Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures

JILIANG ZHANG, CONGCONG CHEN, and JINHUA CUI, Hunan University, Changsha, China
KEQIN LI, State University of New York, New York, USA

Microarchitectural vulnerabilities, such as Meltdown and Spectre, exploit subtle microarchitecture state to steal the user's secret data and even compromise the operating systems. In recent years, considerable discussion lies in understanding the attack-defense mechanisms and exploitability of such vulnerabilities. Unfortunately, there have been few investigations into a systematic elaboration of threat models, attack scenarios and requirements, and defense targets of the resulting attacks. In this article, we fill this gap and make the following contributions. We first propose two sets of taxonomies for classifying microarchitectural timing side-channel attacks and their countermeasures according to various attack conditions. Based on the taxonomies proposed, we then review published attacks and existing defenses and systematically analyze their internals. In particular, we also provide a comprehensive analysis of the similarities and differences among those attacks, uncovering the corresponding practicality and severity by identifying the attack targets/platforms and the security boundaries that can be bypassed to reveal information. We further examine the scalability of those defenses through specifying expected defense goals and costs. We also discuss corresponding detection methods based on different classifications. Finally, we propose several key challenges of existing countermeasures and the attack trends, and discuss directions for future research.

CCS Concepts: • **Security and privacy** → **Side-channel analysis and countermeasures;**

Additional Key Words and Phrases: Microarchitecture, timing side-channel attacks, transient execution, side-channel countermeasures

ACM Reference Format:
Jiliang Zhang, Congcong Chen, Jinhua Cui, and Keqin Li. 2024. Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures. *ACM Comput. Surv.* 56, 7, Article 178 (April 2024), 40 pages. https://doi.org/10.1145/3645109

1 INTRODUCTION

Over the past few decades, processor performance has seen significant advancements due to continued breakthroughs in manufacturing processes, but the semiconductor processing is apparently

This work was supported by the National Natural Science Foundation of China under Grants No. 62122023 and U20A20202, the Science and Technology Innovation Program of Hunan Province under Grant No.2021RC4019, the Natural Science Foundation of Hunan Province (Grant No. 2023JJ40160), the Natural Science Foundation of Changsha City (Grant No. kq2208212), the Fundamental Research Funds for the Central Universities and the Research Foundation of Education Bureau of Hunan Province (Grant No. 23B0036).
Authors' addresses: J. Zhang (Corresponding author), C. Chen, and J. Cui, Hunan University, Building E13-1, Smart Valley Park, Yuelu District, Changsha, China, 410082; e-mails: zhangjiliang@hnu.edu.cn, chencongcong@hnu.edu.cn, jhcui@hnu.edu.cn; K. Li, State University of New York, Science Hall 249, 1 Hawk Drive, New Paltz, New York, USA; e-mail: lik@newpaltz.edu.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM 0360-0300/2024/04-ART178
https://doi.org/10.1145/3645109

approaching physical limits. As a result, vendors have shifted their focus toward enhancing microarchitecture design by introducing various optimization strategies [73]. However, in recent years, these microarchitecture optimizations have raised a series of important security issues. Numerous types of **side-channel attacks (SCAs)** have been proposed, which exploit various subtle signals on the computer system such as power, electromagnetism, and execution time to infer confidential information from user programs and compromise the **operating systems (OSes)**. Among them, **microarchitectural timing side-channel attacks (MTSCAs)** are particularly prevalent and thus are the focus of this survey. Formerly, MTSCAs can only utilize timing differences to infer the victim's access patterns and leak specific secrets [58, 102, 174], such as encryption keys. However, since 2018, a new generation of microarchitectural attacks [15, 20, 22, 46, 77, 88, 89, 100, 104, 123, 129, 130, 138, 146, 151, 152] has shattered this standpoint. These attacks can provide a reliable and fine-grained manner for arbitrary data leakage across security boundaries.

Numerous microarchitectural techniques designed to boost processor performance likely cause serious security risks. Concretely, **simultaneous multi-threading (SMT)** enables multiple tasks to execute in parallel on a single CPU core. Although this feature significantly enhances the concurrency capability of processors, it exacerbates resource contention (e.g., cache, execution ports, etc.) among program instructions. Another prime example is the pipeline design in modern processors, where the out-of-order execution and branch prediction jointly work to maximize the efficiency of the pipeline by pre-executing future instructions. The two above undoubtedly create chances for construction of a timing channel. In particular, two innovative attack vectors, dubbed Spectre [88] and Meltdown [100], respectively, have demonstrated that those optimization or lookahead mechanisms are prone to information leakage vulnerabilities. More specifically, when the CPU check encounters an error, it rollbacks all beforehand executed instructions. However, any change to microarchitectural state (normally invisible to the program) made by such speculative execution still remains. Thus, an adversary can craft a (timing) channel to recover such states, thereby stealing the confidential data. Further, based on the aforementioned attacks, a plethora of variants have been proposed to either expand the attack surface or enhance the practicability, such as Foreshadow [20], ret2spec [104], LVI [146], **microarchitecture data sampling (MDS)** [22, 129, 151, 152], and many more. So far, a long line of work belongs to the well-known *transient execution attack*.

Particularly, MTSCAs are not restricted to several security-aware cryptographic applications [115], and attack variants have been developed to target different scenarios, such as bypassing **address space layout randomization (ASLR)** [40, 53, 68] and inferring keystroke behavior [97]. More recently, breaking the basic memory isolation enforced by the OS [88, 100] and even the hardware-enforced security guarantees like Intel **software guard extensions (SGX)** [92, 161] have been demonstrated. However, it is impractical to eliminate those hardware features from microarchitecture designs exclusively for security gains. As such, it is still very challenging to remove all side-channel vulnerabilities from both hardware and software designs without a deepgoing understanding of their nature and factors related to timing side-channel attacks and defenses.

This article presents a comprehensive survey of MTSCAs and their defenses with a particular focus on attack conditions/requirements. The attack condition refers to prerequisites that need to be met for a particular attack to be successfully launched. It typically includes specific resources, system configurations, or attacker actions that are necessary for the attack to work. Understanding that the attack condition is crucial for responding to potential risks and implementing effective countermeasures. We conduct our survey based on such knowledge. In this sense, our work departs from prior works. Specifically, we first provide a comprehensive overview of CPU microarchitecture and introduce various CPU optimization strategies. We then propose two sets of taxonomies for classifying the timing side-channel attacks and their defenses according to various

attack conditions. Subsequently, we review the published attacks and existing defenses and systematically analyze their internals, including the similarities and differences between distinct attack types and the scalability of those defenses. We also show the practicality and severity of each attack vector by specifying the attack targets/platforms and the security boundaries that can be breached. Ultimately, we propose several key challenges of existing mitigations and the attack trends, and discuss potential research directions.

Existing Surveys. Past investigations on MTSCAs have been published. Ge et al. [48] provide a detailed survey of MTSCAs and group them according to the shared resource categories. They highlight the importance of boosting hardware isolation in cloud environments. Biswas et al. [16] focus on other timing channels, such as network and system channels. Canella et al. [23] present the first comprehensive survey of transient execution, which classifies the attacks into Meltdown-type and Spectre-type. It emphasizes the intrinsic differences between the trigger causes of the two types of attacks. Lou et al. [103] focus on SCAs and defenses in cryptographic applications and analyze the exploitability of vulnerabilities from software and hardware perspectives. Xiong et al. [166] investigate transient execution attacks from 2018 to 2020 and discuss a taxonomy based on the root cause of the transient attacks.

However, several major problems are not fully discussed or clarified. First, existing investigations lack a comprehensive discussion of the attack requirements, together with the attack goals and models as a whole, so understanding the practicality of the attacks and defenses in different scenarios is challenging. Second, most of the taxonomies for transient execution attacks focus on emphasizing the differences between the attacks while ignoring the similarities between the attacks based on their attack conditions. As a result, it is difficult to formulate generic and potent countermeasures. Third, there is no complete discussion yet about both the attack trends and the defense challenges based on the above taxonomies.

Contributions. Our survey pinpoints three important contributions. First, we propose two sets of taxonomies of MTSCAs and their defenses based on different attack conditions. Second, in light of the attack taxonomy, we systematically analyze the existing timing side-channel attacks. We present a comprehensive analysis of the similarities and differences among these attacks, and further uncover the practicality and severity of the vulnerabilities by specifying the attack targets/platforms and the security boundaries. Third, we summarize the published defenses against these timing side-channel attacks based on our proposed defense taxonomy and examine their scalability through specifying wanted defense goals and costs. We also provide a concise discussion and classification of detection schemes. Finally, we flesh out several key challenges and discuss seven future research directions. We hope this survey can eliminate potential ambiguities amongst these attacks and defenses from more practical perspectives and can help one better understand the research area and spark more follow-up work.

Scope of investigation. This survey specifically examines timing side-channel attacks and defenses in processor microarchitectures, excluding other SCAs such as power consumption and electromagnetic attacks. We mainly take cache as an example to analyze the attack conditions and attack targets/platforms of non-speculative SCAs and transient execution attacks. We only mention SCAs involving other components when relevant. The transient execution attacks discussed pertain to a rich execution environment that supports versatile applications on the same platform. Note that attacks against **trusted execution environment (TEE)**, such as Intel SGX [17, 109, 118, 147, 149, 168], are not the main emphasis here. We discuss them whenever necessary.

2 BACKGROUND

This section revisits details of architecture and microarchitecture, such as various microarchitectural components and optimization techniques. It also covers the fundamentals of

microarchitectural timing attacks, such as covert channels, side channels, and transient execution attacks, as well as knowledge related to exceptions and **microcode assists (MAs)**. More details can be found in Reference [64].

2.1 Architecture and Microarchitecture

Architecture. Processor architecture refers to the overall design and organization of a processor, defining its functionality, organization, and implementation. The interface specifications between the processor and external systems are prescribed and implemented typically in an **instruction set architecture (ISA)**. Common processor architectures include ×86, ARM, RISC-V, and others. **Architectural state** refers to the visible state of a computer system to external observers. It defines the registers and memory states that software programmers can directly manipulate, as well as the interaction methods with the external environment.

Microarchitecture. CPU microarchitecture refers to the internal implementation details of a processor designed for a specific architecture. It involves the design and layout of components, such as core organization, pipeline, cache hierarchy, execution units, and more. *Microarchitectural state* refers to the state of instructions running inside the processor, which is used to record and manage various resources and data during execution. It encompasses buffers, caches, instruction queues, **branch predictors (BPs)**, and many more, all of which are invisible to software programs.

Instruction execution. In modern processors, the execution process of instructions typically includes five main stages: **instruction fetch (IF)**, **instruction decode (ID)**, **execution (EX)**, **memory access (MEM)**, and **write back (WB)**. The specific process may vary across different processor architectures. For more details, please refer to Reference [44]. Taking Intel processors as an example, the below is a breakdown of each stage:

IF: The processor fetches the next instruction's address based on the Program Counter and reads the instruction from the instruction cache.

ID: The instruction decoder parses instructions, determines their types and relevant operands and decodes them into smaller **micro-operations (μOPs)**. To predict execution paths of subsequent branches, processors introduce **branch predictors** to record the history of branch directions using **pattern history table (PHT)**, branch targets using **branch target buffer (BTB)**, and return addresses using **return stack buffer (RSB)**.

EX: Instructions are first sent to the **re-order buffer (ROB)**, which records the sequence of instruction execution to support out-of-order execution. Then they are dispatched to the corresponding execution units, such as arithmetic logic units and **floating-point units (FPU)**, to perform the expected operations.

MEM: When an instruction requires memory access, the processor uses the **memory management unit (MMU)** to translate virtual addresses to physical addresses and stores the related mappings into the **translation lookaside buffer (TLB)** for fast indexing. This stage also involves cache, which resides between the CPU and main memory to store recently or frequently accessed data, reducing memory access latency. The cache hierarchy consists of multiple levels, namely L1, L2, and L3. L1 and L2 caches are private to each core, while the L3 (also called the last level cache, LLC) is shared among all cores. Higher-level caches have larger capacities but slower access speeds accordingly. If the required data are in the cache, then the processor loads it into the register via the **load port (LP)**. Otherwise, the processor sends a request to the **line fill buffer (LFB)**. The LFB reads the entire data line from the main memory and stores it temporarily in the cache for subsequent load operations.

WB: The execution results are written back to the register file or the cache for usage of other instructions. This stage also includes the **store buffer (SB)**, which temporarily stores related operations that are retired but not yet written to the L1 cache, thereby improving WB efficiency.

When an instruction reaches the last stage of the pipeline, called the commit stage, it is marked as "completed" state in the ROB. Note that this state of the instruction indicates that the related computation was finished, but it still stays in the pipeline. Only when all previous instructions in the ROB were retired (which means that their results become architecturally visible to the program) can the instruction be allowed to leave the pipeline. In particular, the unretired loads and stores are held in the load-store queue according to the AMD manual [7]. It is similar to the ROB but is used to track the status and order of load and store instructions. Retired loads already have their values in the corresponding registers, while retired stores may have to wait in the SB for the relevant data to reach the L1 cache. In the Intel manual [73], the term used is only "store buffer," which includes both unretired and retired but not yet written to L1 stores.

2.2 CPU Optimization Techniques

Out-of-order execution. The processor can boost performance of program execution through adjustment to the sequence of code instructions executed. For example, when a long-latency instruction (e.g., memory load) occurs, the CPU dispatches the subsequent instructions that do not involve any data and control dependencies to proceed instead of being blocked in place. The states of those instructions are recorded in the ROB, and the CPU commits them to the architecture level only when the previous instructions have been retired. In this way, it ensures correctness of program execution.

Branch prediction. There are many branch instructions in programs, which can be speculatively executed to avoid pipeline stalling. The BPs are introduced in processor microarchitecture to store the information of history branches. When it is a correct prediction, the CPU retains the corresponding predicted results, thus gaining significant performance improvement. On the contrary, when it is a misprediction, code executed during the speculation window is rolled back to the previously saved state. The processor then re-executes the correct code branch.

Store-to-load forwarding. The processor utilizes a **store queue (SQ)** or SB to buffer store operations. To avoid read-after-write hazards, any subsequently issued load must check whether the latest target value is in the SQ or SB. If a match is found, then the value from the preceding store can be directly forwarded to the load operation without waiting for its result to be written back to the memory subsystem. This optimization is known as store-to-load forwarding, and it can eliminate unnecessary stalls, enhancing the efficiency of memory access.

Memory disambiguation. In out-of-order processors, loads that follow earlier stores to the same (or overlapping) address are first satisfied from the SQ or SB rather than relying on stale data from the L1 cache. Such loads are called aliasing previous stores. Since the SB buffers stores before committing them to the L1 cache, loads cannot be executed until all previous store addresses are known, limiting processor performance. Thus, processors introduce memory disambiguation to predict which loads are unlikely to alias with previously unknown store addresses, making them execute ahead of time. Note that the disambiguator only matches a subset of addresses determined by the page size. For instance, when the OS uses 4KB-sized regular pages (rather than large pages), the disambiguator performs prediction on the lower 12-bit addresses (i.e., 4KB aliasing).

Speculative dependency checking. Intel processors use proprietary memory disambiguation and dependency resolution logic to predict address dependencies related to speculative loads. Speculative dependency checking can determine whether the lower 20 bits of the load and store addresses (i.e., 2MB physical address aliasing) are the same. If it is the case, then the checking logic predicts a dependency between the addresses and blocks the load until the preceding store is completed. Beyond this point, the load can be executed ahead of time.

2.3 Microarchitectural Timing Attacks

This article focuses on timing attacks in microarchitecture. This section introduces covert channels, side-channels, and transient execution attacks, all of which are based on timing. The high-precision timer required in MTSCAs is described.

Covert channels. Covert channels exploit timing or delay variations to transmit information instead of an architecturally/OS-defined communication interface [163]. In a covert channel, the communicating entities are called the sender and receiver. The sender modifies the state of shared resources while the receiver observes the state changes (through timing) to infer the sent information. In a real-world attack, the sender is typically a Trojan process, which is a malicious program on the target system and capable of stealing secret data and encoding it into the microarchitectural state. The receiver is a Spy process responsible for decoding and recovering the secret data according to a predefined protocol. Covert channels can be seen as an instrumented attack that is often combined with other attacks, such as transient execution attacks.

Side-channel attacks. SCAs are similar to covert channels where the receiver (called an attacker) infers sensitive information by observing state changes. However, the sender is not a Trojan deliberately leaking information but a trusted entity (called a victim) that inadvertently leaks sensitive data (such as encryption keys) during its execution.

Transient execution attacks. Transient execution is a mechanism present in modern processors, where the processor performs operations (often speculatively) before determining the validity of instructions but does not submit their results. If an instruction is valid, then it is eventually retired; otherwise, it is squashed. Transient execution attacks rely on the side effects of instruction execution to disclose data. These side effects, which may or may not update the architectural state (retired or not retired), invariably leave traces in the microarchitectural state. Attackers can observe these traces with the assistance of covert channels and retrieve sensitive data. Nevertheless, the covert channel in transient execution attacks diverges from that in classical attacks, as the former transmits during transient execution, while all operations of the latter are non-transient. Instructions that access secrets may be transient or non-transient. However, non-transient secret access is typically caused by programming errors, which can be deduced and prevented by programmers and compilers [178]. Thus, it is not the focus of this article.

High-precision timer (HPT). MTSCAs are typically combined with high-precision/fine-grained timers to detect and analyze subtle microarchitecture states. The ×86 architecture provides an unprivileged read timestamp counter (`RDTSC`) instruction, which allows for cycle-level time measurements. This instruction provides a resolution of one to three cycles for Intel CPUs. In contrast, on AMD CPUs, particularly after the Zen microarchitecture, the resolution of `RDTSC` is significantly lower, updating only once every 20 to 35 cycles [99]. Although the AMD Ryzen microarchitecture offers highly accurate APERF counters [7], these counters can only be accessed from kernel space. However, even in the absence of HPT, the counting thread [98] can provide a high-resolution measurement. This method continuously increases a global variable serving as a timestamp independent of microarchitectural details. Intel's **transactional synchronization extensions (TSX)** [70] can also be used for the same purpose.

2.4 Exceptions and Microcode Assists

Many transient execution attacks rely on exceptions and MAs to trigger transient execution. This section introduces some knowledge related to them.

Exceptions. An exception refers to an error that occurs when the CPU is executing instructions, such as division by zero, illegal instructions, and memory access errors. Common exceptions include page fault (#PF), device not accessible fault (#NM), general protection fault (#GP), and bound-range-exceeded exception (#BR). In particular, the #PF is extensively triggered by the OS

and applications due to requests for paging and permission bit violations. The permission bits include user/privilege (U/S), page present (P), read/write (R/W), and so on.

Microcode assists. The ×86 architecture involves sophisticated instruction sets. Certain instructions, such as XSAVE, can be decoded into around 200 μOPs. It is impractical for the CPU to cope with all these instructions by pure hardware. Because the fast-decode path usually only supports handling instructions of 4 to 6 μOPs. For those decoded more than 4 μOPs, the CPU issues a MA and processes it with the aid of a microcode routine.

The microcode routine acquires microcode events [32] from the MSROM, which includes hardware exceptions, assists, and interrupts. The patent [126] describes an event table with 64 entries, the first 16 of which point to exception handlers, and the remaining are used by MAs. Each CPU core is able to issue exceptions and MAs while they can be handled only at the retirement of the μOPs.

3 OUR TAXONOMIES

In this section, we propose two sets of taxonomies as classification rules for the surveyed work. One is used to classify the MTSCAs, while the other is used to evaluate the defenses against those attacks. Our proposed taxonomies are based on the threat models, attack conditions, and scenarios of various attack and defense methods, all of which are not fully discussed together. We fill these gaps by covering the above points. Our taxonomies only consider timing SCAs introduced by the CPU microarchitectures. Other physical SCAs and software-driven SCAs, such as Hertzbleed [158], are beyond the scope of this article.

3.1 The Attack Taxonomy

We classify MTSCAs into conventional/classic attacks (a.k.a. non-speculative SCAs) and transient execution attacks (a.k.a. speculative SCAs) based on the requirement of transient transmission. The conventional attack observes shared state changes of microarchitecture components to infer information passively, where all instructions are executed non-speculatively. Conversely, the transient execution attack triggers programs to gain transient (or architectural) access to secret data and then transiently transmits it by a covert channel.

Conventional attacks. Conventional attacks can be classified into SCAs and covert channels according to their threat models (see Section 4.1). To better understand the attack elements, we divided SCAs into cache-based and non-cache-based attacks. This survey focuses on cache-based attacks, which can further be divided into eviction-based and control-based attacks depending on how the attacker initializes the state of the target cache line under different attack requirements. Eviction-based attacks implicitly modify cache states by accessing eviction sets or large arrays and thus are influenced by cache replacement policies. Control-based attacks utilize cache control instructions [73] such as CLFLUSH and PREFETCH to directly modify cache states without considering cache replacement policies. Non-cache-based attacks exploit components outside the cache, such as FPU, AVX units.

According to the continuity of the state, the covert channel can be divided into volatile and persistent channels. In the volatile channel, both the sender and receiver contend for hardware resources dynamically, which leaves no side effect after usage. In the persistent channel, the sender accesses the shared resources and leaves microarchitecture state there, which hence enables the receiver to be able to observe such state to infer secret information. To show practicality of different scenarios, we further divide the covert channel into same core, cross core, and cross CPU/GPU based on different shared levels.

Transient execution attacks. Transient execution attacks can be divided into speculation-based and exception-based attacks depending on the triggering cause of transient execution. According to the conditions of triggering speculation, we further divide the speculation-based

attacks into control-flow, data, and address. The control-flow speculation discloses secret information by mis-training the BPs to induce the victim to execute critical code along a speculated path. The data speculation misleads the victim to use incorrect data for speculative execution by mis-training the value predictor. The address speculation means an attacker can disclose secret information by triggering store-load address speculation.

Exception-based attacks can be divided into permission violation exceptions and illegal instruction/operand exceptions. Based on the conditions of triggering exceptions, we classify permission violation exceptions into memory access and register read exceptions. The former refers to the occurrence of permission violation exceptions during memory access, including #PF, memory overflow, segmentation faults, and misalignment, among others. The latter means an exception occurs when reading certain privileged registers (e.g., **model specific register (MSR)**). It is important to note that not all exceptions can be used for attacks. An illegal instruction/operand exception occurs when the executed instruction itself generates an exception, such as divide-by-zero and unrecognized instruction exceptions, which have not yet been exploited. Exception-based attacks exploit the time window during exception handling to expose sensitive information to attackers.

3.2 The Defense Taxonomy

We summarize the necessary conditions for the conventional and transient execution attacks and categorize published defense techniques based on those attack conditions.

Conventional defenses. For conventional attacks, we divide the defenses into two categories: hiding time differences and limiting resource interference. Both eliminate observable time differences and measurable resource contention for the attacker respectively, as they are necessary for the majority of SCAs (see Section 4.3). The method of hiding time differences can further be divided into the eliminating timer, constant-time operation, and noise injection. The eliminating timer disables the HPT available to the attacker. The constant-time operation requires careful checks by software developers to ensure that the access patterns of code and data are secret-independent at runtime. Noise injection interferes with the attacker's measurements by obscuring the HPT. A common way to limit resource interference is to partition and randomize caches for different processes or security domains. To be more specific, cache partitioning divides shared caches into different ways or set regions, which are not accessible to each other (non-interfering). Randomization makes the address mapping to the cache unrecognizable to attackers, thereby increasing the difficulty of constructing precise memory access to interfere with the victim's cache.

Transient execution defenses. For transient execution attacks, we divide the defenses into four categories: isolating shared states, limiting speculation, disabling unauthorized access, and invisible state changes. These defenses prevent the secret information from being leaked under transient execution by restricting different attack conditions. Isolating shared states impedes sharing the microarchitectural states between the attacker and the victim. Restricting speculation stops transient execution of wrong predictions by the victim. Disabling unauthorized access hinders operations to unauthorized target addresses. Invisible state changes aim to obstruct the observation to microarchitecture state changes through side channels.

3.3 The Detection Taxonomy

We discuss some approaches for detecting MTSCAs separately and categorize them into code detection and behavior detection. Code detection identifies vulnerable code locations (gadgets) beforehand and then mitigates them. Behavior detection identifies potential attack behaviors, such as cache hit/miss rates, attacker preemption, and conflict patterns.

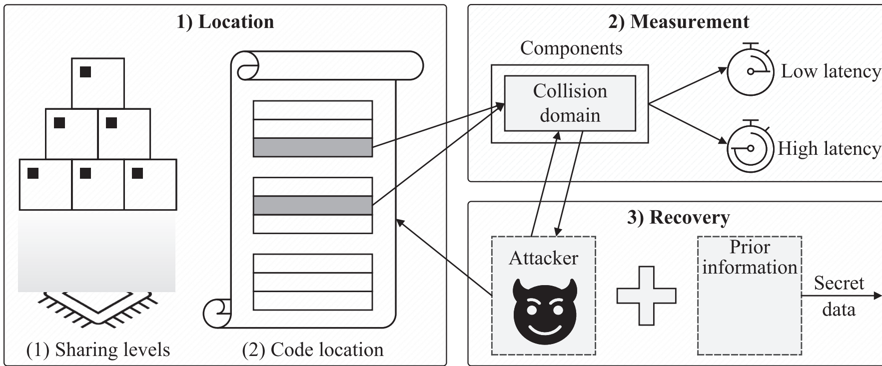

Fig. 1. The workflow of conventional attacks.

4 CONVENTIONAL ATTACKS

In this section, we discuss conventional attacks under our taxonomies from the following aspects: threat model, attack steps, typical examples, attack conditions, and attack targets.

4.1 Threat Model

In a conventional attack, we assume that an attacker has the following three capabilities: (1) sharing the same hardware components with the victim, (2) being capable of causing the microarchitectural state changes when processing data, and (3) observing the resulting impact in microarchitecture. These capabilities can be achieved conditionally in modern computer systems or cloud environments. For (1), sharing components may be core-private (such as BPs), across-core-shared (such as LLC), or even CPU-shared (such as the CPU directory). For (2), many microarchitectural components are competitively shared or time-multiplexed among different threads or processes to improve data processing efficiency. For example, the BP is used to cache the history of recently executed branches to predict the execution of the next branch. In other words, the current states may be influenced by previously used metadata and data in the components. For (3), since microarchitectural components do not provide an interface to query the related state, the attacker can observe the subtle state through timing channels.

Conventional attacks include SCAs and covert channels, both of which involve two entities: the sender and the receiver. For SCAs, the sender is a victim process, and the receiver is a Spy process (attacker). It usually contains the following assumptions: First, the victim may perform some secret-related operations (e.g., a simple if-else statement to check the value of a key); second, the Spy has knowledge of the victim's program, such as the gadget address. For covert channels, the sender and the receiver are two colluding processes, i.e., a Trojan and a Spy. It is assumed that the Trojan has acquired the secret but cannot communicate over the network (e.g., in a sandbox). In contrast, the Spy is located in a non-confined environment. The Trojan and the Spy are prohibited from communicating with each other due to enforcement of the security policy. To this end, the Trojan attempts to send confidential data to the Spy via a covert channel. Covert channels are composed of volatile and persistent channels. For the former, two entities must run concurrently, e.g., on an SMT core. However, for the latter, two entities do not need to execute in parallel.

4.2 Procedure of Attacks

To clarify the implementation of attacks and assist in adopting targeted countermeasures, we divide the conventional attack into three steps, as shown in Figure 1.

- **Locating the vulnerability.** One premise of conventional attacks is that the attacker and the victim can jointly manage the resource usage of a microarchitectural component. Since these components are versatile, two entities must first determine what resources they share. For instance, an adversary in a cross **virtual machine (VM)** attack must first determine the "co-resident VMs." That is, whether the attacker's VM and the victim's are located on the same or different processor core(s), or on the different VMs assigned to different processors. The attacker then needs to find vulnerable code gadgets. For example, a branch associated with secret data performs different operations based on its results.
- **Measuring microarchitectural state changes.** After locating vulnerabilities, the attacker needs to determine the collision domain with the user program, which refers to a set of addresses that contend for the same component region and collectively affect its microarchitectural state. The collision domain may generate some impacts on the execution process, which are summarized as follows: (1) the differences of execution time. Since code or data may be located in different levels of the memory hierarchy, it likely causes varying access latency. For example, it takes ~33–43 cycles to access data from L1 and about 230 cycles from memory on Dell PowerEdge T420 with an Intel Xeon E5-2430 processor [174]. (2) Hardware resource usage. During the program execution, some hardware units may be exclusive or locked [5, 164], so the resources will become temporarily unavailable to other programs. An attacker can measure these hardware resources to analyze the user's behaviors. (3) Transaction aborts. With Intel TSX instructions, a transaction will trigger an abort when a cache line of "read set" is evicted from the L3 cache. Thus, the attacker can capture these abort events for further analysis [38].
- **Recovering secret information.** After obtaining the changes of the microarchitectural state, the attacker processes the learned memory access patterns by combining some prior information. For example, she can recover control flow and divulge memory access data. With these capabilities, the attacker can extract the user's secret key [174], behaviors [139], distribution information in kernel space [40], and so on. This step can be performed through offline analysis in an end-to-end attack. Prior information refers to the additional knowledge that an attacker has about the attack target before launching an attack. For example, in terms of the implementation of RSA algorithm by Montgomery's large number modulo multiplication [133], the attacker knows that a Square-Reduce-Multiply-Reduce sequence signifies an exponent of "1" while Square-Reduce represents the opposite "0." By combining this information, attackers can observe the usage of these functions to retrieve the key after recovering the exponent.

4.3 Side-channel Attacks

Conventional SCAs can be divided into cache-based and non-cache-based attacks according to the aforementioned taxonomy. Of these SCAs, the cache-based is best known in modern processors. The root cause is that caches are widely present in various CPU microarchitectures and are shared among processes, VMs, or even multiple processor cores. Due to the numerous variants and significant differences in cache-based attack methods, this article primarily focuses on the implementation of this type of attack, while the non-cache-based attack is only discussed briefly in Section 4.3.5. As shown in Table 1, we provide a detailed analysis of the attack types, typical examples, attack conditions, and attack targets for cache-based attacks.

Hu [67] first proposed that caches are suitable covert channels for secret information extraction in 1992. Kesley et al. [80] mentioned the possibility of cache attacks based on cache hit/miss rates. Later, Page [113] theoretically studied cache attacks, and Tsunoo et al. [145] researched timing leaks caused by internal table lookup conflicts. However, it was not until 2005 that the first

Table 1. The Conditions and Targets of Cache-based Attacks

Attack type		Condition†								Variant	Target
		ES	*CLF*	*PRE*	*SM*	*SCR*	*HPT*	*TSX*	*HC*		
Eviction	**TS**	○	○	○	○	●	●	○	○	Evict+Time [12, 111]	AES
		●	○	○	○	●	●	○	○	Prime+Probe [102]	GnuPG, AES
		●	○	○	○	●	○	●	○	Prime+Abort [38, 84]	GnuPG, AES
		○	○	○	○	●	○	○	●	Occupancy [136]	Website fingerprint
	MS	●	○	○	◐	●	●	○	○	LRU states [18, 119, 165]	GnuPG, AES
Control	**TS**	○	●	○	●	●	●	○	○	Flush-based [58, 174]	GnuPG, AES, keystrokes
	MS	○	○	●	●	●	●	○	○	Prefetch-based [57, 62, 96]	GnuPG, keystroke, KASLR
		◐	◐	○	●	●	●	○	○	Invalidate+Transfer [76]	AES, ElGamal

†TS: tag state; MS: metadata state; ES: eviction set; CLF: `CLFLUSH` instruction; PRE: `PREFETCH` instruction; SM: shared memory; SCR: shared cache resources; HC: performance counters. ● indicates that the condition is required; ○ indicates that the condition is not required; ◐ indicates that the condition is optional.

practical implementation of cache attacks was demonstrated [12]. Percival [114] demonstrated that shared access to memory caches provides a covert channel between threads and allows a malicious thread to monitor the execution of another thread to steal encryption keys. Early cache-based attacks were mainly time-driven SCAs that collect the overall encryption/decryption times in the cryptographic process. This is determined by the number of cache hits and misses. One representative attack was Evict+Time [111], where an attacker first measures the execution time of the victim's program as a baseline. She then evicts certain specific cache sets and measures the victim's execution time again to determine if the victim has accessed memory locations mapped to the previously evicted cache sets. In 2010, access-driven cache attacks began to attract attention [1]. Subsequently, some classical access-driven attacks such as Flush+Flush [58], Flush+Reload [174], and Prime+Probe [102] were proposed, which are the focus of this section.

4.3.1 Attack Types. In a cache-based attack, the attacker first interferes with the cache access pattern of the victim and then exploits the access latency in different states to infer the victim's behaviors. We classify this type of attack into eviction-based and control-based. In an eviction-based attack, the attacker fills the cache set (that the victim's data occupies) with his own data to initialize the cache state through set conflicts. In a control-based attack, it is usually assumed that the attacker and the victim share data, which allows the attacker to directly manipulate the victim's cache state using cache control instructions, such as `CLFLUSH` and `PREFETCH`. The cache state can be divided into cache tag state and metadata state [86]. The metadata state includes cache coherence metadata (e.g., MESI [62]) and replacement metadata (e.g., LRU [165]). Therefore, eviction-based attacks can be further categorized into **eviction of tag state (Eviction-TS)** and **eviction of metadata state (Eviction-MS)** attacks. Likewise, control-based attacks can be divided into **control of tag state (Control-TS)** and **control of metadata state (Control-MS)** attacks.

4.3.2 Typical Examples. Prime + Probe [102] first populates one or more cache sets with the specified data or code (*Prime*) and then waits for the victim to execute. Afterward, the attacker measures the load time while reloading the primed data or code (*Probe*). If the victim has accessed the cache set, then the previously populated data will be evicted, thus resulting in a higher latency in the *Probe* phase. Flush + Reload [174] first flushes the target cache line using the `CLFLUSH` instruction (*Flush*). After a period of time, the attacker re-accesses the target line and measures the reload time (*Reload*). A lower reload latency indicates that the victim has accessed the target

cache line. By comparison, the prefetch-based attack [62] first prefetches the target data using the PREFETCHW instruction, which will change the target cache line to a modified (exclusive) state. When the victim accesses the target line, it goes into a shared state. Otherwise, this line remains unchanged (i.e., modified state). Finally, the attacker can infer whether the victim accessed the target data by measuring the time of re-accessing it. Because the time of fetching the cached data from the LLC is obviously different from that from the remote private cache.

4.3.3 Attack Conditions. Various MTSCAs manifest differences as well as similarities in terms of essential conditions. Many prior studies have sought to develop more powerful attack variants by relaxing necessary attack conditions. In this section, we summarize the related attack conditions in Table 1 and discuss how they affect the attack process.

First, we list the differences of common attack conditions as follows.

— **Eviction set.** Eviction-based attacks need to construct eviction sets, each of which is a set of addresses mapped to the same cache set. However, it is necessary for the attacker to know certain physical address information. To find eviction sets, Vila et al. [153] proposed a new algorithm that reduces the time complexity from quadratic to linear scale. To perform high-frequency measurements using a quick eviction strategy, Lipp et al. [98] evaluated over 4,200 access patterns on smartphones to seek the ideal eviction strategy, where the eviction time is reduced to 1578 CPU cycles. Purnal et al. [119] find a specific *Prime* pattern according to the cache replacement policy, which allows accessing only a cache line in the *Probe* phase, thus improving the measurement speed. In particular, several eviction-based attacks do not rely on eviction sets. For example, Evict+Time [12, 111] and Occupancy channel [136] evict the victim's data by accessing large arrays.
— **Cache control instructions.** To get rid of the need for eviction sets or large shared arrays, control-based attacks use a cache control instruction [73] with only user privileges to modify the cache state of the victim. For instance, Guo et al. [62] discovered that the PREFETCHW instruction can change the cache line state to "Modified" even if the target data are read-only for the attacker. Based on this observation, the attacker can set the target data to be L1 private. Once the victim accesses this data, it becomes LLC-shared. Since an LLC hit takes less time than a remote private cache hit, the attacker can measure access time to infer whether the victim has accessed target data. The limitation of control-based attacks is that the CLFLUSH is invalid in ARM processors (except ARMv8-A [98]) and JavaScript [53].
— **Shared memory.** Eviction-based attacks do not require shared memory between the attacker and the victim, whereas control-based attacks require shared memory.

Second, we summarize the similarities of common attack conditions as follows.

— **High-precision timer.** It is important for the cache-based attacks to have an HPT to measure subtle cache state. Typically, an attacker can invoke a RDTSC instruction to read the current timestamp. However, the RDTSC is unavailable in the JS browser and ARM architecture. Instead, other timing-related APIs or counting threads are proposed in Reference [98]. Disselkoen et al. [38] introduced a timer-independent attack that exploits the Intel TSX feature. When a cache line is evicted during a transaction, TSX will trigger an exception and terminate the transaction. Shusterman et al. [136] proposed a cache occupancy-based channel, which can collect cache traces with the assistance of performance counters. Moreover, it measures the access time of the whole LLC cache (rather than a cache line) to avoid the usage of the HPT. Other studies are also dedicated to non-timing attacks. Cache Storage Channels [60] exploit virtual aliases by misconfiguring the memory system with mismatched memory attributes and self-modifying code. This allows attackers to place inconsistent copies

of the same physical address into the cache and observe which address is cached. S2C [176] discovered that the execution of synchronization instructions causes architectural state changes when L1 cache evictions occur, enabling attackers to monitor the microarchitectural state changes of addresses without the need for a timer. Zhang et al. [181] leverage the unprivileged idle-loop optimization instructions introduced by the Intel microarchitecture to provide architectural feedback on the transient usage of specified memory regions.

— **Shared cache resources.** Cache-based attacks are based on shared resources between the attacker and the victim. For instance, control-based attacks exploit shared memory and the same cache line, and eviction-based attacks share the same cache set. Different shared resources may affect the accuracy or granularity of an attack.

4.3.4 Attack Target. Most cache-based attacks attempt to infer the secret keys generated from encryption algorithms (e.g., RSA) by probing the victim's cache traces during encryption. Gruss et al. [57] introduce a prefetch SCA that allows an unprivileged local attacker to bypass access control completely. This way can breach protections of supervisor mode access prevention and supervisor mode execution prevention in a ret2dir attack [81] and break the **kernel address space layout randomization (KASLR)** and launch the return oriented programming attack in kernel space. Thus, the whole system may be compromised. Shusterman et al. [136] collect traces of cache occupancy when a browser downloads and renders websites. Further, it utilizes deep neural networks to analyze and classify the collected traces. Finally, it can identify which website the user is browsing.

4.3.5 Non-cache-based Attacks. Non-cache-based attacks target other CPU components outside of the cache, such as the FPU [46], AVX units [130], TLB [52], and BTB [40]. Wang et al. [159] investigated how the enhanced features of modern processor architectures, such as SMT and control speculation, open up new opportunities for SCAs. Andrysco et al. [8] discovered that the execution time of FP `ADD` and `MUL` instructions in ×86 processors shows a big difference between different operands. They exploited the FP data timing variability to carry out SCAs. A timing channel was constructed by leveraging the power-saving characteristics of AVX2 units [130]. TLBleed [52] built an SCA over shared TLBs to extract secret information from a victim program protected by cache defenses. Evtyushkin et al. [40] utilized a BTB-based SCA to break user-level and kernel ASLR.

4.4 Covert Channels

The covert channel is a type of communication channel that attempts to avoid detection or bypass OS isolation. It utilizes the time variation of shared resources to transmit information between two entities. Covert channels can be divided into volatile and persistent channels, as shown in Table 2.

4.4.1 Volatile Channels. The volatile channels do not leave any footprint in the microarchitecture after usage. In this case, the sender and receiver have to execute concurrently for contention of hardware resources. For instance, two entities use the same execution port on a physical core [5]. According to a pre-agreed protocol, the sender can perform different operations to vary the throughput of the receiver (e.g., the effect of high latency). The receiver then measures the time of occupying the port to infer the "0" or "1" transmitted.

The root cause of volatile channels is the limited bandwidth of shared resources. The exploitable resources include cache banks [78, 108, 175], execution ports [5], interconnect paths [2, 112, 155], memory buses [164], and PCIe [139]. This article divides these resources into same-core, cross-core, and cross-CPU/GPU. Different levels indicate what cases the Trojan and the Spy can co-locate. For example, two entities based on the covert channel of execution port [5] must run on the same physical core, which requires the support of SMT. Note that SpectreRewind [46] constructs a new

Table 2. The Taxonomy of Covert Channels

Attack type	Level	Sharing resource	Time resolution (cycles)	Bandwidth (KB/s)	Bit error rate (%)
Volatile channels	Same core	L1 cache bank [78, 108, 175]	~10–20	Not given	Not given
		Execution ports [5] [15]	~30	Not given	Not given
		FP division unit [46]	~20–30	~53–115	<0.5
	Cross core	CPU ring interconnect [112]	~30	~688	<5
	Cross CPU/GPU	Memory bus [164]	~2,500–8,000	~0.093	0.09
		PCIe [139]	~1,200–1,300	Not given	Not given
		Network-on-Chip [2]	Not given	3000	~0
		CPU mesh [155]	~10,000–40,000	Not given	Not given
Persistent channels	Same core	L1/L2 [114, 167]	~10–30	~400	Not given
		LRU states [165]	~5–15	72.5	8
		Way predictor [99]	~50–60	588	~1–3
		TLB [52] [68]	~30	~5	~0
		PHT [39, 41, 42]	~25	~15	~4
		BTB [40]	~9	Not given	Not given
		AVX unit [130]	~300–400	0.125	0.58
	Cross core	LLC [18, 58, 59, 62, 98, 102, 119, 127, 174]	~100	~75–1,801	<1
		NUMA [173]	~200	23.75	0
		Cache directories [170]	360	25	Not given
		Dirty states [172]	~100–200	~162.5–550	<5
		Cache coherence [35]	~25–150	~87.5–137.5	<10
		MMU [150]	Not given	~0.9	~2
	Cross CPU/GPU	Directory protocol [76]	~50–250	Not given	Not given
		DRAM row buffer [116]	~50	250	<1

volatile channel where the sender and receiver are two code snippets on the same thread, which makes it feasible on a system without SMT. Due to out-of-order execution, the new instructions and the old ones may cause contention on the same port. Therefore, the sender can choose whether to cause resource contention according to the message bits to be transmitted, and the receiver measures the execution time of the same type of instructions to infer the sent message.

4.4.2 Persistent Channels. The persistent channels leave lasting footprints in the microarchitectural state. To be specific, the Spy first initializes the shared component into an expected state and then waits for the Trojan to execute. Subsequently, the Trojan decides whether the shared states will be changed according to the transmission of "0" or "1." Finally, the Spy checks the state changes by the timing difference to infer the secret. In this covert channel, the Trojan leaves some secret traces, which are preserved until the Spy finishes probing them.

The root cause of persistent channels is the limited storage space for shared microarchitectural resources. The shared resources available for persistent channels include cache [12, 18, 19, 58, 62, 98, 102, 119, 127, 174], way predictor [99], TLB [52, 68], PHT [39, 42], and BTB [40]. Cache-based covert channels exploit the fact that the timing of cache operations depends on the presence of the target address in the cache. Most existing channels may occur cache misses (either evicting or flushing cache lines) by changing the cache states. However, with cache replacement policies, any cache access (either hits or misses) may trigger a new attack (e.g., LRU [18, 119, 165]).

4.4.3 Measurable Indicators for Covert Channels. Developing a more aggressive attack or evaluating the harmfulness of an attack must have a thorough understanding of the attack indicators. We describe two important measurable indicators of covert channels, i.e., the time resolution and bandwidth, as shown in Table 2.

Time resolution refers to the time difference caused by the state changes of shared resources. For example, transmitting "0" and "1" may take varying time. The higher the resolution of the state changes, the better the visibility of data and control flow access of victim applications. For the cases, when the time resolution needs little dependencies, the channel established will be more stable or reliable. The receiver can improve the time resolution through accumulation methods. For example, multiple division instructions are chained together in a `fdiv` channel [46]. The higher the time resolution, the transmission rate may become low accordingly. Therefore, a reliable channel should keep a balance between the transmission rate and the time resolution.

Bandwidth refers to the capacity of a covert channel to transmit data per unit of time. Typically, bandwidth may suffer from noise effects from other software and OS in a real system. If the error rate is controllable, then higher bandwidth will be better. As recommended by trusted computer system evaluation criteria [95], the bandwidth of a channel should not be lower than 1 b/s as the "acceptable" threshold. The lower channel capacity may bring negligible risks. Saileshwar et al. [127] proposed Streamline, which utilizes asynchronous communication to improve the transmission rate. The main idea is to use a series of shared addresses (larger than the cache size) to induce cache thrashing and reduce the latency of inter-symbol synchronizations through asynchronous communication. In this way, they improve the rate of Prime+Probe from 75 KB/s to 1,801 KB/s.

5 TRANSIENT EXECUTION ATTACKS

Transient execution attacks exploit various optimization mechanisms to perform unauthorized computations, thereby accessing the secret and transiently encoding it into the microarchitectural state. The attacker then recovers the secret through covert channels in the normal (non-speculative) state. In this section, we group transient execution attacks into two categories, as shown in Table 3. Moreover, we discuss the threat model, attack steps, typical examples, attack conditions, platforms, leakage channels, gadget types, and limitations for transient execution attacks.

5.1 Threat Model

In a transient execution attack, we usually assume that an attacker has the following three capabilities. First, the attacker has partial control over a process that is a Spy program with only normal user privileges. With the assistance of the Spy, she can execute malicious code snippets (NetSpectre [130] is an exception). Second, the attacker cannot manipulate the control flow or memory of the victim. That means the victim's integrity is guaranteed well during the normal execution. Finally, the attacker has knowledge of the gadget in the victim code and can trigger the execution of transient instructions (speculative instructions bound to squash) to execute the gadget. Transient execution attacks can leak encryption keys and even data from protected regions. In general, we hypothesize that the attacker only snoops on the victim's behavior rather than attempting to compromise data integrity. For example, Rowhammer attack [56] can cause bit flips in adjacent rows by accessing a row of memory repeatedly, which results in corruption of in-memory data and code. In addition, a combination of Rowhammer and Spectre is proposed to enhance speculative execution attacks [142]. However, we focus on microarchitectural attacks that only break data confidentiality, other types of attacks are beyond the scope of this article.

5.2 Procedure of Attacks

Transient execution attacks execute instructions in advance to access unauthorized secret data and leave measurable side effects in the microarchitecture state. Although different attacks vary in terms of implementation details [23], they all can be divided into three phases, as shown in Figure 2.

Table 3. The Taxonomy of Transient Execution Attacks

Attack type		Trigger condition	Attack variant	Component / privilege	Platforms†			
					INTEL	*AMD*	*ARM*	*RISC-V*
Speculation-based	**Control-flow**	Conditional branch	Spectre-PHT (V1.0/1.1) [29, 88, 130, 144]	PHT/BHB	☑	☑	☑	☑
		Indirect branch	Spectre-BTB (V2) [27, 88]	BTB/BHB	☑	☑	☑	☑
		Return branch	Spectre-RSB (V5) [89, 104]	RSB/BTB	☑	☐	☑	☑
	Address	Lower 12 bits	Spectre-STL (V4) [65]	STL	☑	☐	☑	☑
			Fallout [22]	SB	☑	☐	☐	☐
			LVI [146]	L1D/SB/LFB/LP	☑	☐	☐	☑
		Lower 6 bits	RIDL [151], ZombieLoad [129], CacheOut [152]	LFB/LP	☑	☐	☐	☑
			CROSSTALK [123]	Staging buffer	☑	☐	☐	☐
		12–19 bits	Spoiler [77]	MOB	☑	☐	☐	☐
	Data	Value	Value predictor [36]	Value predictor	☐	☐	☐	☐
Exception-based	**Permission violation**	Memory access	Meltdown (V3) [100]	US	☑	☐	☑	☑
			Spectre V1.2 [87]	RW	☑	☐	☑	☐
			Foreshadow [20, 161]	P	☑	☐	☐	☑
			Meltdown-PK [23]	Protection key (PK)	☑	☐	☑	☐
			LVI [146]	US/P/RW	☑	☐	☐	☑
			MicroScope [137]	P	☑	☐	☐	☐
			Meltdown-BR [23]	MPX	☑	☑	☐	☐
		Register read	Lazy FP [138]	FPU	☑	☐	☐	☐
			Meltdown-GP (V3a) [93]	MSR	☑	☐	☑	☐
	Illegal instruction / operand	Not implement	Not implement	Not implement	Not implement			

†☑ indicates this platform is vulnerable; ☐ indicates this attack is not demonstrated on the platform (BHB: branch history buffer; STL: store-to-load; MOB: memory order buffer; and MPX: memory protection eXtensions).

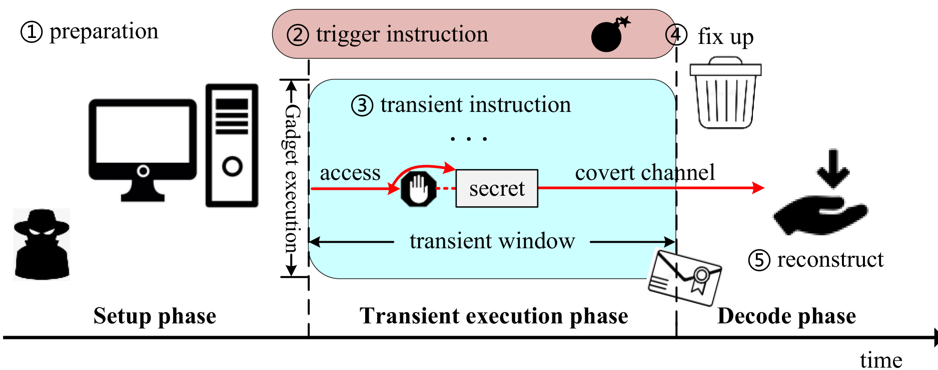

Fig. 2. The steps of transient execution attacks.

- **Setup.** This phase is to determine the attack target and set up necessary conditions (①). In a Spectre attack [88] combined with a Flush+Reload covert channel, for example, the attacker needs to run on the same core with the victim. She then trains the BP and flushes the target cache line. Another case is SMoTherSpectre [15] with a port contention covert channel, in which the attacker needs to share the same physical core with the victim. She then uses **branch target injection (BTI)** to poison the BP, causing it to predict a jump to the smother gadget. The execution time of this gadget sequence overlaps with the execution (timing) of the attacker's sequence to ensure port contention (see more covert channels in Section 5.6).
- **Transient execution.** The phase first executes a trigger instruction (②), which will stall the commit stage so that younger (transient) instructions can execute. The trigger instruction may be an exception or unsolved branch. Then, a chain of transient instructions is executed within the transient window before completion (architecturally visible) of the trigger instruction (③). For Spectre, the transient instructions access secret data and encode it into the microarchitecture state (cache). For SMoTherSpectre, the encoding process is done by choosing whether or not to compete for the same port with the attacker. The process is similar to the transmitter that executes instructions by a Trojan process in covert channels.
- **Decode.** At the retirement of the trigger instruction (i.e., reaching the head of ROB), if there is a misprediction, then the CPU will flush the pipeline to discard the results of transient instructions (④). However, the microarchitectural state is not flushed. For Spectre, the attacker can later reconstruct the secret data by decoding from the state(⑤). For SMoTherSpectre, the secret data are decoded by a concurrent attacker by timing port contention. This decoding process is similar to the receiver that executes instructions by a Spy process in covert channels.

5.3 Speculation-based Attacks

5.3.1 Attack Types. Speculation-based attacks can be divided into control-flow, data, and address speculation attacks. In a control-flow speculation (also called Spectre-type [23]), the CPU predicts the most likely execution path based on the BPs when encountering an unresolved branch. Such prediction can speed up execution of programs by eliminating the waiting latency on each branch. However, the BPs are shared among different processes. Based on this fact, an attacker can mislead the target program to execute a vulnerable code sequence by training a branch. According to the type of training branch, control-flow speculation attacks are divided into three categories: conditional branch, indirect branch, and return branch. The data speculation induces the victim to execute with incorrect data through mis-training multiple times. The only attack (of this type) is based on value predictors [36]. In an address speculation, the processor predicts the dependencies between load and store addresses and then forwards or blocks the outcome ahead of time to avoid waiting. Particularly, only partial addresses are used for judgement of the dependency prediction, this way can reduce storage overhead and improve performance [129]. However, the attacker may exploit such prediction to launch information leakage attacks. According to the number of address bits for the dependency checking, we divide the address speculation attack into the lower 6 bits, lower 12 bits, and 12–19 bits. The first two are to match the corresponding virtual address bits of the load and store while the third is concerning the physical address bits.

5.3.2 Typical Examples. Conditional branch: Spectre-PHT [88] (V1) exploits a branch direction predictor in processors, namely the PHT, which is a table composed of *n*-bit (e.g., 2-bit or 3-bit) saturating counters. Each counter implements a simple finite-state machine, and the number of bits depends on the microarchitecture implementation [13]. Take a 2-bit PHT attack as an example, the attacker first poisons the PHT by mis-training conditional branch to strong "taken" (11) or strong

"not-taken" (00). She then waits for the victim to mis-speculate and load the secret data. Ultimately, the attacker is able to recover the secret via a covert channel. In addition to leakage on loads, Kiriansky and Waldspurger [87] found that a write operation could be used to implement the transient attack, called Spectre V1.1. It leverages the speculative store to overflow a buffer, which likely causes modifications of data or code pointers on arbitrary out-of-bounds addresses (as demonstrated in a typical buffer overflow attack [106]). *Indirect branch:* Spectre-BTB [88] (V2) utilizes the indirect BP inside the processor, namely the BTB, which is used to store the mapping of source and destination addresses of indirect jump instructions. In such an attack, the attacker first executes an indirect branch that has a congruent source address with the target branch's to inject a destination address into the BTB. When the victim program comes into the target branch, it is misled to jump to the injected destination address. This causes the victim to execute the attacker-specified gadget speculatively, leading to the leakage of sensitive information. *Return branch:* Spectre-RSB [89, 104] utilizes the RSB, which stores the return addresses of the last N call instructions. Upon a ret instruction, the CPU pops up the top entry from the RSB to predict the return address. Therefore, an attacker can deliberately cause the misprediction and trick the RSB into using the crafted gadget address.

The address speculation attacks mostly leak on-the-fly data in CPU internal buffers (e.g., the SB). Further filtering is needed after the data are extracted. Therefore, most address speculation attacks are also called MDS attacks. An exception is the Spoiler [77], because it leaks physical address information and does not require filtering sampled data. *Lower 12 bits:* Fallout [22] exploits two features of the SB to leak data. One is to utilize write transient forwarding to forward data from a store to the subsequent load, even if the load address is different from the store's. The other is to exploit the interaction between the TLB and SB to leak data from the store address (store-to-leak). *Lower 6 bits:* RIDL [151] demonstrated a data leakage attack based on the LFB. When the accessed address is not in the **L1 data (L1D)** cache, the CPU will read the data from the LFB speculatively using partial address bits. Consequently, if the attacker synchronizes the LFB well ahead, then it is possible to read the victim's secret data speculatively. *12–19 bits:* Spoiler exploits speculative address dependency resolution logic to leak physical address information. When the lower 12 bits and upper 20 bits of virtual addresses of the store and load are identical, the dependency check will block the load upon consistent 12–19 bits of physical address.

5.3.3 Attack Conditions. To visualize the differences and similarities between various speculation-based attacks, we summarize their attack conditions and the security boundaries they across, as shown in Table 4.

First, we summarize the conditions for control-flow and data speculation attacks as follows:

- **Running on the same CPU core.** The predictors are shared among different processes on each core, so an attacker must run on the same core as the victim's to poison the branch state. This condition can be met by enabling the SMT or running in the same thread.
- **Gadgets in the victim's space.** In a control-flow or data speculation, it is assumed that the attacker can control or trigger the gadget's execution in the victim process. In this case, the victim code must contain exploitable gadgets. For example, a gadget in the shared library can be exploited easily. The victim is then induced to perform the procedures of transient execution. This requirement provides chances for a detector to detect vulnerabilities by searching and matching gadgets, which is discussed in Section 8.1.
- **Mis-training predictors.** The attacker needs to mistrain the predictors ahead of time to induce the victim to deviate from its intended execution path or use incorrect data. This mistraining may take place entirely in the attacker's process. However, SpecROP [14] showed that Intel processors from the eighth generation share the BP between processes but do not

Table 4. The Conditions and Targets of Transient Execution Attacks

Attack type		Condition†							Variant	Boundaries†					
		SC	*MT*	*VG*	*EP*	*MA*	*AA*	*AR*		*pro.*	*pri.*	*VM*	*enc.*	*san.*	*net.*
Speculation-based	**Control-flow**	●	●	●	○	○	○	●	Spectre-PHT [88]	☑	☐	☐	☐	☐	☐
		●	●	●	○	○	○	●	Spectre-BTB [27, 88]	☑	☐	☐	☑	☐	☐
		●	●	●	○	○	○	●	Spectre-RSB [89, 104]	☑	☐	☐	☑	☐	☐
		●	●	●	○	○	○	●	NetSpectre [130]	☑	☐	☐	☐	☐	☑
	Data	●	●	●	○	○	○	●	Value predictor [36]	☑	☐	☐	☐	☐	☐
	Address	●	○	○	○	●	●	○	Spectre-STL [65]	☑	☑	☑	☑	☑	☐
		●	○	○	○	●	●	○	MDS [22, 129, 151, 152]	☑	☑	☑	☑	☑	☐
		●	○	○	○	○	●	○	Spoiler [77]	☐	☐	☐	☐	☐	☐
		○	○	○	○	●	●	○	CROSSTALK [123]	☑	☑	☑	☑	☑	☐
		●	○	●	◐	◐	●	○	LVI [146]	☑	☑	☑	☑	☑	☐
Exception-based	**Memory access**	○	○	○	●	○	○	●	Meltdown-US/PK/BR [23, 100]	☐	☑	☐	☐	☐	☐
		○	○	○	●	○	○	●	Spectre V1.2 [87]	☐	☑	☐	☐	☑	☐
		●	○	○	●	○	○	●	Foreshadow [20]	☐	☑	☐	☑	☐	☐
		●	○	○	●	○	○	●	Foreshadow-NG [161]	☑	☑	☑	☐	☐	☐
	Register read	●	○	○	●	○	○	○	Lazy FP [138]	☑	☑	☑	☑	☑	☑
		○	○	○	●	○	○	●	Meltdown-GP [93]	☐	☑	☐	☐	☐	☐

†SC: same core; MT: mis-training predictors; VG: victim gadget; EP: exception; MA: microcode assist; AA: address aliasing; AR: addressing restrictions. pro.: process; pri.: privilege; enc.: enclave; san.: sandbox; net.: network. ● indicates that the condition is required; ○ indicates that the condition is not required; ◐ indicates that the condition is optional. ☑ indicates that the attacker can bypass the boundary; ☐ on the contrary.

use cross-process entries, thus preventing cross-process BTI. In this case, it is difficult to find the alias address in the BTB to set up the training.

— **Addressing restrictions.** Although transient execution attacks can leak sensitive information across security boundaries, they are susceptible to addressing restrictions [151]. Specifically, the control-flow and data speculation only allow data leakage within the virtual address space accessible from the victim domain. This condition leaves chances for many software mitigations, such as stopping guessing when accessing untrusted pointers [117] or ignoring vulnerable branches [79]. We will describe how this condition can be relaxed in the following.

Second, we summarize the conditions required for address speculation attacks as follows:

— **Running on the same CPU core.** Address speculation attacks exploit per-core private internal buffers, so both the attacker and the victim need to run on the same core. However, CROSSTALK [123] discovers a staging buffer shared by all cores, which is used to implement the first cross-core address-speculative attack.

— **Gadgets in the attacker's space.** Unlike the control-flow speculation, the address speculation can trigger transient execution directly in the attacker's process space and leak secrets during the execution. Two different scenarios can be used to exemplify this behavior. The first is that only the attacker's code is running and the gadget code can be transiently executed at any time (e.g., Spoiler [77]). The second involves concurrent execution of the attacker and the victim (e.g., MDS [22, 129, 151, 152]), where the attacker first synchronizes with the victim and then extracts information from buffers through transient execution.

LVI [146] is an exception, where the attacker first executes a fill gadget to inject malicious values in buffers. The victim then executes an exception or assist gadget to trigger transient execution, and finally encodes the secret data into the microarchitectural state using a disclosure gadget.

- **Triggering MAs.** In an address speculation, the attacker can initiate transient execution by triggering MAs. Common microcode events include paging requests, CPUID, and so on. An exception is Spoiler, which exploits the address dependency checking mechanism. In addition, LVI can use MAs or exceptions to trigger transient execution.
- **Address aliasing.** Address translation is a time-consuming process that involves page table walks and multiple memory look-ups. When the address of a load cannot be translated in time (i.e., the address is not in the TLB or the physical address is unavailable), the CPU predicts whether the load can be executed ahead of time and uses only some lower address bits to match those of a prior store. Address speculation attacks rely on this address matching, also known as address aliasing, to speculatively retrieve on-the-fly data from internal buffers.
- **Addressing restrictions.** As discussed above, the address and data speculation only leak data within the accessible address space (architecturally accessible). Exception-based attacks require the target physical address to be present in the TLB structure. Therefore, these attacks all leak data from a valid (known) address. However, address speculation attacks loosen the addressing restriction. For instance, Fallout allows the attacker to access addresses that point to invalid page frames. Thus, the attack still works even if the mitigation of **page table entries (PTE)** inversion [31] is enforced. RIDL, ZombieLoad, and CacheOut further relax the requirements for the translation data structure, so the attack target no longer relies on a specific address.

5.3.4 Security Boundaries. The control-flow and data speculation attacks can leak arbitrary data from the victim's accessible space, so they can cross process boundaries. References [88, 104] demonstrate Spectre attacks cross the SGX enclave boundary. NetSpectre [130] demonstrated the first generic remote Spectre attack on a device that is not running any potential attacker-controlled code. It trains the target branch by sending two types of network packets (tagged as valid or invalid) to the victim's network interface. The address speculation presents a unique characteristic: the ability to induce data leakage that is agnostic to address translation. This allows attackers to access on-the-fly data passing through CPU buffers without the security checks. As a result, such attacks can typically cross any security boundary. It is important to note that Spoiler attacks [77] leak some unknown physical address bits and thus do not cross any security boundary to disclose data.

5.4 Exception-based Attacks

5.4.1 Attack Types. Exception-based attacks (also called Meltdown-type [23]) leak architecturally inaccessible data by exploiting illegal dataflow from exception instructions. Based on the type of exception, they can be categorized into permission violation and illegal instruction/operand. Depending on the trigger conditions, exceptions are further divided into memory access exceptions and register read exceptions. The former also refers to permission violation exceptions that occur when accessing the memory system, which involves the TLB, caches, and PTEs. The results of an unauthorized load are still forwarded to subsequent transient operations that may encode the data before eventually raising an exception. Register read exceptions occur when accessing certain registers (such as FP registers [138] and MSR [93]). Before the exception is handled, attackers have accessed the data in the registers and encoded it into the microarchitectural state using covert channels.

5.4.2 Typical Examples. In Meltdown [100], an attacker accesses kernel memory from user space. When an unauthorized kernel address is dereferenced, it leads to a #PF. However, before the fault becomes visible in the architecture, the kernel data has already been forwarded to subsequent transient instructions. In Foreshadow [20], the attacker exploits the technique of page aliasing to create an additional virtual-to-physical mapping for the physical address of a secret within an enclave. The attacker then uses `mprotect` to clear the "present" bit of PTE associated with the secret in the TLB, ensuring that subsequent accesses will trigger a #PF. In this case, the CPU should immediately abort the address translation. However, since the L1 data cache is indexed in parallel with address translation, the physical address field (i.e., frame number) of a PTE may still be forwarded to the L1 cache. This allows the attacker to temporarily gain access to the enclave's physical address and its associated data using page aliasing. Variant 3a [93] raises a #GP through unauthorized access to privileged system registers. Lazy FP [138] proposed an attack against the lazy state switching mechanism, where the CPU does not clear the FPU registers and instead marks them as "not available" when a context switch occurs. Assuming the FPU register is owned by the victim process, when an attacker attempts to access the register, it will generate an #NM fault to tell the OS that the FPU is disabled. However, as the CPU has speculatively used the data from the FPU before retirement of the fault instruction, it likely causes data leakage. Note that while most Meltdown-type attacks are caused by out-of-order execution microarchitectures, an in-order pipeline can also be exploited [148]. The only requirement for Meltdown-type attacks is to forward data to other dependent instructions when the instruction triggers an exception.

5.4.3 Attack Conditions. We summarize the conditions for exception-based attacks below:

- **Running on the same CPU core.** In exception-based attacks, the attacker and the victim may run concurrently on the same core (e.g., lazy FP [138]), or only the attacker is running and can choose any core (e.g., Meltdown-GP [93]).
- **Gadgets in the attacker's space.** An attacker can directly trigger transient execution to obtain the secret in an exception-based attack. That means the operation of transient execution can be done purely by the attacker, i.e., exploiting the gadget in the attacker's own process space. This condition makes the type of attacks more flexible and detection more difficult, as the victim is only responsible for accessing the secret without the need for gadgets that transfer the secret.
- **Triggering exceptions.** An attacker first needs to trigger an exception for exception-based attacks. Due to deferred exception handling by design, the attacker is able to continue transient execution on an unintended access path.
- **Addressing restrictions.** As mentioned above, The control-flow/data speculation attack can only leak accessible target addresses in the victim's space. The exception-based attack loosens the restriction and is able to access data from architecturally inaccessible addresses. An important requirement for both Meltdown and Foreshadow is that the target physical address is present in the loaded address translation data structures. Specifically, Meltdown requires the privileged address that is going to be accessed in a transient window to be mapped to the virtual memory map of the attacker's process. Foreshadow, however, requires the physical address of the enclave to be mapped into the TLB, even if the mapping entry is marked as "no present." LazyFP is an exception, because it leaks data from stale FP registers without performing a tag check before retrieving data. In this sense, it is similar to address speculation attacks, allowing access to old data without any checks.

5.4.4 Security Boundaries. Exception-based attacks can expose architecturally inaccessible data, so they can cross privilege boundaries. Spectre V1.2 [87] can bypass page table-based read

and write access permissions within the current privilege level. The ability to transiently overwrite read-only data allows bypassing software sandboxes that enable hardware-enforced read-only memory. Foreshadow [20] targets Intel SGX enclaves while Foreshadow-NG [161] extends it to cross-process and cross-VM leaks. Variant 3a [93] mainly leaks data from some special registers. Lazy FP [138] transiently reads data from the FPU that has not been cleared during context switches, thereby causing data leakage across arbitrary security boundaries. Note that MicroScope [137] does not attempt to leak data directly but to denoise MTSCAs. The victim only needs to run once, but the attacker can trigger a #PF to replay the execution results.

5.5 Attacks on Commercial Platforms

Currently, researchers pay most attention to Intel processors, partly just due to their high availability, as shown in the last column of Table 3. The table also lists known attacks targeting processors from other vendors, including AMD [6, 23], ARM [23, 93], and RISC-V [45, 49, 50]. Spectre attacks, as observed from the table, work on all platforms, because the BP is a necessary foundation for all processors. Other types of transient execution attacks are influenced by specific microarchitecture implementations.

5.6 Leakage Channel

Many covert channels in Table 2 can be used for leakage channels of transient execution. Some common channels for transient execution include execution ports [15], AVX unit [130], FP div unit [46], PHT [29], and caches [20, 22, 27, 88, 89, 100, 104, 129, 138, 144, 146, 151, 161]. Among them, the cache is the most commonly used due to the following advantages. (1) Persistent shared resource: Unlike per-core private branch state and volatile execution ports, the cache state can be persistently shared among different processor cores. This sharing means that attackers can infer the behavior of other active entities by monitoring the shared cache state without the need for concurrent interaction with the target entity. This makes cache-based channels more flexible in leaking information across security boundaries. (2) The state setup and its measurement: An attacker can use the `CLFLUSH` to clear the cache state or construct an eviction set to evict target cache lines. The measurement process involves either timing the load/store operations or monitoring cache hit/miss events. (3) The time difference: There is a significant time difference between cache hits and misses, which may take up to 150–300 cycles. Therefore, compared to other channels [15, 29, 46], cache-based channels are more robust against noise that can impact the execution time of instructions, such as preemption or contention for execution resources by other processes.

5.7 Gadget Types

As depicted in Figure 3, we present potential gadgets used for well-known transient execution attacks. It is essential to note that we use cache-based channels as an illustration, and gadgets for other covert channels [130] may vary. We categorize these gadgets into four types based on how secret data are accessed. The first type involves *index*, where the attacker mis-trains the target branch with the controlled input x, causing the victim to speculatively access secret data through the out-of-bounds index of $arr1[x]$. Canella et al. [23] introduced additional variants of Spectre-PHT gadgets, such as *prefetch*, *compare*, and *execute*. The second type is known as *aliasing*, where the attacker exploits overlapping addresses of loads and stores to access previously sensitive data from internal buffers. The third type of gadget, termed *jump*, speculatively jumps to secret locations by injecting or polluting the destination address of the target branch. The fourth type is *access*, where the attacker directly accesses unauthorized secret data (such as kernel and system register data) and leaks it during exception handling.

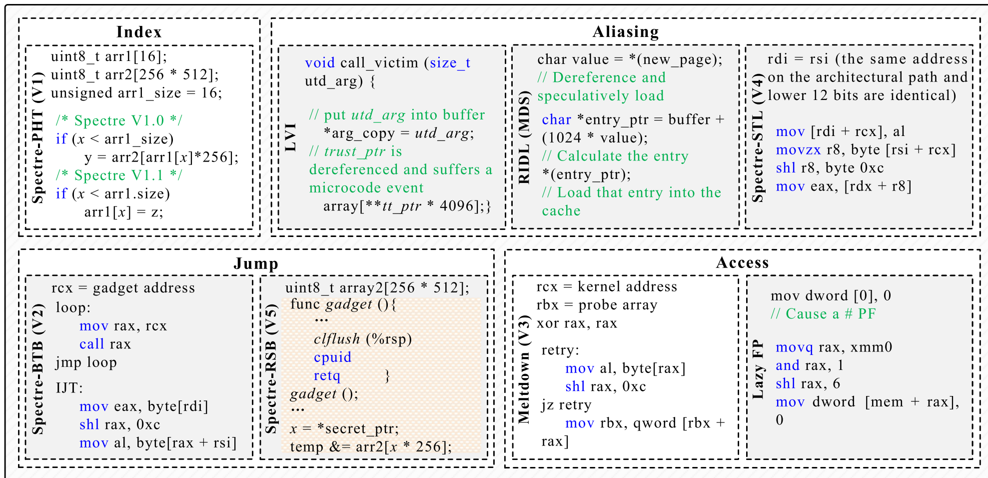

Fig. 3. The gadgets of transient execution attacks.

5.8 Limitations of Attacks

Transient execution attacks have two main limitations, i.e., the speculative window size and the maximum latency for branch resolution or microcode event processing.

The speculative window size. Modern high-performance processors can tolerate long latency by buffering unaccomplished (unretired) operations in the ROB. The CPU then proceeds to execute the following instructions. Note that these instructions are not interdependent. The number of instructions that can be executed in a speculative window is limited by the size of the ROB. Current processors support a large speculative window, e.g., Intel Skylake contains an ROB of 224 entries, which can hold about 200 instruction codes. This means that the attacker needs to complete the malicious functions within the instruction window.

The maximum latency. The time of branch/address resolution or microcode event processing decides how many transient instructions can be executed. However, while loading one cache line from DRAM theoretically takes around 150–300 cycles [82], the actual speculation window may be much longer in terms of cycles. Attackers can utilize/create pointer-chasing gadgets that have a control-flow dependency on a chain of pointers. These gadgets typically give attackers extremely large speculation windows (with even thousands of cycles) to access secrets and leak the corresponding values.

6 DEFENSES AGAINST CONVENTIONAL ATTACKS

MTSCAs exploit the time difference introduced by microarchitecture components to transmit secret data, which poses serious security risks to local and cloud environments. Various encryption algorithms, VMs, browsers, and even TEEs are affected. For this reason, the academic and industry have been devoted to how to mitigate such attacks. In this section, we discuss the published defenses against cache-based attacks and illustrate their advantages and limitations.

Recall that we have introduced various attack conditions for cache-based attacks, such as HPT, cache control instructions, and shared resources. Based on these conditions, prior studies have proposed many countermeasures, which include interfering with the measurement time and resource sharing, eliminating HPTs, using constant-time instructions, and so on. Different defense methods often mitigate against a specific attack condition. However, by undermining the necessary attack

conditions summarized (see Table 1), they can be divided into two categories in essence: hiding timing differences and limiting resource interference.

6.1 Hiding Time Difference

Due to limited cache size, the data used for program execution cannot be entirely put into the cache, thus inevitably inducing cache hits and misses. The macroscopic manifestation is the varying execution time of programs or instructions. Thus, a straightforward method to mitigate this type of attack is to hide the measurable time difference from attackers.

6.1.1 Eliminating Timers. Timing SCAs typically require HPTs for attackers to differentiate microarchitectural states. Therefore, *Cloudflare Workers* [30] utilize a modified JavaScript sandbox that disables all local timers and known primitives for constructing timers [53]. In this scenario, an attacker can start a timing measurement by sending a network request to a remote server. But this way obviously has a lower resolution. Thus, the attacker has to rely on amplification techniques to amplify the latency between a cache hit and miss [131]. Nonetheless, some attacks [38, 60, 84, 136, 176, 181] that do not rely on timers (as discussed in Section 4.3.3) still cannot be mitigated through this method.

6.1.2 Constant-time Operation. In an SCA, if the victim takes different actions depending on the secret value, then it could cause measurable time differences for the attacker. Therefore, some defenses impede such an attack by fixing the execution time of different secret values.

Intel [75] pointed out that most SCAs can be mitigated by using the constant-time principle for all code interacting with secrets. It suggested that software developers scrutinize runtime, code, and data access patterns of programs, all of which should be secret independent. The MbedTLS library [9] used symmetric execution paths to balance branches, where both paths of a branch should execute the same function so that there is no observable time difference on different paths. Andrysco et al. [8] discovered that the runtime of FP `ADD` and `MUL` instructions varies by two orders of magnitude depending on the operands. For this reason, they designed a fixed-point constant-time math library (LibFTFP) to alleviate the FP data timing channel. LibFTFP uses three simple strategies to support its constant-time operations: (1) calculating all values on different code paths; (2) ensuring that no data-oriented branches are used in the code; and (3) using input-independent basic integer operations. All above operations take the same time on the fast and slow branch paths. Rane et al. [124] further proposed a compiler-based approach to provide fixed-time FP operations by a SIMD channel in the ×86 SSE and SSE2. Gruss et al. [58] suggest modifying the `CLFLUSH` instruction to be a constant-time instruction so that the execution time is constant regardless of cache hits or misses.

Although it works to defend against SCAs through avoiding branch execution, memory access, and instruction execution closely related to secrets, constant-time implementations cannot fully prevent data leakage. For instance, Frontal attack [118] shows that the execution time of an instruction depends on not only preceding or following instructions but also its virtual address. This is because the different addresses have different offsets in the instruction fetch window. Thus, the Frontal attack successfully exploits the relationship of execution time and virtual address to compromise the Intel SGX. In addition, the constant-time operation introduces high overhead. For example, *Escort* [124] causes up to 32.6× overhead on the SPECfp2006. Furthermore, the constant-time is also an extremely labor-intensive solution, which requires software developers to carefully create algorithms for each input that should be executed in constant time.

6.1.3 Noise Injection. The method makes the results of the attacker's measurement futile. However, it does not completely remove the channel but instead makes the attacker collect more samples to make a confident guess about the transmitted bits.

Table 5. Defenses against Cache Side-channel Attacks

Method	Type	Example	Target	Overhead
Cache partitioning	Software partitioning	Chameleon [135] STEALTHMEM [85] CacheBar [186]	Eviction-TS and control-TS	∼3%–12% ∼5.9% <25%
	Hardware partitioning	CATalyst [101] Cloak [54] HYBCACHE [37]	Eviction-TS and control-TS	∼0.5%–0.7% ∼0.8%–1.2%, ∼3.5%–5%
		DAWG [86]	Eviction-based and control-based	<2%
Randomization	Index randomization	SCATTERCACHE [162]	Eviction-TS and control-TS	2% (5%)†
		CEASER [121] CEASER-S [122] PhantomCache [140]	Eviction-TS	∼1% (<24B)† <1% (<100B)† ∼0.5%–1.2% (0.5%)†
	Code randomization	DSD [33]	Eviction-TS	∼25%–110%

†Indicates storage overhead.

TimeWarp [105] disturbs timing channels by using the idea of fuzzy time. It modifies the implementation of the ×86 RDTSC timer so that the program always executes a predefined and randomly generated epoch-size time. In addition, TimeWarp provides a hardware monitoring mechanism to detect software clocks, such as inter-thread communication via shared variables. However, the effectiveness of this scheme relies on two systematic assumptions: (1) external events cannot provide sufficient resolution to measure microarchitectural events, and (2) software clocks are unavailable. Trilla et al. [143] propose the time predictable secure cache, which reduces the time predictability of SCAs by perturbing the access time of caches. Wang et al. [157] propose the MemJam protection framework that utilizes back-end DRAM refreshes as a free noise source to eliminate temporal correlations between the applications of the victim and attacker. It intentionally introduces interference to the shared memory controller to prevent a malicious attacker from snooping on the memory access patterns of sensitive programs. The Reuse-trap [43] counts the reuse distance in each cache set, i.e., the number of cache misses suffered by a non-victim process between two consecutive cache misses of the victim. The counter of reuse distance is sent to a scorer to identify potential adversaries. When an exception pattern of reuse distance is detected, the scorer forwards the cache set index and the corresponding process ID to the prefetcher. The prefetcher then prefetches critical memory lines into the cache, thereby fuzzing the access latency that is observable for an attacker.

6.2 Limiting Resource Interference

Another necessary condition of SCAs is intrinsic resource sharing in processors. This causes the cache state of the attacker and the victim to interfere with each other. Therefore, this type of attack can be obstructed by limiting resource interference. An attacker can exploit the resource interference under two conditions: (1) shared memory pages/cache states among mutually distrusting processes and (2) deterministic cache index and fixed set associativity. For condition (1), it can be blocked by enabling cache partitioning for each security domain, and for (2), one primary scheme is to randomize cache indexes, which disturbs the eviction set prepared by an attacker. We summarize current defenses based on the idea of limiting resource interference, as shown in Table 5.

6.2.1 Cache Partitioning. Modern processor caches are typically set-associative, i.e., they are organized into multiple sets and ways. Therefore, cache partitioning can divide cache sets or ways among processes, which have been implemented in software and hardware methods.

Software partitioning. The method disrupts the cache state of covert channels by disallowing data of different protected domains to occupy the same cache sets. Shi et al. [135] designed a dynamic page coloring solution to limit SCAs. They utilized different colors to tag the pages, which are no longer assigned to other processes for safety-critical operations. STEALTHMEM [85] manages a set of locked cache lines that are never evicted from the cache for each core. Moreover, these lines are efficiently multiplexed so that each VM can load its own sensitive data into them. Zhou et al. [186] designed a memory management subsystem named CacheBar, which provides two techniques for defending against access-driven SCAs: (1) It replicated the page when accessing a physical page shared by multiple security domains to prevent Flush+Reload, and (2) it enforced cacheability management of pages to restrict an attacker from occupying the whole cache set, thus mitigating the Prime+Probe. Software partitioning allows communication between different security domains without breaking cache coherency. However, cache coloring at page granularity is not well compatible with huge pages, which may reduce the number of TLB hits, thus bringing a significant performance overhead. In addition, some privileged entities may need to move huge data blocks in memory when allocating cache sets. Since the allocation of cache sets is associated with the physical address's. For example, when a 1/8 cache space is allocated to a protection domain, the same 1/8 physical address space must be allocated to the corresponding process.

Hardware partitioning. The method allows for more fine-grained way partitioning. As these defenses require modifications to either instruction [58] or hardware architecture [10], it is not easy to deploy them in practice. Therefore, an alternative is to reuse deployed hardware technologies. For instance, CATalyst [101] uses leverage cache allocation technology to divide secure partitions that contain cache-fixed secure code or data pages. The non-secure partitions can be used by any application freely. Cloak [54] makes use of the intel TSX to prevent the attacker's adversarial observations on sensitive code or data by cache misses. During a TSX transaction, once the security-sensitive data cache is evicted, the transaction will be terminated. Constant-time operations have been shown to be vulnerable to CacheBleed attacks [175]. To address this, Yu et al. [177] proposed data oblivious ISA extensions called OISA to provide side-channel resistance. OISA employs the way partitioning technique to implement **oblivious memory partition (OMP)** as an isolated region in the L1D cache. When an instruction accesses the OMP, all concurrent cache accesses are halted to avoid cache contention [175]. HYBCACHE [37] presents a generic mechanism for a flexible partitioning of set-associative caches. It can be configured to apply side-channel-resistant cache behavior for isolated execution domains while providing regular cache behavior, capacity, and performance for the non-isolated execution domains. DAWG [86] endows each hardware thread with a notion of domain ID, where both cache hits and cache line replacements are limited by the assignment of protected domains. Based on this operation, cache-related metadata, such as the cache coherence and replacement states, can be also partitioned safely. DAWG is the first strategy to mitigate metadata state-based SCAs.

6.2.2 Randomization. The deterministic cache indexes and fixed set associativity allow an attacker to pre-calculate eviction set of the target address, which may cause eviction-based SCAs. Therefore, it is effective against such attacks by randomizing the address mapping to cache sets. The randomization scheme can be categorized into index randomization and code randomization.

Index randomization. By randomizing the fixed mapping relationship between memory lines and cache lines, the attacker cannot determine the cache set of the target address. In other words, it is impossible to construct an eviction set in a limited time interval. SCATTERCACHE [162]

Table 6. Defenses against Transient Execution Attacks

Method	Target	Example	Overhead
Isolating shared states	Control-flow speculation	Zigzagger [66]	~18% (code size: 1.2 times)
		BRB [154]	~3.5%–5.5%
		XOR-BP [185]	~1.5%
		LS-BP [26]	<3%
Limiting speculation	Control-flow and address speculation	Retpoline [51]	~5–10%
		`FENCE` after each branch [169]	88%
		SLH [24]	29%~36%
Disabling un-authorized access	Control-flow speculation, memory access exception	WebKit [117]	<2.5%
		Site Isolation [125]	Memory overhead: ~9%–13%
		KAISER [55]	~1–10%
Invisible state changes	Speculation-based, exception-based	InvisiSpec [169]	22% (area overhead: 3.5%)
		CleanupSpec [128]	5.1% (storage overhead: ~1KB)
		SafeSpec [83]	−3% (area overhead: ~2%–17%)
		SpectreGuard [47]	~8%–20%
		NDA [160]	~10.7%–125%
		STT [178]	~8.5%–14.5%

proposed that each address should be combined with a key and SDID as inputs of the mapping function to generate a cache set index. Thus, it is able to eliminate the fixed cache set coherency required for eviction-TS attacks. In addition, it can also alleviate control-TS attacks such as Flush+Reload, because it implements security domains in the mapping function and prevents any cache line from being accessed across security boundaries. CEASER [121] uses low-latency-block-ciphers to translate the physical address into the encrypted cache line address. It initializes and randomizes the encryption key on every reboot and executes dynamic remapping to improve robustness. Qureshi et al. [122] discovered that the remapping rate of CEASER may cause an impractical overhead. For this reason, they proposed an improved method called CEASER-S, which divides the cache ways into multiple partitions with different keys. This design greatly enhances the robustness of CEASER, because an attacker must evict cache lines from many possible locations to construct the eviction set. PhantomCache [140] proposed a localized randomization technique to avoid inefficient dynamic remapping, which maps a memory address within only a limited number of cache sets. The small randomization space provides a faster set searching, as it allows checking all possible cached locations in parallel during searching for the target address.

Code randomization. The code randomization means that the code and data of a program are randomized during execution to guarantee that the same copies are different at the machine instruction level. This approach has the advantage of hiding observable execution features. DSD [33] generates diversified replicas and randomly and frequently switches between these replicas at runtime. Every replica differs due to the insertion of `NOP` instructions, permutation of function or basic block layout, and randomization of register assignments. A program can dynamically select control-flow paths to generate different results, which hence disrupts the attacker's measurement.

7 DEFENSES AGAINST TRANSIENT EXECUTION ATTACKS

In this section, we discuss the published defenses against transient execution attacks. By removing the necessary conditions for this type of attack, the defenses can be divided into the following categories, as shown in Table 6.

— **Isolating shared states.** It keeps the microarchitectural state of both the attacker and the victim from interfering with each other.
— **Limiting speculation.** It selectively restricts transient execution by looking for certain locations where secret data are prone to leakage.
— **Disabling unauthorized access.** It prevents attackers from directly accessing unprivileged or sensitive target addresses through transient execution.
— **Invisible state changes.** It makes the microarchitectural state invisible to the attacker.

7.1 Isolating Shared States

The cache partitioning schemes discussed in the previous section can also mitigate transient execution attacks that utilize cache covert channels. However, to avoid repetition, we will not elaborate on them in this section. Instead, this section focuses on other approaches for isolating microarchitectural states. For example, in a control-flow speculation, any context switch may be a potential leak point, as an adversary can exploit that to manipulate the BP. Therefore, isolating the BP's state among different processes is an effective way to mitigate such attacks. Intuitively, clearing BP state [41] for each context switch can prevent attackers from manipulating or snooping on the branch execution. However, frequently refreshing the entire states (at every context switch) significantly deteriorates prediction accuracy, thus drastically decreasing performance.

Paverd et al. [66] replaced a set of conditional branches with many indirect jumps at compile time, because it is much more difficult to infer the state of an indirect branch than that of a conditional branch. Furthermore, they randomized the code at runtime to hide the targets of indirect branches. Thus, an adversary cannot mis-train and perceive the state of the BP. BRB [154] is an advanced hardware implementation that provides a dedicated branch reservation buffer to retain the BP state for each context switch. This reduces cold-start effects and improves transient accuracy. Despite attempts to limit hardware costs, it is generally impractical for the BRB to assign solely a branch history table for each thread and privilege space. Tagging shared structure entries is another way to isolate shared states. BTB entries can be tagged by a process context identifier like TLB entries [91] to prevent cross-process BTI. Intel [70] also proposes adding ASID (address-space identifier) information for each PHT entry, although it may come at a relatively high cost.

Furthermore, physical isolation is inadequate for mitigating all control-flow speculative attacks, since the branch state is still multiplexed in time by different threads. To this end, XOR-BP [185] proposed encrypting the branch target addresses stored in the BP at each context switch. The idea is to generate a random number for each thread context and XOR the target address with it when updating the branch state. Although this method has eliminated malicious branch training among different processes, it cannot prevent leakage due to misprediction in the same address space, such as Spectre-PHT. On Skylake and younger microarchitectures, Intel [79] suggested using the RSB padding. Concretely, when the processor executes a `RET` instruction in an empty RSB, it will perform speculation based on the BTB, which may lead to the Spectre V2 attack. To prevent the RSB from returning to the BTB due to underfilling, Intel proposed using a benign gadget address to fill the RSB at every context switch. In addition, some ISA-level solutions can also isolate the branch prediction state. For instance, Intel and AMD extended the ISA to control indirect branches, whose addition to the ISA consists of three controls [74]: indirect branch restricted speculation, single thread indirect branch prediction, and indirect branch predictor barrier.

7.2 Limiting Speculation

In the control-flow and address speculation, the attacker induces the victim to speculatively execute along a wrong path. Therefore, an effective way to mitigate this type of attack is to limit the speculative execution, which can be achieved with the aid of existing ×86 instructions. An

example is the FENCE instruction that forces sequential program execution, i.e., it guarantees that the preceding instructions must be executed and retired completely before any subsequent instruction proceeds. Obviously, disabling all speculation with the FENCE can effectively prevent both types of attacks. However, it incurs significant performance penalties [72].

Google proposed Retpoline [51] that replaces indirect branches with return instructions. This approach makes indirect branch instructions always speculate into an infinite loop through the RSB. It then performs some invalid operations until the target address is resolved correctly. Although Retpoline can prevent branch mistraining, it only targets BTI attacks and is ineffective against other attacks such as Spectre-PHT. In addition, Intel mentioned that Retpoline might bring false positives in future CPUs with the deployment of control-flow execution technology. SLH [24] used branchless code to make sure that the control-flow paths of loads are valid. Concretely, it transforms the code at the compiler level and adds data dependencies on conditions. If it turns out to be a mis-speculation, then the pointer is zeroed to prevent speculative execution. This approach requires hardware support to allow branchless implementation and updates for unpredictable conditions, which is only available in LLVM with ×86.

7.3 Disabling Unauthorized Access

Control-flow speculation and memory access exception attacks can only leak data at specific addresses. In these attacks, the attacker first needs to know the target address of secret data, so that she can divulge the sensitive data through unauthorized transient access. Thus, hiding or isolating the target address may be a targeted and effective mitigation for eradicating data leakage.

WebKit [117] proposed two approaches to mitigate Spectre-type attacks. One is index masking, which binds the mask to array bound checks. While this approach is not a complete fix for out-of-bounds accesses, it defines a maximum range of the out-of-bounds violation. Another is pointer poisoning. It uses a random number generator to pick a large poison value (e.g., 2ˆ40) at compile-time, and then XORs the value with the pointer. Based on these operations, any access to the poisoned pointer is likely to hit unmapped memory. Thus, misprediction in a branch-type check may cause an error of unmapped pointer. Moreover, it is almost impossible for an attacker to guess this poisoned value in advance, thus protecting the pointer from being misused. Google proposed Site Isolation [125], which isolates each site in different processes and limits any process obtaining sensitive information from other websites. Therefore, even if the attacker has privileged memory reads, she can only access data from its own process. Gruss et al. [55] proposed KAISER, a practical isolation in which kernel address information is never mapped to user space. This makes most kernel addresses invisible to the user-space code, thus preventing information leakage across privileges by Meltdown-type attacks. However, in addition to the performance impact, the KAISER scheme has one important limitation: some privileged memory locations must be mapped to user space on the ×86 architectures. This inevitably leaves a small fraction of kernel addresses accessible from user space, which may be exploited to leak privileged data (e.g., Meltdown).

To make the target address inaccessible, Intel [71] provides a microcode update to allow the OS to assist in clearing the PTE of the target address or setting it to non-present physical memory. Thus, it can mitigate the Foreshadow attacks. Specifically, in the context of the OS, it ensures that vulnerable PTEs only point to specifically selected physical addresses, such as an address outside the available cache or one that does not contain secrets; In Intel SGX [69], it ensures that different authentication keys are derived according to the hyperthreading enabled or disabled; In the **system management mode (SMM)** [71], it checks all logical cores to make sure that no non-SMM software is running when the data in SMM is located in L1 cache; in the virtual machine monitor [71], it guarantees that no other untrusted threads are running on the same core.

7.4 Invisible State Changes

After illegal access to the secret, the transient execution attack attempts to transfer it from a confined environment to outside via a covert channel. The attacker then can recover the secret data by observing the microarchitectural state changes. Consequently, a basic idea of the defense against such attacks is to figure out how to make microarchitectural state changes invisible to attackers.

One approach is to isolate the state changes caused by speculative execution and hide them from the cache hierarchy. InvisiSpec [169] stores unsafe speculative load data in a new speculative buffer instead of reading it into the cache. Additionally, only when a speculative load is considered safe does InvisiSpec send it back to the memory system, making it visible to other processes in the OS. CleanupSpec [128] proposed an “undo”-based scheme to secure speculative execution. This way allows all loads to modify the cache speculatively. Moreover, CleanupSpec activates corrective action to clean up all cached state changes caused by illegal transient loads. SafeSpec [83] introduced a shadow structure that separates the speculative state from the retired state of instructions.

A second approach is to restrict unsafe speculative instructions from being propagated to subsequent instructions. SpectreGuard [47] marks sensitive memory blocks as non-speculative memory regions, in which the result of speculative execution is not immediately forwarded. Instead, it is retained in the ROB until all preceding branches have been resolved. Weisse et al. [160] proposed **non-speculative data access (NDA)**, which limits the propagation of speculative data by blocking the tag broadcast of unsafe instructions. Although NDA is flexible enough with multiple policy variants, it cannot benefit from high performance and security at the same time. Moreover, it also fails to recognize all covert channels. To this end, STT [178] proposes a new classification abstraction for all covert channels, including explicit and implicit channels. STT issues loads as long as the branch is resolved in the correct prediction, so it can improve the performance of NDA schemes.

However, invisible state changes do not completely prevent against transient execution attacks that utilize cache state. Behnia et al. [11] introduced speculative interference attacks, which indirectly monitor timing effects caused by the interaction between secret-dependent instructions and old non-speculative instructions. They found that invisible state change schemes cannot protect resource usage patterns, execution time, or branch prediction. Thus, although an attacker (monitoring the cache) is incapable of directly discerning the execution operations associated with secrets. Instead, she can indirectly observe timing disparities through the resource contention between secret-dependent speculative instructions and the non-speculative portion of the pipeline (called interference targets), thereby leaking secret data. GhostMinion [3] introduced strict ordering that allows information (and side channels) to flow in any case, but speculative operations that are not committed architecturally are never leaked to those committed. It aims to mitigate transient execution attacks, including speculative interference. Yang et al. [171] proposed Pensieve, a security evaluation framework against early microarchitectural defenses targeting speculative execution attacks that is capable of precisely capturing timing variations through exploiting resource contention and microarchitectural optimizations. They used Pensieve to assess a range of invisible state change defenses and discovered a variant of speculative interference attacks that leveraged unrelated instructions in program order to bypass GhostMinion.

8 DETECTION SCHEMES FOR MTSCAS

The deployment of defenses discussed throughout the article may incur significant overhead. Therefore, researchers usually recommend detection before defense. These detection techniques aim to identify either the location of vulnerabilities or attack behaviors. As a result, we can categorize them into two types: code detection and behavior detection.

8.1 Code Detection

In control-flow speculation attacks, the exploitable gadgets are in the victim's code space. Therefore, a variety of static and dynamic analysis methods have been proposed to find gadgets in target programs. We will discuss them in detail as follows.

Static analysis. OO7 [156] utilizes control-flow extraction, taint analysis, and address analysis to detect tainted conditional branches and speculative memory accesses. It then prevents control-flow speculation by inserting a few `FENCE` instructions. The detection results of OO7 depend on the integrity of the control-flow-graph extraction. Spectector [61] endeavors to trace all memory accesses and jump targets along different execution paths to confirm that no gadget is involved in the target program. To do this, the program has to run twice. For the first time, it records some memory access traces where no misprediction occurs. Next time, it simulates the mispredicted branch paths and records a certain number of memory access traces. If the Spectector detects a mismatch between the two traces, then it reports a gadget found.

Dynamic analysis. SpecFuzz [110] simulates speculative execution at compile time by forcefully executing the code paths that can trigger misprediction. It then uses an integrity checker (e.g., AddressSanitizer [132]) to check for out-of-bounds memory accesses (e.g., buffer overflow) during the simulation. SpecTaint [120] simulates speculative execution at runtime to capture dataflow patterns along speculative paths. In addition, it deploys a semantic-based detector to discover exploitable Spectre gadgets. CSF [141] injects `FENCE` instructions between the control-flow and payload with the assistance of a context-sensitive decoder in the front-end. It can dynamically and surgically insert the `FENCE` into potentially malicious code without recompilation or binary translation. SPECCFI [90] verifies dangerous speculations through forward-edge **control flow integrity (CFI)**. It makes sure that the control-flow instructions can jump to legitimate destinations, thereby preventing Spectre-BTB attacks. Meanwhile, it augments a unified shadow call stack for enforcement of backward-edge CFI, which can defend against Spectre-RSB attacks.

8.2 Behavior Detection

Zhang and Reiter [184] analyzed the impact of periodically flushing the L1 instruction and data caches in kernel-level threads and suggested automatically switching to a defensive mode when malicious preemptive behavior is detected. Gruss et al. [58] suggested using **performance monitoring units (PMUs)** to detect LLC SCAs. Similarly, Zhang et al. [182] proposed a system that utilizes PMU counters to monitor cache misses or hit rates when executing encryption operations, thus detecting cache-based SCAs. However, detection relying on PMU counters may suffer practical issues. First, the number of events available for counters is limited on the hardware platform. Second, the counters are accessible by software, so attackers may manipulate them and adjust their attacks to evade detection.

CC-Hunter [28] detects the presence of timing covert channels by dynamically tracking conflict patterns on shared processor components. Cyclone [63] leverages a characteristic shared by cache contention channels—cross security domain cyclic interference—to detect microarchitectural attacks. PerSpectron [107] designs a hardware-based neural predictor that uses perceptron learning to identify and classify microarchitectural attacks. Further, it can actively mitigate attacks by triggering appropriate countermeasures. EVAX [4] significantly reduces the overhead of existing defenses by enabling the corresponding countermeasure only when an attack is detected. For example, it reduces the overhead of InvisiSpec to 1.26% and the overhead of Fencing to 3.45%. SPOILER-ALERT [34] identifies attack behaviors based on three traits of Spoiler: (1) The attacker relies on a large number of store instructions with the same address offset, (2) the SB is filled up with those store instructions, and (3) the store instructions on each sequential

page are iteratively forwarded to the CPU. According to these traits, SPOILER-ALERT leverages a cuckoo filter module to dynamically screen buffer addresses, thus detecting Spoiler attacks in real time.

9 CHALLENGES AND FUTURE DIRECTIONS

9.1 Key Challenges

MTSCAs and defenses have been the focus of today's processor security. In this section, we summarize several key challenges of microarchitectural security for future research.

(1) *The CPU vendor's dilemma.* A fundamental way of mitigating MTSCAs is to redesign the corresponding hardware component, which only incurs low performance cost. However, the opposite is often the case today: Hardware manufacturers remain overly focused on performance improvement and continuously integrate new components and optimization techniques into the microarchitecture design. Obviously, the security factor is often overlooked in their design choices. Consequently, it is a separate challenge for the research community to draw the CPU vendor's attention to the security designs.
(2) *Tradeoffs between security and overhead.* Although restricting transient instructions is able to thwart most speculative execution attacks, it often suffers from a high performance overhead. Further, a targeted mitigation measure can effectively reduce the overhead, but they may mitigate only one type of attack. Thus, it is quite difficult to strike a balance between security and overhead.
(3) *The need for complex reverse engineering.* As processor vendors do not release their architecture designs in detail, it is extremely hard to have a full understanding of the working mechanism of microarchitectural components. However, discovering new microarchitectural vulnerabilities and designing secure microarchitectures have to rely on those detailed information. Thus, it is an important technical challenge for researchers to develop efficient reverse engineering approaches based on publicly available component information.

9.2 Future Research Directions

MTSCAs pose the risk of leaking arbitrary memory data from the privileged mode or victim memory. In response to those hazards, various defenses have been proposed successively, but many issues remain unresolved. Next, we point out some research directions from two perspectives: microarchitectural timing side-channel attacks and their countermeasures.

We propose the following future directions in the research of MTSCAs:

(1) *Enhancing attack metrics.* Boosting existing attacks with novel/well-known exploit primitives is a very interesting research direction. Researchers can improve the effectiveness of attacks by promoting some metrics, such as transmission rate and time resolution. For instance, Streamline [127] proposes to use asynchronous communication to improve the transmission rate. Moreover, Prime+Scope [119] constructs new prime patterns by exploiting cache replacement policies. It requires only observation on a single cache line during the attacker's measurement phase, thereby offering a better time resolution.
(2) *Hunting for common attack conditions.* It is much more difficult to launch attacks when sophisticated attack conditions are necessarily required, thus greatly limiting the attack's applicability. For this reason, exploring common attack conditions or relaxing existing conditions is an essential direction, which could inspire researchers to develop more general attack techniques. For example, future research could focus on developing attacks that require fewer special instructions and shared resources, relaxing the requirements for platform and execution environment. A promising direction is to explore more

attacks that converts microarchitectural states into architectural states without relying on high-precision timing measurements [176, 181].

(3) *Uncovering new attack surfaces.* Further exploring new attack surfaces is always an important direction in the research of microarchitecture security, which can be explored from the aspects of attack conditions, attack scenarios, and threat models. The related approaches can be generalized as follows: (1) exposing new microarchitecture components, such as various CPU buffers; (2) discovering new leakage points through probing other microcode events (e.g., interrupt [183]); (3) diverting the exploitable resource sharing level from CPU component or core to higher system level [134], such as those shared in OSes [180] and databases; and (4) combining transient execution with covert channels to develop new attack scenarios.

Several potential directions for the defenses of MTSCAs include the following:

(1) *Efficiently automatic gadget finding.* Automatically finding exploitable and effective gadgets has been a separate body of research, because one defense can benefit from it by reducing performance overhead. However, current methods primarily focus on identifying Spectre-PHT gadgets. From Section 5.3, it is evident that both control-flow and data speculation attacks require victim gadgets, making mitigation possible through gadget discovery. Future research could explore more comprehensive gadget discovery methods, addressing issues present in current approaches, such as false positives, false negatives, and analytical costs. One potential approach involves providing more generalized gadget modeling and integrating static and dynamic analyses to enhance efficiency.

(2) *General mitigation measures.* Some high-security scenarios, such as confidential computing in the cloud, often require deployment of various effective mitigations to thwart information leakage as much as possible. However, most current defenses tend to address only one aspect of attacks. For example, literature [66, 154, 185] can effectively counter Spectre V1 and V2 attacks, but not other types of speculative execution attacks (e.g., MDS [22, 104, 129]). Therefore, developing more general mitigation strategies by blocking common attack conditions is an important research direction. For instance, given that most transient execution attacks need to share resources on the same core, it is still worth exploring strategies with lower overhead to isolate physical resources across different logical cores or eliminate microarchitectural resource sharing for distinct security domains.

(3) *Customizable defenses.* Blocking all leaks fundamentally has to rely on hardware support from the CPU vendors, which inevitably introduces high overhead. However, in some high-performance computing scenarios, it is usually not necessary to cover all leaks. Similarly to the design space exploration in Reference [94], a compromise approach for reducing performance costs is to customize mitigation measures according to different security requirements. Thus, an important research branch is to explore customizable defenses according to either security or performance (or both) requirements. For example, control-flow speculation attacks require mis-training BPs. Therefore, effectively reducing defense overhead can be achieved by focusing on preventing branch mis-training [26].

(4) *Formal verification and analysis.* Although substantial hardware- and software-based mitigations against MTSCAs have been proposed successively, almost all of them lack provable security guarantees. To identify microarchitecture vulnerabilities and verify the security of hardware designs as much as possible, a promising research direction is using formal verification and analysis. This requires a fine-grained microarchitecture- or RTL-level formal analysis model [21, 25, 179], which can effectively weigh the accuracy and complexity.

10 CONCLUSIONS

MTSCAs break various security boundaries and severely threaten the security of modern processors. In this survey, we first recall the microarchitectural optimization techniques on modern processors. Subsequently, we propose two sets of taxonomies for microarchitecture vulnerabilities and their countermeasures based on the varying attack conditions. With the proposed taxonomies, we classify the published attacks and existing defenses into different categories respectively and offer an in-depth analysis from distinct perspectives. In particular, we discuss the similarities and differences of those attacks and uncover the practicality and severity of the vulnerabilities by specifying attack targets/platforms and security boundaries that they can bypass. We also examine the scalability of those defenses through specifying wanted defense goals and costs, and discuss corresponding detection methods that can be combined with defenses to reduce overhead. Last, we finalize this survey by highlighting some key challenges and proposing a series of interesting research directions. We hope our survey initiates comprehensive understanding and careful consideration for MTSCAs and their defenses in existing processors as well as in future microarchitecture designs.

REFERENCES

[1] Onur Acıiçmez, Billy Bob Brumley, and Philipp Grabher. 2010. New results on instruction cache attacks. In *CHES*, 110–124.
[2] Jaeguk Ahn, Jiho Kim, Hans Kasan, Leila Delshadtehrani, Wonjun Song, Ajay Joshi, and John Kim. 2021. Network-on-chip microarchitecture-based covert channel in gpus. In *MICRO*, 565–577.
[3] Sam Ainsworth. 2021. GhostMinion: A strictness-ordered cache system for spectre mitigation. In *MICRO*, 592–606.
[4] Samira Mirbagher Ajorpaz, Daniel Moghimi, Jeffrey Neal Collins, Gilles Pokam, Nael Abu-Ghazaleh, and Dean Tullsen. 2022. EVAX: Towards a practical, pro-active & adaptive architecture for high performance & security. In *MICRO*, 1218–1236.
[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *S&P*, 870–887.
[6] AMD. 2023. AMD Product Security. Retrieved August 2023 from https://www.amd.com/en/corporate/product-security
[7] AMD. 2023. AMD64 Architecture Programmer's Manual. Retrieved August 2023 from https://www.amd.com/system/files/TechDocs/40332.pdf
[8] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On subnormal floating point and abnormal timing. In *S&P*, 623–639.
[9] ARM Corporation. 2010. mbedTLS (Formerly Known as PolarSSL). Retrieved April 2022 from https://tls.mbed.org/
[10] Sahan Bandara and Michel A Kinsy. 2019. Adaptive caches as a defense mechanism against cache side-channel attacks. In *ASHES*, 55–64.
[11] Mohammad Behnia, Prateek Sahu, Riccardo Paccagnella, Jiyong Yu, Zirui Neil Zhao, Xiang Zou, Thomas Unterluggauer, Josep Torrellas, Carlos Rozas, Adam Morrison, Frank Mckeen, Fangfei Liu, Ron Gabor, Christopher W. Fletcher, Abhishek Basak, and Alaa Alameldeen. 2021. Speculative interference attacks: Breaking invisible speculation schemes. In *ASPLOS*, 1046–1060.
[12] Daniel J. Bernstein. 2005. Cache-timing Attacks on AES. Retrieved August 2021 from http:https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.140.2835&rep=rep1&type=pdf
[13] Sarani Bhattacharya, Clémentine Maurice, Shivam Bhasin, and Debdeep Mukhopadhyay. 2020. Branch prediction attack on blinded scalar multiplication. *IEEE Trans. Comput.* 69, 5 (2020), 633–648.
[14] Atri Bhattacharyya, Andrés Sánchez, Esmaeil M. Koruyeh, Nael Abu-Ghazaleh, Chengyu Song, and Mathias Payer. 2020. SpecROP: Speculative exploitation of ROP chains. In *RAID*, 1–16.
[15] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: Exploiting speculative execution through port contention. In *CCS*, 785–800.
[16] Arnab Kumar Biswas, Dipak Ghosal, and Shishir Nagaraja. 2017. A survey of timing channels and countermeasures. *Comput. Surv.* 50, 1 (2017), 1–39.
[17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 1–12.
[18] Samira Briongos, Pedro Malagón, José M. Moya, and Thomas Eisenbarth. 2020. RELOAD+ REFRESH: Abusing cache replacement policies to perform stealthy cache attacks. In *USENIX Security*, 1967–1984.

[19] Billy Bob Brumley and Risto M. Hakala. 2009. Cache-timing template attacks. In *ASIACRYPT*, 667–684.
[20] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the keys to the intel SGX kingdom with transient out-of-order execution. In *USENIX Security*, 991–1008.
[21] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendraminetto. 2019. Model-checking speculation-dependent security properties: Abstracting and reducing processor models for sound and complete verification. *Electronics* 8, 9 (2019), 1057–1065.
[22] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, et al. 2019. Fallout: Leaking data on meltdown-resistant CPUs. In *CCS*, 769–784.
[23] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A systematic evaluation of transient execution attacks and defenses. In *USENIX Security*, 249–266.
[24] C. Carruth. 2018. Speculative Load Hardening (A Spectre Variant #1 Mitigation Technique). Retrieved November 2022 from https://releases.llvm.org/8.0.1/docs/SpeculativeLoadHardening.html
[25] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A formal approach to secure speculation. In *CSF*, 288–28815.
[26] Congcong Chen, Chaoqun Shen, and Jiliang Zhang. 2022. Lightweight and secure branch predictors against spectre attacks. In *ASPDAC*, 25–30.
[27] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H. Lai. 2019. SgxPectre: stealing intel secrets from SGX enclaves Via speculative execution. In *EuroS&P*. 142–157.
[28] Jie Chen and Guru Venkataramani. 2014. CC-hunter: Uncovering covert timing channels on shared processor hardware. In *MICRO*, 216–228.
[29] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. 2020. BranchSpec: Information leakage attacks exploiting speculative branch instruction executions. In *ICCD*, 529–536.
[30] Cloudflare. 2019. Cloudflare Workers. Retrieved August, 2023 from https://www.cloudflare.com/products/cloudflare-workers/
[31] Jonathan Corbet. 2018. Meltdown Strikes Back: The L1 Terminal Fault Vulnerability. Retrieved October 2022 from https://lwn.net/Articles/762570/
[32] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. Cryptology ePrint Archive. https://eprint.iacr.org/2016/086
[33] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. 2015. Thwarting cache side-channel attacks through dynamic software diversity. In *NDSS*, 1–14.
[34] Jinhua Cui, Yiyun Yin, Congcong Chen, and Jiliang Zhang. 2023. SPOILER-alert: Detecting spoiler attack using cuckoo filter. In *DATE*, 1–6.
[35] Yujie Cui, Chun Yang, and Xu Cheng. 2022. Abusing cache line dirty states to leak information in commercial processors. In *HPCA*, 82–97.
[36] Shuwen Deng and Jakub Szefer. 2021. New predictor-based attacks in processors. In *DAC*, 697–702.
[37] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. 2020. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 451–468.
[38] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+ abort: A timer-free high-precision L3 cache attack using intel TSX. In *USENIX Security*, 51–67.
[39] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: A feasibility study. In *HASP*, 1–8.
[40] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *MICRO*, 1–13.
[41] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *ACM Trans. Arch. Code Optimiz.* 13, 1 (2016), 1–23.
[42] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. Branchscope: A new side-channel attack on directional branch predictor. *ACM SIGPLAN Not.* 53, 2 (2018), 693–707.
[43] Hongyu Fang, Miloš Doroslovački, and Guru Venkataramani. 2020. Reuse-trap: Re-purposing cache reuse distance to defend against side channel leakage. In *DAC*, 1–6.
[44] Agner Fog. 2023. The Microarchitecture of Intel, AMD, and VIA CPUs. Retrieved August 2023 from https://www.agner.org/optimize/microarchitecture.pdf
[45] Franz A. Fuchs, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. 2021. Developing a test suite for transient-execution attacks on risc-v and cheri-risc-v. In *CARRV*.

[46] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. Spectrerewind: Leaking secrets to past instructions. In *ASHES*, 117–126.
[47] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. Spectreguard: An efficient data-centric defense mechanism against spectre attacks. In *DAC*, 1–6.
[48] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27.
[49] Moein Ghaniyoun, Kristin Barber, Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2023. TEESec: Pre-silicon vulnerability discovery for trusted execution environments. In *ISCA*, 1–15.
[50] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanovic. 2019. Replicating and mitigating spectre attacks on an open source RISC-V microarchitecture. In *CARRV*, 1–7.
[51] Google. 2018. Retpoline: A Software Construct for Preventing Branch-target-injection. Retrieved August 2021 from https://support.google.com/faqs/answer/7625886
[52] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2018. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security*, 955–972.
[53] Ben Gras, Kaveh Razavi, Erik Bosman, Herbert Bos, and Cristiano Giuffrida. 2017. ASLR on the line: Practical cache attacks on the MMU. In *NDSS*, 1–15.
[54] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 217–233.
[55] Daniel Gruss, Moritz Lipp, Michael Schwarz, Richard Fellner, Clémentine Maurice, and Stefan Mangard. 2017. Kaslr is dead: Long live kaslr. In *ESSoS*, 161–176.
[56] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. 2018. Another flip in the wall of rowhammer defenses. In *S&P*, 245–261.
[57] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. 2016. Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR. In *CCS*, 368–379.
[58] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A fast and stealthy cache attack. In *DIMVA*, 279–299.
[59] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security*, 897–912.
[60] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *S&P*, 38–55.
[61] Marco Guarnieri, Boris Köpf, José F. Morales, Jan Reineke, and Andrés Sánchez. 2020. Spectector: Principled detection of speculative information flows. In *S&P*, 1–19.
[62] Yanan Guo, Andrew Zigerelli, Youtao Zhang, and Jun Yang. 2022. Adversarial prefetch: New cross-core cache side channel attacks. In *S&P*, 1458–1473.
[63] Austin Harris, Shijia Wei, Prateek Sahu, Pranav Kumar, Todd Austin, and Mohit Tiwari. 2019. Cyclone: Detecting contention-based cache information leaks through cyclic interference. In *MICRO*, 57–72.
[64] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture, Fifth Edition: A Quantitative Approach.*
[65] Jann Horn. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. Retrieved March 2022 from https://bugs.chromium.org/p/project-zero/issues/detail
[66] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. 2018. Mitigating branch-shadowing attacks on Intel SGX using control flow randomization. In *SysTEX*, 42–47.
[67] Wei-Ming Hu. 1992. Lattice scheduling and covert channels. In *RISP*, 52–61.
[68] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *S&P*, 191–205.
[69] INTEL. 2016. Intel Software Guard Extensions (Intel SGX). Retrieved February 2024 from https://cdrdv2-public.intel.com/671581/intel-sgx-developer-guide.pdf
[70] Intel. 2016. Intel 64 and IA-32 Architectures Software Developer's Manual. Retrieved August 2023 from https://www.intel.cn/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developers-manual.pdf
[71] Intel. 2018. Deep Dive: Intel Analysis of L1 Terminal Fault. Retrieved December 2022 from https://www.intel.com/content/www/us/en/developer/topic-technology/software-security-guidance/overview.html
[72] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. Retrieved September 2020 from https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf
[73] Intel. 2018. Intel 64 and IA-32 Architectures Optimization Reference Manual. Retrieved August 2023 from https://www.intel.com/content/dam/doc/manual/64-ia-32-architectures-optimization-manual.pdf
[74] Intel. 2018. Speculative Execution Side Channel Mitigation. Retrieved September 2020 from https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf

[75] Intel Corporation. 2019. Guidelines for Mitigating Timing Side Channels against Cryptographic Implementations. Retrieved June 2022 from https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/secure-coding/mitigate-timing-side-channel-crypto-implementation.html
[76] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2016. Cross processor cache attacks. In *Asia CCS*, 353–364.
[77] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative load hazards boost rowhammer and cache attacks. In *USENIX Security*, 621–637.
[78] Zhen Hang Jiang and Yunsi Fei. 2017. A novel cache bank timing attack. In *ICCAD*, 139–146.
[79] Mohd Fadzil Abdul Kadir, Jin Kee Wong, Fauziah Ab Wahab, Ahmad Faisal Amri Abidin Bharun, Mohamad Afendee Mohamed, and Aznida Hayati Zakaria. 2019. Retpoline technique for mitigating spectre attack. In *ICEEE*, 96–101.
[80] John Kelsey, Bruce Schneier, David Wagner, and Chris Hall. 1998. Side channel cryptanalysis of product ciphers. In *ESORICS*, 97–110.
[81] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. 2014. ret2dir: Rethinking kernel isolation. In *USENIX Security*, 957–972.
[82] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Des. Autom. Embed. Syst.* 12, 3 (2008), 221–230.
[83] Khaled N. Khasawneh, Esmaeil Mohammadian Koruyeh, Chengyu Song, Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2019. Safespec: Banishing the spectre of a meltdown with leakage-free speculation. In *DAC*, 1–6.
[84] Sowoong Kim, Myeonggyun Han, and Woongki Baek. 2022. DPrime+DAbort: A high-precision and timer-free directory-based side-channel attack in non-inclusive cache hierarchies using Intel TSX. In *HPCA*, 67–81.
[85] Taesoo Kim, Marcus Peinado, and Gloria Mainar-Ruiz. 2012. STEALTHMEM: System-level protection against cache-based side channel attacks in the cloud. In *USENIX Security*, 189–204.
[86] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*, 974–987.
[87] Vladimir Kiriansky and Carl A. Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. arXiv:1807.03757. Retrieved from https://arxiv.org/abs/1807.03757
[88] 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. 2019. Spectre attacks: Exploiting speculative execution. In *S&P*, 1–19.
[89] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *WOOT*, 1–12.
[90] Esmaeil Mohammadian Koruyeh, Shirin Haji Amin Shirazi, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2020. SpecCFI: Mitigating spectre attacks using cfi informed speculation. In *S&P*, 39–53.
[91] Jakob Koschel, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2020. TagBleed: Breaking KASLR on the isolated kernel address space using tagged TLBs. In *EuroS&P)*, 309–321.
[92] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *USENIX Security*, 557–574.
[93] ARM LIMITED. 2018. Vulnerability of Speculative Processors to Cache Timing Side Channel Mechanism. Retrieved December 2022 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3640
[94] Antoine Linarès, David Hely, Frank Lhermet, and Giorgio Di Natale. 2021. Design space exploration applied to decurity. In *DTIS*, 1–4.
[95] Steven B. Lipner. 2015. The birth and death of the orange book. *IEEE Ann. Hist. Comput.* 37, 2 (2015), 19–31.
[96] Moritz Lipp, Daniel Gruss, and Michael Schwarz. 2022. AMD prefetch attacks through power and time. In *USENIX Security*, 643–660.
[97] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. 2017. Practical keystroke timing attacks in sandboxed javascript. In *ESORICS*, 191–209.
[98] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache attacks on mobile devices. In *USENIX Security*, 549–564.
[99] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take a way: Exploring the security implications of AMD's cache way predictors. In *Asia CCS*, 813–825.
[100] 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. 2018. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 973–990.
[101] Fangfei Liu, Qian Ge, Yuval Yarom, Frank Mckeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. 2016. Catalyst: Defeating last-level cache side channel attacks in cloud computing. In *HPCA*, 406–418.
[102] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level cache side-channel attacks are practical. In *S&P*, 605–622.

[103] Xiaoxuan Lou, Tianwei Zhang, Jun Jiang, and Yinqian Zhang. 2021. A survey of microarchitectural side-channel vulnerabilities, attacks, and defenses in cryptography. *Comput. Surv.* 54, 6 (2021), 1–37.

[104] Giorgi Maisuradze and Christian Rossow. 2018. Ret2spec: Speculative execution using return stack buffers. In *CCS*, 2109–2122.

[105] Robert Martin, John Demme, and Simha Sethumadhavan. 2012. Timewarp: Rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In *ISCA*, 118–129.

[106] Marius Iulian Mihailescu and Stefania Loredana Nita. 2021. *Brute Force and Buffer Overflow Attacks.* 423–434.

[107] Samira Mirbagher-Ajorpaz, Gilles Pokam, Esmaeil Mohammadian-Koruyeh, Elba Garza, Nael Abu-Ghazaleh, and Daniel A. Jiménez. 2020. PerSpectron: Detecting invariant footprints of microarchitectural attacks with perceptron. In *MICRO*, 1124–1137.

[108] Ahmad Moghimi, Jan Wichelmann, Thomas Eisenbarth, and Berk Sunar. 2019. Memjam: A false dependency attack against constant-time crypto implementations. *Int. J. Parallel Program.* 47, 4 (2019), 538–570.

[109] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. 2020. CopyCat: Controlled instruction-level attacks on enclaves. In *USENIX Security*, 469–486.

[110] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing spectre-type vulnerabilities to the surface. In *USENIX Security*, 1481–1498.

[111] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *CT-RSA*, 1–20.

[112] Riccardo Paccagnella, Licheng Luo, and Christopher W. Fletcher. 2021. Lord of the ring (s): Side channel attacks on the CPU On-Chip ring interconnect are practical. In *USENIX Security*, 645–662.

[113] D. Page. 2002. Theoretical use of cache memory as a cryptanalytic side-channel. Cryptology ePrint Archive. https://eprint.iacr.org/2002/169.pdf

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

[115] Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. 2016. Make sure DSA signing exponentiations really are constant-time. In *CCS*, 1639–1650.

[116] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM addressing for cross-CPU attacks. In *USENIX Security*, 565–581.

[117] Filip Pizlo. 2018. *What Spectre and Meltdown Mean for Webkit.* Retrieved December 2022 from https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/

[118] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Čapkun. 2021. Frontal attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, 663–680.

[119] Antoon Purnal, Furkan Turan, and Ingrid Verbauwhede. 2021. Prime + Scope: Overcoming the observer effect for high-precision cache contention attacks. In *CCS*, 2906–2920.

[120] Zhenxiao Qi, Qian Feng, Yueqiang Cheng, Mengjia Yan, Peng Li, Heng Yin, and Tao Wei. 2021. SpecTaint: Speculative taint analysis for discovering spectre gadgets. In *NDSS*, 1–14.

[121] Moinuddin K. Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *MICRO*, 775–787.

[122] Moinuddin K. Qureshi. 2019. New attacks and defense for encrypted-address cache. In *ISCA*, 360–371.

[123] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. Crosstalk: Speculative data leaks across cores are real. In *S&P*, 1852–1867.

[124] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2016. Secure, precise, and fast floating-point operations on x86 processors. In *USENIX Security*, 71–86.

[125] Charles Reis, Alexander Moshchuk, and Nasko Oskov. 2019. Site isolation: Process separation for web sites within the browser. In *USENIX Security*, 1661–1678.

[126] Scott Dion Rodgers, Rohit Vidwans, Joel Huang, Michael A. Fetterman, and Kamla Huck. 1999. Method and apparatus for generating event handler vectors based on both operating mode and event type. US Patent 5,889,982.

[127] Gururaj Saileshwar, Christopher W. Fletcher, and Moinuddin Qureshi. 2021. Streamline: A fast, flushless cache covert-channel attack by enabling asynchronous collusion. In *ASPLOS*, 1077–1090.

[128] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. Cleanupspec: An "undo" approach to safe speculation. In *MICRO*, 73–86.

[129] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*, 753–768.

[130] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *ESORICS*, 279–299.

[131] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2022. Robust and scalable process isolation against spectre in the cloud. In *ESORICS*, 167–186.

[132] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX ATC*, 309–318.
[133] Chaoqun Shen, Congcong Chen, and Jiliang Zhang. 2021. Micro-architectural cache side-channel attacks and countermeasures. In *ASPDAC*, 441–448.
[134] Chaoqun Shen, Jiliang Zhang, and Gang Qu. 2023. MES-attacks: Software-controlled covert channels based on mutual exclusion and synchronization. In *DAC*, 1–6.
[135] Jicheng Shi, Xiang Song, Haibo Chen, and Binyu Zang. 2011. Limiting cache-based side-channel in multi-tenant cloud using dynamic page coloring. In *DSN-W*, 194–199.
[136] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. 2019. Robust website fingerprinting through the cache occupancy channel. In *USENIX Security*, 639–656.
[137] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. 2019. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, 318–331.
[138] Julian Stecklina and Thomas Prescher. 2018. Lazyfp: Leaking fpu register state using microarchitectural side-channels. arXiv:1806.07480. Retrieved from https://arxiv.org/abs/1806.07480
[139] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. 2021. Invisible probe: Timing attacks with pcie congestion side-channel. In *S&P*, 322–338.
[140] Qinhan Tan, Zhihua Zeng, Kai Bu, and Kui Ren. 2021. PhantomCache: Obfuscating cache conflicts with localized randomization. In *NDSS*, 1–17.
[141] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*, 395–410.
[142] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. 2022. SpecHammer: Combining spectre and rowhammer for new speculative attacks. In *S&P*, 681–698.
[143] David Trilla, Carles Hernandez, Jaume Abella, and Francisco J. Cazorla. 2018. Cache side-channel attacks and time-predictability in high-performance critical real-time systems. In *DAC*, 1–6.
[144] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. 2018. MeltdownPrime and SpectrePrime: Automatically-synthesized attacks exploiting invalidation-based coherence protocols. arXiv:1802.03802. Retrieved from https://arxiv.org/abs/1802.03802
[145] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. 2003. Cryptanalysis of DES implemented on computers with cache. In *CHES*, 62–76.
[146] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lippi, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*, 54–72.
[147] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 1–6.
[148] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2018. Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic. In *CCS*, 178–195.
[149] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *USENIX Security*, 1041–1056.
[150] Stephan Van Schaik, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. 2018. Malicious management unit: Why stopping cache attacks in software is harder than you think. In *USENIX Security*, 937–954.
[151] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue in-flight data load. In *S&P*, 88–105.
[152] Stephan van Schaik, Marina Minkin, Andrew Kwong, Daniel Genkin, and Yuval Yarom. 2021. CacheOut: Leaking data on Intel CPUs via cache evictions. In *S&P*, 339–354.
[153] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and practice of finding eviction sets. In *S&P*, 39–54.
[154] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M. Al-Hashimi, and Geoff V. Merrett. 2019. BRB: Mitigating branch predictor side-channels. In *HPCA*, 466–477.
[155] Junpeng Wan, Yanxiang Bi, Zhe Zhou, and Zhou Li. 2022. MeshUp: Stateless cache side-channel attack on CPU mesh. In *S&P*, 1506–1524.
[156] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead defense against spectre attacks via program analysis. *IEEE Trans. Softw. Eng.* 47, 11 (2019), 2504–2519.
[157] Ying Wang, Wen Li, Huawei Li, and Xiaowei Li. 2018. Lightweight timing channel protection for shared DRAM controller. In *ITC*, 1–10.
[158] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. 2022. Hertzbleed: Turning power side-channel attacks into remote timing attacks on x86. In *USENIX Security)*, 679–697.
[159] Zhenghong Wang and Ruby B. Lee. 2006. Covert and side channels due to processor architecture. In *ACSAC*, 473–482.

[160] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *MICRO*, 572–586.

[161] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F. Wenisch, and Yuval Yarom. 2018. Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-order Execution. Retrieved September 2021 from https://foreshadowattack.eu/foreshadow-NG.pdf

[162] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. ScatterCache: Thwarting cache attacks via cache set randomization. In *USENIX Security*, 675–692.

[163] J.C. Wray. 1991. An analysis of covert timing channels. In *RISP*, 2–7.

[164] Zhenyu Wu, Zhang Xu, and Haining Wang. 2015. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM Trans. Netw.* 23, 2 (2015), 603–615.

[165] Wenjie Xiong and Jakub Szefer. 2020. Leaking information through cache LRU states. In *HPCA*, 139–152.

[166] Wenjie Xiong and Jakub Szefer. 2021. Survey of transient execution attacks and their mitigations. *Comput. Surv.* 54, 3 (2021), 1–36.

[167] Yunjing Xu, Michael Bailey, Farnam Jahanian, Kaustubh Joshi, Matti Hiltunen, and Richard Schlichting. 2011. An exploration of L2 cache covert channels in virtualized environments. In *CCSW*, 29–40.

[168] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, 640–656.

[169] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*, 428–441.

[170] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack directories, not caches: Side channel attacks in a non-inclusive world. In *S&P*, 888–904.

[171] Yuheng Yang, Thomas Bourgeat, Stella Lau, and Mengjia Yan. 2023. Pensieve: Microarchitectural modeling for security evaluation. In *ISCA*, 1–15.

[172] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage? In *HPCA*, 168–179.

[173] Fan Yao, Guru Venkataramani, and Miloš Doroslovački. 2017. Covert timing channels exploiting non-uniform memory access based architectures. In *GLSVLSI*, 155–160.

[174] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *USENIX Security*, 719–732.

[175] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A timing attack on OpenSSL constant-time RSA. *J. Cryptogr. Eng.* 7, 2 (2017), 99–112.

[176] Jiyong Yu, Aishani Dutta, Trent Jaeger, David Kohlbrenner, and Christopher W. Fletcher. 2023. Synchronization storage channels (S2C): Timer-less cache side-channel attacks on the apple M1 via hardware synchronization instructions. In *USENIX Security*, 1973–1990.

[177] Jiyong Yu, Lucas Hsiung, Mohamad El Hajj, and Christopher W. Fletcher. 2019. Data oblivious ISA extensions for side channel-resistant and high performance computing. In *NDSS*, 1–15.

[178] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2019. Speculative taint tracking (STT) : A comprehensive protection for speculatively accessed data. In *MICRO*, 954–968.

[179] Danfeng Zhang, Yao Wang, G Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. *ACM SIGPLAN Not.* 50, 4 (2015), 503–516.

[180] Jiliang Zhang, Chaoqun Shen, and Gang Qu. 2023. Mex+Sync: Software covert channels exploiting mutual exclusion and synchronization. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 42, 12 (2023), 4491–4504.

[181] Ruiyi Zhang, Taehyun Kim, Daniel Weber, and Michael Schwarz. 2023. (M)WAIT for it: Bridging the gap between microarchitectural and architectural side channels. In *USENIX Security*, 7267–7284.

[182] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2016. CloudRadar: A real-time side-channel attack detection system in clouds. In *Research in Attacks, Intrusions, and Defenses*, 118–140.

[183] Xin Zhang, Zhi Zhang, Qingni Shen, Wenhao Wang, Yansong Gao, Zhuoxi Yang, and Jiliang Zhang. 2024. SegScope: Probing fine-grained interrupts via architectural footprints. In *HPCA*, 1–15.

[184] Yinqian Zhang and Michael K. Reiter. 2013. Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud. In *CCS*, 827–838.

[185] Lutan Zhao, Peinan Li, Rui Hou, Michael C. Huang, Jiazhen Li, Lixin Zhang, Xuehai Qian, and Dan Meng. 2021. A lightweight isolation mechanism for secure branch predictors. In *DAC*, 1267–1272.

[186] Ziqiao Zhou, Michael K. Reiter, and Yinqian Zhang. 2016. A software approach to defeating side channels in last-level caches. In *CCS*, 871–882.

Received 7 February 2023; revised 28 December 2023; accepted 31 January 2024