNTA places the target cache line into the requesting core's private cache; with an inclusive LLC, this cache line has to be also brought into the LLC (if not already present). However, prefetching a cache line into the LLC may replace cache lines from other threads and degrade their performance. According to Intel [3], using PREFETCHNTA can reduce this disturbance to other data cached in the LLC: a cache line prefetched with this instruction will not be placed into the most recently used position (in the LLC set) and may be chosen for replacement faster than a regular LLC fill. Thus, when the target cache line is only accessed once in the entire execution path, the user should prefetch it using PREFETCHNTA. We reverse engineer the detailed cache behavior of PREFETCHNTA in this section, and will explain why this instruction raises severe security concerns in the next section.

Experiment platform. The experiments in this chapter are all performed on two Intel processors, Core i7-6700 and Core i7-7700K. The processor parameters are listed in Table 4. In this section we only show the results on the Core i7-6700 processor due to limited space. Note that in this section we disable the hardware prefetcher to get accurate reverse engineering results. In the following sections, we enable the hardware prefetcher when evaluating the attacks for generality, and avoid triggering the hardware prefetcher using the techniques in prior work [55, 74].

4.3.2 Key Properties

4.3.2.1 Insertion Policy

We first verify that a prefetched cache line is evicted faster/sooner than a loaded cache line in the same LLC set, using a four-step experiment. To prepare the experiment, we

Table 4: The specifications of the tested processors.

Platform	Core i7-6700	Core i7-7700K
Microarchitecture	Skylake	Kaby Lake
Num of cores	4	4
Frequency	3.4 GHz	4.2 GHz
L1 associativity	8	8
L1 type	Private	Private
L2 associativity	4	4
L2 type	Private, non-inclusive	Private, non-inclusive
LLC associativity	16	16
LLC type	Shared, inclusive	Shared, inclusive

construct an eviction set, i.e., a group of cache lines that are all mapped to one specific set (the target set) in the LLC. This eviction set consists of $w+1$ cache lines where w is the set associativity of the LLC (16 for our processors). These cache lines are named l_0, l_1,..., l_w. As shown in Figure 13, this experiment consists of the following steps:

Step 1: We make the target LLC set empty. This can be achieved by loading the cache lines in the eviction set into the LLC and then flushing all of them with `CLFLUSH`.

Step 2: We pick l_a from the eviction set, where $0 \leq a \leq w-1$. Then, we first load the cache lines before l_a in the eviction set into the LLC (l_0 to l_{a-1}, if $a \neq 0$), and then prefetch l_a. After the prefetch, we load the rest of the cache lines until the LLC set is full (l_{a+1} to l_{w-1}, if $a \neq w-1$). We add `LFENCE` after each load/prefetch operation to ensure that the cache lines are filled into the LLC in order.

Step 3: We load l_w into the LLC which evicts one of the existing cache lines in the target set.

Step 4: We load l_a and time the load to learn whether this prefetched line was evicted in Step 3. If l_a was evicted, it takes longer (typically more than 150 cycles) to load, otherwise it takes much shorter to load (less than 100 cycles).

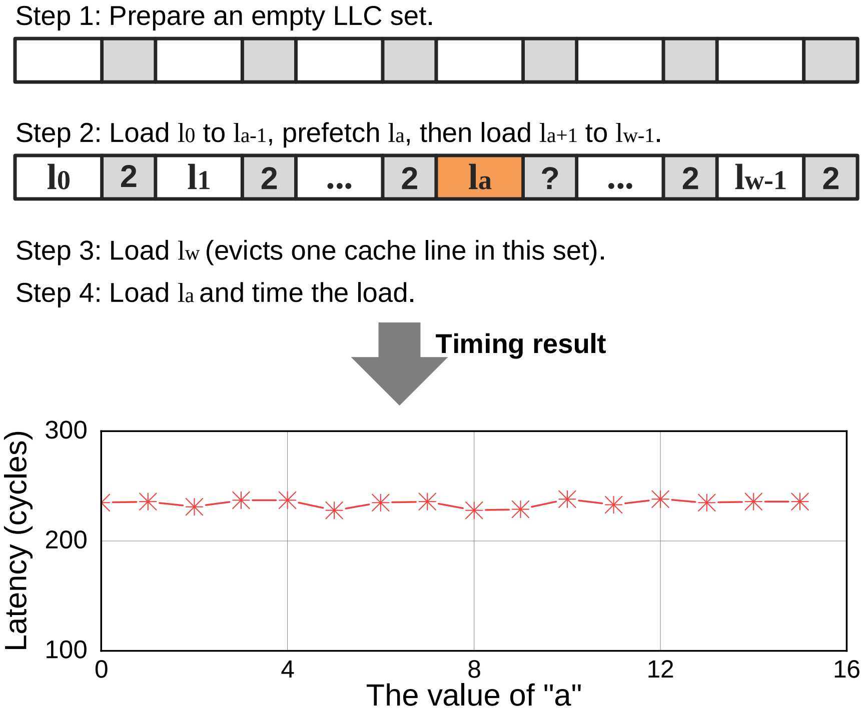
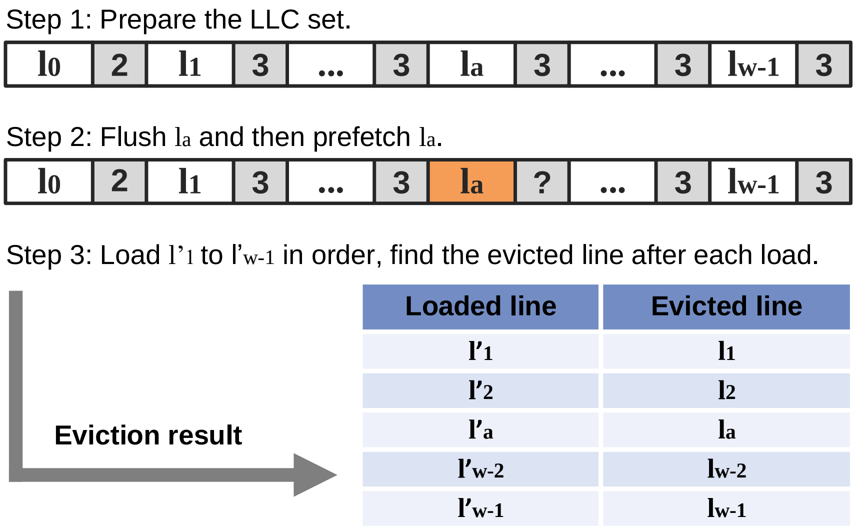


Figure 13: The experiment steps and results for verifying that prefetched data are evicted earlier than other data.

We run the above experiment with a changing from 0 to $w-1$ and repeat the experiment 10000 times for each value of a. The average load latencies in Step 4 are shown in Figure 13: it always takes over 200 cycles to reload the prefetched line (l_a), meaning l_a was always evicted from the LLC in Step 3, regardless of its position in the set. This proves that a prefetched cache line is easier to be evicted than cache lines loaded into the LLC.

The above experiment indicates that a prefetched cache line is *distinctively inserted into the LLC*, so that the replacement policy will choose it to be evicted sooner than other cache lines. We hypothesize two possible hardware-level implementations to achieve this: 1) a prefetched cache line is inserted into the LLC set with the age initialized to be 3 instead of 2 (cf. Section 4.2); 2) a prefetched cache line is inserted into the LLC set with age 2 as



Step 1: Prepare the LLC set.

Step 2: Flush la and then prefetch la.

Step 3: Load l'1 to l'w-1 in order, find the evicted line after each load.

Loaded line	Evicted line
l'1	l1
l'2	l2
l'a	la
l'w-2	lw-2
l'w-1	lw-1

Figure 14: The experiment steps and results for learning the insertion policy of `PREFETCHNTA`.

normal, but this line is flagged for "early eviction" in the LLC, i.e., a prefetched line and a line with age 3 are treated unequally by the replacement policy. As shown in Figure 14, we then perform the following experiment to know which option has more likely been chosen by Intel. In this experiment, we use two eviction sets that are mapped to the same LLC set (l_0 to l_w and l'_0 to l'_w).

Step 1: We prepare the target LLC set as shown in Step 1 of Figure 14. This can be achieved by first filling the set with l_w and l_1, l_2,..., l_{w-1} in that order, and then loading l_0 to evict l_w (cf. Section 4.2).

Step 2: We flush l_a ($1 \leq a \leq w-1$) and then prefetch l_a. It is brought back to this flushed location.

Step 3: We load l'_1 to l'_{w-1} into the LLC in order and check which cache line in this set is evicted after loading each of them.[2]

[2] Checking if a cache line is evicted can be done by loading it and timing the load. Note that we should start over the experiment before checking the next cache line to avoid the noise caused by the measurement.

With each possible value of a, we run the experiment 10000 times. The eviction results in Step 3 are shown in Figure 14. We get the same results in each trial regardless of the value of a: when loading l'_1 to l'_{w-1} into this LLC set, l_1 to l_{w-1} are evicted in order (from the left to right in Figure 14). This indicates that the prefetched cache line (l_a) is treated equally as a cache line whose age is 3. Thus, we believe that a prefetched cache line is inserted into the LLC set with age 3 instead of being flagged for early eviction. This experiment also verified the replacement policy introduced in Section 4.2.

Property #1: On an LLC miss, `PREFETCHNTA` inserts the target cache line into the LLC with the age initialized as 3.

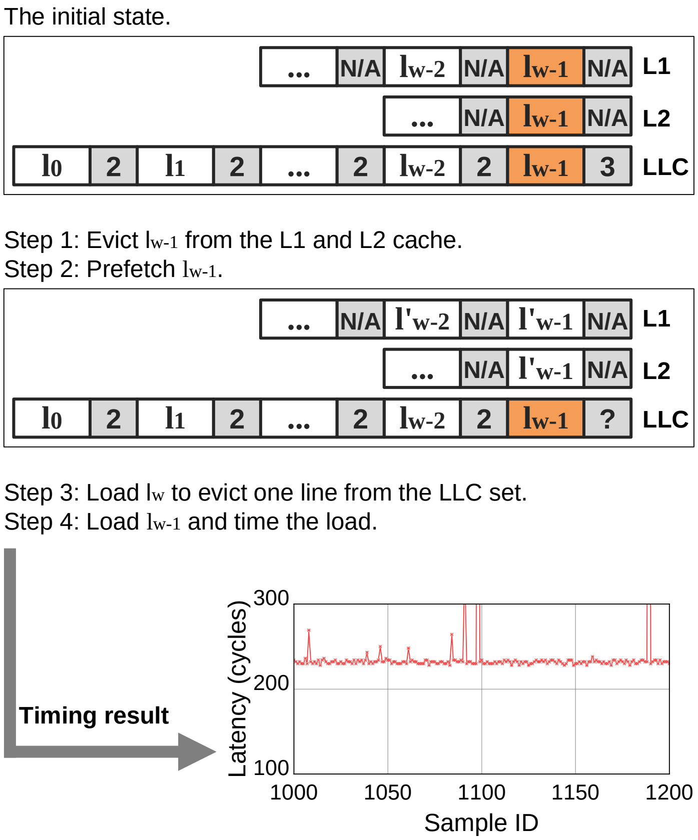

The initial state.
... N/A lw-2 N/A lw-1 N/A L1
... N/A lw-1 N/A L2
l0 2 l1 2 ... 2 lw-2 2 lw-1 3 LLC
Step 1: Evict lw-1 from the L1 and L2 cache.
Step 2: Prefetch lw-1.
... N/A l'w-2 N/A l'w-1 N/A L1
... N/A l'w-1 N/A L2
l0 2 l1 2 ... 2 lw-2 2 lw-1 ? LLC
Step 3: Load lw to evict one line from the LLC set.
Step 4: Load lw-1 and time the load.
Timing result
Latency (cycles)
300
200
100
1000
1050
1100
1150
1200
Sample ID

Figure 15: The experiment steps and results for learning the updating policy of `PREFETCHNTA`.

4.3.2.2 Updating Policy

In this section, we study the cache behavior of `PREFETCHNTA` when the target cache line is already in the LLC (but not in the private cache). Specifically, we are interested in whether an LLC hit caused by `PREFETCHNTA` updates the age of the target cache line in the LLC like a load instruction (cf. Section 4.2). In this experiment, we need two eviction sets: one for the LLC (l_0 to l_w) and one for the L1 and L2 cache (l'_0 to l'_w); l'_0 to l'_w are all mapped into the same L1/L2 set with l_0 to l_w, but different LLC sets. Then, we prepare the target LLC set as the initial state shown in Figure 15: the LLC set is filled with l_0 to l_{w-1}; the ages of l_0 to l_{w-2} are 2 but the age of l_{w-1} is 3. Thus, l_{w-1} is the eviction candidate in the LLC set. l_{w-1} may be also present in the L1/L2 cache. After the preparation, we run the following four steps (as shown in Figure 15) to learn whether prefetching l_{w-1} updates its age from 3 to 2:

Step 1: We access l'_0 to l'_w multiple times to ensure that l_{w-1} is no longer present in the L1 and L2 cache.[3] This step is necessary because if l_{w-1} is present in the L1 or L2 cache, when prefetching l_{w-1}, the request will not reach the LLC and we cannot learn the updating policy in the LLC.

Step 2: We prefetch l_{w-1}. This results in an LLC hit and may update the age of l_{w-1} in the LLC.

Step 3: We load a new cache line l_w into the LLC which evicts one of the existing lines in the LLC set.

Step 4: We access l_{w-1} and time the access to learn whether it was evicted in Step 3: if l_{w-1} was not evicted, then `PREFETCHNTA` updated its age in Step 2 (from 3 to 2), otherwise `PREFETCHNTA` did not update the age in Step 2.

We repeat the above experiment 10000 times and Figure 15 shows a segment of the collected timing results (in Step 4). It always takes over 200 cycles to load l_{w-1}, meaning l_{w-1} was likely in DRAM before it's loaded in Step 4. Thus, we can safely conclude that `PREFETCHNTA` did not update its age (from 3 to 2) so it was chosen by the replacement policy

[3] $w+1$ cache lines are enough to evict l_{w-1} from both the L1 and L2 cache because $L1\_Associativity + L2\_Associativity < LLC\_Associativity$ on our processors.

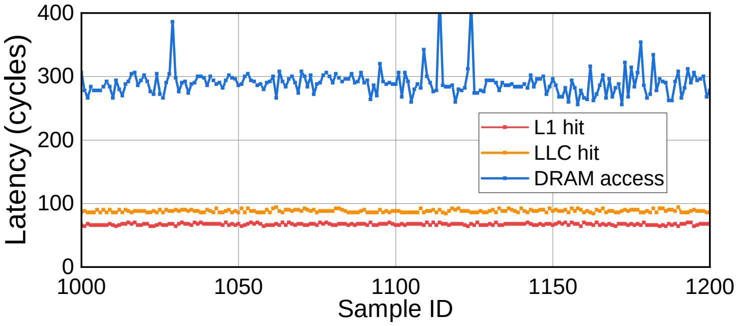


Figure 16: The execution times of PREFETCHNTA when the target data is the L1 cache, LLC, and DRAM.

and got evicted from the LLC in Step 3. Similarly, we have verified that PREFETCHNTA does not update the age of a cache line from 2 to 1, or 1 to 0 either, when hitting in the LLC.

Property #2: On an LLC hit, PREFETCHNTA does not update the age of the target cache line in the LLC.

4.3.2.3 Timing Variance

Prior work (e.g., [134, 132, 81, 66]) has shown that the execution time of a regular load instruction is related to the location of the target data in the memory hierarchy. Here we analyze whether PREFETCHNTA also has such timing variance. Specifically, we measure the execution time of PREFETCHNTA in three scenarios where the target cache line (l_t) is present in the L1 cache, not in the L1/L2 cache but present in the LLC, and not cached at all, respectively. The detailed operations for each scenario are as follows:

Scen. 1: We load l_t so that it is brought into the L1 cache; then we prefetch l_t and time the prefetch.

Scen. 2: We still load l_t first, as done in Scen. 1. However, before we prefetch l_t and measure the timing, we build set conflicts in the L1 and L2 cache to ensure that l_t is evicted from them.

Scen. 3: We first build set conflicts in the LLC to ensure that l_t is evicted from the entire cache hierarchy, and then we time the prefetch on it.

We test each scenario 10000 times and a segment of the collected timing results are shown in Figure 16. When the target cache line is present in the L1 cache, it takes around 70 cycles to prefetch it; it takes 90 to 100 cycles to prefetch when the cache line is only in the LLC, and over 200 cycles when the cache line is not cached at all.

Property #3: The execution time of `PREFETCHNTA` is related to the cache level of the target cache line.

4.4 A Covert Channel Based on `PREFETCHNTA`

Based on the properties of `PREFETCHNTA` that are reverse-engineered in Section 4.3, we build a new conflict-based cache covert channel. In this section, we first introduce the threat model, then discuss the details of this channel.

4.4.1 Threat Model

We use a similar threat model to previous conflict-based cache covert channels (e.g., [81]). We assume that the two essential parties for the channel, the sender and receiver are two unprivileged processes running on the same processor (but potentially different cores) with an inclusive LLC. We also assume that the sender and receiver are able to construct eviction sets for the LLC (e.g., using methods proposed in prior work [81, 93, 94, 121]). In addition, the sender and receiver should agree on the pre-defined channel protocols, including the synchronization, data encoding, target LLC set(s), and error correction protocols. Note that we do not assume any shared data between the sender and receiver, resulting in a more practical channel than channels relying on data sharing (e.g., [134, 55, 28, 67, 133, 59]).

Algorithm 4: NTP+NTP Covert Channel

```
d_s: the sender's data (cache line) for transmitting signals
d_r: the receiver's data (cache line) for transmitting signals
message[n]: the n-bit long message to be transferred
Th0: the timing threshold for distinguishing prefetch hit and miss
```

Sender Algorithm

```
// Send 1 bit in each iteration.
for $i = 0; i < n; i++$ do
    synchronization();
    if message[i] == 1 then
        Prefetch $d_s$;
    else
        Do not prefetch;
    wait_for_receiver();
```

Receiver Algorithm

```
// Detect 1 bit in each iteration.
for $i = 0; i < n; i++$ do
    synchronization();
    wait_for_sender();
    Prefetch $d_r$ and time the prefetch;
    if prefetch_time > Th0 then
        Received a bit "1";
    else
        Received a bit "0";
```

4.4.2 NTP+NTP

4.4.2.1 Channel Protocol

When prefetching a cache line into the LLC, it replaces the current eviction candidate of the LLC set. According to the replacement policy explained in Section 4.2, this eviction candidate is the first cache line in the set whose age is 3. Since the prefetched cache line's age is also set as 3 (cf. Property #1), it now becomes the first cache line in the set with age 3. This means that prefetching a cache line into the LLC evicts the current eviction candidate in the set, and then the prefetched cache line becomes the *new eviction candidate.* With this knowledge, we can build a covert channel where the sender and receiver communicate by competing (or not) for one way in an LLC set (i.e., the eviction candidate way). The sender and receiver can simply achieve this by prefetching their own cache lines which are mapped into the same LLC set. We name this covert channel NTP+NTP (Non-Temporal Prefetch+Non-Temporal Prefetch).

Basic channel protocol. In NTP+NTP, the sender and receiver first need to ensure that the sender's cache line d_s and the receiver's cache line d_r are mapped to the same LLC set, as done in prior work [81, 93, 66]. Then, the receiver prepares the channel by prefetching d_r into the LLC.[4] After this, the sender and receiver can communicate following Algorithm 4. One bit is transmitted in each iteration: the sender sends "1" by prefetching d_s into this target LLC set, or sends "0" by not prefetching. After this, the receiver receives the bit by prefetching d_r and times the prefetch. If the sender sends "1", then d_r should have been evicted from the LLC (by d_s); it takes longer for the receiver to prefetch. In contrast, if the sender sends "0", d_r is still in the LLC so it is faster for the receiver to prefetch. The sender and receiver can synchronize using the time stamp counters (TSCs).

The state change details in the target LLC set during the covert channel are shown in Figure 17. Before the sender and receiver start the covert channel, the target LLC set is in a random state, i.e., it is filled with random cache lines in random ages. When the receiver prefetches d_r for channel preparation, d_r becomes the first (left-most) cache line in the set with age 3, i.e., the eviction candidate. Thus, if now the sender prefetches d_s (to send "1"), it evicts d_r and then d_s becomes the new eviction candidate since it is now the first cache line with age 3. Therefore, when the receiver later prefetches d_r (for receiving the bit), it takes over 200 cycles to finish the prefetch (cf. Property #3). This prefetch also evicts d_s and then d_r is the eviction candidate again, i.e., this LLC set is ready for transmitting the next bit. In contrast, if the sender does not prefetch d_s in this iteration (to send "0"), then d_r is not evicted. Later when the receiver prefetches d_r, it will get an LLC hit (or a private cache hit) which takes less than 100 cycles. In addition, this prefetch does not update the age of d_r (cf. Property #2). Thus, d_r is still the eviction candidate and this LLC set is ready for the next iteration. In summary, the receiver's operation, prefetching d_r and timing the prefetch, is able to measure the bit from the sender in the current iteration, as well as reset

[4] We assume that the target set does not have empty ways which is true for most cases. The receiver can also prepare an eviction set and load it before the channel starts to ensure there is no empty way.

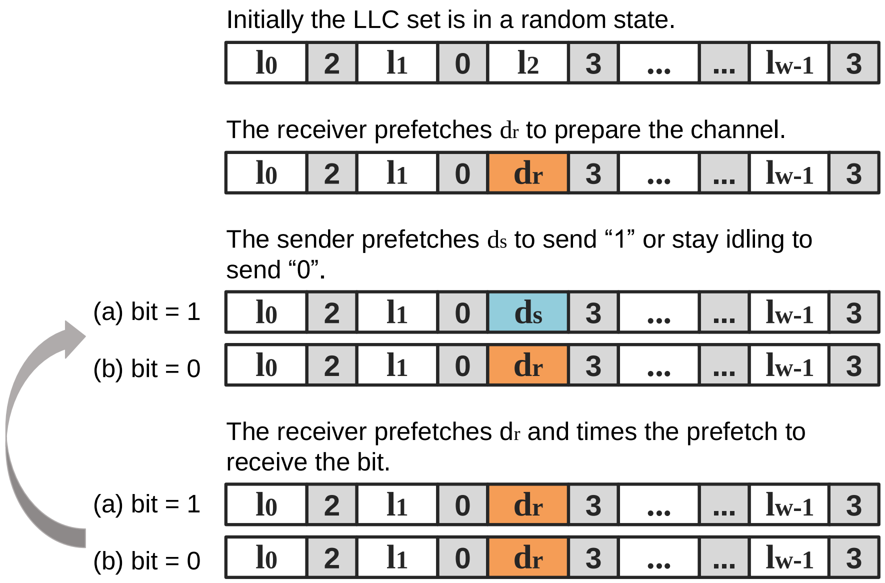
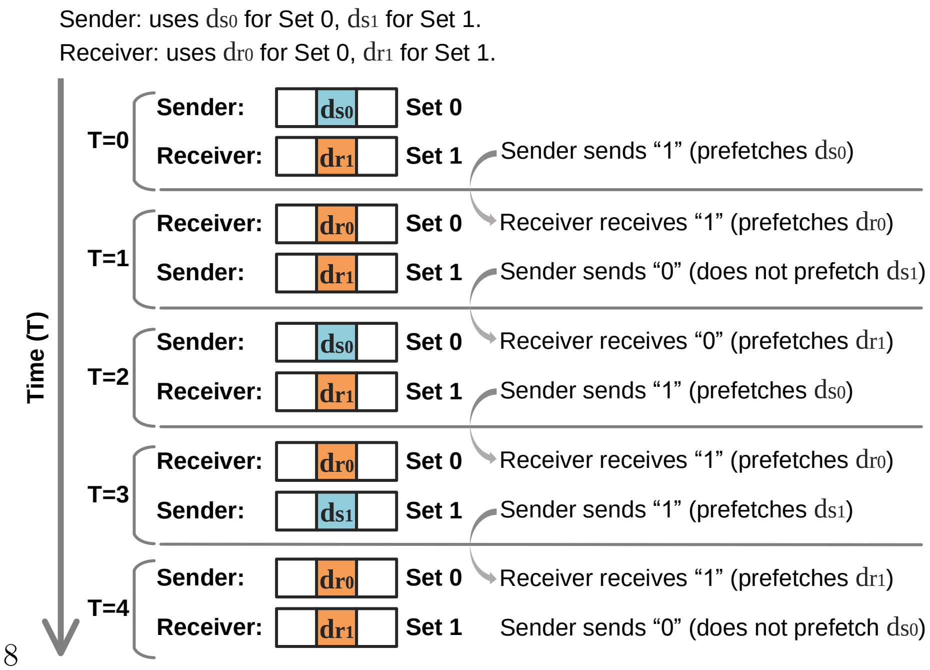

Initially the LLC set is in a random state.
l0 2 l1 0 l2 3 ... ... lw-1 3
The receiver prefetches dr to prepare the channel.
l0 2 l1 0 dr 3 ... ... lw-1 3
The sender prefetches ds to send "1" or stay idling to send "0".
(a) bit = 1
l0 2 l1 0 ds 3 ... ... lw-1 3
(b) bit = 0
l0 2 l1 0 dr 3 ... ... lw-1 3
The receiver prefetches dr and times the prefetch to receive the bit.
(a) bit = 1
l0 2 l1 0 dr 3 ... ... lw-1 3
(b) bit = 0
l0 2 l1 0 dr 3 ... ... lw-1 3

Figure 17: How the state of the target LLC set changes during the NTP+NTP covert channel.

the state of the target LLC set so that it is ready for transmitting the next bit.

Compared to Prime+Probe. Prior conflict-based covert channels such as Prime+Probe and its variants [81, 93] require the sender and receiver together access at least $w + 1$ cache lines in each iteration to cause cache conflicts; w is the set associativity of the LLC. For example, if the sender sends a bit by loading (or not) a single cache line d_s, in each iteration the receiver needs to prime the target LLC set (by accessing at least w cache lines) to evict d_s and get ready for the next iteration. This is because after the sender loads d_s, it may become the youngest cache line in the target LLC set. To evict d_s, the receiver needs to first access all other $w - 1$ cache lines in the set to "refresh" their ages and make d_s the oldest cache line in the set. Then, the receiver accesses a cache line that is not present in the LLC, causing set conflict and thus evicting d_s.

In NTP+NTP, the sender inserts d_s into the target LLC set as the oldest cache line. Thus, the receiver is able to evict it using only *one operation*. Essentially, the sender and receiver can use `PREFETCHNTA` to bypass the w-way associativity of the LLC and use it as a one-way associative LLC. This results in much more efficient LLC conflicts and thus a faster covert channel.



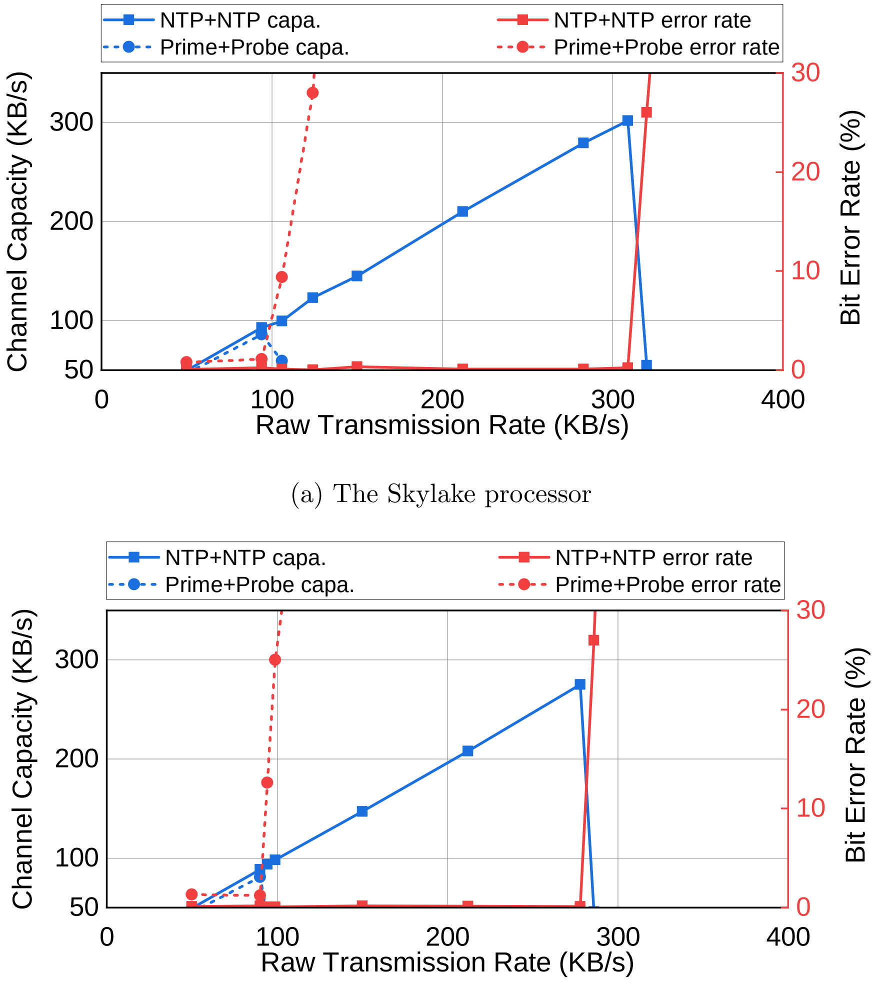
Sender: uses ds0 for Set 0, ds1 for Set 1.
Receiver: uses dr0 for Set 0, dr1 for Set 1.
Time (T)
T=0
Sender:
ds0
Set 0
Receiver:
dr1
Set 1
Sender sends "1" (prefetches ds0)
T=1
Receiver:
dr0
Set 0
Receiver receives "1" (prefetches dr0)
Sender:
dr1
Set 1
Sender sends "0" (does not prefetch ds1)
T=2
Sender:
ds0
Set 0
Receiver receives "0" (prefetches dr1)
Receiver:
dr1
Set 1
Sender sends "1" (prefetches ds0)
T=3
Receiver:
dr0
Set 0
Receiver receives "1" (prefetches dr0)
Sender:
ds1
Set 1
Sender sends "1" (prefetches ds1)
T=4
Sender:
dr0
Set 0
Receiver receives "1" (prefetches dr1)
Receiver:
dr1
Set 1
Sender sends "0" (does not prefetch ds0)
8

Figure 18: The operations of the sender and receiver in each iteration of NTP+NTP, when using two LLC sets; the receiver always detects the bit sent in the last iteration instead of the current iteration.

4.4.2.2 Channel Capacity

We implement NTP+NTP and Prime+Probe on two Intel processors (as listed in Table 4) to test their bandwidths. For Prime+Probe, we use the example implementation discussed above: the sender accesses (or not) one cache line, and the receiver primes with w cache lines.

The bandwidth of NTP+NTP is limited when using one target LLC set: if the cache line in an LLC way is in-flight (e.g., waiting for the memory response), this cache line cannot be evicted regardless of its age. This means d_r cannot evict d_s if d_s is still in-flight when the prefetch request of d_r reaches the LLC. Thus, we need to space out the prefetches from the



(a) The Skylake processor

(b) The Kaby Lake processor

Figure 19: The capacities and bit-error-rates of NTP+NTP and Prime+Probe.

sender and receiver (in each iteration). To avoid the slowdown caused by the spacing, we use two LLC sets in NTP+NTP and let the sender and receiver access different sets in each iteration. As shown in Figure 18, the receiver is always detecting the bit that was sent one iteration earlier. For fair comparison, we also use two sets in Prime+Probe. However, we do not use the sets as in Figure 18 since it does not benefit Prime+Probe much. Instead, we just use the two sets to transfer two bits in each iteration.

We measure the channel capacities and bit error rates of both channels, under different transmission intervals. Although the raw transmission rate increases when decreasing the transmission interval, the bit error rate may also increase, especially when the interval is too short. To find the best transmission rate, we use the channel capacity metric (as in [92, 90]). This metric is computed by multiplying the raw transmission rate with $1 - H(e)$, where e is the bit error rate and H is the binary entropy function. The results are shown in Figure 19. The bit error rates of both channels stay low (lower than 0.5% for NTP+NTP, 1.5% for Prime+Probe) and are almost constant, when the raw transmission rate is under a threshold (e.g., 304 KB/s for NTP+NTP in Figure 19 (a)). Thus, the channel capacity increases proportionally to the raw transmission rate. It reaches the peak when the raw transmission rate is around this threshold. Beyond this threshold, the increasing error rate causes a decrease in the channel capacity. The peak capacities of the two channels are summarized in Table 5.

Table 5: The maximum channel capacities of NTP+NTP and Prime+Probe.

Platform	Skylake	Kaby Lake
NTP+NTP	302 KB/s	275 KB/s
Prime+Probe	86 KB/s	81 KB/s

4.4.2.3 Channel Reliability

Similar to prior conflict-based covert channels, NTP+NTP is also affected by noise from other processes accessing data mapped to the target LLC set. For example, in a transmission iteration, although the victim sends “0” by not prefetching d_s, the receiver may receive “1” if other processes access their data and evict d_r, i.e., a false positive occurs.

This problem can be solved by using a more reliable data encoding method [81, 66, 84], rather than the very simple method in Algorithm 4. For example, multiple LLC sets can be used to send one bit. Note that the error caused by other processes’ accesses in one attack iteration will not affect the next iteration: once the receiver prefetches d_r, d_r is the eviction

```
/* evset is the eviction set for priming */
/* the scope line addr is in evset[0] */
for(i = 0; i < 3; i++) {
    for(j = 0; j < 13; j+=4) {
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[0]);
        memaccess((void *) evset[j+3]);
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[j+3]);
        memaccess((void *) evset[j+0]);
        memaccess((void *) evset[j+1]);
        memaccess((void *) evset[j+2]);
        memaccess((void *) evset[j+3]);}}
```

Listing 4.1: The preparation step in Prime+Scope.

candidate again. If other processes flush their data in the target LLC set, it will create empty ways, which can also impact the performance of NTP+NTP. However, `CLFLUSH` is rarely used in daily applications [96, 134, 128], and this problem can also be avoided by using a more reliable channel encoding method.

4.5 Side Channel Attacks Based on `PREFETCHNTA`

NTP+NTP introduced in the last section is unlikely a side channel because the sender is transmitting the signal by "prefetching (or not) a cache line". In other words, the attacker (receiver) can only detect the victim's (sender's) prefetch patterns on a cache line, resulting in very limited attack opportunities to normal applications. However, the properties of

PREFETCHNTA reverse-engineered in Section 4.3 make it much easier for users to manipulate the replacement states (ages) of cache lines in the LLC than before. Thus, attackers can also use PREFETCHNTA to improve the existing cache attacks that are based on cache replacement states, making them more efficient and accurate. In this section, we use two cache attacks that were proposed very recently as examples to show how they can benefit from using PREFETCHNTA.

4.5.1 Prime+Scope with PREFETCHNTA

4.5.1.1 Prime+Scope

Prime+Scope [93] proposed in 2021 is an LLC attack based on set conflicts. Prime+Scope is similar to Prime+Probe, but it has much higher temporal resolution. In each iteration of Prime+Scope, the attacker first primes the target LLC set with a *special pattern* to ensure two things. First, the target LLC set is occupied by the attacker's cache lines. Second, the current eviction candidate (a.k.a. the scope line, l_s) in the LLC set is also present in the attacker's private cache. Then, the attacker repeatedly accesses the scope line and times the access to detect the victim's access to her own cache line (which is also mapped to this LLC set). When the victim has not yet accessed her cache line in the current iteration, the attacker's accesses to l_s always hit in the private cache; once the victim accesses her cache line and brings it to the LLC, l_s is evicted and the attacker reaches an LLC miss. Then, the current iteration ends; the attacker primes this set again and moves to the next iteration. Note that the attacker can repeatedly access l_s without disturbing its replacement state in the LLC and changing the eviction candidate. This is because private cache hits do not update the replacement state of the LLC copy.

Prime+Scope is an important attack because it has very high temporal resolution. On our processors, loading a cache line that is in the private cache and timing the load together only take around 70 cycles. Thus, with Prime+Scope, the attacker can locate the victim's access in the time domain with a granularity of 70 cycles. For example, the attacker can know that the victim's access happened when $70 < current\_time < 140$ or when $140 < current\_time < 210$. In comparison, the resolution of Prime+Probe is over 2000 cycles [93].

```
/* prime the eviction set n times */
for(i = 0; i < n; i++)
    for(j = 1; j <= 16; j++)
        memaccess((void *) evset[j]);

/* prefetch the scope line after priming */
prefetch_nta((void*) evset[0]);
```

Listing 4.2: The preparation step with PREFETCHNTA.

There are two necessary conditions for building this high-resolution attack. First, the attacker needs to know the eviction candidate (l_s) of the target LLC set after priming. Second, l_s needs to be present in the private cache after priming, otherwise once the attacker accesses l_s, it is no longer the eviction candidate in the LLC. These two requirements make the attack very challenging because they are intuitively contradictory: being the eviction candidate means l_s is accessed *less frequently* than other cache lines; being present in the private cache means l_s is accessed *more frequently* than other cache lines. To satisfy the requirements, the original Prime+Scope uses very long and complicated access sequences to manipulate the replacement states of both the private cache and the LLC. The access sequence[5] for our Skylake processor is shown in Listing 4.1. It contains 192 cache references in total. This long access sequence results in a slow preparation (priming) step. Thus, although Prime+Scope has high temporal resolution in each attack iteration, it requires a long preparation step between two consecutive iterations. Therefore, the attacker may miss the victim's accesses when the victim is repeatedly accessing her cache line with a high frequency, resulting in a high attack error rate.

4.5.1.2 Prime+Prefetch+Scope

The two key requirements in Prime+Scope can be satisfied in a much easier way when using PREFETCHNTA. As explained in Section 4.4.2, when prefetching a cache line, it is installed

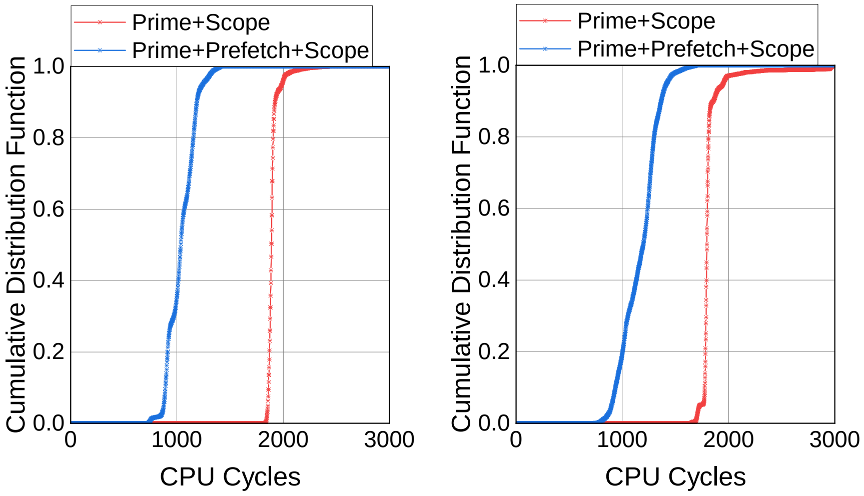
[5] This pattern is not optimal. For example, it could be more efficient with knowing the details of the L1 replacement policy. Prime+Scope does not assume that knowledge for generality.

in the LLC set as the eviction candidate, and at the same time it is brought into the L1 cache. Thus, the preparation step in Prime+Scope can be done using the operations shown in Listing 4.2. We first prime the LLC set by accessing the eviction set (consisting of w cache lines without l_s) several times, so that the victim's data in this set gets old and can be reliably evicted. Then we prefetch l_s to install it into the L1 cache, as well as the LLC as the eviction candidate. Note that on tested processors, priming the eviction set twice is enough for reliably evicting the victim's data (with over 99.99% probability). Thus, on our Skylake processor, we only need 33 cache references (compared to 192 in the original Prime+Scope), resulting in a more efficient attack.

4.5.1.3 Faster Preparation Step

We test the total latency of the preparation step in each attack iteration. For the original Prime+Scope, to prepare the attack iteration, the attacker primes the target LLC set with a long pattern that takes a long period of time to finish. As shown in Figure 20, the preparation takes on average 1906 cycles on our Skylake processor (and 1762 on Kaby Lake). In contrast, with `PREFETCHNTA`, although the attacker needs to first prime the LLC set and then prefetch the scope line, the priming pattern is much shorter. The entire preparation step only takes 1043 cycles on the Skylake processor (and 1138 on Kaby Lake).

The faster preparation step can make Prime+Prefetch+Scope more reliable and accurate than Prime+Scope. To prove this, we use three threads (T1 and T2) pinned on two different cores. T1 accesses a predetermined address every 1.5K cycles, as ground truth. T2 continuously monitors the LLC set for events using one of the attacks. We consider it a false negative if an event (from T1) is not detected. From the experiments on our Skylake processor, the false negative rate is about 50% for Prime+Scope. However, when using Prime+Prefetch+Scope, this rate is reduced to less than 2%.



(a) The Skylake processor

(b) The Kaby Lake processor

Figure 20: The total latency of the preparation step, for the two attack primitives.

4.5.2 Reload+Refresh with `PREFETCHNTA`

4.5.2.1 Reload+Refresh

Reload+Refresh [28] is one of the first attacks that leak the victim's information by monitoring *the replacement state changes* to the victim's cache line. Reload+Refresh is an LLC attack and it assumes shared data between the attacker and victim. To learn the victim's access pattern on the shared cache line (d_t), the attacker needs to prepare an eviction set (l_0 to l_{w-1}) that is mapped to same LLC set with d_t. Each attack iteration in Reload+Refresh consists of five steps, as shown in Figure 21. In Step 1, the attacker fills the target LLC set with d_t and l_0 to l_{w-2} in order. After this, all the cache lines in this set are in age 2. Since d_t is the first line in the set, it is the eviction candidate. Then in Step 2, the attacker waits for the victim; if the victim accesses d_t, its age is updated to 1, and l_0 becomes the eviction candidate. Then in Step 3, the attacker forces replacement in this set by loading l_{w-1}. Either d_t or l_0 is evicted depending on whether the victim accessed d_t in

Step 2. Then in Step 4, the attacker reloads d_t and times the load to learn whether it was evicted in the last step and infer whether the victim accessed it in Step 2. Finally in Step 5, the attacker reverts the changes in this set to prepare for the next attack iteration. The attacker first flushes d_t and l_{w-1} and then loads d_t and l_0 so that the states of d_t and l_0 are reset. After this, the attacker accesses l_1 to l_{w-2} in order, to refresh their ages from 3 back to 2.

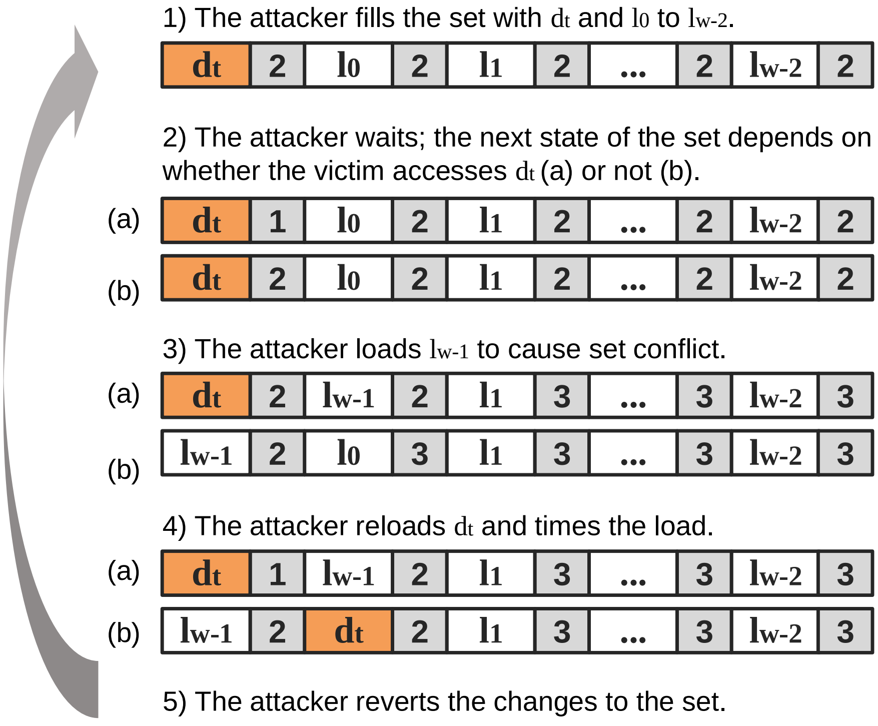

1) The attacker fills the set with dt and l0 to lw-2.
dt 2 l0 2 l1 2 ... 2 lw-2 2
2) The attacker waits; the next state of the set depends on whether the victim accesses dt (a) or not (b).
(a) dt 1 l0 2 l1 2 ... 2 lw-2 2
(b) dt 2 l0 2 l1 2 ... 2 lw-2 2
3) The attacker loads lw-1 to cause set conflict.
(a) dt 2 lw-1 2 l1 3 ... 3 lw-2 3
(b) lw-1 2 l0 3 l1 3 ... 3 lw-2 3
4) The attacker reloads dt and times the load.
(a) dt 1 lw-1 2 l1 3 ... 3 lw-2 3
(b) lw-1 2 dt 2 l1 3 ... 3 lw-2 3
5) The attacker reverts the changes to the set.

Figure 21: Sequence of the LLC set states during Reload+Refresh.

Reload+Refresh is a powerful attack since it is much stealthier (on the victim's side) compared to prior LLC attacks such as Flush+Reload [134]. However, similar to Prime+Scope, many operations are needed in Reload+Refresh to reset the LLC state (in Step 5). In Flush+Reload, after measuring the victim's behavior (by reloading the shared data), the attacker only needs to flush this data to reset the state. In contrast, in Reload+Refresh the attacker needs to perform two flushes, two memory accesses, and $w - 2$ serialized LLC

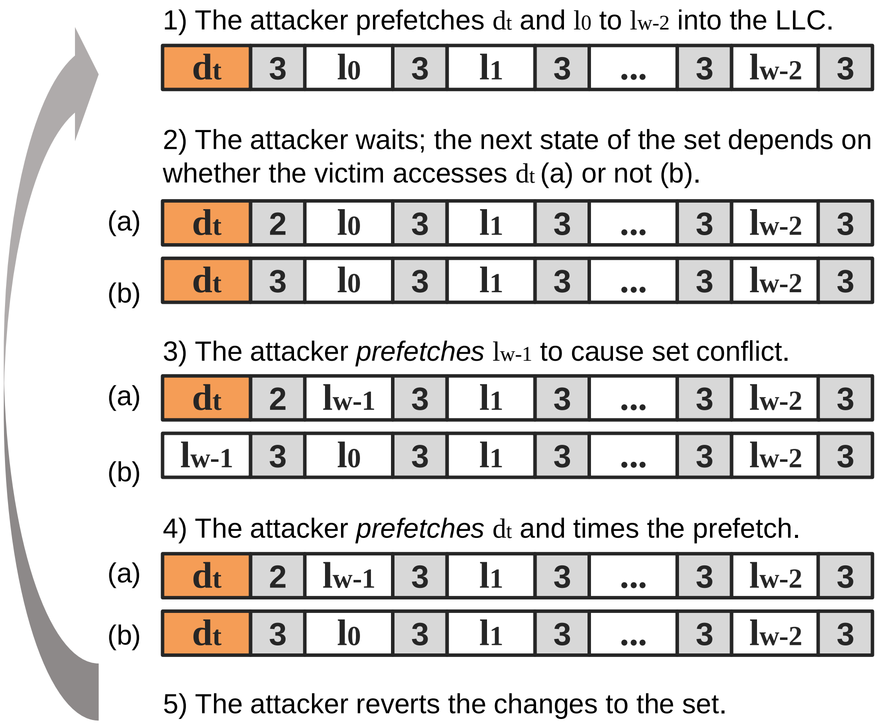
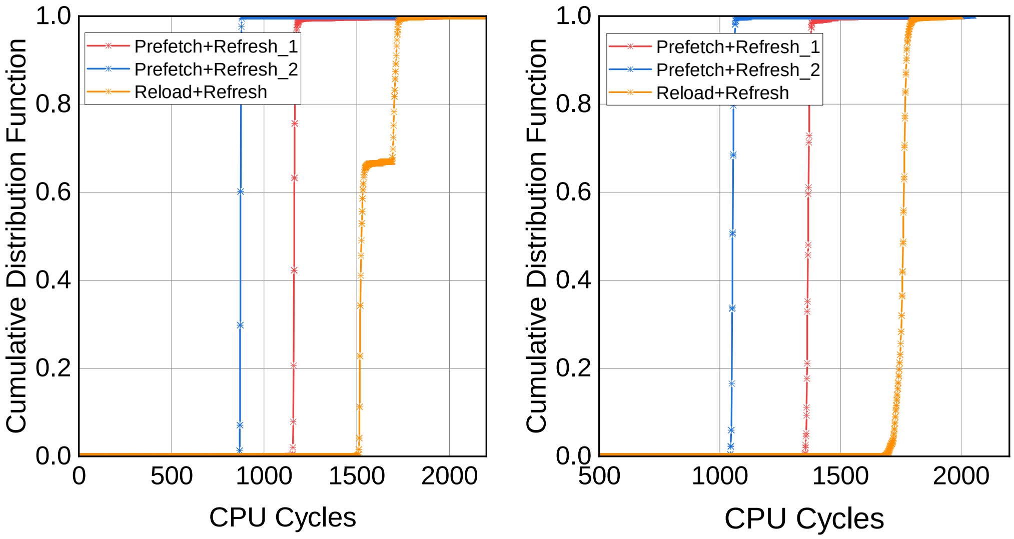

1) The attacker prefetches dt and l0 to lw-2 into the LLC.
dt 3 l0 3 l1 3 ... 3 lw-2 3
2) The attacker waits; the next state of the set depends on whether the victim accesses dt (a) or not (b).
(a) dt 2 l0 3 l1 3 ... 3 lw-2 3
(b) dt 3 l0 3 l1 3 ... 3 lw-2 3
3) The attacker prefetches lw-1 to cause set conflict.
(a) dt 2 lw-1 3 l1 3 ... 3 lw-2 3
(b) lw-1 3 l0 3 l1 3 ... 3 lw-2 3
4) The attacker prefetches dt and times the prefetch.
(a) dt 2 lw-1 3 l1 3 ... 3 lw-2 3
(b) dt 3 l0 3 l1 3 ... 3 lw-2 3
5) The attacker reverts the changes to the set.

Figure 22: Sequence of the LLC set states during Prefetch+Refresh.

accesses. Due to these operations, the state reset step and the entire attack iteration take very long to finish.

4.5.2.2 Prefetch+Refresh

We propose a new attack named Prefetch+Refresh which works in a similar way to Reload+Refresh but with much less operations for resetting the state in each iteration. As shown in Figure 22, this attack also consists of five steps. In Step 1, the attacker prepares the target LLC set similar to the one in Reload+Refresh; however, the attacker initializes the age of each cache line to 3 instead of 2. Then in Step 2, the attacker waits for the victim; if the victim accesses d_t, its age is changed from 3 to 2. Later in Step 3, the attacker *prefetches* l_{w-1} (instead of loading it) to cause conflict in this set. Then in Step 4, the attacker *prefetches* d_t, as well as measures the prefetch latency to learn the victim's behavior in Step 2. Eventually in Step 5, the attacker reverts the changes to this LLC set. If we compare the state of this LLC set after Step 4 and the state in Step 1, only the two left most lines are potentially changed: if the victim accessed d_t, now its age is 2 instead of 3, and the second cache line from the left is l_{w-1} instead of l_0. Thus, the attacker does not need to access l_1 to l_{w-2} to change their ages, as done in Reload+Refresh. This results in a faster state reverting step in Prefetch+Refresh.



(a) The Skylake processor

(b) The Kaby Lake processor

Figure 23: The total latency of the attacker's operations in each attack iteration, for Reload+Refresh and the two versions of Prefetch+Refresh.

We propose two options for the attacker to revert the states of these two cache lines in Step 5. First, the attacker can simply flush d_t and l_{w-1} and then reload d_t and l_0 to undo the state changes. In the second option, the attacker still flushes and reloads d_t. But she does not flush l_{w-1} and reload l_0, she instead uses l_0 to cause set conflict (in Step 3) in the next attack iteration, if the victim accessed d_t. In other words, the attacker exchanges the roles of l_0 and l_{w-1}. The second option makes Step 5 even faster compared to the first option; however, it slightly increases the complexity of the attack. The attacker needs to dynamically determine the cache line to use in Step 3 in each iteration. Table 6 shows the operations needed in Step 5 in Reload+Refresh and the two versions of Prefetch+Refresh.

Table 6: # of operations for reverting the cache state with a 16-way associative LLC.

Attack Method	# of flushes	# of DRAM accesses	# of LLC accesses
Reload+Refresh	2	2	14
Prefetch+Refresh v1	2	2	0
Prefetch+Refresh v2	1	1	0

4.5.2.3 Faster Attacks

We test the total latency of performing all the attacker operations in each iteration. For Reload+Refresh, the operations include loading l_{w-1} (to cause conflict), reloading d_t, flushing d_t and l_{w-1}, reloading d_t and l_0, and accessing l_1 to l_{w-2} (with pointer chasing). As shown in Figure 23, the average latency of a Reload+Refresh iteration (without the waiting window) is 1601 cycles on our Skylake processor (and 1767 cycles on the Kaby Lake processor). In contrast, when using Prefetch+Refresh (v1), the attacker does not need to access l_1 to l_{w-1}, and thus the average latency of an iteration is reduced to 1165 (and 1369) cycles. In Prefetch+Refresh (v2), flushing l_{w-1} and reloading l_0 are eliminated and the average latency is only 873 (and 1054) cycles.

4.6 Discussion

4.6.1 Fast Eviction Set Construction

Conflict-based cache attacks such as Prime+Probe require the attacker to build eviction sets: given a target address, the attacker needs to find groups of addresses that are mapped into the same set with it (i.e., congruent with it) in the target cache such as the LLC. As mentioned in Section 4.4.2, the properties of PREFETCHNTA allow us to achieve one-way competition in an LLC set. As a result, with PREFETCHNTA, set conflicts occur more frequently than before when searching for congruent addresses. This leads to a more efficient algorithm for constructing eviction sets. Algorithm 5 shows our eviction set construction method. It repeatedly measures the prefetch latency of the target cache line l_t, and before each measurement, it prefetches a new candidate line l_c (which is potentially congruent with l_t). If the prefetched l_c is congruent with l_t, l_t is evicted and later it takes longer to prefetch it; then this l_c is added to the congruent address list. If the prefetched l_c is not congruent with l_t, it takes shorter to prefetch and l_t remains being the eviction candidate in the set after the prefetch; the algorithm then moves on to test the next candidate l_c. The algorithm keeps looking for congruent addresses until enough are found.

Algorithm 5: Eviction Set Construction

```
Input: l_t, the target cache line for which an eviction set is desired
Output: EV, the eviction set
EV ⟵ an empty list
ev_count ⟵ 0
while ev_count < ev_desired_size do
    prefetch l_t
    do
        l_c ⟵ a candidate line
        prefetch l_c
    while prefetch l_t is fast;
    EV[ev_count] ⟵ l_c
    ev_count + +
end
```

The state-of-the-art eviction set construction method [93] uses a similar algorithm with ours. However, it accesses l_t and l_c in each searching iteration instead of prefetching them (in line 4, line 7, and line 8 of Algorithm 5). With this approach, a congruent cache line can only be observed (l_t can be evicted) if about w congruent lines have been tested/accessed since the last time l_t was brought into the LLC (in line 4), where w is the LLC associativity. This is because when accessing l_t (in line 4), it becomes the youngest cache line in the LLC set which will not be evicted until about w congruent cache lines are accessed (accessing EV between line 4 and line 5 can slightly reduce this number). When l_t is finally evicted, we only know that the last accessed l_c is congruent with it. In contrast, when using `PREFETCHNTA`, prefetching each congruent cache line can evict l_t since l_t is installed as the eviction candidate, making the algorithm much more efficient compared to the state-of-the-art. We test the execution time of these two approaches, and the results are shown in Figure 24.

We provide an analysis on the time complexities of Algorithm 5 and the state-of-the-art [93]. We only analyze the number of memory references as they dominate the execution time. Similar to prior work [121], we assume that from the physical address, c bits are used for set index, the s-bit hash value of the c set index bits and higher bits are used for slice index, and users control the lowest γ bits of set index bits. We also assume that the attacker samples from a large set of addresses M. M contains addresses that coincide with l_t (the target address) on the user-controlled γ set index bits. We denote the event that $l_c \leftarrow M$ (uniform sampling) is congruent with l_t as C, which causes a set and slice conflict. Then, the probability of this event is given by:

$$p := P(C) = 2^{\gamma - c - s} \tag{4-1}$$

The attacker's goal is to find a minimal set of addresses $EV = \{ev_0, ev_1, ...\}$ ($EV \subseteq M$) that can be used to reliably evict l_t. For simplicity, in this analysis we assume the LRU policy. Thus, $|EV| = w$, where w is the associativity. Repeatedly sampling and counting the number of successes follows a binomial distribution, where the success probability p is as above. Suppose the number of trials in the experiment is $n := p^{-1}$, the expected number of successes is given by np. We then expect $np = p^{-1} \cdot p = 1$ success, i.e., we find one member of EV after p^{-1} samples, on average.

The state-of-the-art approach is the same with Algorithm 5, but using load instead prefetch in line 4, line 7, and line 8. To find one ev_x, in Step 1 it loads l_t into cache (line

4); l_t is now the MRU line in the LLC. Then in Step 2 it repeatedly samples l_c and checks if l_t is evicted after accessing l_c, until l_t is evicted (line 5-8). Since l_t is the MRU line, w congruent addresses need to be sampled (and accessed) in Step 2 to evict l_t. Thus, using linearity of the expectation, Step 2 samples l_c $w \cdot p^{-1}$ times on average. Then, to find all w ev_x, it repeats Step 1 and Step 2 w times, resulting in $O(w + w \cdot w \cdot p^{-1}) = O(w^2 p^{-1})$ memory references. We should mention that in a noiseless environment, we can take advantage of LRU order to access $O(p^{-1})$ in Step 2 every other round. However, this does not change the asymptotics of the algorithm. Note that line 8 does not update the replacement state of l_t in the LLC since this access should be filtered by lower-level caches.

For our approach (Algorithm 5), in Step 1 (line 4) l_t is the LRU line after prefetching. Thus, in Step 2, one congruent address needs to be sampled and prefetched to evict l_t; Step 2 only needs to sample p^{-1} times. Then, to obtain ev_0 to ev_{w-1}, the total amount of memory references is $O(w + w \cdot p^{-1}) = O(wp^{-1})$.

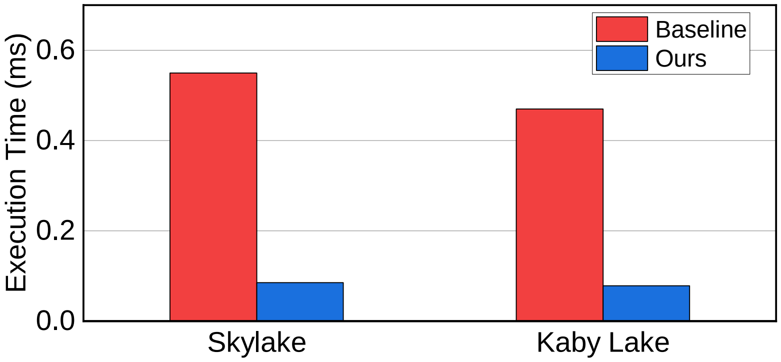


Figure 24: The execution time of the two algorithms.

4.6.2 `PREFETCHNTA` with Non-Inclusive LLCs

Most Intel server processors use non-inclusive LLCs. On such processors, `PREFETCHNTA` brings data only to the L1 cache and the coherence directory, but not the LLC [3]. Thus, the covert channels and side channel attacks discussed in this chapter cannot directly work on those processors. However, if prefetched data are easier to be evicted from a set-associative coherence directory than loaded data, it may be possible for us to build fast set conflicts in the directory, resulting in a directory version of NTP+NTP. Verifying this vulnerability requires comprehensively understanding the replacement policy of the directory. Unfortunately, the directory policy has not yet been fully reverse engineered [132, 93, 94]. We leave it as future work.

Note that according to [8], on some AMD processors prefetched data are placed into a software-invisible buffer (instead of cache/directory). Therefore, it may be possible to build conflicts using PREFETCHNTA in this buffer and create a new covert channel.

4.7 Chapter Summary

In this chapter, we reverse-engineered the detailed cache behaviors of `PREFETCHNTA`, the non-temporal data prefetch instruction, on Intel processors. From the results, we found that using `PREFETCHNTA`, two cache lines that are mapped into the same LLC set can compete for the eviction candidate way in the set, achieving cache conflicts without priming the cache set for the first time. Based on this, we proposed Prefetch+Prefetch, a conflict-based cache covert channel which has much higher bandwidth compared to existing conflict-based channels such as Prime+Probe. In addition, we showed how `PREFETCHNTA` can be used in cache side channel attacks to improve their performance. We also demonstrated a new LLC eviction set construction algorithm which is significantly faster than the state-of-the-art.

5.0 Cache Attacks Based on Uncore Frequency Scaling

5.1 Overview

Following Intel's terminology, a multi-core processor consists of multiple cores and an uncore. The uncore typically includes the last-level cache (LLC), the on-chip interconnect, the memory controllers (MCs), and other components. Over the last two decades, the microarchitectural resources in both the cores (e.g., branch predictor [16, 46, 47]) and the uncore (e.g., the LLC [81, 66, 132]) have been exploited to mount covert channels (and side channel attacks). However, covert channels based on the uncore components are a more serious threat to the security of modern systems, as the uncore is shared among all the applications running on the processor.

Fortunately, in recent years, there has been a growing focus on uncore covert channels, leading to the development of countermeasures against them (e.g., [62, 21, 38, 86, 18]). Since most uncore covert channels are based on *uncore resource contention/conflict*, partitioning the uncore hardware resources among users is a promising countermeasure approach. Various partitioning strategies with different granularities have been proposed to mitigate different uncore covert channels. For example, inside the uncore of a processor, LLC set partitioning [113] can defend covert channels based on LLC set conflicts (e.g., Prime+Probe [84]); tile partitioning may mitigate covert channels based on interconnect contention (e.g., the mesh contention [37]). In addition, for a multi-processor (multi-socket) system, one can also use a coarse-grained mechanism which assigns each user to a separate processor. In this scenario, each user has its own uncore, and users are not allowed to make cross-socket memory allocations/accesses, resulting in no cross-socket uncore contention. With all these partitioning designs, we ask the following questions:

Can uncore partitioning prevent all uncore covert channels? Can we build a practical uncore covert channel that remains functional even with one or more partitioning mechanisms in place?

Recently, Chen et al. [31] provided a preliminary answer to these questions by proposing a covert channel based on uncore idle power management. This channel cannot be stopped by existing uncore partitioning designs; however, it requires an "idle" system, making it highly susceptible to noise from co-located users and thus likely impractical on real systems. Therefore, we must explore alternative solutions.

There is an ever-growing need for improving power efficiency, as higher power usage directly translates into higher operational costs for data centers. The power consumption of a processor is closely related to its frequency. Thus, adjusting the processor frequency based on the workloads has been widely used on Intel processors to reduce power usage. Earlier Intel processors use a common frequency for the cores and the uncore. On more recent processors, the uncore frequency is controlled independently and can be different than the core frequency. In addition, Intel has introduced a mechanism called uncore frequency scaling (UFS) for their Xeon processors, which adjusts the uncore frequency based on uncore needs [60, 25]. This UFS mechanism, however, may actually introduce practical uncore covert channels that cannot be prevented by uncore partitioning.

In this chapter, we conduct a series of experiments to study the detailed behavior of UFS. We have three important observations from the results. First, the power monitoring unit (PMU) continuously monitors the system status, and adjusts the uncore frequency by increasing, decreasing, or maintaining it approximately every 10 ms. Second, when there is low demand for uncore resources, the uncore frequency remains relatively low. There are (at least) two situations that can cause the uncore to operate at a higher frequency: 1) high uncore utilization, such as frequent LLC accesses and dense interconnect traffic, and 2) a significant proportion of active cores being stalled. Third, we found that for a multi-processor system, all the uncores in different processors always maintain similar frequencies.

Next, based on these observations, we propose a new uncore covert channel that can operate as both a cross-core and cross-processor channel. We name this covert channel UF-variation. Specifically, the sender manipulates the uncore frequency (e.g., by controlling the density of LLC accesses) and encodes the data into the uncore frequency variation within each transmission interval. For example, the sender increases the uncore frequency in the interval to send a bit "1", and decreases it to send a bit "0". The receiver then obtains the data by observing the uncore frequency variation within a transmission interval. We found that the LLC access latencies differ significantly when the uncore operates at different frequency levels. Thus, the receiver can indirectly determine the uncore frequency by timing the LLC accesses. We test the channel capacities of UF-variation and show that UF-variation can achieve a capacity of 46 bit/s in the cross-core case, and 31 bit/s in the cross-processor case. Compared to other covert channels, the capacities of UF-variation are limited. However, we show that UF-variation remains functional even with one or more uncore partitioning mechanisms enabled, while most prior uncore covert channels can be prevented by those mechanisms.

Finally, we demonstrate how UFS can be used for side channel attacks to profile the activities of co-located users. For example, when used for website fingerprinting, the UFS-based attack can achieve a top-1 accuracy of 82.18%.

5.2 CPU On-Chip Interconnect

On multi-core processors, an on-chip interconnect is used to connect the processor cores, LLC slices, MCs, and other components (e.g., the PCIe controller). This interconnect facilitates efficient data transmission and coordination between these essential components. Early generations of Intel server-grade processors (Intel Xeon processors) use a ring interconnect (often referred to as a ring bus), allowing data to circulate in a loop-like manner. Recent Intel Intel Xeon processors use a mesh interconnect which has a grid-like layout with multiple horizontal and vertical channels, enabling more direct on-chip communication.

As shown in Figure 25, with a mesh interconnect, the processor chip is structured as a 2D matrix of tiles; each tile can be either a *core tile* which consists of a core (and an LLC+directory slice), or a *controller tile* which consists of an integrated MC. Note that there are three types of CPU dies for Intel Xeon processors based on Skylake: LCC, HCC, and XCC, which represent low, high, and extreme core counts, respectively. The XCC die features 30 tiles (28 core tiles + 2 controller tiles), arranged in a 5×6 grid. However, some tiles might be intentionally disabled[1] by Intel. For example, our Xeon Gold 6142 processor which uses the XCC die, has 16 cores and 16 LLC slices, meaning 12 out of 28 core tiles are disabled (Figure 25).

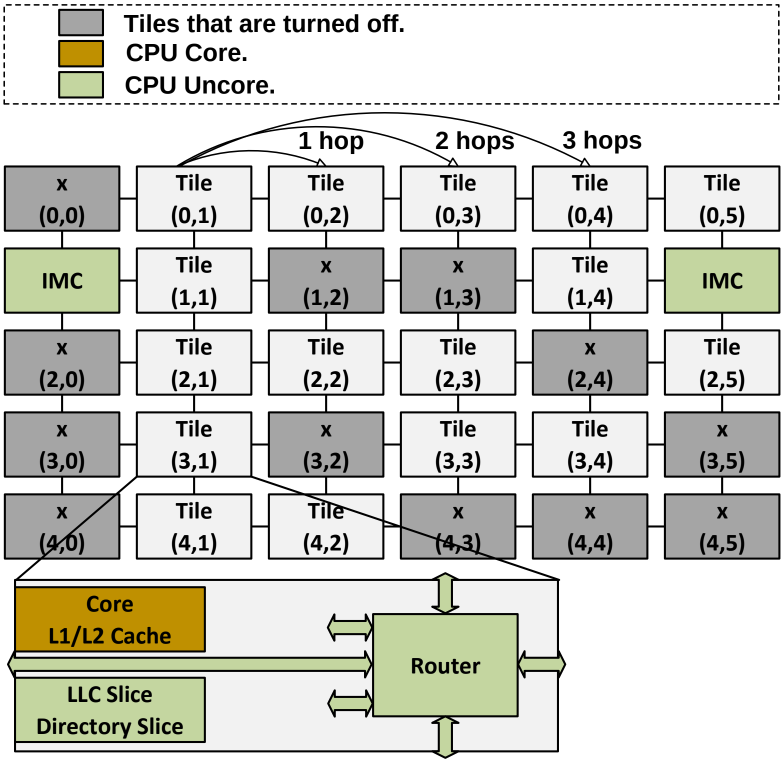


Figure 25: The architecture of our Intel Xeon Gold 6142 processor; the I/O controllers are omitted.

On an Intel Xeon processor, physical addresses are uniformly distributed to LLC slices using a slice hash function. This function is static, meaning a given physical address will always be mapped to a particular LLC slice in this processor. The specific hash function used in a processor is determined by the number of tiles in the processor. For example, all processors with 28 active core tiles use the same hash function, which has been reverse engineered [85]. Note that an unprivileged user, who cannot access the physical address of a given virtual address, may not directly know the LLC slice a virtual address is mapped

[1] The routers in the disabled tiles are still functional.

to. However, the user can infer this mapping indirectly using timing information, as access latencies (from a specific core) may vary across different LLC slices.

CPU uncore. According to Intel's terminology, a multi-core processor is composed of several cores and an uncore. A core is a logically independent computing unit with ALUs, FPUs, registers, and private caches. The uncore, on the other hand, comprises the components that are not part of the individual cores but are essential for the overall functionality and performance of the processor. Typically, the uncore includes the LLC, on-chip interconnect, and other components (e.g., MCs). The uncore is shared by all the cores on the processor.

5.3 CPU Power Management

Reducing the power consumption of processors (especially server processors) has become increasingly vital these days. As a result, Intel has integrated many power efficiency features into its processors. In this section, we only focus on the features that are related to our design.

5.3.1 CPU Frequency Scaling

Core frequency scaling. Intel processors use the common power saving approach, Dynamic Voltage and Frequency Scaling (DVFS) [4], to adjust the frequencies of the cores, based on the workloads. This frequency adjustment works at the granularity of P-states. Each P-state corresponds to a different operating point of the core, in 100 MHz frequency increments. Recent Intel processors offer two mechanisms for P-state selection, namely SpeedStep and SpeedShift. With SpeedStep, the operating system (OS) is responsible for controlling and selecting P-states. In contrast, when SpeedShift is enabled, the P-state selection is controlled by the hardware rather than the OS. However, the OS can give hints to the hardware, such as restricting the range of allowed P-states.

Uncore frequency scaling. Early Intel processors use either a fixed uncore frequency (e.g., for Nehalem and Westmere) or a common frequency for both cores and the uncore (e.g., for Sandy Bridge and Ivy Bridge). Since Haswell, the uncore frequency can be set independently of the core frequencies, and UFS was introduced on Intel Xeon processors to dynamically control the frequency of the uncore, based on the needs for the uncore [60]: UFS increments, decrements, or leaves unchanged uncore frequency based on whether the uncore is under stress, under-utilized, or stable, respectively. This ensures that the uncore components can deliver optimal performance when required, while conserving energy during periods of reduced activity or demand. Unlike DVFS (for cores), the selection of uncore frequency is always managed by hardware using built-in power management algorithms. However, the OS can restrain the uncore frequency selection, through model specific registers (MSRs). Specifically, the OS can specify the maximum and minimum uncore frequencies by writing to `UNCORE_RATIO_LIMIT`, as shown in Figure 26, and the hardware will only adjust the uncore frequency within this range. On Intel Xeon Scalable Processors, the default minimum frequency is 1.2 GHz, and the maximum frequency is 2.4 GHz.

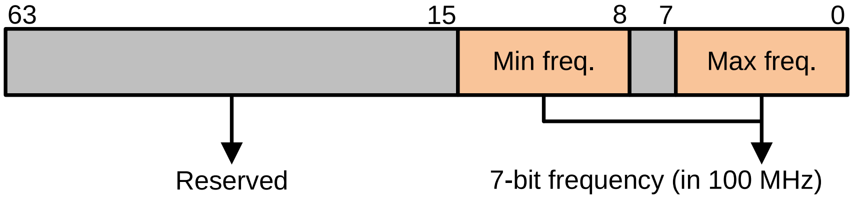


Figure 26: The layout of the uncore freq. limitation register.

The specifics of UFS on Intel processors are undocumented. According to our experiments (and prior studies such as [60]), in general the uncore frequency is dynamically adjusted only when all the active cores are running at a frequency lower than (or equal to) the base frequency[2] (i.e., UFS is enabled). When at least one core is running at a higher frequency, the uncore consistently stays at the maximum frequency specified in `UNCORE_RATIO_LIMIT` (i.e., UFS is disabled). Note that UFS is also disabled if the OS sets the minimum and maximum uncore frequencies to be the same.

[2] The conditions may vary depending on the processor model.

5.3.2 CPU Idle Power Management

Computing tasks often involve idle periods, during which the processor core enters a low-power state to save energy. Modern processors support multiple core power states, known as C-states [3]. A specific C-state is denoted as Cn, where n is the index. $C0$ is the normal operating state where the core is 100% active, while other states (C1 to Cn) represent idle states (also called sleep states) where the core is inactive and some components of the core are powered down. A deeper C-state indicates more powered-down components and better power efficiency; however, it also means that it takes more time for the core to become fully active (i.e., longer *exit latency*). The OS primarily manages the C-state selection. Typically, it chooses a C-state based on the intensity of the workloads running on the core. If the workloads are intense, the core is more likely to stay in a shallow C-state during idle periods; otherwise, it stays in a deeper C-state.

When all the cores of a processor are idle, the uncore is also (partially) turned off to further reduce the idle power consumption. Similar to C-states, modern processors support several package C-states (i.e., PC-states), which indicate the power state of the uncore [31]. Again, the uncore is fully active when it is in PC0, with deeper PC-state meaning more uncore components are turned off and there is a longer exit latency for the uncore. The selection of PC-state is usually driven by the selection of C-state. On Intel processors, the PC-state index is *no larger than* the smallest C-state index (among all the cores on the processor).

5.4 Prior Uncore Covert Channels

Over the last few years, researchers have proposed many covert channels exploiting the uncore components. We discuss these covert channels in this section.

Covert channels based on the LLC. In this type of covert channels, the sender intentionally modifies the LLC state to send the data; the receiver then checks the LLC state (e.g., through timing information) to receive the data [55, 134, 81, 71, 28, 93, 133]. For example, in Prime+Probe [81], the data is transmitted through set conflicts. To send a bit “1”, the sender loads its own data into an LLC set, evicting the receiver’s data in this set; to send a bit “0”, the sender does not load the data and the receiver’s data remains in this set. The receiver then determines the bit by checking whether its data is evicted from the LLC, based on the access latency of this data.

Covert channels based on the interconnect. As explained in Section 5.2, modern processors use interconnects for on-chip data transmission. Recent work [90, 122, 37, 43] discovered that LLC accesses from different cores may contend for interconnect bandwidth (for both ring and mesh interconnects), resulting in longer access latencies. Thus, the interconnect can be utilized for a covert channel: the sender sends a bit by generating LLC accesses that are transmitted through the interconnect (for “1”) or not (for “0”). At the same time, the receiver generates LLC accesses that contend with the sender’s accesses on the interconnect, and measures the LLC access latencies to receive the data.

Covert channels based on other uncore components. Some covert channels exploit contention in other uncore components, such as MCs, PMUs, and PCIe controllers [124, 49]. Although the specifics may vary across channels, the overall concept remains the same: contention for limited hardware resources can affect the access time to those resources, enabling covert communication.

Covert channels based on the idle power states. Since multiple PC-states exist, it is possible to encode information into these PC-states and form a covert channel. Specifically, as PC-states are driven by C-states, the sender can force the uncore to enter a certain PC-state by controlling the workload on a core (assuming there is no other active cores). The receiver can then infer the PC-state by examining the exit latency of the uncore (cf. Section 5.3.2). For example, this latency can be measured through a network interface card: the receiver records the timestamp when a packet arrives (T_1), and the timestamp when the interrupt service routine starts (T_2). Since serving this package requires waking up the uncore and a core, $T_2 - T_1$ is the sum of this core’s exit latency and the uncore’s exit latency. If the receiver is aware of the core’s C-state and its exit latency, the receiver can infer the uncore’s exit latency and thus the PC-state from $T_2 - T_1$. Compared to other uncore covert channels, this Uncore-idle channel is much less reliable, as it can only work in an “idle” environment. If there are other workloads running on the same processor, keeping at least one core fully active, the uncore stays in PC0, and this channel no longer functions.

As explained above, most previous uncore covert channels are based on uncore resource contention/conflict. This means that they are likely to be mitigated by partitioning uncore components among users. In this dissertation, we demonstrate that even with such partitioning mechanisms in place, whether it is partitioning spatially or temporally, the very fact that uncore is shared by all the tenants could lead to covert channels. Given that the entire uncore operates in a single frequency domain, a covert channel based on uncore frequency variation could be a good candidate. However, the reliability of the existing design, Uncore-idle [31], is quite low, as it requires an "idle" system. Therefore, in this dissertation, we investigate the feasibility of covert channels based on an alternative source of uncore frequency variation, specifically, UFS.

5.5 UFS Characterization

In order to exploit UFS for a practical covert channel, we must first answer a fundamental question: "how do Intel processors dynamically adjust the uncore frequency using UFS?". For example, we must understand which factors lead to uncore frequency changes. Thus, in this section, we study the details of UFS.

Experiment platform. Unless otherwise specified, all the experiments in this chapter are performed on a dual-socket system with two Intel Xeon processors. An overview of this system is given in Table 7. Figure 25 shows the architectural details of one of these processors. Note that the basic architectures of the two processors are the same; however, the tiles that are turned off are different (cf. Section 5.2). Figure 25 corresponds to Processor 0 on our platform; the details of Processor 1 are omitted due to the limited space. In this section, the uncore frequency is obtained by reading the MSR. Specifically, Intel provides the MSR, `U_PMON_UCLK_FIXED_CTR`, which is incremented by one at each tick of the uncore clock. Thus, by repeatedly reading this MSR, we can indirectly obtain the current uncore frequency.

Table 7: Platform details.

Processor	2× Intel Xeon Gold 6142
Microarchitecture	Skylake-SP
Num of cores	2×16
Core base frequency	2.6 GHz
UFS	1.2-2.4 GHz
L1 cache	8-way associative, private, 32KB+32KB
L2 cache	16-way associative, private, inclusive, 1024KB
LLC	11-way associative, shared, non-inclusive, 22528KB
Operating system	Ubuntu 22.04.1
Frequency driver	Intel_cpufreq
Frequency governor	powersave

5.5.1 UFS with LLC/Interconnect Utilization

The idea behind UFS is to adjust the uncore frequency based on the needs for uncore. Naturally, we then expect that the uncore frequency is higher when there is higher uncore utilization. In this section, we conduct experiments to verify this hypothesis.

Experiments. We study the uncore frequency under various uncore utilization levels, focusing on the utilization of the LLC and the interconnect. To control the uncore utilization level, we need to regulate the amount of LLC accesses and interconnect traffic. To achieve this, we use a group of threads to generate *LLC accesses* and pin each thread to a different core on the same processor. All the accesses from the same thread target the same LLC slice, while accesses from different threads target different LLC slices. Then, we can manipulate the uncore utilization by varying two parameters: 1) the total number of threads and 2) the on-chip distance (hops, cf. Figure 25) between the CPU core and the target LLC slice for each thread. The first parameter mainly affects the LLC utilization, with more threads indicating higher LLC access density. The second parameter mainly affects the interconnect utilization, with longer core-to-LLC distance indicating more interconnect traffic.

```
/* n is the number of eviction lists. */
/* m is the number of addresses in each list. */
/* EV_lists[n][m] is used to store all the eviction
   lists, EVs(0) to EVs(n-1). */
while ((i++) < Total_Rounds) {
    for(j = 0; j < m; j++)
        for(k = 0; k < n; k++)
            memaccess(EV_lists[k][j]);}
```

Listing 5.1: The loop used in each thread to creat LLC accesses (and traffic on the interconnect), referred to as the traffic loop.

Generate LLC accesses. To create LLC accesses, we must bypass the L2 cache (and L1). We achieve this using eviction lists: we define an eviction list, $EV_j(i)$, as a group of cache lines (addresses) that are mapped into the i^{th} L2 **set**, as well as the j^{th} LLC **slice**. Then, a thread that targets the s^{th} LLC slice operates as follows:

Step 1: Create n eviction lists, $EV_s(0)$ to $EV_s(n-1)$; each list contains m addresses.

Step 2: Repeatedly access the addresses in $EV_s(0)$ to $EV_s(n-1)$ and in each round, alternate the accesses to addresses in different eviction lists, as shown in Listing 5.1. With a proper m and n, all these accesses are likely to miss in the L2 cache and hit in the s^{th} LLC slice (explained below).

Let W_{L2} and W_{LLC} be the associativities of the L2 cache and the LLC, respectively. Then, to ensure that all the accesses in Step 2 are likely to be served by the LLC, m should be large enough (e.g., larger than W_{L2}) to avoid L2 hits, and small enough (e.g., smaller than $W_{L2}+W_{LLC}$) to avoid LLC misses. For our processor where $W_{L2} = 16$ and $W_{LLC} = 11$, we use 20 cache lines in each eviction list, i.e., $m = 20$. In addition, to guarantee L2 misses, the m addresses in the same L2 set (same eviction list) must always be accessed in a fixed order (assuming the LRU policy). To guarantee this order, we use multiple L2 sets (multiple

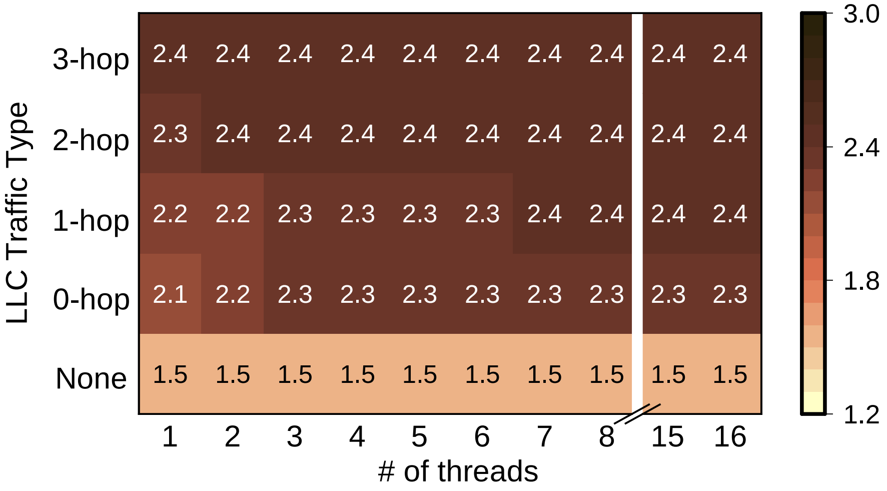


Figure 27: The median uncore frequencies (in GHz) with different thread counts and LLC access types.

eviction lists) and alternate the accesses to different sets: in this case the accesses to the same L2 set are separated (by accesses to other L2 sets) in the program order, and are thus unlikely to be close enough to get reordered by hardware. Here we use 64 L2 sets (64 eviction lists), i.e., $n = 64$.

Results. We measure the uncore frequency with varying thread counts and core-to-LLC distances. Our results show that when we first launch the thread(s), the uncore frequency adjusts accordingly and eventually stabilizes at a certain level (since each thread executes a loop). The stabilized frequencies are given in Figure 27. First of all, for a given core-to-LLC distance (for all the threads), executing more threads results in higher uncore frequencies. For example, when all the threads are accessing their local LLC slices (i.e., 0-hop traffic), the uncore frequency increases from 2.1 GHz to 2.3 GHz if the thread count increases from 1 to 16. Likewise, given a specific thread count, the uncore frequency is higher if the threads are accessing further LLC slices. In addition, when accessing LLC slices that are 3 hops away from the core, the uncore frequency reaches 2.4 GHz (i.e., the maximum uncore frequency, cf. Table 7), even with just one thread running.

For reference, in Figure 27 we also show the uncore frequency with only L2 accesses and no LLC access. In this scenario, the uncore does not stay at a certain frequency; instead, it alternates between 1.4 GHz and 1.5 GHz. *For simplicity, we refer to this situation as "staying at 1.5 GHz" in the rest of this chapter.*

These results confirm that the uncore frequency changes based on the uncore utilization, including both the LLC and the interconnect utilization. Higher utilization results in higher uncore frequency. Without any traffic on the interconnect, the frequency can only go up to 2.3 GHz; in contrast, it can go up to the maximum uncore frequency (2.4 GHz) with interconnect traffic.

5.5.2 UFS with Core Stalling

We conducted several supplementary experiments and studies, in addition to those presented in Section 5.5.1, to determine if other factors influence uncore frequency changes. It is shown that the uncore frequency is also related to the number of cores that are stalled due to waiting for load or store operations.[3]

Experiments. We use a similar approach to the one in Section 5.5.1. We again launch a group of threads accessing the LLC slices; however, instead of accessing each address independently, we access them through pointer chasing. That is, the data at a pointer address dictates the subsequent pointer address. This ensures that the subsequent load cannot be executed until the current load is completed, i.e., the CPU core is stalled due to waiting for a load to finish. The example code is shown in Listing 5.2; here we only use one eviction list since the access order is already guaranteed by pointer-chasing.

Results. Our experiments show that with pointer chasing, the uncore frequency always stabilizes at 2.4 GHz, regardless of the thread count and the core-to-LLC distance. This means, the uncore frequency reaches 2.4 GHz even when running just one thread accessing the local LLC slice. Recall that without pointer chasing (as in Section 5.5.1), with this setup the uncore frequency is only 2.1 GHz. To better understand this difference, we use Linux perf tools to profile the pointer-chasing threads and gather data from two counters:

[3] This aligns with the design in Intel's patent [25].

```
/* EV_list[m] contains all the addrs in the eviction list. */
/* *EV_list[i] = EV_list[i+1], with i in [0, m−2]. */
/* *EV_list[m−1] = EV_list[0]. */
current_addr = EV_list[0];
while ((i++) < Total_Rounds) {
    current_addr = *(current_addr);}
```

Listing 5.2: The loop used in each thread to stall the core, referred to as the stalling loop.

1) `cycle_activity.stalls_mem_any`, which represents the total time that the execution is stalled due to an outstanding memory operation, and 2) `cycles`, which is the total execution time. The results show that, the ratio of these two data is approximately **0.77** for each pointer-chasing thread. For comparison, this ratio is only about **0.3** for the traffic threads used in Section 5.5.1. It is notable that if the pointer chasing happens within L2 (no uncore activity), the stalling ratio is **0.14**, and uncore will not boost its frequency. Thus, we hypothesize that the uncore frequency increases (to the maximum uncore frequency) when the stalling time within a given time period for one or more cores surpasses a certain threshold. In the rest of this chapter, we use the term "a core is stalled" to indicate that "the stalling time of a core in almost every time period is above this threshold".

In the above experiments, all the threads running on the processor are the pointer-chasing threads. As a result, all the cores that are active are stalled. We found that, the uncore frequency may not reach 2.4 GHz when only some of the active cores are stalled, and others are not. We test this by launching some threads that do not stall the CPU cores alongside the pointer-chasing threads. As shown in Figure 28, when two active cores are stalled, if there are four (or more) other active cores which are not stalled, the uncore frequency stabilizes at 1.8 or 1.5 GHz, rather than 2.4 GHz. Similarly, when three cores are stalled and six (or more) cores are active but not stalled, the uncore frequency is 1.8 or 1.5 GHz. These observations indicate that the uncore frequency is indeed influenced by the proportion of the active cores that are stalled; the uncore frequency only rises to 2.4 GHz if more than **1/3** active cores are stalled.

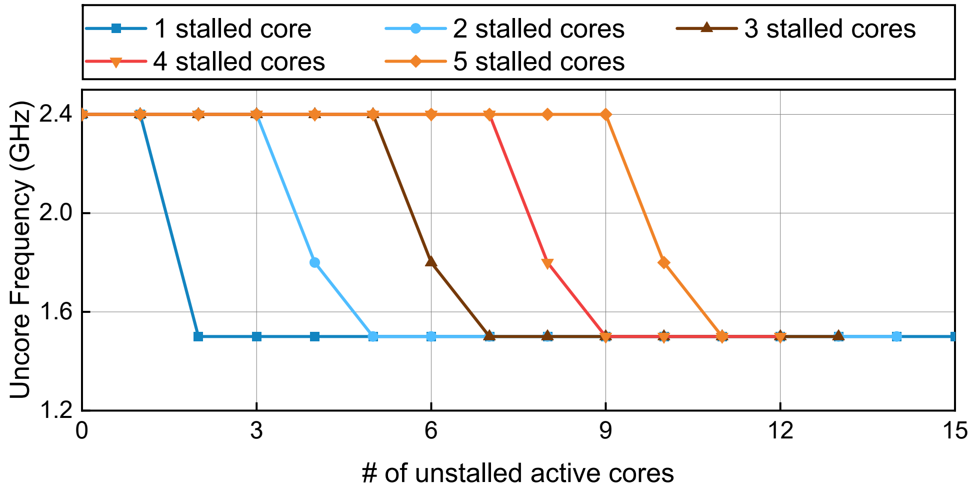


Figure 28: The uncore frequencies based on the number of stalled cores and active but not stalled cores.

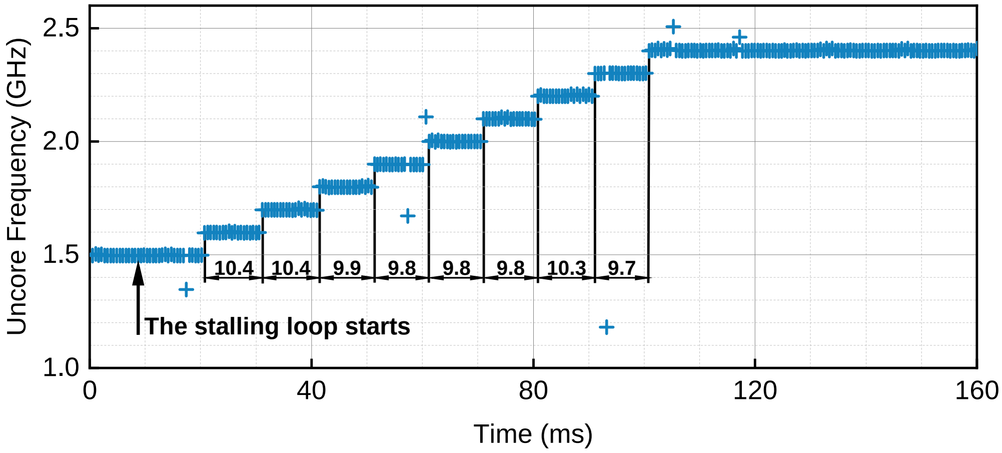


Figure 29: Uncore frequency trace upon initiating the stalling loop.

5.5.3 UFS Granularity

In the previous experiments, we focused on the stabilized uncore frequencies under specific workloads. However, for building a covert channel with UFS, it is also important to

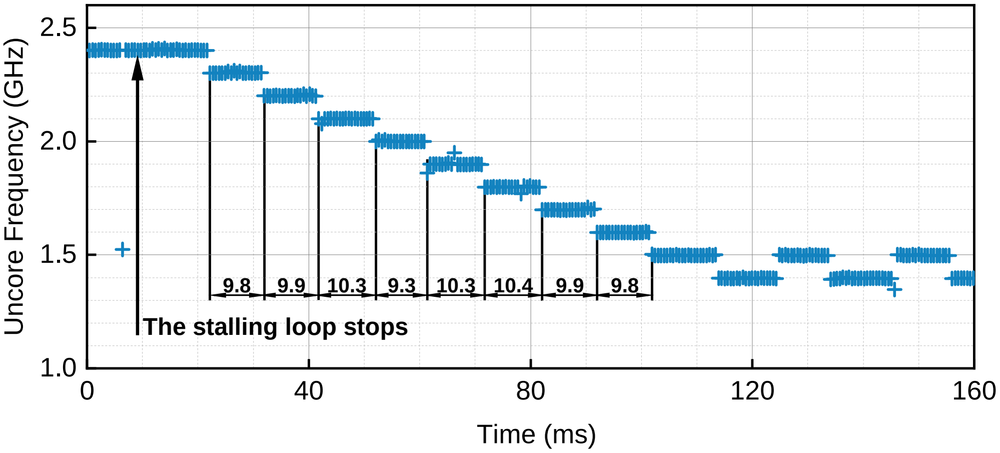


Figure 30: Uncore frequency trace upon stopping the stalling loop.

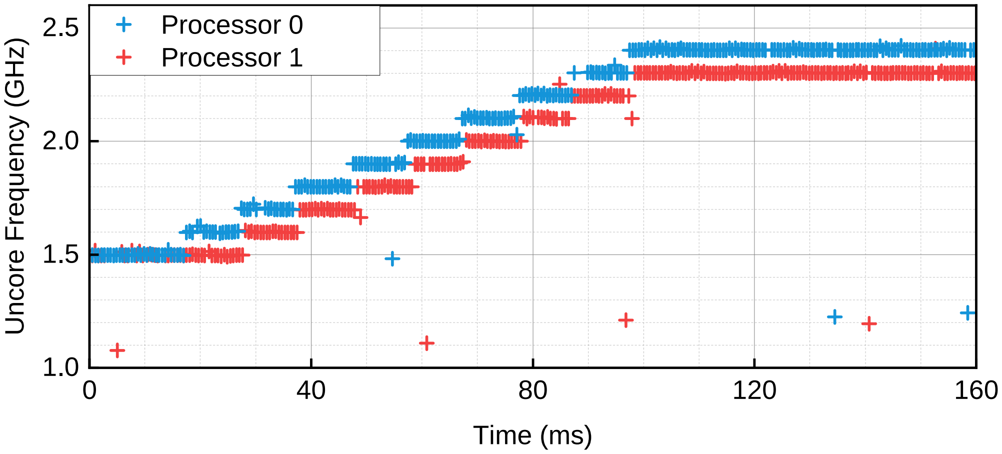


Figure 31: Uncore frequency trace upon initiating the stalling loop on Proc. 1.

understand the details of the frequency adjustment period (before stabilization). For instance, we need to know the total time it takes for the frequency to rise from 1.5 GHz to 2.4 GHz after a core becomes stalled. In this section, we investigate this aspect.

Frequency increase. We launch a thread that first runs a `nop` loop, and then switches to a stalling loop (cf. Listing 5.2). We record the uncore frequency trace while the thread is running, collecting the uncore frequency every 200 μs. The result is shown in Figure 29. Before the stalling loop starts, the uncore frequency is 1.5 GHz. Once the stalling loop starts, the uncore frequency increases by 100 MHz approximately every 10 ms; after the frequency reaches 2.4 GHz, it stabilizes. In addition, we tried launching multiple such threads together and letting each thread access a further LLC slice, but neither of these options can make the uncore frequency increase faster, i.e., it still only changes every 10 ms. Similar results apply when using a traffic loop (cf. Listing 5.1) instead of the stalling loop. Thus, we believe that the frequency control hardware checks the system status approximately every 10 ms and decides whether and how to update the uncore frequency. Additionally, similar to the P-states for cores, the uncore also has different operating points in 100 MHz frequency increments. Moreover, it takes slightly longer than 10 ms to change from 1.5 GHz to 1.6 GHz. We believe this is because the starting time of the stalling loop is not aligned with the frequency update periods.

Frequency decrease. We use a similar method to measure how the uncore frequency decreases. Specifically, we launch a thread which first runs a stalling loop and then switches to a `nop` loop. The recorded uncore frequency trace is shown in Figure 30: once the stalling loop stops, the frequency decreases by 100 MHz every 10 ms, until it reaches 1.5 GHz (and starts to fluctuate around 1.5 GHz). Again, similar results apply when using a traffic loop.

5.5.4 UFS across Processors

After analyzing UFS within a processor, now we study how UFS works across processors (sockets). Figure 31 shows the uncore frequency traces for both processors when starting a stalling loop on a core of Processor 0. As discussed earlier, the uncore frequency of Processor 0 increases after the loop starts. Interestingly, the uncore frequency of Processor 1 also increases, even though there is nothing running on Processor 1 that can trigger this increment. In addition, the frequency increment on Processor 1 starts about 10 ms later than the increment on Processor 0. Thus, during the frequency adjustment period, the uncore frequency of Processor 1 is always 100 MHz less than the uncore frequency of Processor 0. Eventually, the uncore frequency of Processor 1 stabilizes at 2.3 GHz instead of 2.4 GHz.

We perform further tests where we run different workloads on Processor 0 to make its uncore frequency stay at different levels (e.g., 2.1 GHz). Then we examine the uncore frequency of Processor 1. It turns out that the uncore frequency adjustment on Processor 1 always starts later than the one on Processor 0. In addition, its stabilized frequency is always the same or slightly lower than the one of Processor 0.

5.5.5 Summary of UFS Behavior

The UFS behavior discussed in this section can be summarized as follows:

- The uncore has different operating points in 100 MHz frequency increments. The system status is checked about every 10 ms to decide whether to increase, decrease, or maintain the uncore frequency.
- The uncore frequency is influenced by the uncore utilization, higher utilization leads to higher frequency (within the allowed frequency range).
- The uncore frequency is also affected by the proportion of active cores stalled due to cache/memory accesses; the uncore remains at the maximum frequency when more than 1/3 of the active cores on the processor are stalled.
- The uncore frequencies of processors (in the same system) are correlated: when the uncore frequency of a processor increases, the ones of other processors also increase.

Note that this summary does not represent the complete design of UFS. There might be other factors that can affect the uncore frequency. Our goal is to utilize UFS to build a covert channel, instead of uncovering every detail about UFS.

5.6 UFS-Based Covert Channel

We use the findings in Section 5.5 to build the first covert channel based on UFS. The basic idea of the sender is to transmit information by manipulating the uncore frequency (through controlling the workload being executed). Simultaneously, the receiver obtains the information by monitoring the uncore frequency. In this section, we first introduce the threat model, and then provide an in-depth discussion of this channel's details.

5.6.1 Threat Model

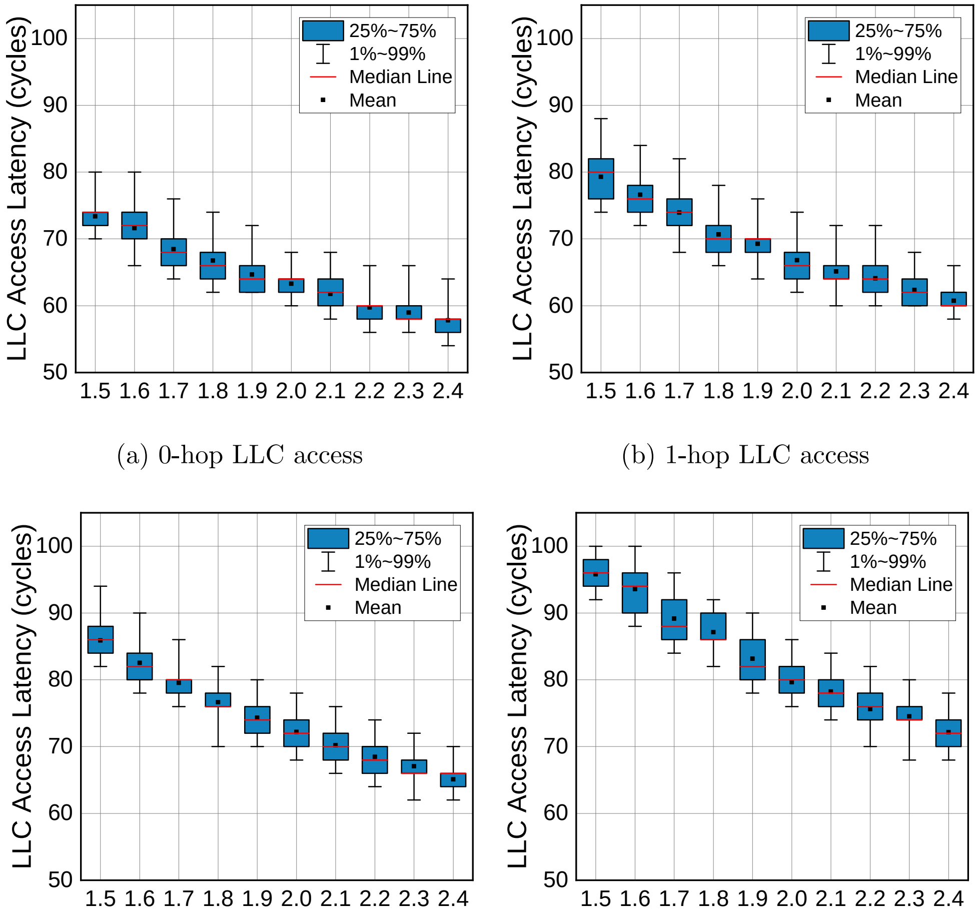
Like all other covert channels, the UFS-based covert channel involves two parties: the sender and the receiver. We assume that the sender and the receiver are two unprivileged processes or virtual machines that are either 1) running on the same processor (but different cores) or 2) running on the same computing system (but different processors). We also assume the processors in the system are Intel processors that dynamically adjust their uncore frequency using UFS. In addition, the sender and the receiver agree on pre-defined channel protocols, such as the synchronization protocol.

Apart from the above, no additional assumptions are made on the sender or the receiver. For example, we do not assume memory sharing techniques (e.g., page deduplication [19]) or HugePages [2] which are required in many uncore covert channels (e.g., [81, 134, 132, 67]). We also do not require memory allocations/accesses across non-uniform memory access (NUMA) domains, unlike some prior cross-processor covert channels [122].

5.6.2 Measuring Uncore Frequency

The receiver in the UFS-based covert channel needs to monitor the uncore frequency, in order to receive information. In Section 5.5, we obtain the uncore frequency by reading the MSR. However, accessing MSRs is generally only allowed for privileged users. Since we do not require the receiver to have privileged permission, we need to find a different and more accessible method to probe the uncore frequency.

Our insight is that, the uncore frequency can be obtained indirectly, by measuring the access latencies to uncore components (e.g., the LLC). Intuitively, a lower uncore frequency means that the uncore components are working at a lower speed, resulting in slower accesses to those components (and vice versa). Figure 32 shows the LLC access latencies at various uncore frequencies. We force the uncore to operate at a certain frequency by setting the minimum and maximum uncore frequencies to be the same (cf. Figure 26). It is shown that,



(a) 0-hop LLC access

(b) 1-hop LLC access

(c) 2-cop LLC access

(d) 3-hop LLC access

Figure 32: The LLC access latencies under different uncore frequencies; the latencies are measured all on core (3,3). 0-hop latencies, 1-hop latencies, 2-hop latencies, and 3-hop latencies are collected when accessing LLC slice (3,3), LLC slice (2,3), LLC slice (2,2), and LLC slice (2,1), respectively. The latencies are collected in a 10 ms window.

for a given LLC slice, the average access latency decreases as the uncore frequency increases. Consequently, the receiver can use the LLC access latency to accurately determine the uncore frequency.

Measurement noise. Listing 5.3 shows the code snippet of the measurement loop in the receiver: it sequentially accesses every cache line in the eviction list and times the access.

```
/* EV_list[m] contains the addrs the evic. list. */
while ((i++) < Total_Measure) {
    for(j = 0; j < m; j++) {
        mfence();
        lfence();
        t1 = rdtscp();
        memaccess(EV_list[j]);
        t2 = rdtscp();
        access_latency[i*m+j] = t2-t1;}}
```

Listing 5.3: The measurement loop in the receiver.

All of these accesses should hit in the LLC. Since this measurement loop creates a lot of LLC accesses, it is essential to know how this loop affects the uncore frequency. If running this loop makes the uncore constantly stay at a very high frequency, it will be difficult or even impossible for the sender to manipulate the uncore frequency (to send data). In fact, we found that when only running this loop, the uncore frequency stays low (at 1.5 GHz). This is because the memory fences used in the loop keep the LLC access density relatively low.

5.6.3 UF-variation

In this section we explain the covert channel in details. For generality, in the rest of this chapter we use `freq_max` to represent the maximum uncore frequency (2.4 GHz on our processor), and use `freq_min` to represent the minimum active uncore frequency (1.5 GHz on our processor).

5.6.3.1 Channel Protocol

Intuitively, with UFS, the sender can encode the data into the uncore frequency values: to send different data, the sender creates different amounts of LLC traffic (cf. Figure 27) or different levels of core stalling to make the uncore frequency stay at different levels. However, we do not use this approach because it results in a very long transmission interval and thus a limited transmission rate. In Figure 29, the uncore frequency increases by 100 MHz every time when the hardware checks the system status (every 10 ms), during the frequency adjustment period. We found that this only happens when we apply heavy LLC traffic or have severely stalled cores (where the stabilized frequency is `freq_max`). In contrast, with lighter LLC traffic or less severely stalled cores where the stabilized frequency is lower than `freq_max`, the uncore frequency is not increased in every 10 ms during the adjustment period. As a result, it takes much longer for the uncore frequency to adjust. For example, when launching one thread accessing the local LLC slice (where the stabilized frequency is 2.1 GHz, cf. Figure 27), it takes over 50 ms for the uncore frequency to even change from 1.5 GHz to 1.6 GHz.

Algorithm 6: The UF-variation Covert Channel

```
Input: $T_{freq\_max}$: the LLC latency at freq_max.
Input: $T_{freq\_min}$: the LLC latency at freq_min.
Input: message[n]: the n-bit message to be transmitted.
Sender:
    // Algorithm steps for the sender
    for $i = 0; i < n; i++$ do
        sync_channel();
        if message[i] $== 1$ then
            stalling_loop(); // Or a heavy LLC traffic loop
Receiver :
    // Algorithm steps for the receiver
    for $i = 0; i < n; i++$ do
        sync_channel();
        $T_1$ = measure_avg_LLC_latency();
        wait();
        $T_2$ = measure_avg_LLC_latency();
        if $T_2 < T_1$ or $T_1 = T_2 = T_{freq\_max}$ then
            Received a bit "1";
        if $T_2 > T_1$ or $T_1 = T_2 = T_{freq\_min}$ then
            Received a bit "0";
```

Thus, in our covert channel, we only use heavy LLC traffic or severely stalled cores (to control the uncore frequency). This ensures that the frequency changes frequently (every 10 ms), which allows a shorter transmission interval and thus a higher transmission rate.

The covert channel we propose, named UF-variation, encodes data into the uncore frequency variation. The channel protocol is shown in Algorithm 6. The sender uses the stalling loop (Listing 5.2) which can severely stall the core to control the uncore frequency.[4] 1-bit of data is transmitted in each transmission interval. To send a bit “1”, the sender executes this loop and the uncore frequency increases every 10 ms (unless it’s already at `freq_max`). To send “0”, the sender does not execute the loop and the frequency decreases every 10 ms (unless it’s already at `freq_min`). On the other hand, the receiver monitors the LLC access latencies, and compares the average latency near the beginning of the interval (`T1`) and the average latency near the end of the interval (`T2`). If 1) `T2` $<$ `T1` or 2) both `T1` and `T2` match the latency at `freq_max`, it means the uncore frequency is increasing or staying at `freq_max` in this interval. Thus, the receiver receives a bit “1”. Otherwise, if 1) `T2` $>$ `T1` or 2) both `T1` and `T2` match the latency at `freq_min`, it means the uncore frequency is decreasing or staying at `freq_min`, and the receiver gets a bit “0”.

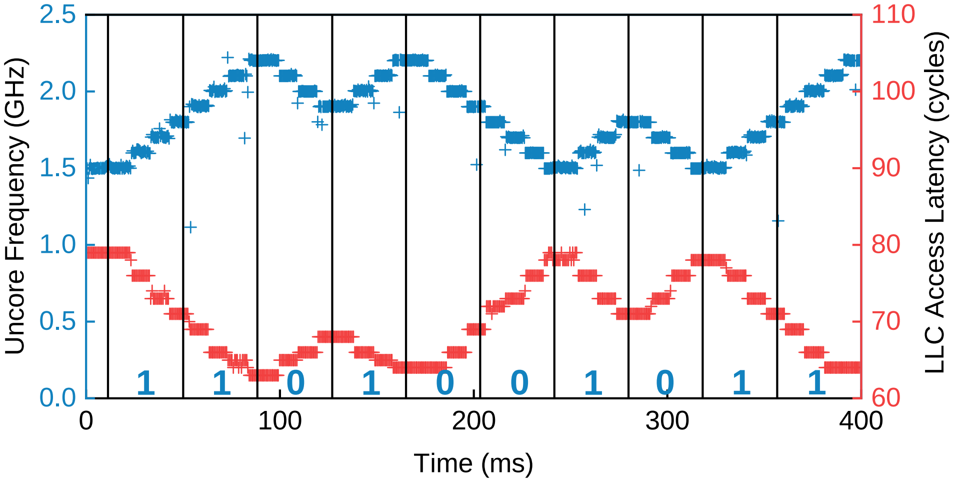


Figure 33: The LLC access latency trace and the corresponding uncore frequency trace when sending “1101001011” through the channel. The transmission interval is 38 ms. The LLC access latencies are 1-hop latencies.

[4] The sender can also use a heavy traffic loop (cf. Listing 5.1) instead of a stalling loop.

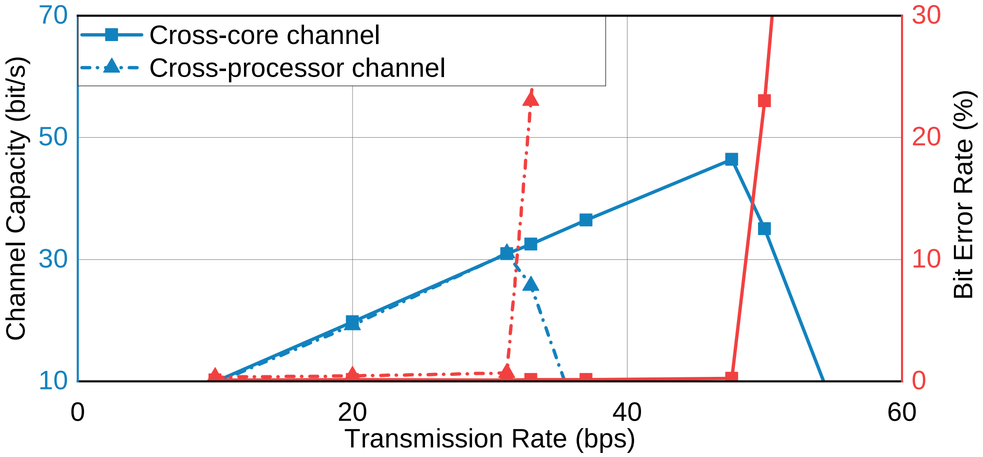


Figure 34: The channel capacities and error rates of UF-variation, in the cross-core and cross-processor scenarios, respectively.

In this channel, the transmission interval should be long enough for the frequency to change (i.e., at least 10 ms). Figure 33 provides an example of sending “1101001011” through this channel. In the first interval, the sender sends “1” by executing the stalling loop, the frequency increases from 1.5 to 1.8 GHz and the LLC latency decreases from 79 to 71 cycles. Then in the second interval, the sender continues the stalling loop to send “1”, the frequency continues to increase from 1.8 to 2.2 GHz and the LLC latency further decreases from 71 to 63 cycles. In the third interval, the sender sends “0” and stops the stalling loop, the frequency thus decreases from 2.2 to 1.9 GHz and the LLC latency increases from 63 to 68 cycles.

5.6.3.2 Channel Capacity

In this section, we evaluate the throughput of UF-variation as a cross-core covert channel and a cross-processor covert channel, respectively.

Configuration. We create a proof-of-concept implementation of UF-variation, where the sender and the receiver are single-threaded processes that synchronize using time stamp counters. The receiver calculates the average LLC latencies for the first and last 5 ms in an interval and compares them. We use the metric *channel capacity* (as in [31]) to quantify the throughput performance. It can be calculated by multiplying the raw transmission rate with $(1 - H(e))$, where e represents the bit error rate and H denotes the binary entropy function.

Figure 34 shows the channel capacities and bit error rates of UF-variation under different raw transmission rates (i.e., different transmission intervals). When the transmission rate is low (e.g., below 47 bit/s for the cross-core channel), the error rate is very low and remains almost constant. Thus, the channel capacity increases proportionally to the transmission rate. When the transmission rate is higher (i.e., intervals are smaller), the error rate starts to increase which causes a decrease in the channel capacity. In the cross-core scenario, the channel capacity peaks at 46 bit/s given a transmission rate of 47.6 bit/s (interval of 21 ms). In the cross-processor scenario, the capacity peaks at 31 bit/s given a transmission rate of 33 bit/s (interval of 33 ms). Although the channel capacity of UF-variation is lower than many prior uncore covert channels, it is effective under a wider range of situations than prior channels. We discuss the details of this later in Section 5.6.4.

5.6.3.3 Channel Reliability

Like other uncore covert channels, UF-variation can be affected by noise from other processes running on this system. There are mainly two categories of noise that can influence UF-variation. First, the execution of other processes may affect the proportion of the active cores that are stalled, and thus affect the uncore frequency. For example, in Algorithm 6, the sender only launches one thread and uses the stalling loop to control the uncore frequency. This works well when only the sender and receiver are using the processor: when the sender sends a “1”, 1/2 of the active cores are stalled, causing the uncore frequency to rise. However, if there are two threads from other processes running on this processor (which do not stall the cores), only 1/4 active cores are stalled when the sender sends a “1”. Consequently, the uncore frequency does not increase, and the receiver cannot differentiate between “1” and “0”. Nevertheless, this type of noise can always be avoided by using the traffic loop to control the uncore frequency instead. Moreover, if the sender can access multiple cores, this issue can be resolved by stalling multiple cores simultaneously. For example, on a 16-core processor, if the sender stalls 6 cores, then it is guaranteed that over 1/3 active cores are stalled when the sender sends a "1".

Second, other processes with heavy LLC utilization or stalling loops may keep the uncore frequency high even when the sender sends a "0". Here we test the channel capacity of UF-variation while running `stress-ng |cache N` to stress the CPU cache in the background (using `N` threads), similar to prior work [37, 90, 59]. The results are shown in Table 8. The channel is affected by the phases where `stress-ng` keeps the uncore frequency at `freq_max`. The channel capacity is lower when those phases appear more often or last longer. As shown in the table, UF-variation can tolerate the cache stressing when `N < 9`. When `N` is higher, the error rate becomes excessive and the covert channel is no longer functional. We compare the reliability of UF-variation to the reliabilities of other covert channels in Section 5.6.4.

Table 8: The maximum channel capacities of UF-variation (as a cross-core channel) with the stress-ng tool.

stress-ng -N	N=1	N=2	N=3	N=4	N=5	N=6	N=7	N=8	N=9
Capacity (bit/s)	8.6	7.2	6.8	5.1	4.4	3.0	2.4	0.2	0

5.6.4 Comparison of Uncore Covert Channels

Table 9 compares UF-variation to the existing uncore covert channel techniques based on prerequisites, robustness against defenses, and reliability.

Prerequisites. Covert channels typically have some requirements for the system setup. The most fundamental requirement for an uncore covert channel is co-location, i.e., the sender and receiver are able to run simultaneously on the same system. In addition to this basic prerequisite, some uncore covert channels have further requirements. Common additional requirements include memory sharing, the presence of the `clflush` instruction (or similar instructions), and transactional memory techniques (e.g., Intel TSX). Although these requirements facilitate powerful covert channels in terms of speed and reliability, they also significantly restrict the channel's applicability. For example, memory sharing is discouraged in cloud environments, and special instructions like `clflush` may not be accessible to users in non-native environments (e.g., within a browser).

Effectiveness under defenses. In recent years, numerous defense approaches against uncore covert channels have been proposed. It is expected that real processors will soon implement one or more of these defenses. Thus, it is important to know whether a covert channel remains functional under a specific defense mechanism. The main lines of defense designs are based on randomization and isolation. First, by randomizing the address-to-set mapping in the LLC (e.g., [52]), it becomes challenging to force or observe LLC set conflicts, which are essential for many LLC covert channels. Arguably, a more promising method is partitioning uncore components among users (e.g., [22, 42, 113, 125]), since most uncore covert channels are based on uncore resource contention/conflict. We discuss two types of partitioning mechanisms here.

The first one is a coarse-grained partitioning approach where the sender and receiver are on different processors in the system and the NUMA-strict policy is enforced, i.e., memory allocations/accesses across NUMA domains are not allowed. The second one is a more fine-grained partitioning mechanism, where the sender and receiver can run on the same processor but in different security domains. In this case, all the uncore buffering structures such as LLC slices and queues in the MCs are partitioned among domains: for example, with two domains, each domain is assigned with half of the LLC slices (8 on our processor). Additionally, all the communication paths such as the interconnect work with a time-multiplexed scheduling policy so that traffic from different domains is partitioned and served in different time periods [127], avoiding contention.[5]

Reliability. We evaluate the reliabilities of the channels by examining their functionalities while running `stress-ng -- cache 4` in the background, i.e., whether the receiver can still distinguish between "1" and "0". Note that we use four stressing threads here because this results in a processor load of 37.5%, which is close to the processor load observed in modern data centers [51].

[5] We do not assume a spatial partitioning design like Intel Sub-NUMA Clustering [5] for preventing interconnect contention, since the traffic to peripheral devices from different domains may still contend in this scenario.

Table 9: The comparison of uncore covert channels; ✓ means the channel is functional while ✗ means it is not.

		Prerequisites			**Defenses**			**Reliability**
Attack technique	**Leakage source**	No shared mem.	No `clflush`	No TSX	Rand. LLC	Fine partition	Coarse partition	`stress-ng -cache 4`
Flush+Reload [134]	Data reuse	✗	✗	✓	✓	✗	✗	✓
Flush+Flush [55]		✗	✗	✓	✓	✗	✗	✓
Reload+Refresh [28]		✗	✗	✓	✗	✗	✗	✓
Prime+Probe [81]	LLC set conflict	✓	✓	✓	✗	✗	✗	✓
Prime+Abort [41]		✓	✓	✗	✗	✗	✗	✓
SPP [120]		✓	✓	✓	✓	✗	✗	✓
Mesh-contention [122]	Interconnect contention	✓	✓	✓	✓	✗	✗	✓
Ring-contention [90]		✓	✓	✓	✓	✗	✗	✓
IccCoresCovert [61]	PMU contention	✓	✓	✓	✓	✓	✗	✓
Uncore-idle [31]	Idle power control	✓	✓	✓	✓	✓	✓	✗
UF-variation	**UFS**	✓	✓	✓	✓	✓	✓	✓

Comparison. As shown in Table 9, covert channels based on data reuse usually require memory sharing and special-purpose instructions, meaning they can be defended by simply disabling the required features. Covert channels based on LLC set conflicts typically do not have additional requirements; however, most of them can be mitigated by randomized LLC designs (other than SPP), and all of them can be prevented by uncore partitioning. Covert channels based on interconnect contention do not have additional requirements as well, but they also cannot work under either of the two partitioning designs. IccCoresCovert relies on the contention for the voltage regulator in the PMU. Thus, it cannot work under the coarse-grained partitioning where the PMU is no longer shared. In addition, all these above-mentioned covert channels remain functional with four cache stressing threads.

Uncore-idle (cf. Section 5.4) and UF-variation are the only two channels that cannot be stopped by any of the listed defenses. Unlike other uncore channels, these two channels are not based on hardware contention or conflict. Thus, randomizing the LLC mapping or partitioning uncore resources cannot prevent them. However, Uncore-idle which is based on the uncore *idle* power management, is highly susceptible to noise: as long as one core in the entire system is fully active, all the uncores (in all the processors) are active and the covert channel no longer exists. In contrast, UF-variation demonstrates better reliability.

5.7 Side Channel Attacks

In this section, we provide a preliminary study on exploiting UFS for side channel attacks. The two factors that affect the uncore frequency (uncore utilization and core stalling) can both be used to construct side channel attacks. Here we focus on the latter.

Attack methodology. As explained in Section 5.5.2, the uncore frequency is related to the proportion of active cores that are stalled. This proportion (and thus the frequency) may change depending on whether the victim's core(s) are active, reflecting the victim's core(s) utilization. Specifically, the attacker executes a stalling thread and a non-stalling thread. Then, when the victim's core(s) are inactive or minimally utilized, the uncore frequency stays at `freq_max`, since more than 1/3 of the active cores are stalled. However, if the victim's core(s) become active (but not stalled), the uncore frequency decreases because less than 1/3 active cores are stalled now. The core(s) activity may be related to some sensitive information of the victim, as shown in prior work [40]. Here we show two example attacks, file size profiling and website fingerprinting.

File size profiling. In this case, the victim is executing a python program to compress a file. The total execution time of the program is correlated with the size of the file. As a result, if the attacker determines the execution time of this program by monitoring the uncore frequency, the attacker can deduce the file size and it might even be able to further infer which specific file the victim is compressing. The attacker collects the uncore frequency every 3 ms. Some of the captured frequency traces are shown in Figure 35. When the victim is compressing a smaller file, the uncore frequency is at `freq_min` for a shorter period. The attacker can distinguish the file sizes at a granularity of 300KB with an accuracy of over 99%. Note that the attacker is running the helper threads (the stalling/non-stalling threads, as explained above) while collecting the trace.

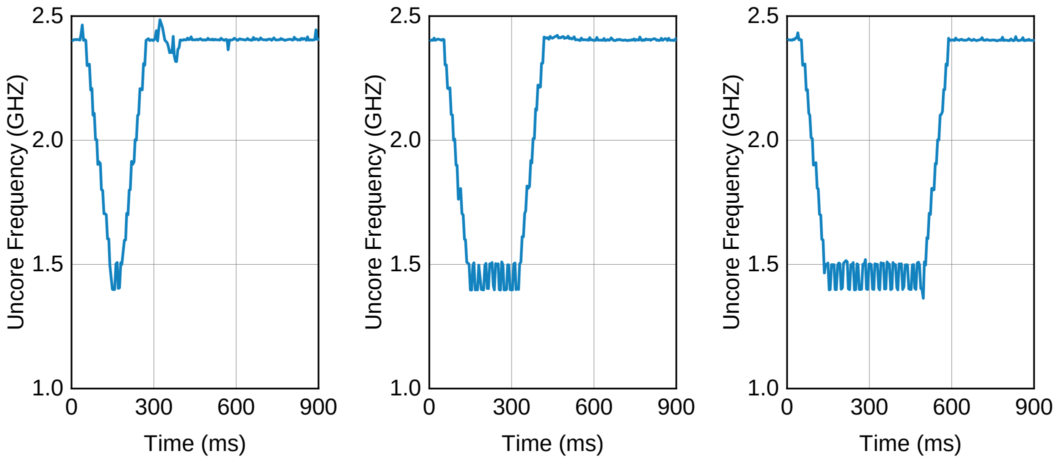


Figure 35: The uncore frequency traces captured while the victim compresses files with varying sizes.

Website fingerprinting. Prior work [40] has shown that core(s) utilization can be used for website fingerprinting. The victim in this attack is a user browsing webpages in a browser, and the attacker aims to determine the website the victim is accessing through the uncore frequency trace. Similar to previous fingerprinting-based attacks [122, 35, 40], we utilize machine learning to develop an attack with two phases: the training phase and the attack phase. In the training phase, the attacker collects uncore frequency traces for each of the top 100 websites according to Alexa [1] while accessing them. The attacker then uses these traces to train an RNN classifier. We use the same model and hyperparameters as [122]. In the attack phase, when the victim is accessing a website, the attacker collects the uncore frequency trace and feeds it to the classifier. In both phases, the attacker records the uncore frequency every 3 ms. Note that in both phases the attacker also needs to run the helper threads (the stalling/non-stalling threads), as explained in the attack methodology before.

Figure 36 shows examples of collected traces. The top-1 accuracy for website fingerprinting is 82.18%, while the top-5 accuracy is 91.48%. In addition to identifying the accessed website, the attacker can also learn how the website is used. For example, the attacker is able to differentiate between successful and unsuccessful login attempts on hotrcrp.com.

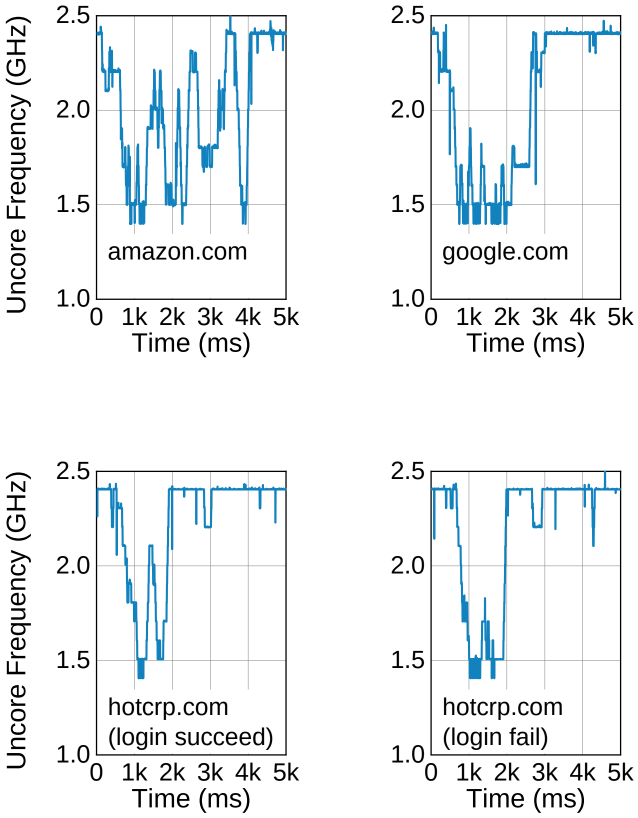


Figure 36: The uncore frequency traces captured while the victim is accessing varying domains.

5.8 Discussion

5.8.1 Evaluation with Background Workloads

Here we examine how running background cloud applications can impact UF-variation (and the side channels). We focus on three types of cloud applications as listed in Table 10. In-memory and graph analytics are derived from CloudSuite [48]. For web serving, we use the Apache server: we run an Apache HTTP server on the same machine with UF-variation, and run a client on a separate machine within the local network. The client repeatedly sends request to the server, similar to the setup in prior work [122].

Table 10: The channel capacities of UF-variation (as a cross-core channel) with background cloud applications.

Application	Web serving	In-mem analytics		Graph analytics	
		Small dataset	Large dataset	Workload PR	Workload CC
Channel capacity (bit/s)	44	40	31	6	2

The evaluation results for UF-variation are shown in Table 10. The web-serving workload has minimal impact on the channel capacity of UF-variation. In contrast, graph analytics workloads which have very high uncore utilization, significantly reduce the channel capacity. Compared to UF-variation, the UFS side channels are more vulnerable to background workloads, especially to the ones with high uncore utilization. For example, with graph analytics (for CC), UF-variation is still functional (although with a very limited capacity), but the accuracy of website fingerprinting is reduced to almost the level of random guessing, unless the attacker can take multiple samples during the attack.

5.8.2 Remote Covert Channel with UFS

Prior work [112] demonstrated a covert channel over the network: the sender attempts to send data through the network to the receiver located in the same physical network. The sender has the ability to send packets to the network, but the receiver is not permitted to use the networking stack (e.g., it is in a container). Using UFS, we can build a reversed form of this covert channel where the sender without network access can transmit bits to a remote receiver. Specifically, the sender manipulates the uncore frequency on the local machine. The receiver (on a different machine) collects the data by pinging the sender's machine and logging the response time. We found that this response time correlates with the uncore frequency (on the sender's machine): when the uncore frequency is at `freq_max`, the average response time is 0.8 ms less than when it's at `freq_min`.

5.8.3 Applicability to Non-Intel Processors

We examined the uncore frequencies of AMD processors and the results suggest that on those processors, the uncore frequency does not correlate with the uncore utilization or core stalling ratio. Thus, we cannot directly apply UF-variation on AMD processors. However, the insight in our study—that uncore frequency variations reflect core workload patterns—can inspire new attack methodologies. For example, we found that the uncore frequency of an AMD processor is tied to the highest core frequency among all the cores. Prior work [19,59] has shown that the core frequency can reveal information of the application running on the core (such as the data being processed). Thus, an adversary could potentially use the uncore frequency to infer the information of the application running on another core, resulting in a new form of uncore covert/side channel.

5.9 Chapter Summary

In this chapter, we presented the first covert channel based on UFS. We showed that the uncore utilization and the proportion of active cores that are stalled are two key factors affecting the uncore frequency. Based on this observation, we developed a new covert channel, UF-variation, with a channel capacity of 46 bit/s. We further showed that although UF-variation has lower capacities than previous uncore covert channels, it remains functional even with uncore partitioning in place, while previous channels do not. We demonstrated that it is possible to build side channel attacks such as website fingerprinting attacks, utilizing UFS.

6.0 Countermeasures

In this chapter, we discuss the potential countermeasures for the three new attack methods proposed in this dissertation. We focus on 1) proposing the countermeasures tailored for each of these attacks and 2) analyzing whether the existing secure cache designs explained in Section 2 can stop these new attacks.

6.1 Countermeasures against Attacks Based on Cache-Controlling Instructions

In general, the attacks introduced by the instructions that allow the software-level users to directly control specific cache states can be prevented by disabling such instructions. However, this unfortunately also prevents benign users from utilizing these instructions to achieve better performance, and is thus not a preferred solution. Therefore, in this section we propose several countermeasures to particularly defend each of these attacks. We also show that these attacks can be mitigated through one or more existing countermeasures discussed in Chapter 2.

Countermeasures against attacks based on `PREFETCHW`. First of all, these attacks based on `PREFETCHW` (cf. Chapter 3) can be prevented through modifications on the microarchitecture behavior of `PREFETCHW`. The complete protection is two-fold. First, `PREFETCHW` should perform write permission checks, just as a regular memory write instruction, and trigger a fault or ignore this instruction if the target data is not writable. Second, `PREFETCHW` should execute in constant time. These modifications may introduce some performance overhead. We do not suggest eliminating `PREFETCHW` since it is important for improving write performance.

Second, similar to many prior cache attacks [134, 55, 133], our attacks also require data sharing between the attacker and the victim. Thus, disabling implicit data sharing across security domains, as suggested in previous defense studies (e.g., [128, 95, 96]) can also be used to stop our attacks. However, it is important to note that this solution may significantly increase the system's memory pressure.

Countermeasures against attacks based on `PREFETCHNTA`. NTP+NTP (cf. Chapter 4) uses a similar threat model with prior conflict-based cache covert channels such as Prime+Probe. Thus, countermeasures to mitigate conflict-based channels (cf. Chapter 2) may also defend NTP+NTP. This includes 1) partitioning-based defenses (e.g., [80, 101, 22, 39, 58]) which partition the cache so that data from different security domains do not interfere with each other, and 2) randomization-based defenses (e.g., [96, 128, 111]) which make it very hard (if not impossible) to build set conflicts by modifying set index mapping. However, for randomization-based designs that rely on the random replacement policy [128], using `PREFETCHNTA` might weaken its security guarantee by a factor of w, where w is the associativity.

In addition, a countermeasure for NTP+NTP specifically is to change the LLC insertion policy for both prefetched and loaded cache lines. For example, cache lines can be loaded into the LLC with age **1** and prefetched into the LLC with age **2**. Then, a prefetched cache line is still evicted sooner than a loaded cache line, but the prefetched cache line is no longer guaranteed to be the eviction candidate in the set. Thus, NTP+NTP can no longer work reliably. In addition, with this modified policy, the speed of our eviction set construction method (Algorithm 5) is significantly reduced. We build Python models of both the original Intel LLC policy and this modified policy, and simulate both our eviction set construction method and the state-of-the-art [93] with these two policies. With Intel LLC policy, our method requires **7.25×** less memory references compared to the state-of-the-art. In contrast, with the modified policy, this improvement is reduced to **1.26×**.

However, this countermeasure also weakens the performance benefit of `PREFETCHNTA`. With the original Intel LLC policy, prefetched cache lines can occupy at most one way in an LLC set, ensuring that the upper bound of LLC pollution is $1/w$, where w is the associativity. This is no longer guaranteed with the modified policy.

6.2 Countermeasures against Attacks Based on UFS

As UF-variation (cf. Chapter 5) does not require data sharing or building cache conflicts, none of the existing countermeasure designs can be used to mitigate UF-variaion. Thus, we propose the following defense mechanisms to prevent this channel. These are all software-level solutions that can be used immediately to protect users.

Fixing the uncore frequency. The prerequisite of UF-variation is that the uncore frequency is dynamically adjusted with UFS (based on the running workloads). Thus, to prevent this covert channel, the system software can disable UFS. That is, the system software can set the minimum and maximum uncore frequencies to be the same (cf. Figure 26), forcing the uncore to operate at a fixed frequency (`freq_fix`). However, it may be difficult to determine the value of `freq_fix`. Using a high frequency increases the energy consumption. For example, for graph analytics applications [48], fixing the uncore frequency at `freq_max` increases the energy consumption by 7%. In contrast, using a low uncore frequency reduces the performance. A more desirable method is to randomize the uncore frequency: instead of always using a particular uncore frequency, every certain period of time, the system software randomly selects a frequency (from within the allowed frequency range) to set as the uncore frequency (i.e., `freq_fix`). This can guarantee security while maintaining a balance between the performance and energy consumption.

Restricting the frequency range for UFS. Using a smaller frequency range for UFS, compared to the default range, can mitigate the website fingerprinting side channel attack. From our experiments, limiting the range for UFS to no larger than 0.2 GHz (e.g., from 1.5 GHz to 1.7 GHz) makes it very difficult to distinguish the uncore frequency traces for different websites. However, this method cannot stop the covert channel (UF-variation). When using a smaller frequency range for UFS, the temporal resolution of UFS remains the same as before (i.e., 10 ms). Further, some of the conditions for triggering frequency increase/decrease also remain the same. For example, with more than 1/3 active cores being stalled, the uncore frequency still increases by 0.1 GHz every 10 ms, until reaching the highest frequency allowed. Thus, the channel capacity of UF-variation remains the same as long as the maximum and minimum frequencies for UFS are not set to be equal (cf. Figure 26).

Maintaining high uncore utilization. The UFS-based covert/side channels can also be prevented by maintaining high uncore utilization: one can use a background thread that is always stressing the uncore to make it stay at `freq_max`.

7.0 Related Work

We have already discussed previous work on cache attacks and defenses in Chapter 2. Here we discuss prior studies on prefetch-based attacks, CPU vulnerabilities related to cache coherence, and attacks based on frequency scaling and power management.

7.1 Prefetch-Based Attacks

Gruss et al. [54] made two observations about prefetch instructions on Intel processors. They found that the execution time of a prefetch instruction, such as `PREFETCHT0`, leaks the translation levels of inaccessible kernel addresses. Using this, they built an attack to break Kernel Address Space Layout Randomization (KASLR). They also observed that prefetch instructions change the cache state of inaccessible kernel memory, but recent work [104] proved this incorrect. In fact, their observation is the result of transient execution caused by a Spectre gadget in the kernel, not the prefetch instruction.

Very recently, Lipp et al. [76] observed that on AMD processors, the timing (and power consumption) of a prefetch instruction on an inaccessible kernel address leaks the translation level and TLB state of this address. They used this to break KASLR and leak kernel memory (with Spectre) on AMD processors. These two prefetch attacks are orthogonal to our attacks. They focus on specifically attacking the kernel; we instead focus on building general cache timing attacks.

Regarding hardware prefetch attacks, Shin et al. [105] attacked OpenSSL, leaking the private key by leveraging the Intel stride prefetcher. Rohan et al. [99] reverse-engineered the stream prefetcher on Intel processors, using it to build a covert channel.

7.2 Cache Coherence Vulnerabilities

Although we are the first to propose cross-core private cache side channel attacks leveraging cache coherence protocol invalidations, cache coherence protocols have been exploited in many different attacks. Trippel et al. [115] discovered that a transient write may change the coherence state of the target data, which can be used as a covert channel in transient execution attacks. In addition, previous studies [63, 74, 50] mention that “bouncing” cache lines between private caches may be used as a replacement for `CLFLUSH` or set conflicts in Spectre and Rowhammer attacks. However, in this method, coherence states are manipulated by write operations. This means it requires that at least part of the target cache line happens to contain writable data (unless Meltdown-RW [73, 44] can be exploited). Unfortunately, as discussed in [50], this requirement is impractical for general side channel attacks.

Prior work [67, 77] built cross-core attacks on AMD and ARM processors, respectively, based on cache coherence. An Evict+Reload attack on Intel processors with non-inclusive LLCs was proposed in [132]. In these three attacks, the attacker learns the victim’s behavior by distinguishing between remote private cache hits and DRAM accesses. A variant of Flush+Reload attack on x86 processors was proposed in [133]. It works by distinguishing between remote private cache hits and LLC hits. These attacks are more general than the ones discussed earlier, but they all suffer from low bandwidth as DRAM accesses are involved in the attacks.

7.3 Attacks Based on Frequency Scaling and Power Management

Covert channels. CPU power management systems dynamically adjust the CPU core frequencies to prevent exceeding the power (thermal) limits: the core frequencies are related with the available power headroom). Khatamifard et al. [70] show that this principle can be used to construct covert channels. Kalmbach et al. [68] propose TurboCC, a new mechanism for creating cross-core covert channels. With Intel Turbo Boost 2, the maximum CPU frequency is selected based on the number of active cores. TurboCC utilizes this and encodes information into the maximum CPU frequency by placing load on a certain amount of CPU cores.

Side channel attacks. Dipta et al. [40] found that DVFS feature on modern processors can be utilized to perform website fingerprinting attacks and keystroke attacks. Specifically, the selection of P-states is related to the utilization of the core, which is further related to private information such as the website loaded in the browser. Wang et al. [126] discover that when turbo boost is enabled, DVFS-induced CPU frequency adjustments depend on the power consumption which is data dependent. This indicates that the CPU frequency adjusts based on the data it is processing, and result in different performance with different data. This observation fundamentally undermines constant-time programming. In addition, Liu et al. [79] show that a privileged adversary can extract AES-NI keys using the frequency side channel after reducing the power limits to fractions of their default values.

8.0 Conclusion and Future Work

In this chapter, we conclude this dissertation and discuss future research directions in the field of cache attacks.

8.1 Conclusion

In this dissertation, we showed that the recent hardware developments in modern processors that aggressively push the performance optimization boundaries enable new and more powerful cache attacks.

Specifically, we first showed that the prefetch-for-write instruction on Intel processors, `PREFETCHW`, allows software-level users to directly control the data's cache coherence state, even for read-only data. This leads to a cross-core private cache eviction method. Based on this method, we developed the first two cross-core private cache attacks that work with both inclusive and non-inclusive LLCs.

Then, we demonstrated that the non-temporal prefetch instruction on Intel processors, `PREFETCHNTA`, can be used by software to manipulate the cache replacement state (i.e., the eviction candidate of a cache set). With this instruction, conflict-based cache attacks become much faster than previously expected. In addition, this instruction enables a new algorithm for building eviction sets, which outperforms the state-of-the-art by several times.

Next, we found that recent Intel processors incorporate an advanced frequency scaling design which dynamically adjusts the frequency of the uncore based on the uncore utilization as well as the core stalling ratio. Based on this design, we developed a highly practical and robust attack that 1) works across different processors (sockets) in the same system and 2) remains effective with existing defense mechanisms for uncore attacks (e.g., uncore partitioning) in place.

Finally, we studied the countermeasures against the above attacks. We found that although the attacks based on the special-purpose prefetch instructions can be defended by one or more existing countermeasures, the attacks based on uncore frequency scaling can bypass all existing countermeasures, calling for more comprehensive defense solutions.

8.2 Future Work on Cache Attacks

This dissertation studied three specific designs on modern CPUs that can trigger new cache attacks. In fact, there are many other advanced designs on CPUs, such as heterogeneous cores, that may also trigger security issues. In the future, it is very important for researchers to continue to investigate these designs and understand their security implications. In addition, modern CPUs are continually evolving and becoming increasingly complex with sophisticated functional designs. Moreover, each new generation of CPUs in recent years has introduced additional instructions. It is very time-consuming to analyze the microarchitectural details and discover potential side channels manually every time new features occur. To streamline this task, it is critical to develop automated tools for side-channel detection. These tools should examine the new processor features with common hardware flaws that can trigger side channels. For example, with a new CPU instruction, we can use these tools to test whether proper permission checks are implemented for it, whether its execution time is data-dependent, and whether its execution has any speculative microarchitectural effects. These tools can substantially reduce the effort required by researchers to identify side channels. They can also be used by processor vendors to proactively resolve security issues before their products reach the market.

Second, unlike CPUs, the security of accelerators like GPUs and FPGAs has received significantly less attention. Given the growing use of these accelerators, it is critical to understand their security implications and develop countermeasures. Similar to CPUs, accelerators are increasingly being shared among users, raising the risk of side channel attacks. However, accelerators may face unique side-channel vulnerabilities compared to CPUs: accelerators are designed for different purposes than CPUs, and thus their hardware naturally differs. For example, while CPUs typically use non-shared translation look-aside buffers (TLBs) among their computing units, GPUs often employ shared TLBs. Therefore, we may not be able to directly apply the attacks and defenses that are well established for CPUs to accelerators. Instead, to protect accelerator users from potential side channels, we need to 1) reverse engineer the accelerator microarchitectural details, 2) identify the associated security issues, and 3) develop countermeasures with low overhead. For instance, recently researchers have revealed that the cache replacement policy in modern GPUs takes into account the 'dirtiness' of cache lines, which is not the case for CPUs. This points to the potential for new (and possibly more severe) cache attacks, calling for specialized secure cache designs for GPUs. However, this is only an initial step; many hardware components on GPUs remain unexplored today and require future efforts.

Bibliography

[1] Alexa top websites. Available at https://www.expireddomains.net/alexa-top-websites.

[2] Huge pages and transparent huge pages. Available at https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/performance_tuning_guide/s-memory-transhuge.

[3] Intel® 64 and IA-32 architectures optimization reference manual. Available at https://cdrdv2.intel.com/v1/dl/getContent/671488.

[4] Intel® 64 and IA-32 architectures software developer's manual. Available at https://cdrdv2.intel.com/v1/dl/getContent/671200.

[5] Intel® Xeon® processor scalable family technical overview. Available at https://www.intel.com/content/www/us/en/developer/articles/technical/xeon-processor-scalable-family-technical-overview.html.

[6] mmap(2) — linux manual page. Available at https://man7.org/linux/man-pages/man2/mmap.2.html.

[7] pthread_mutex_lock(3p) — linux manual page. Available at https://man7.org/linux/man-pages/man3/pthread_mutex_lock.3p.html.

[8] Software optimization guide for the AMD family 15h processors. Available at https://www.amd.com/system/files/TechDocs/47414_15h_sw_opt_guide.pdf.

[9] SPEC CPU 2017. Available at https://www.spec.org/cpu2017.

[10] Spectre PoC. Available at https://github.com/crozone/SpectrePoC.

[11] taskset(1) — linux manual page. Available at https://man7.org/linux/man-pages/man1/taskset.1.html.

[12] 3dnow! technology manual, 2000. Available at https://www.amd.com/system/files/TechDocs/21928.pdf.

[13] Intel quickpath architecture, 2012. Available at http://www.intel.com/pressroom/archive/reference/whitepaperQuickPath.pdf.

[14] Intel® Xeon® scalable processor: The foundation of data centre innovation, 2017. Available at https://simplecore-ger.intel.com/swdevcon-uk/wp-content/uploads/sites/5/2017/10/UK-Dev-Con_Toby-Smith-Track-A_1000.pdf.

[15] Onur Acıiçmez. Yet another microarchitectural attack: Exploiting I-cache. In *ACM workshop on Computer security architecture*, 2007.

[16] Onur Acıiçmez, Çetin Kaya Koç, and Jean-Pierre Seifert. Predicting secret keys via branch prediction. In *Topics in Cryptology–CT-RSA 2007: The Cryptographers' Track at the RSA Conference*, 2006.

[17] Onur Acıiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In *Topics in Cryptology–CT-RSA 2008: The Cryptographers' Track at the RSA Conference 2008*, 2008.

[18] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. Evax: Towards a practical, pro-active & adaptive architecture for high performance & security. In *55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022.

[19] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the linux symposium*, 2009.

[20] Lucian Armasu. OpenBSD will disable Intel Hyper-Threading to avoid Spectre-like exploits (updated), 2018. Available at https://www.tomshardware.com/news/openbsd-disables-intel-hyper-threading-spectre,37332.html.

[21] Zelalem Birhanu Aweke and Todd Austin. Øzone: Efficient execution with zero timing leakage for modern microarchitectures. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2018.

[22] Raad Bahmani, Ferdinand Brasser, Ghada Dessouky, Patrick Jauernig, Matthias Klimmek, Ahmad-Reza Sadeghi, and Emmanuel Stapf. CURE: A security architecture with CUstomizable and Resilient Enclaves. In *30th USENIX Security Symposium*, 2021.

[23] Mohammad-Mahdi Bazm, Thibaut Sautereau, Marc Lacoste, Mario Sudholt, and Jean-Marc Menaud. Cache-based side-channel attacks detection through Intel cache monitoring technology and hardware performance counters. In *Third International Conference on Fog and Mobile Edge Computing (FMEC)*, 2018.

[24] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, et al. Speculative interference attacks: Breaking invisible speculation schemes. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[25] Malini K Bhandaru, Ankush Varma, James R Vash, Monica Wong-Chan, Eric J DeHaemer, Christopher Allan Poirier, Scott P Bobholz, et al. Dynamically controlling interconnect frequency in a processor, April 26 2016. US Patent 9,323,316.

[26] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. SMoTherSpectre: Exploiting speculative execution through port contention. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[27] Samira Briongos, Gorka Irazoqui, Pedro Malagón, and Thomas Eisenbarth. Cacheshield: Detecting cache attacks through self-observation. In *Eighth ACM Conference on Data and Application Security and Privacy*, 2018.

[28] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. Reload+Refresh: Abusing cache replacement policies to perform stealthy cache attacks. In *29th USENIX Security Symposium*, 2020.

[29] Leon Groot Bruinderink, Andreas Hülsing, Tanja Lange, and Yuval Yarom. Flush, Gauss, and Reload-a cache attack on the BLISS lattice-based signature scheme. In *International Conference on Cryptographic Hardware and Embedded Systems*, 2016.

[30] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. Fallout: Leaking data on Meltdown-resistant CPUs. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[31] Paizhuo Chen, Lei Li, and Zhice Yang. Cross-VM and cross-processor covert channels exploiting processor idle power management. In *30th USENIX Security Symposium*, 2021.

[32] Jonghyeon Cho, Taehun Kim, Soojin Kim, Miok Im, Taehyun Kim, and Youngjoo Shin. Real-time detection for cache side channel attack using performance counter monitor. *Applied Sciences*, 10(3):984, 2020.

[33] Thomas Claburn. RIP Hyper-Threading? ChromeOS axes key Intel CPU feature over data-leak flaws – Microsoft, Apple suggest snub, 2019. Available at `https://www.theregister.co.uk/2019/05/14/intel_hyper_threading_mitigations/`.

[34] Pat Conway and Bill Hughes. The AMD Opteron Northbridge Architecture. *IEEE Micro*, 2007.

[35] Jack Cook, Jules Drean, Jonathan Behrens, and Mengjia Yan. There's always a bigger fish: A clarifying analysis of a machine-learning-assisted side-channel attack. In *49th Annual International Symposium on Computer Architecture*, 2022.

[36] Yujie Cui and Xu Cheng. Abusing cache line dirty states to leak information in commercial processors. In *2022 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2022.

[37] 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 *31st USENIX Security Symposium*, 2022.

[38] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. Analysis of secure caches using a three-step model for timing-based attacks. *Journal of Hardware and Systems Security*, 3(4):397–425, 2019.

[39] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *29th USENIX Security*, 2020.

[40] Debopriya Roy Dipta and Berk Gulmezoglu. DF-SCA: Dynamic frequency side channel attacks are practical. *arXiv preprint arXiv:2206.13660*, 2022.

[41] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A timer-free high-precision L3 cache attack using Intel TSX. In *26th USENIX Security Symposium*, 2017.

[42] 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)*, 8(4):1–21, 2012.

[43] Sankha Baran Dutta, Hoda Naghibijouybari, Nael Abu-Ghazaleh, Andres Marquez, and Kevin Barker. Leaky buddies: Cross-component covert channels on integrated CPU-GPU systems. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[44] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[45] T. Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Trans. Inf. Theor.*, 2006.

[46] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

[47] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. Understanding and mitigating covert channels through branch predictors. *ACM Transactions on Architecture and Code Optimization (TACO)*, 13(1):1–23, 2016.

[48] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: A study of emerging scale-out workloads on modern hardware. *Acm sigplan notices*, 47(4):37–48, 2012.

[49] Andrew Ferraiuolo, Yao Wang, Danfeng Zhang, Andrew C Myers, and G Edward Suh. Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[50] Anders Fogh. Row hammer, java script and MESI, 2016. Available at `https://dreamsofastone.blogspot.com/2016/02/row-hammer-java-script-and-mesi.html`.

[51] Peter Garraghan, Paul Townend, and Jie Xu. An analysis of the server characteristics and resource utilization in Google Cloud. In *2013 IEEE International Conference on Cloud Engineering (IC2E)*, 2013.

[52] 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. In *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[53] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 1998.

[54] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016.

[55] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+Flush: A fast and stealthy cache attack. In *13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[56] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In *24th USENIX Security Symposium*, 2015.

[57] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *2011 IEEE Symposium on Security and Privacy (S&P)*, 2011.

[58] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. IVcache: Defending cache side channel attacks via invisible accesses. In *Proceedings of the 2021 on Great Lakes Symposium on VLSI (GLSVLSI)*, 2021.

[59] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. Adversarial Prefetch: New cross-core cache side channel attacks. In *2022 IEEE Symposium on Security and Privacy (S&P)*, 2022.

[60] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. An energy efficiency feature survey of the Intel Haswell processor. In *2015 IEEE international parallel and distributed processing symposium workshop*, 2015.

[61] Jawad Haj-Yahya, Lois Orosa, Jeremie S Kim, Juan Gómez Luna, A Giray Yağlıkçı, Mohammed Alser, Ivan Puddu, and Onur Mutlu. Ichannels: Exploiting current management mechanisms to create covert channels in modern processors. In *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.

[62] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2019.

[63] Jann Horn. CPU security bug: information leak using speculative execution, 2017. Available at `https://bugs.chromium.org/p/project-zero/issues/attachmentText?aid=287305`.

[64] Taylor Hornby. Side-channel attacks on everyday applications: Distinguishing inputs with Flush+Reload. *BlackHat USA*, 2016.

[65] Ralf Hund, Carsten Willems, and Thorsten Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy (S&P)*, 2013.

[66] 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 *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[67] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cross processor cache attacks. In *11th ACM on Asia conference on computer and communications security (Asia CCS)*, 2016.

[68] Manuel Kalmbach, Mathias Gottschlag, Tim Schmidt, and Frank Bellosa. Turbocc: A practical frequency-based covert channel with intel turbo boost. *arXiv preprint arXiv:2007.07046*, 2020.

[69] Mehmet Kayaalp, Nael Abu-Ghazaleh, Dmitry Ponomarev, and Aamer Jaleel. A high-resolution side-channel attack on last-level cache. In *53rd Annual Design Automation Conference (DAC)*, 2016.

[70] S Karen Khatamifard, Longfei Wang, Amitabh Das, Selcuk Kose, and Ulya R Karpuzcu. Powert channels: A novel class of covert communicationexploiting power management vulnerabilities. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2019.

[71] Jinchun Kim, Seth H. Pugsley, Paul V. Gratz, A. L. Narasimha Reddy, Chris Wilkerson, and Zeshan Chishti. Path confidence based lookahead prefetching. In *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

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

[73] Vladimir Kiriansky and Carl Waldspurger. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757*, 2018.

[74] 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 *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[75] Michael Kurth, Ben Gras, Dennis Andriesse, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. NetCAT: Practical cache attacks from the network. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[76] Moritz Lipp, Daniel Gruss, and Michael Schwarz. AMD prefetch attacks through power and time. In *31st USENIX Security Symposium*, 2022.

[77] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. Armageddon: Cache attacks on mobile devices. In *25th USENIX Security Symposium*, 2016.

[78] 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 *27th USENIX Security Symposium*, 2018.

[79] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency throttling side-channel attack. In *2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[80] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2016.

[81] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B Lee. Last-level cache side-channel attacks are practical. In *2015 IEEE Symposium on Security and Privacy (S&P)*, 2015.

[82] Adam Malamy, Rajiv N Patel, and Norman M Hayes. Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature, 1994.

[83] Andrew Marshall, Michael Howard, Grant Bugher, Brian Harden, Charlie Kaufman, Martin Rues, and Vittorio Bertocci. Security best practices for developing Windows Azure applications. *Microsoft Corp*, 42:12–15, 2010.

[84] 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.

[85] John D. McCalpin. Address hashing in Intel processors, 2018.

[86] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A Jiménez. Perspectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2020.

[87] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. Memjam: A false dependency attack against constant-time crypto implementations. *International Journal of Parallel Programming*, 47:538–570, 2019.

[88] 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 *2015 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2015.

[89] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: the case of AES. In *2006 Cryptographers' track at the RSA conference*, 2006.

[90] 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 *30th USENIX Security Symposium*, 2021.

[91] Colin Percival. Cache missing for fun and profit, 2005.

[92] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *25th USENIX Security Symposium*, 2016.

[93] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Prime+Scope: Overcoming the observer effect for high-precision cache contention attacks. In *2021 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2021.

[94] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. Double trouble: Combined heterogeneous attacks on non-inclusive cache hierarchies. In *31st USENIX Security Symposium*, 2022.

[95] Moinuddin K Qureshi. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

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

[97] 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 *2009 ACM SIGSAC conference on Computer and Communications Security (CCS)*, 2009.

[98] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[99] Aditya Rohan, Biswabandan Panda, and Prakhar Agarwal. Reverse engineering the stream prefetcher for profit. In *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020.

[100] Gururaj Saileshwar, Christopher W Fletcher, and Moinuddin Qureshi. Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2021.

[101] Gururaj Saileshwar, Sanjay Kariyappa, and Moinuddin Qureshi. Bespoke cache enclaves: Fine-grained and scalable isolation from cache side-channels via flexible set-

partitioning. In *2021 International Symposium on Secure and Private Execution Environment Design (SEED)*, 2021.

[102] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *2019 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2019.

[103] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read arbitrary memory over network. In *24th European Symposium on Research in Computer Security*, 2019.

[104] Martin Schwarzl, Thomas Schuster, Michael Schwarz, and Daniel Gruss. Speculative dereferencing: Reviving Foreshadow. In *25th International Conference on Financial Cryptography and Data Security (FC)*, 2021.

[105] Youngjoo Shin, Hyung Chan Kim, Dokeun Kwon, Ji Hoon Jeong, and Junbeom Hur. Unveiling hardware-based data prefetcher, a hidden source of information leakage. In *2018 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2018.

[106] 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 *30th USENIX Security Symposium*, 2021.

[107] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust website fingerprinting through the cache occupancy channel. In *28th USENIX Security Symposium*, 2019.

[108] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: Enabling microarchitectural replay attacks. In *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, 2019.

[109] Kimming So and Rudolph N Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6), 1988.

[110] Dawn Xiaodong Song, David A Wagner, and Xuqing Tian. Timing analysis of keystrokes and timing attacks on ssh. In *10th USENIX Security Symposium*, 2001.

[111] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. PhantomCache: Obfuscating cache conflicts with localized randomization. In *Network and Distributed System Security Symposium (NDSS)*, 2020.

[112] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. Packet chasing: Spying on network packets over a cache side-channel. In *ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020.

[113] Daniel Townley, Kerem Arıkan, Yu David Liu, Dmitry Ponomarev, and Oğuz Ergin. Composable cachelets: Protecting enclaves from cache side-channel attacks. In *31st USENIX Security Symposium*, 2022.

[114] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. Checkmate: Automated synthesis of hardware exploits and security litmus tests. In *51st IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[115] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. *arXiv preprint arXiv:1802.03802*, 2018.

[116] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *27th USENIX Security Symposium*, 2018.

[117] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *2020 IEEE Symposium on Security and Privacy (S&P)*, 2020.

[118] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue in-flight data load. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[119] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. CacheOut: Leaking data on Intel CPUs via cache evictions. In *2021 IEEE Symposium on Security and Privacy (S&P)*, 2021.

[120] Tarunesh Verma, Achilleas Anastasopoulos, and Todd Austin. These aren't the caches you're looking for: Stochastic channels on randomized caches. In *2022 IEEE Inter-*

national Symposium on Secure and Private Execution Environment Design (SEED), 2022.

[121] Pepe Vila, Boris Köpf, and José F Morales. Theory and practice of finding eviction sets. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[122] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. MeshUp: Stateless cache side-channel attack on CPU mesh. In *2022 IEEE Symposium on Security and Privacy (SP)*, 2022.

[123] Daimeng Wang, Ajaya Neupane, Zhiyun Qian, Nael B Abu-Ghazaleh, Srikanth V Krishnamurthy, Edward JM Colbert, and Paul Yu. Unveiling your keystrokes: A cache-based side-channel attack on graphics libraries. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[124] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing channel protection for a shared memory controller. In *2014 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2014.

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

[126] 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 *31st USENIX Security Symposium*, 2022.

[127] Hassan MG Wassel, Ying Gao, Jason K Oberg, Ted Huffmire, Ryan Kastner, Frederic T Chong, and Timothy Sherwood. SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. *ACM SIGARCH Computer Architecture News*, 41(3):583–594, 2013.

[128] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium*, 2019.

[129] Zhenyu Wu, Zhang Xu, and Haining Wang. Whispers in the Hyper-Space: High-speed covert channel attacks in the cloud. In *21st USENIX Security Symposium*, 2012.

[130] Wenjie Xiong and Jakub Szefer. Leaking information through cache LRU states. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2020.

[131] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel atacks. *ACM SIGARCH Computer Architecture News*, 45(2):347–360, 2017.

[132] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *2019 IEEE Symposium on Security and Privacy (S&P)*, 2019.

[133] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. Are coherence protocol states vulnerable to information leakage? In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2018.

[134] Yuval Yarom and Katrina Falkner. Flush+Reload: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*, 2014.

[135] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant-time RSA. *ACM SIGARCH Computer Architecture News*, 45(2):347–360, 2017.

[136] Younis A Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. A new Prime and Probe cache side-channel attack for cloud computing. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, 2015.

[137] Kehuan Zhang and Xiaofeng Wang. Peeping tom in the neighborhood: Keystroke eavesdropping on multi-user systems. In *18th USENIX Security Symposium*, 2009.

[138] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM side channels and their use to extract private keys. In *2012 ACM SIGSAC conference on Computer and communications security (CCS)*, 2012.

[139] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-tenant side-channel attacks in PaaS clouds. In *2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2014.