Survey of Transient Execution Attacks and Their Mitigations

WENJIE XIONG and JAKUB SZEFER, Dept. of Electrical Engineering, Yale University

Transient execution attacks, also known as speculative execution attacks, have drawn much interest in the last few years as they can cause critical data leakage. Since the first disclosure of Spectre and Meltdown attacks in January 2018, a number of new transient execution attack types have been demonstrated targeting different processors. A transient execution attack consists of two main components: transient execution itself and a covert channel that is used to actually exfiltrate the information.Transient execution is a result of the fundamental features of modern processors that are designed to boost performance and efficiency, while covert channels are unintended information leakage channels that result from temporal and spatial sharing of the micro-architectural components. Given the severity of the transient execution attacks, they have motivated computer architects in both industry and academia to rethink the design of the processors and to propose hardware defenses. To help understand the transient execution attacks, this survey summarizes the phases of the attacks and the security boundaries across which the information is leaked in different attacks.This survey further analyzes the causes of transient execution as well as the different types of covert channels and presents a taxonomy of the attacks based on the causes and types. This survey in addition presents metrics for comparing different aspects of the transient execution attacks and uses them to evaluate the feasibility of the different attacks. This survey especially considers both existing attacks and potential new attacks suggested by our analysis. This survey finishes by discussing different mitigations that have so far been proposed at the micro-architecture level and discusses their benefits and limitations.

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

Additional Key Words and Phrases: Transient execution, speculative execution, timing channels, covert channels, secure processor architectures

ACM Reference format:
Wenjie Xiong and Jakub Szefer. 2021. Survey of Transient Execution Attacks and Their Mitigations. *ACM Comput. Surv.* 54, 3, Article 54 (May 2021), 36 pages.
https://doi.org/10.1145/3442479

1 INTRODUCTION

In the past decades, computer architects have been working hard to improve the performance of computing systems. Different optimizations have been introduced in the various processor micro-architectures to improve the performance, including pipelining, out-of-order execution, and branch prediction [51]. Some of the optimizations require aggressive speculation of the executed

This work was supported by NSF grants 1651945 and 1813797, and through SRC award number 2844.001.

Authors' addresses: W. Xiong and J. Szefer, Dept. of Electrical Engineering, Yale University 10 Hillhouse Ave, Room 505, New Haven, CT, 06511; email: wenjie.xiong@aya.yale.edu, jakub.szefer@yale.edu.

0360-0300/2021/05-ART54 $15.00
https://doi.org/10.1145/3442479

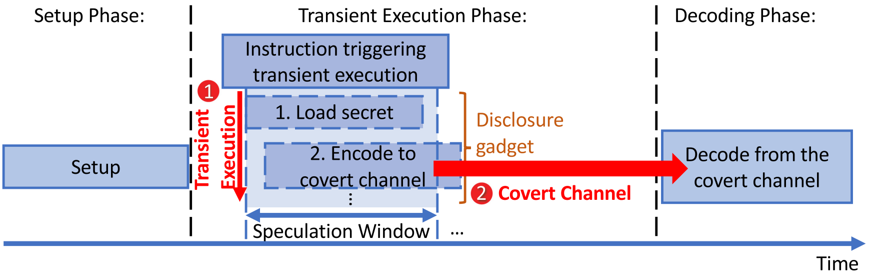

Setup Phase:
Transient Execution Phase:
Decoding Phase:
Setup
Instruction triggering transient execution
1
Transient Execution
1. Load secret
2. Encode to covert channel
⋮
Disclosure gadget
2 Covert Channel
Decode from the covert channel
Speculation Window
...
Time

Fig. 1. Phases of transient execution attacks.

instructions, which leads to transient execution of certain instructions. At the **Instruction Set Architecture (ISA)** level, today's processors behave correctly and return correct results. However, in the majority of today's processors, the complicated underlying micro-architectural states are modified during the transient execution, and data can be leaked from these micro-architectural states, even if there is no leak at the ISA level. For example, while waiting for a conditional branch to be resolved, branch prediction is used to predict whether the branch will be taken or not, and the processor begins to speculatively execute down the predicted control flow path before the outcome of the branch is known. Such speculative execution of instructions causes the micro-architectural state of the processor to be modified, even if instructions are executed down an incorrectly speculated control flow path. The execution of the instructions down the incorrectly speculated path is called *transient execution*—because the instructions execute transiently and should ideally disappear with no side effects if there was mis-speculation. When a mis-speculation is detected, the architectural and micro-architectural side effects should be cleaned up—but this is not done in the majority of today's processors, leading to a number of recently publicized transient execution attacks [16, 64, 72, 84, 102, 113, 115, 123]. The attacks enable data leakage across different security boundaries in computing systems. The micro-architectural states of a processor are today especially not captured by the ISA specification, and the micro-architectural vulnerabilities cannot be found or analyzed by only examining the processor's ISA.

The complicated micro-architectural states arise from the various optimizations realized in today's processors. Besides focusing on pure performance optimization, many processors are also designed to share hardware units in order to reduce area and improve power efficiency. For example, hyper-threading allows different programs to execute concurrently on the same processor pipeline by sharing the execution and other functional units among the hardware threads in the pipeline. Also, because supply voltage does not scale with the size of the transistors [87], modern processors use multi-core designs. In multi-core systems, caches, memory-related logic, and peripherals are shared among the different processor cores. Sharing of the resources has led to numerous timing-based side and covert channels [50, 76, 109, 142]—the channels can occur independent of transient execution or together with transient execution, which is the focus of this survey.

Transient execution combined with covert channels results in *transient execution attacks,* which can compromise the confidentiality of the system. As shown in Figure 1, during such attacks, the secret (a.k.a., sensitive data) is available in the duration of the transient execution—this differentiates the transient execution attacks from conventional covert channel attacks where the data is assumed to be always available to the sender, not just during transient execution.[1] After the secret

[1] There are also attacks using the timing difference in transient execution, e.g., [33, 34, 36, 36, 56]. These attacks are still conventional covert channel attacks, where the timing difference comes from the prediction units. Thus, these attacks are not in the scope of this article but are listed in Section 8.

data is accessed during transient execution and encoded into a covert channel, the secret data can later be extracted, i.e. decoded, by the attacker from the covert channel.

A number of transient execution attack variants have been demonstrated, e.g., Spectre [10, 21, 64, 65, 77, 78, 104, 111], Meltdown [16, 23, 63, 72, 111], Foreshadow [113, 123], LazyFP [107], **Micro-architectural Data Sampling (MDS)** [84, 102, 115], and **Load Value Injection (LVI)** [114]. These attacks have been shown to allow data leaks across different security boundaries, e.g., across different privilege levels, between the SGX enclave and the rest of the system, across sandbox isolation, and so forth. The transient execution attacks have been assigned 9 **Common Vulnerabilities and Exposures (CVE)** IDs out of 14 CVE IDs that correspond to vulnerabilities about gaining information on Intel products in 2018, and 4 out of 9 in 2019, according to the CVE Details database [24]. These attacks also affect other vendors, such as AMD or Arm [4, 5, 16].[2]

In addition, these attacks have raised a lot of interest and motivated computer architects to rethink the design of processors and propose a number of hardware defenses [8, 38, 60–62, 68, 97, 98, 100, 122, 125, 132]—this survey summarizes the attacks and the hardware defenses, while software-based defenses are summarized in existing work [16].

1.1 Outline and Contributions

This article provides a survey of existing *transient execution attacks* from January 2018 to December 2020. We start by providing background on the micro-architectural features that lead to the attacks. We then define the transient execution attacks and summarize the phases and attack scenarios. We analyze the types of transient execution and covert channels leveraged by the transient execution attacks to show the root causes of these attacks. In the end, we discuss the mitigation strategies for the transient execution and covert channels. The contributions of this survey are the following:

- We summarize different attack scenarios and the security boundaries across which secrets are leaked in the different attacks.
- We provide a taxonomy of the existing transient execution attacks by analyzing the causes of transient execution that they leverage, and we propose metrics to compare the feasibility of using different transient execution types for attack.
- We summarize and categorize the existing and potential timing-based covert channels in micro-architecture states that can be used with transient execution attacks, and also propose metrics to compare these covert channels.
- We discuss the feasibility of the existing attacks based on the metrics we propose.
- We compare the different mitigation strategies that have been so far designed at the micro-architectural level in various publications.

2 BACKGROUND

This section gives background about various optimizations in micro-architecture that lead to the transient execution, and thus in turn contribute to the transient execution attacks. Many more details about processor micro-architecture and pipeline details are available in computer architecture textbooks, e.g., [51]. This section also discusses conventional side channels and covert channels based on timing; many more details about the timing channels are given in other surveys, e.g., [109]. The two concepts combined lead to the transient execution attacks, which is the focus of the rest of this survey.

[2] No CVEs were published in 2020 when this survey was being prepared.

2.1 Performance Optimizations That Enable Transient Execution Attacks

Due to the data dependencies between instructions, the CPU pipeline sometimes has to stall until the dependencies are resolved. To reduce the stalls in the pipeline and to keep the pipeline full, many performance-improving optimizations have been proposed and implemented in today's commodity processors.

Out-of-Order Execution (OoO): In OoO, some of the younger instructions can be executed earlier than the older instructions (based on program order) if all the dependencies of the younger instructions are available. OoO helps to improve instruction-level parallelism. In OoO, the life cycle of instruction is fetch, dispatch, issue (a.k.a. execute), and retire (a.k.a. commit). The instruction is first fetched into the pipeline, and after decoding, micro-ops are dispatched. Once all the dependencies of the instruction are satisfied, an instruction (or micro-op) is issued for execution. OoO uses a **reorder buffer (ROB)** to hold the instructions (or micro-ops) and execution results and to retire each instruction (or micro-op) in the program order. Instructions (or micro-ops) are retired (committed) when they reach the head of ROB.

Speculative Execution: Speculative execution occurs when there is a control flow instruction for which the processor does not know the result yet. For example, a branch condition needs to be computed before the branch result (taken or not taken) can be obtained. To improve performance, processors often speculate the outcome of such instructions and begin to execute instructions down the speculated path. As the instructions execute down the mis-speculated path, they may access data that should otherwise not be accessible if the speculation was not allowed. Later, either speculation is confirmed to be correct or there is a mis-speculation, and instructions that began to execute down the mis-speculated path are squashed (cleared from the pipeline). After mis-speculation, the processor should clean up all architectural and micro-architectural states to create an illusion as if these instructions never executed. In addition to control flow, speculation may also happen in prefetching and value prediction (not used in commodity processors).

Delayed Fault Validation: Similar to speculation for control flow, processors speculate that there is no fault and continue execution even when there is a potential fault. Fault checking especially can be delayed, and some processors allow for instructions to continue executing, during which time the data (that should have been inaccessible due to the fault) can be accessed and micro-architecture states can be modified. If there is any fault, the processor then should clean up all the architectural and micro-architectural state changes.

Caching: One of the key optimizations for memory-related operations is caching. Modern processors have multiple levels of caches, typically L1, L2, and **Last Level Cache (LLC)**. A cache hit occurs when a piece of data is found in a cache, and the processor does not need to go to the main memory to fetch the data. Based on the level of cache where data is found, it can be up to about 500× faster to get data if it is a cache hit. Other cache-like structures in processors, such as **translation lookaside buffer (TLB)**, can also offer performance improvement by caching data or metadata.

Sharing of Functional Units in Hyper-Threading within a Processor Core: To reduce area and power consumption, functional units are shared among hardware threads in hyper-threading. Today, most processors are built with two-thread **Simultaneous Multi-Threading (SMT)** configurations. In SMT, in the execution stage, the processor decides with instructions from which threads can execute based on the available execution units. It is expected that not all programs need all the units at the same time, so the sharing allows for good performance while reducing the need to duplicate all units among all SMT threads. However, sharing also creates situations when two processes try to use the same unit—there may be contention and difference in the timing of the execution.

Sharing of Resources among Cores in Multi-Core Processors: Similar to sharing within the processor core, many resources are shared among cores. These include caches, prefetchers, various buffers, memory controllers, directories, and other cache coherence-related structures, memory busses, I/O, and so forth. Rather than each core having a separate one of these units, they are shared to reduce hardware resources. However, the sharing, again, can create timing differences based on the different operations.

2.2 Covert Channels

Covert channels are communication channels that are not originally designed for information transmission but can indirectly exfiltrate the information across security boundaries [109]. The entity sending the information is called the sender in covert channels.[3] Execution of some instructions by the sender causes the state of the processor to be modified. The states can be internal processor states or physical states such as temperature or EM emanations. Finally, the receiver observes the state change to learn about a bit of information transmitted (or leaked) by the sender.

Timing-based covert channels, which are the focus of this survey, leverage the changes in the processor, which can be observed by measuring the execution time of some operations in software. The state modification can be the architectural state or the micro-architectural state. However, because by design the processors should not leak any information from the architectural state (e.g., two processes in different memory spaces cannot access each other's memory), the majority of the timing channels abuse the micro-architectural state changes, which cannot be directly read, but which can be observed through timing differences. These covert channels are the result of the various performance optimizations discussed in Section 2.1.

3 TRANSIENT EXECUTION ATTACK SCENARIOS

We define transient execution attacks as data exfiltration attacks that access data during transient execution and then leverage a covert channel to leak information. The phases of these attacks are shown in Figure 1. Although not indicated in the "transient execution attacks" name, covert channels are an essential component of the transient execution attacks, because the micro-architectural states changed during transient execution are not visible at the architectural level and are only accessible by using a covert channel to learn the micro-architectural state change (and thus the secret). In this section, we summarize the attack scenarios, e.g., the attacker's goal, the location of the attacker, and so forth.

3.1 Attacker's Goal: Breaking Security Boundaries

There are many security boundaries (between different privilege levels or security domains) in a typical processor, as shown in Figure 2. The goal of the attacker of the transient execution attacks is to learn information related to the victim's protected data across the different boundaries. In Figure 2, we categorize the possible privilege levels, or security domains, where the attack can originate and wherefrom it is trying to extract data as follows:

(1) **Across user-level applications:** The attacker and the victim are two separate user applications, and the attacker process tries to learn the memory content of another process; e.g., [10] demonstrates how an attacker process learns the private key of a separate victim OpenSSH process.

[3] Side channel is similar to covert channel. In a side channel, the sender (a.k.a the victim) does not transmit information on purpose, but has a vulnerability in the code to leak information.

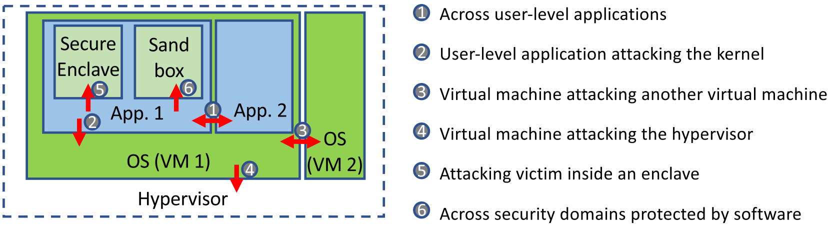

Secure Enclave
Sand box
App. 1
App. 2
OS (VM 1)
OS (VM 2)
Hypervisor
1
2
3
4
5
6
1 Across user-level applications
2 User-level application attacking the kernel
3 Virtual machine attacking another virtual machine
4 Virtual machine attacking the hypervisor
5 Attacking victim inside an enclave
6 Across security domains protected by software

Fig. 2. Security boundaries in computer systems that are broken by transient execution attacks.

(2) **User-level program attacking the kernel:** The attacker runs in the user level and wants to read the privileged data of the kernel; e.g., [72] demonstrates an attack that allows an unprivileged application to dump kernel memory.
(3) **Virtual machine attacking another VM:** The attacker and the victim reside in two different guest **virtual machines (VMs)**; e.g., [10] shows it is possible for an attacker VM to learn the private key of the OpenSSH server in the victim VM.
(4) **Virtual machine attacking the hypervisor:** The attacker is a guest OS and the victim is the host hypervisor; e.g., [64] demonstrates an attack against KVM that leaks the hypervisor's memory when the attacker has full control of the OS inside a VM.
(5) **Attacking the victim running inside an enclave:** The victim runs inside a security domain protected by some hardware scheme, such as Intel SGX [25], XOM [69], Aegis [108], Bastion [19], Sanctum [26], and Keystone [66], and the attacker code runs outside of it; e.g., [21] demonstrates such an attack that retrieves secret from inside the SGX enclave.
(6) **Across security domains protected by software:** The victim runs inside the security domain protected by some software scheme, such as sandboxes in JavaScript, and the attacker code runs outside of it, as shown in [64].

All of the security boundaries listed above are broken by one or more of the existing transient execution attacks. Details about each attack type will be discussed in Section 4.7.

3.1.1 Coherent and Non-Coherent Data. We especially categorize all the data in the processor state into coherent data and non-coherent data. The existing attacks have been shown to be able to retrieve coherent data, as well as non-coherent data.

- *Coherent data* are those coherent with the rest of the system; e.g., data in caches are maintained by cache coherence protocol. Coherent data can be accessed by its address.
- *Non-coherent data* are temporarily fetched into micro-architectural buffers or registers, are not synchronized with the rest of the system, and may not be cleaned up after use, e.g., data in the **store-to-load (STL)** buffer. Thus, non-coherent data may be stale. Non-coherent data that is left in the buffer can be of a different privilege level or security domain, so the attacker will break the security domain when accessing the non-coherent stale data. Some attacks [77, 107] focus on attacking buffers to retrieve such non-coherent data, which in turn breaks the security boundaries.

3.2 Phases of the Attack

As shown in Figure 1, we divide the transient execution attacks into three phases:

(1) **Setup Phase**: The processor executes a set of instructions that modify the micro-architectural states such that it will later cause the transient execution of the desired code (called *disclosure gadget*) to occur in a manner predictable to the attacker. An example of

modifying the micro-architectural states can be achieved by performing indirect jumps to a specific address to "train" the branch predictor. The setup can be done by the attacker running some setup code or triggering a *setup gadget* in the victim's code so that the micro-architectural state is set up as the attacker expects.

(2) **Transient Execution Phase**: The transient execution is actually triggered in this phase. The piece of code that accesses[4] and transmits the secret into the covert channel is called *disclosure gadget*, following the terminology in [112]. The desired disclosure gadget executes transiently in this phase due to the prior training in the setup phase. The instructions belonging to the disclosure gadget are eventually squashed, and the architectural states of the transient instructions are rolled back, but as many of the attacks show, the micro-architectural changes caused by the disclosure gadget remain in the majority of today's processors, so secret data can be later decoded from the covert channel. This phase can be executed either by the victim or by the attacker.

(3) **Decoding Phase**: The attacker is able to recover the data via the covert channel by running the attacker's code or by triggering a *decoding gadget* in the victim's code and observing the behavior or result of the execution.

During an attack, the *Setup Phase* and the *Transient Execution Phase* cause the transient execution of the disclosure gadget to occur. Then, the *Transient Execution Phase* and the *Decoding Phase* leverage the covert channel to transmit data to the attacker. Thus, the Transient Execution Phase is critical for both accessing the secret and encoding it into a channel.

3.3 Transient Execution of the Victim vs. the Attacker

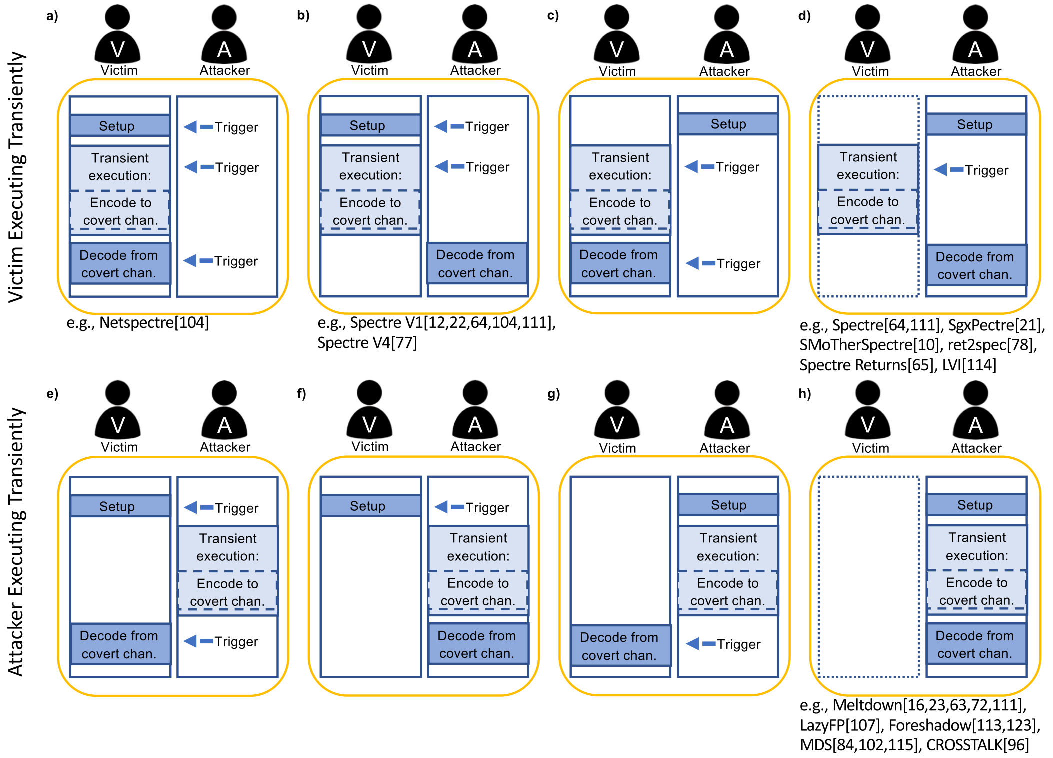
Each phase listed above can be performed by the attacker code or by the victim code, resulting in eight attack scenarios shown in Figure 3. When a phase is performed by the victim, the attacker is assumed to have the ability to trigger the victim to execute the disclosure gadget. We categorize the attacks based on who is executing transiently.

3.3.1 Victim Is Executing Transiently. Figures 3(a) through 3(d) show the scenarios where the victim is triggered to execute a disclosure gadget transiently and the attacker obtains the secret by decoding the data from the covert channel. Because accessing the secret is conducted by the victim transiently, all the data accessible by the victim may be exfiltrated. In these scenarios, the attacker is assumed to be able to control or trigger the execution of the disclosure gadget in the victim's code. The attacker can do this by calling some victim functions with certain parameters. For example, in SGXpectre [21], the attacker can call the target enclave program.

Different from the conventional side and covert channels, here, the encoding phase is executed transiently, and thus, the attack cannot be detected by simply analyzing the software semantics of the victim code. This attack vector leverages the difference between the expected semantics of software execution and the actual execution in hardware and is a fundamental problem in current computer architectures.

There are two options for the setup phase. First, if the hardware component that causes transient execution, e.g., the prediction unit, is shared between the attacker and the victim, then the attacker's execution can manipulate the states of the prediction unit, as shown in Figures 3(c) and 3(d). The second option is that the attacker triggers a *setup gadget* in the victim code to set up the transient execution, as shown in Figures 3(a) and 3(b). For the first option, the attacker is required

[4] In most existing attacks, the secret is not in the register file and needs to be fetched and then transmitted during transient execution. But it is also a transient execution attack if the secret is already in the register but is transmitted during transient execution.



e.g., Netspectre[104]

e.g., Spectre V1[12,22,64,104,111], Spectre V4[77]

e.g., Spectre[64,111], SgxPectre[21], SMoTherSpectre[10], ret2spec[78], Spectre Returns[65], LVI[114]

e.g., Meltdown[16,23,63,72,111], LazyFP[107], Foreshadow[113,123], MDS[84,102,115], CROSSTALK[96]

Fig. 3. Possible scenarios of transient execution attacks: (a)–(d) The attacker triggers part of the victim code to execute transiently to leak secret data through the covert channel. (e)–(h) The attacker executes transiently to access data that he or she does not have permission to access and encodes it into the covert channel.

to co-locate with the victim to share the prediction and to prepare some code to set up the hardware to lure the victim into the desired transient execution. For the second option, the attacker is required to understand the victim's code and be able to trigger the setup gadget to execute with a controlled input, e.g., by calling a function of the victim code.

Decoding data from the covert channel can be done by the attacker, as shown in Figures 3(b) and 3(d), or by the victim, as shown in Figures 3(a) and 3(c). If the decoding is done by the victim, the attacker may leverage the results of the victim code to infer information, or the attacker may trigger the execution of a *decoding gadget* in the victim's code and measure the time or other side effect of the execution.

3.3.2 Attacker Is Executing Transiently. As shown in Figures 3(e) through 3(h), alternatively, the attacker can directly obtain the secret in transient execution, then encode the data into a covert channel and decode it to obtain the secret in the architectural state, such as in his or her memory. Transient execution allows the attacker to access more data than is allowed in the architecture (i.e., ISA) level; thus, this attack is powerful. The attacker can also launch different software threads for the setup or the decoding phases. The attacker's code shown in Figures 3(e) through 3(h) might be in different threads, even on different cores.

During the attack, the attacker directly obtains the secret during transient execution, and thus, the attacker should be able to have a pointer to the location of the victim data. There might be only the attacker code running, or the attacker and the victim running in parallel. In the scenario

Table 1. Required Control of the Victim's Execution in Different Attack Scenarios

Scenario in Figure 3	Setup Phase	Transient Execution Phase	Decoding Phase	Number of Victim Gadgets to Be Triggered*	Sharing Required during Transient Execution**	Sharing Required for Covert Channel**
a	Victim	Victim	Victim	2–3	No	No
b	Victim	Victim	Attacker	1–2	No	Yes
c	Attacker	Victim	Victim	2	Yes	No
d	Attacker	Victim	Attacker	1	Yes	Yes
e	Victim	Attacker	Victim	2	Yes	Yes
f	Victim	Attacker	Attacker	1	Yes	No
g	Attacker	Attacker	Victim	1	No	Yes
h	Attacker	Attacker	Attacker	0	No	No

*The number shows the number of different code gadgets in the victim's code to be triggered by the attacker. We assume the decoding gadget is different from the disclosure gadget. The setup gadget may or may not be the same code as the disclosure gadget, so the two gadgets can be counted as either 1 (same) or 2 (different) gadgets, giving a range of gadgets required, as shown in the fifth column of the table.

**Here, we refer to sharing of hardware between the attacker and the victim. In addition, the attacker (or the victim) could also have multiple software threads running and sharing hardware between the threads. We assume co-location between the threads is possible and do not list that here.

when there is only the attacker code running, the victim's protected data should be addressable to the attacker or the data is in some register in the hardware; i.e., the attacker should have a way to point to the data. In Meltdown [72], the attacker code first loads protected data by its virtual address to a register and then transfers the data through a covert channel. When the attacker and the victim are running concurrently, the attacker should be able to partially control the victim's execution or synchronize with the victim execution. For example, in MDS attacks [84, 102, 115], the attacker needs to synchronize with the victim execution to extract useful information from the non-coherent data of the victim in the processor's buffers.

In micro-architectural implementations, transient execution allows the attacker to access more data than is allowed in the architecture level. Thus, this attack leveraging the attacker's transient execution is implementation dependent and does not work on all the CPUs; e.g., Meltdown [72], Foreshadow [113, 123], and MDS [84, 102, 115] are reported to work only on Intel processors.

Similar to the case when the victim is executing transiently, the setup phases and decoding phases can also be done by the victim, resulting in four attack scenarios in Figures 3(e) through 3(h). However, in the current known attacks, the attacker always sets up, triggers the transient execution, and decodes from the channel, which is more practical.

3.3.3 *Feasibility of the Attack Scenarios.*

The required number of gadgets in the victim code to be triggered and required sharing in different transient execution scenarios are summarized in Table 1. In addition, Figure 3 shows the attack scenarios demonstrated in different publications. In a practical attack, it is desired to have most phases to be executed by the attacker's code and less required sharing of hardware.

In most of the existing attacks, the attacker completes setup and decoding phases, as shown in Figures 3(d) and 3(h), because they use fewer gadgets in the victim code and are more practical for the attacker. Attack scenarios are also demonstrated in Figures 3(a) and 3(b) that have fewer requirements for shared hardware. In Spectre V1, since the victim disclosure gadget can be reused as the setup gadget for training the predictor, triggering the victim to run the setup phase does not require additional effort for the attacker, and thus, Figure 3(b) is also feasible. The attacker can also use the victim's code to complete both setup and decoding steps, as shown in Figure 3(a). In this case, the attacker can launch the attack remotely [104].

The scenarios in Figures 3(c) and 3(e) through 3(g)require more gadgets in the victim code and are not demonstrated in the publications so far. However, if the attacker has the ability to trigger the victim to execute certain gadgets (as required by some of the attacks already), those scenarios are still feasible and should be considered when designing future mitigations.

4 CAUSES OF TRANSIENT EXECUTION

Transient execution is the phenomenon where code is executed speculatively, and it is not known if the instructions will be committed or squashed until the retirement of the instruction or a pipeline squash event. Upon an instruction squash, not all the micro-architectural side effects are cleaned up properly in the majority of today's processors, causing different transient execution attacks. Hence, all causes of pipeline squash are also causes of transient execution and need to be understood to know what caused transient execution attacks to occur. In this section, we first discuss all possible causes of transient execution and then propose a set of the metrics to evaluate feasibility of the transient execution attacks.

4.1 Mis-speculation

The first possible cause of transient execution is mis-speculation. Modern computer architectures make predictions to make full use of the pipeline to gain performance. When prediction is correct, the execution continues and the results of the predicted execution will be used. In this way, prediction boosts performance by executing instructions earlier. If the prediction is wrong, the code executed transiently down the incorrect (mis-predicted path) will be squashed.

(1) **Control Flow Prediction:** Control flow prediction predicts the execution path that a program will follow. **Branch prediction unit (BPU)** stores the history of past branch directions and targets and then leverages the locality in the program control flow to make predictions for future branches. BPU predicts whether the branch is to be taken or not (i.e., branch direction) by a using **pattern history table (PHT)**, and what is the target address (i.e., branch or indirect jump target) by using the **branch target buffer (BTB)** or **return stack buffer (RSB)**. The implementation details of PHT, BTB, and RSB in Intel processors will be discussed in Section 4.9.1.

(2) **Address Speculation:** Address speculation is a prediction of the address when the physical address is not fully available yet, e.g., whether two addresses are the same. It is used to improve performance in the memory system, e.g., STL forwarding in the load-store queue and **line-fill buffer (LFB)** in the cache use address speculation. The implementation details of STL and LFB in Intel processors will be discussed in Section 4.9.2.

(3) **Value Prediction:** To further improve the performance, while the pipeline is waiting for the data to be loaded from the memory hierarchy on a cache miss, value prediction units have been designed to predict the data value and to continue the execution based on the prediction. While this is not known to be implemented in commercial architectures, value prediction had been proposed in the literature [70, 71].

4.2 Exceptions

The second possible cause of transient execution to occur is exceptions. If an instruction causes an exception, the handling of the exception is sometimes delayed until the instructing is retired, allowing code to (transiently) execute until the exception is handled. There are a number of causes of exceptions, such as a wrong permission bit (e.g., present bit, reserved bit) in **Page Table Entry**

(PTE), and so forth. A list of all the exception types or permission bit violations is summarized in [16]. In addition, Xiao et al. developed a software framework to automatically explore the vulnerabilities using exceptions on a variety of Intel and AMD processors [127].

Sometimes the exceptions are suppressed due to another fault, e.g., nested exceptions. For example, when using transactional memory (Intel TSX [55]), if a problem occurs during the transaction, all the architectural states in the transaction will be rolled back by a transaction abort, suppressing the exception that occurred in the middle of the transaction [102, 115]. Another way is to put the instruction that would cause exception in a mis-predicted branch. In this survey, even if the exception is suppressed later, we categorize the attack to be due to exceptions.

4.3 Interrupts

Another cause of transient execution is (external) interrupts. If a peripheral device or a different core causes an interrupt, the processor stops executing the current program, saves the states, and transfers control to the interrupt handler. In one common implementation, when stoping execution, the oldest instruction in the ROB will finish execution, and all the rest of the instructions in the ROB will be squashed; the instructions that were executed after the oldest instruction (but end up being squashed) are executed transiently. After the interrupt is handled, the current program may continue the execution; i.e., the instructions that are squashed will be fetched into the pipeline again.

4.4 Load-to-Load Reordering in Multi-Core Processors

The fourth possible cause of transient execution is load-to-load reordering. Current x86 architectures use the **total store order (TSO)** memory model [105]. In TSO, all observable load and store reordering are not allowed except store-to-load reordering, where a load bypasses an older store of a different address. To prevent a load-to-load reordering, if a load has executed but not yet retired and the core receives a cache invalidation for the line read by the load, the pipeline will be squashed. Transient execution occurs between the instruction issue and when the load-to-load reordering is detected.

4.5 Causes of Transient Execution in Known Attacks

Not all transient execution can be leveraged in an attack, and Table 2 shows the causes of transient execution in existing attacks. (Mis-)speculation is leveraged in Spectre attacks, e.g., [64]. Address speculation is leveraged in MDS attacks [84, 102, 115] and LVI [114]. Exceptions of loads or stores are leveraged in Meltdown attacks [72], Foreshadow attacks [113, 123], LVI [114], and so forth. Other types of exceptions, interrupts, and load-to-load reordering are not considered to be exploitable. Because the instructions that get squashed due to exceptions, interrupts, and load-to-load reordering are legal to be resumed later on, no extra data is accessible to the attacker during the transient execution.

Example code of different variants is shown in Figure 4. The victim code should allow a potential mis-speculation or exception to happen. In Spectre V1 [64], to leverage PHT, a conditional branch exists in the victim code followed by the gadget. Even if the condition `offset < arr1_len` is not true, the disclosure gadget may still execute transiently. The disclosure gadget first fetches the secret at `arr1[offset]` and then conducts a memory access whose address (`arr2[sec*c]`) depends on the secret value that encodes the secret in the cache states. Similarly, in Spectre V2 [64] and V5 [65, 78], the victim code has an indirect jump (or a return from a function) that uses BTB (or RSB) for prediction of the execution path. In Spectre V4 [77], to use STL, the victim code has a store following a load having potential address speculation. In LVI [114], a load that triggers a page fault (accessing `trusted_ptr`) will forward non-coherent data in the store buffer, which is

Table 2. Data Leaked by Different Transient Execution Attacks

	Causes of Transient Execution			**Example Attacks**	Coh. Data**						**Non-coh. Data****
					Hypervisor	Across VM	Kernel data	Across user app.	SGX	Sandbox	
Victim Executes Transiently	Speculation	Ctrl Flow	PHT	Spectre V1 [12, 22, 64, 104, 111]	☒	☒	☒	☒	☒	☒	☐
			BTB	Spectre V2 [10, 21, 64]	☒	☒	☒	☒	☒	☒	☐
			RSB	Spectre V5 [65, 78]	☒	☒	☒	☒	☒	☒	☐
		Addr.	STL	Spectre V4, LVI [77, 114]	☒	☒	☒	☒	☒	☒	☒
			LFB	LVI [114]	☒	☒	☒	☒	☒	☒	☒
		Value		No commercial implementation							
	Exception		*	LVI [114]	☒	☒	☒	☒	☒	☒	☒
	Interrupts			No known attack							
	Load-to-load reordering			No known attack							
Attacker Executes Transiently	Speculation	Ctrl Flow	*	No known attack							
		Addr.	STL	Fallout [84]	☐	☐	☐	☐	☐	☐	☒
			LFB	RIDL, ZombieLoad [102, 115]	☐	☐	☐	☐	☐	☐	☒
			Staging buf.	CROSSTALK [96]	☐	☐	☐	☐	☐	☐	☒
		Value		No commercial implementation							
	Exception		PF-US	Meltdown (V3) [72, 111]	☐	☐	☒	☐	☐	☐	☐
			PF-P	Foreshadow (L1TF) [113, 123]	☒	☒	☒	☒	☒	☐	☐
			PF-RW	V1.2 [63]	☐	☐	☐	☐	☐	☒	☐
			NM	LazyFP [107]	☐	☐	☐	☐	☐	☐	☒
			GP	V3a [23]	☐	☐	☒	☐	☐	☐	☐
	Interrupts			No known attack							
	Load-to-load reordering			No known attack							

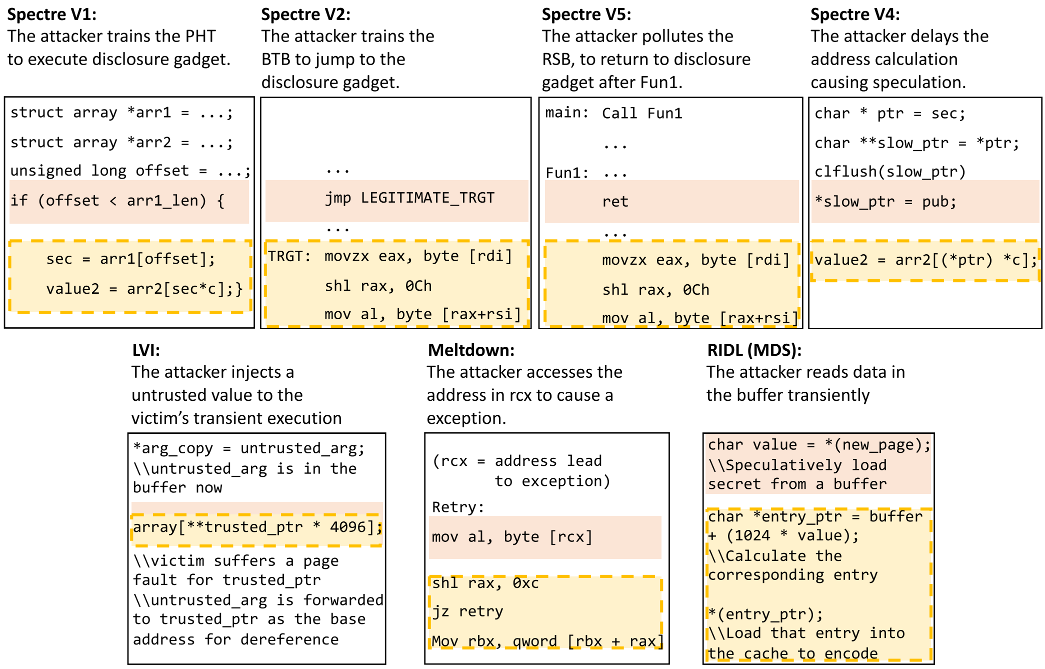
☒ indicates that the attack can leak the protected data; ☐ indicates that the attack cannot leak the data.
*indicates all hardware components that cause the corresponding transient execution; we combine them in the same row because the data leaked in the attacks are the same.
***Coh. Data* is short for coherent data. *Non-coh. Data* is short for non-coherent data.

injected by a malicious store (`*arg_copy = untrusted_ptr`), and then the secret data addressed by the injected value (`**untrusted_ptr`) is leaked. In Meltdown [72], the attacker code makes an illegal load to cause an exception. In MDS attack [84, 102, 115], a faulty load (`value=*(new_page)`) forwards non-coherent data in the buffer.

4.6 Metrics for Causes of Transient Execution

If the attacker wants to launch a transient execution attack, the attacker should be able to cause transient execution of the disclosure gadget in a controlled manner. We propose the following metrics to evaluate the different causes of transient execution in an attack:

- **Security Boundaries That Are Broken**: This metric indicates the security boundaries that are broken during the transient execution attacks—this will be discussed in Section 4.7.



Spectre V1: The attacker trains the PHT to execute disclosure gadget.

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {
    sec = arr1[offset];
    value2 = arr2[sec*c];}
```

Spectre V2: The attacker trains the BTB to jump to the disclosure gadget.

```
         ...
         jmp LEGITIMATE_TRGT
         ...
TRGT:  movzx eax, byte [rdi]
       shl rax, 0Ch
       mov al, byte [rax+rsi]
```

Spectre V5: The attacker pollutes the RSB, to return to disclosure gadget after Fun1.

```
main:  Call Fun1
         ...
Fun1:  ...
         ret
         ...
         movzx eax, byte [rdi]
         shl rax, 0Ch
         mov al, byte [rax+rsi]
```

Spectre V4: The attacker delays the address calculation causing speculation.

```
char * ptr = sec;
char **slow_ptr = *ptr;
clflush(slow_ptr)
*slow_ptr = pub;
value2 = arr2[(*ptr) *c];
```

LVI: The attacker injects a untrusted value to the victim's transient execution

```
*arg_copy = untrusted_arg;
\\untrusted_arg is in the
buffer now
array[**trusted_ptr * 4096];
\\victim suffers a page
fault for trusted_ptr
\\untrusted_arg is forwarded
to trusted_ptr as the base
address for dereference
```

Meltdown: The attacker accesses the address in rcx to cause a exception.

```
(rcx = address lead
    to exception)
Retry:
mov al, byte [rcx]
shl rax, 0xc
jz retry
Mov rbx, qword [rbx + rax]
```

RIDL (MDS): The attacker reads data in the buffer transiently

```
char value = *(new_page);
\\Speculatively load
secret from a buffer
char *entry_ptr = buffer
+ (1024 * value);
\\Calculate the
corresponding entry

*(entry_ptr);
\\Load that entry into
the cache to encode
```

Fig. 4. Example code of transient execution attacks. Code highlighted in orange triggers transient execution. Code highlighted in yellow with dashed frame is the disclosure gadget.

- **Required Control of the Victim's Execution**: This metric evaluates whether the attacker needs to control the execution of victim code—details will be discussed in Section 4.8.
- **Required Level of Sharing**: This metric evaluates how close the attacker should co-locate with the victim and whether the attacker should share memory space with the victim to trigger the transient execution in a controlled manner—details will be discussed in Section 4.9.
- **Speculative Window Size**: This metric indicates how many instructions can be executed transiently—the speculation window size will be discussed in more detail in Section 4.10.

4.7 Security Boundaries That Are Broken in Different Attacks

As discussed in Section 3.1, the attacker's goal is to access the coherent or non-coherent data across the security boundaries in the system. Table 2 lists the type of data and the security boundaries across which the data can be leaked in the known transient execution attacks, assuming all the instructions in the disclosure gadget can execute transiently and the covert channel can transmit information to the attacker.

If the victim is executing transiently, the disclosure gadget can read any coherent data that the victim could access architecturally, even if the semantics of the victim code do not intend it to access the data [64]. Hence, in these attacks, the attacker can break the isolation between the victim and the attacker and learn data in the victim's domain. For example, the SWAPGS instruction is a privileged instruction that is usually executed after switching from user-mode to kernel-mode. If SWAPGS is executed transiently in the kernel-mode in the incorrect path, kernel data can be leaked [12]. When the victim is executing transiently, the attacker can also learn the non-coherent data (e.g., stale data) and also data that depends on non-coherent data (e.g., data in an address that depends on non-coherent data). For example, in Spectre V4 [77], stale data that contains the address of the secret data in the store buffer is forwarded to the younger instructions transiently, and the disclosure gadget accesses and transmits the secret data to the attacker. As another example, in LVI attack [114], the attacker injects malicious value through buffers, such as STL or LFB, causing a victim's transient execution that depends on a value controlled by the attacker and potentially leaks the value in an address controlled by the attacker.

If the attacker is executing transiently, transient execution allows the attacker to access illegal data directly. As shown in Table 2, the security boundaries that are broken depend on the causes of transient execution. In some processor implementations, even if a load causes an exception due to permission violation, the coherent data might still be propagated to the following instructions and learned by the attacker. For example, in Meltdown [72], privileged data is accessible transiently to an unprivileged user even if the privileged bit in the page table is set. In **L1 terminal fault (L1TF)** [123], secret data in the L1 cache is accessible transiently even if the present bit in the page table is not set. In Table 2, the attacks leveraging exceptions are categorized by the cause of the exception, e.g., **page fault (PF)**, and the related permission bit. Non-coherent data present in the micro-architecture buffers, e.g., LFBs or **store buffer (STB)**, can sometimes be accessed by the attacker in transient execution [84, 102, 115]. In addition, in CROSSTALK [96], a hardware buffer called the staging buffer is discovered. The staging buffer is used for some type of off-core reads; e.g., it is used by the RDRAND instruction that requests **Digital Random Number Generator (DRNG)**, and the CPUID instruction that reads from **Machine-Specific Registers (MSRs)**. The staging buffer is shared across cores, and thus, the CROSSTALK paper demonstrated a cross-core attack where the victim fetched some data from DRNG, and the attacker then learned the random number stored in the staging buffer during transient execution.

4.8 Required Control of the Victim's Execution

For the attacks leveraging mis-speculation, (mis-)training is an essential setup phase to steer the control flow to execute the desired disclosure gadget. The (mis-)training can be part of victim code, which is triggered by the attacker, as shown in Figure 3(b) and Table 1. In the example of Spectre V1, the attacker can first provide inputs to execute the setup gadget to train the branch predictor (i.e., PHT) to execute the target branch, because in this way the training code will always share the branch predictor with the attack code. In this case, the attacker should be able to control the execution of victim code. The (mis-)training code can also be a part of the attacker's code and run in parallel with the victim code, as shown in Figure 3(d), e.g., in Spectre V2. Then, it does not require control of the victim's execution for setup, but it requires the attacker's training thread and the victim's thread should be co-located to share the same prediction unit (e.g., BTB). Further, to share the same entry of the prediction unit, if the prediction unit is indexed by physical address, the attacker and the victim should also share the same memory space to share the entry, which will be discussed in the next subsection.

For the attacks that leverage exceptions, the instructions that follow the exception will be executed transiently, and thus, no mis-training is required, but the attacker needs to make sure the disclosure gadget is located in the code such that it is executed after the exception-causing instruction.

4.9 Required Sharing to Setup Transient Execution

As shown in Table 1, in some scenarios, the setup phase and the transient execution phase are run by different parties, e.g., Figures 3(c) through 3(f), or in the attacker's different software threads, e.g., Figures 3(g) and 3(h). These cases require that the setup phase shares the same prediction unit (entry) with the transient execution phase. One common attack scenario is that the attacker mis-trains the prediction unit to lure the execution of the disclosure gadget of the victim, e.g., Figure 3(d). Hardware sharing can be the following:

- **Same thread:** The attacker and the victim (if both of them are executing) or the attacker's software threads (if only the attacker is executing) are running on the same logical core (hardware thread) in a time-sliced setting, and there might be context switches in between.
- **Same core, different threads:** The attacker and the victim (if both of them are executing) or the attacker's threads (if only the attacker is executing) are running on different logical cores (hardware threads) through SMT on the same physical core.
- **Same chip, different cores:** The attacker and the victim (if both of them are executing) or the attacker's threads (if only the attacker is executing) are on different CPU cores but are sharing LLC, memory bus, and other peripheral devices.
- **Same motherboard, different chip:** The attacker and the victim (if both of them are executing) or the attacker's threads (if only the attacker is executing) share memory bus and peripheral devices.

Some prediction units have multiple entries indexed by address or thread ID, and in that case, the attacker needs to share the same entry of the prediction unit with the victim during the setup. To share the same entry, the attacker needs to control the address to map to the same predictor entry as the victim. The address space can be one of the following:

- **In the same address space:** In this case, the attacker and the victim have the same virtual-to-physical address mapping.
- **In different address spaces with shared memory:** In this case, the attacker and the victim have different virtual-to-physical address mappings, but some of the attacker's pages and the victim's pages map to the same physical pages. This can be achieved by sharing dynamic libraries (e.g., `libc`).
- **In different address spaces without shared memory:** The attacker and the victim have different virtual to physical address mapping. Further, their physical addresses do not overlap.

In the following, we discuss the level of sharing required to trigger transient execution of disclosure gadget for an attack leveraging mis-speculation. In particular, the scenario depends on the implementation, and thus, we discuss each of the prediction units in Intel processors in detail.

4.9.1 Control Flow Prediction. To predict the branch direction, modern branch predictors use a hybrid mechanism [32, 57, 81, 83, 106]. One major component of the branch predictor is the PHT. Typically, a PHT entry is indexed based on some bits of the branch address, so a branch at a certain virtual address will always use the same entry in the PHT. In each entry of the PHT, a saturating counter stores the history of the prior branch results, which in turn is used to make future predictions.

To predict the branch targets, a BTB stores the previous target address of branches and jumps. Further, a return instruction is a special indirect branch that always jumps to the top of the stack. The BTB does not give a good prediction rate on return instructions, and thus, RSB has been introduced in commercial processors. The RSB stores N most recent return addresses.

Table 3. Level of Sharing and (Mis-)training the Prediction Unit on Intel Processors

	Prediction Unit	**Same Thread**	**Same Core, Different Thread**	**Same Chip, Different Core**	**Same Motherboard**
Ctrl Flow	PHT [36, 63]	f(virtual addr)	f(virtual addr)	–	–
	BTB [34, 64]	f(virtual addr)	f(virtual addr)[a]	–	–
	RSB [78]	Not by address[b]	–	–	–
Address	STL [56, 84]	f(physical addr)[c]	–	–	–
	LFB [102, 115]	Not by address	Not by address	–	–
	Other[d]				
Value	No commercial implementation				

"–" indicates the prediction unit is not possible to be trained under the corresponding sharing setting. Otherwise, the prediction unit can be trained and "f(virtual addr)" indicates the prediction unit is indexed by a function of the virtual address, "f(physical addr)" indicates the prediction unit is indexed by a function of the physical address, and "not by address" indicates the prediction unit is not indexed by addresses.

[a] Conflicting results are presented in different publications [34, 64].

[b] Most OSs overwrite RSBs on context switches.

[c] STL is possible after context switch but not on SGX enclave exit.

[d] In [102], it is indicated that there could be other structures that forward data speculatively.

In Intel processors, the PHT and BTB[5] are shared for all the processes running on the same physical core (same or different logical core in SMT). The RSB is dedicated to each logical core in the case of hyper-threading [78]. Table 3 shows whether the prediction unit can be trained when the training code and the victim are running in parallel in different settings. The results are implementation dependent and Table 3 shows the result from Intel processors.

The prediction units sometimes have many entries, and the attacker should use the same entry as the victim for mis-training. The attacker and the victim will use the same entry only if they are using the same index. When the prediction unit is indexed by virtual address, the attacker can train the prediction unit from another address space using the same virtual address as the victim code. If only part of the virtual address is used as the index, which is shown as `f(virtual addr)` in Table 3, the attacker can even train with an aliased virtual address, which maps to the same entry of the prediction unit as the victim address. The RSB is not indexed by the address; rather, it overflows when many nested calls are made, and this creates conflicts when there are more than N nested calls, and will cause mis-speculation.

[5] In [34], the authors did not observe BTB collision between logical cores. However, it is demonstrated that the attacker can mis-train the indirect jump of a victim when they are two hyper-threads sharing the same physical core in [64]. Thus, we think BTB is shared across hyper-threads in some of the processors.

4.9.2 Address Speculation. One of the uses of address speculation is in the memory disambiguation to resolve read-after-write hazards, which are the data dependencies between instructions in out-of-order execution. In Intel processors, there are two known uses of address speculation. First, loads are assumed not to conflict with earlier stores with unknown addresses, and speculatively STL forwarding will not happen. When the address of a store is later resolved, the addresses of younger loads will be checked. And if store-to-load forwarding should have happened and data dependence has been violated, the loads will be flushed, and the new data is reloaded from the store, as shown in the attacks [77, 99]. Second, for performance, when the address of a load *partially* matches the address of a preceding store, the store buffer will forward the data of the store to the load speculatively, even though the full addresses of the two may not match [84]. In the end, if there is mis-speculation, the load will be marked as faulty, flushed, and reloaded again.

Another use of address speculation is in conjunction with the LFB, which is the buffer storing cache-lines to be filled to the L1 cache. LFB may forward data speculatively without knowledge of the target address [102, 115]. Address speculation may also be used in other hardware structures in Intel processors, as indicated in [102].

To trigger address speculation, the availability of the address should be delayed to force the hardware to predict the address. One way is to make the address calculation depend on some uncached data, as in Spectre V4 [77]. Another way is to use a newly mapped page, so that the physical address is available only after OS handles the page-in event, as in [115]. In an extreme case, the speculation can even be caused by a NULL pointer or an invalid address, and then the error is suppressed in the attacker code, as in attack [102]. In STL, the entries are indexed by a function of physical addresses. In this case, the training code needs to share memory space with the victim to achieve an attack.

4.9.3 Value Prediction. There is no commercial processor that implements value prediction yet. Thus, there are no known exploits that abuse value prediction. However, similar to control flow prediction, if the predictor is based on states that are shared between different threads and not cleaned up during context switch, the prediction can be hijacked by the attacker.

4.10 Speculative Window Size

To let an attack happen, there should be a large enough speculative window for the disclosure gadget to finish executing transiently, as shown in Figure 1. The speculative window size is the window from the time the transient execution starts (instruction fetch) to the time the pipeline is squashed. In attacks leveraging mis-speculation, the speculative window depends on the time the prediction is resolved. In a conditional branch, the time depends on the time to resolve the branch condition; in indirect jump, this depends on the time to obtain the target address; and in address speculation, this depends on the time to get the virtual and then the physical address. In [79], a tool called *Speculator* is proposed to reverse-engineer the micro-architecture using hardware performance counters. The results of *Speculator* show that the speculative window of branches that depend on uncached data is about 150 cycles on Intel Broadwell, about 300 cycles on Intel Skylake, and about 300 cycles on AMD Zen, and the speculative window of STL is about 55 cycles on Intel Broadwell. In attacks leveraging exceptions, the speculative window depends on the implementation of exceptions. To make the speculative window large enough for the disclosure gadget, the attacker can delay obtaining the result of the branch condition or the addresses by leveraging uncached loads from main memory, chains of dependent instructions, and so forth.

5 COVERT CHANNELS

Transient execution enables the attacker to access the secret data transiently, and a covert channel[6] is required for the attacker to eventually obtain the secret data. There is a distinction between *conventional channels,* where the encoding happens in the software execution path, and *transient execution channels,* where the encoding phase is executed transiently. In this survey and this section, we focus on covert channels that can be used in transient execution attacks; however, these can also be used as conventional covert channels. There are two parties in a covert channel: the sender and the receiver. In the covert channels, the sender execution will change some micro-architectural states and thus encode information into the channel. Meanwhile, the receiver will observe the change to extract information, e.g., by observing the execution time.

5.1 Assumptions about Covert Channels

This survey focuses on covert channels that do not require physical presence and that only require the attacker's software (or software under the attacker's control) to be executing on the same system as the victim. Thus, we do not consider physical channels, such as power [39], EM field [80], acoustic signals [6, 40], and so forth. There are certain physical channels that can be accessed from software and not require physical presence, such as temperature [128]. However, thermal conduction is slow and the bandwidth is limited.

When considering channels that do not require a physical presence, in general, any sharing of hardware resources between users or programs can lead to a covert channel [119]. The receiver user or program can try to observe the states of the hardware through the execution time, the values of **hardware performance counters (HPCs)**, or system behavior. The most commonly used observation by the receiver of the covert channels is the timing of execution. In today's processors, components are designed to achieve a better performance, and thus, the execution time contains information about whether a certain hardware unit is available during execution (e.g., port), whether the micro-architectural states are optimal for the code (e.g., cache hits or misses), and so forth. To observe the hardware states via timing, a timer is needed. In x86, the *rdtscp* instruction can be used to read a high-resolution timestamp counter of the CPU, and thus can be used to measure the latency of a chosen piece of code. When the *rdtscp* is not available, a counting thread can be used as a timer [103].

The receiver can also gain information from HPCs. HPCs have information about branch prediction, cache, TLB, and so forth, and have been used in covert channel attacks [36]. However, HPCs must be configured in kernel mode [27], and thus are not suitable for unprivileged attackers.

The receiver can further observe the state of the hardware by the system behaviors. In Prime+Abort attack [30], for example, Intel **Transactional Synchronization Extensions (TSXs)** [55] can be exploited to allow an attacker to receive an abort (call-back) if the victim process accessed a critical address.

In transient execution attacks, several covert channels can also be used in series. Here, we focus on the channels where the receiver can eventually decode data from the architectural states, e.g., values in register files. For example, in the Fetch+Bounce covert channel [99], first, the secret is encoded into the TLB states, which affect the STL forwarding, and then a cache Flush+Reload covert channel is used to observe the STL forwarding results. The first channel can only be observed by instructions in transient execution, and the states will be removed when the instruction retires. We only consider the second covert channel to be critical for transient execution attack because it allows the attacker to observe the secret from the architectural states.

[6] The channel is considered a covert channel, not a side channel [64], because the attacker has control over the disclosure gadget, which encodes the secret.

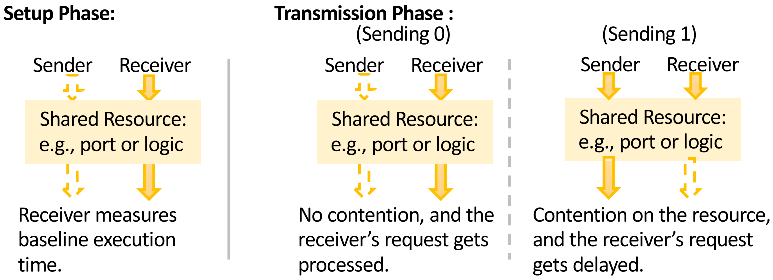

Setup Phase:
Sender
Receiver
Shared Resource: e.g., port or logic
Receiver measures baseline execution time.
Transmission Phase :
(Sending 0)
Sender
Receiver
Shared Resource: e.g., port or logic
No contention, and the receiver's request gets processed.
(Sending 1)
Sender
Receiver
Shared Resource: e.g., port or logic
Contention on the resource, and the receiver's request gets delayed.

Fig. 5. Steps for the sender and the receiver to transfer information through volatile covert channels. The yellow box shows the shared resource. The solid (dashed) arrow shows the shared resource is (is not) requested or used by the corresponding party.

5.2 Types of Covert Channels

We categorize the covert channels into volatile channels and persistent channels. In volatile channels, the sender and the receiver share the resource on the fly, and no states are changed; e.g., the two parties use a port or some logic concurrently. There is resource contention when the sender and the receiver communicate using this type of channel. In persistent channels, the sender changes the micro-architectural states, and the receiver can observe the state changes later, e.g., change of cache state. Although the states may be changed later, we call them persistent channels to differentiate from the volatile channels.

5.3 Volatile Covert Channels

In a *volatile covert channel*, there is contention for hardware resources between the sender and the receiver on the fly, and thus, the two should run concurrently, for example, as two hyper-threads in SMT processors, or running concurrently on two different cores. Another scenario is that the sender and the receiver are two parts of code in the same software thread that their instructions are scheduled to execute concurrently due to OoO [37]. As shown in Figure 5, in a volatile channel, the receiver first measures the baseline execution time when the sender is not using the shared resource. Then, the sender causes contention on the shared resource or not, depending on the bit of the message to be sent, while the receiver continues to measure the execution time. If the execution time increases, the receiver knows the sender is using the shared resource at the moment.

Execution units, ports, and buses are shared between the hyper-threads running concurrently on the same physical core and can be used for covert channels [2, 10]. There is also a covert channel leveraging the contention in the floating-point division unit [37]. L1 cache ports are also shared among hyper-threads. In Intel processors, L1 cache is divided into banks, and each cache bank can only handle a single (or a limited number of) requests at a time. CacheBleed [137] leverages the contention L1 cache bank to build a covert channel. Later, Intel resolved the cache bank conflicts issue with the Haswell generation. However, MemJam [86] attack demonstrates that there is still a false dependency of memory read-after-write requests when the addresses are of the same L1 cache set and offset for newer generations of Intel processors. This false dependency can be used for a covert channel. As shown in Table 4, the covert channel in execution ports and L1 cache ports can lead to covert channels within the same thread when the sender and the receiver code are executed in parallel due to OoO and between hyper-threads in the SMT setting.

Memory bus serves memory requests to all the cores using the main memory. In [126], it is shown that the memory bus can act as a high-bandwidth covert channel medium, and covert channel attacks on various virtualized x86 systems are demonstrated.

Table 4. Known Micro-architectural Covert Channels

Covert Channel Type		Level of Sharing				Bandwidth	Required Time Resolution of the Receiver (CPU cycles)
		Same Thread	Same Core, Different Thread	Same Chip, Different Core	Same Motherboard		
Volatile Covert Channels	Execution Ports [2, 10, 119]	☒	☒	□	□	Not given	50 vs. 80
	FP division unit [37]	☒	☒	□	□	~70kB/s	314 vs. 342
	L1 Cache Ports [86, 137]	☒	☒	□	□	Not given	36 vs. 48
	Memory Bus [126]	☒	☒	☒	☒	~700 B/s	2,500 vs. 8,000
Persistent Covert Channels	AVX2 unit [104]	☒	☒	□	□	>0.02B/s	200 vs. 550
	PHT [35, 36]	☒	☒	□	□	Not given	65 vs. 90
	BTB [34, 122]	☒	☒	□	□	Not given	56 vs. 65
	STL [56]	☒	□	□	□	Not given	30 vs. 300
	TLB [42, 53, 99]	☒	☒	□	□	~5kB/s per set	105 vs. 130[a]
	L1, L2 (tag, LRU) [62, 129, 130]	☒	☒	□	□	~1MB/s per cache entry	5 vs. 15[b]
	LLC (tag, LRU) [15, 76]	□	□	☒	□	~0.7MB/s per set	500 vs. 800
	Cache Coherence [111, 135]	□	□	☒	☒	~1MB/s per cache entry	100 vs. 250[c]
	Cache Directory [134]	□	□	☒	□	~0.2MB/s per slice	40 vs. 400
	DRAM row buffer [93]	□	□	☒	☒	~2MB/s per bank	300 vs. 350

☒ indicates that the attack is possible to leak the protected data; □ indicates that the attack cannot leak the data.

[a] Depending on the level of TLB used, the required time resolution varies. The biggest one is shown.

[b] Shows the time resolution for covert channel using L1 cache.

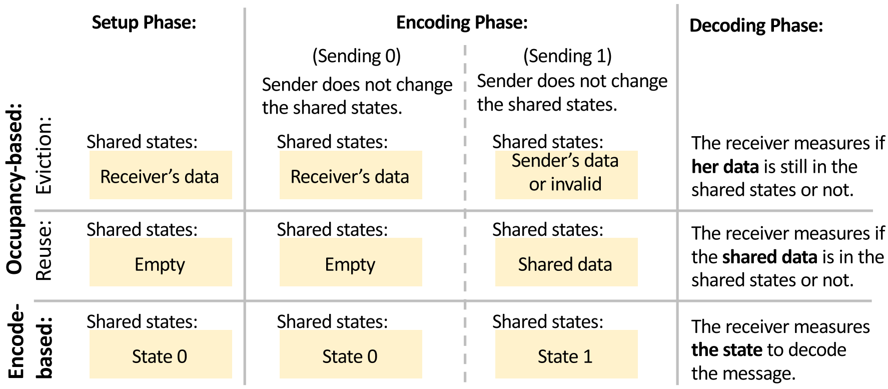
[c] Depending on the setup, the required time resolution varies. The biggest one is shown.

5.4 Persistent Covert Channels

In a *persistent channel*, the sender and the receiver share the same micro-architectural states, e.g., registers, caches, and so forth. Different from volatile covert channels, the state will be memorized in the system for a while. And the sender and the receiver do not have to execute concurrently. Depending on whether the state can only be used by one party or can be directly accessed by multiple parties, we classify the persistent channels into occupancy based and encode based, as shown in Figure 6.

5.4.1 Occupancy-Based Persistent Covert Channels. When a state or data can only be used by one party (e.g., registers, cache), an occupancy-based covert channel may be built. The sender can occupy the states or data to affect the execution of the receiver.

• *Eviction-Based Persistent Channels:* In this channel, the sender and the receiver will compete and evict the other party to occupy some states to store their data or metadata to (de-)accelerate their execution. One example of the eviction-based channel is the Prime+Probe attack [45, 91, 92, 131, 134]. The receiver first occupies a cache set (i.e., primes). Then, the sender may use the state for his or her data or not, depending on the message to be sent. And in the end, the receiver reads (i.e., probes) his or her data that were used to occupy the cache set in the first step to see whether those data are still in the cache by measuring the timing, as shown in the first row of Figure 6. Other



		Setup Phase:	Encoding Phase: (Sending 0) Sender does not change the shared states.	Encoding Phase: (Sending 1) Sender does not change the shared states.	Decoding Phase:
Occupancy-based:	Eviction:	Shared states: Receiver's data	Shared states: Receiver's data	Shared states: Sender's data or invalid	The receiver measures if **her data** is still in the shared states or not.
	Reuse:	Shared states: Empty	Shared states: Empty	Shared states: Shared data	The receiver measures if the **shared data** is in the shared states or not.
Encode-based:		Shared states: State 0	Shared states: State 0	Shared states: State 1	The receiver measures **the state** to decode the message.

Fig. 6. Steps for the sender and the receiver to transfer information through different types of persistent covert channels.

examples of the eviction-based channel are cache Evict+Time attack [9, 91], the covert channel in the DRAM row buffer [93].

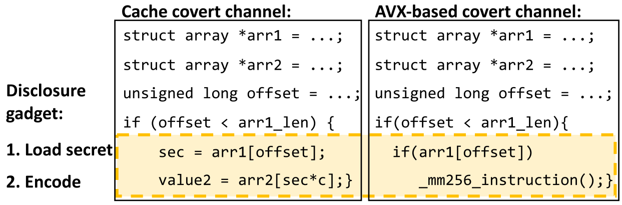
Another possible contention is that the sender needs to use the same piece of data (e.g., needs exclusive access to the data for write), and thus, the receiver's copy of data can be invalidated. Some state is used for tracking the relationship of data in different components, which can cause the data in one component to be invalidated. For example, a cache coherency policy can invalidate a cache line in a remote cache, and thus, it results in a covert channel between threads on different cores on the same processor chip [111, 135]. The cache directory keeps the tags and cache coherence state of cache lines in the lower levels of cache in a non-inclusive cache hierarchy and can cause eviction of a cache line in the lower cache level (a remote cache relative to the sender) to build a covert channel [134].

- *Reuse-Based Persistent Channels:* In this channel, the sender and the receiver will share some data or metadata, and if the data is stored in the shared state, it could (de-)accelerate both of their execution. The cache Flush+Reload attack [44, 136] transfers information by *reusing* the same data in the cache. The receiver first cleans the cache state. Then, the sender loads the shared data or not. And in the end, the receiver measures the execution time of loading the shared data, as in Figure 6. If the sender loads the shared data in the second step, the receiver will observe faster timing compared to the case when the sender does not load the shared data. There are other reuse-based attacks, such as the Cache Collision attack [13] and the cache Flush+Flush attack [43].

Prediction units can also be leveraged for such covert channels due to a longer latency for misspeculation. For example, PHT [22, 33, 35, 36, 139], BTB [34, 122], and STL [56] have been demonstrated to be usable for constructing covert channels. For example, when sharing BTB, the sender and the receiver use the same indirect jump source, ensuring the same BTB entry is used. If the receiver has the same destination address as the sender, the BTB will make a correct prediction, resulting in a faster jump.

5.4.2 *Encode-Based Persistent Covert Channels.*

In encode-based persistent covert channels, the sender and the receiver can both directly change and probe the shared state. One example of such a channel is the AVX channel [104]. There are two AVX2 unit states: power-off and power-on. To save power, the CPU can power down the upper half of the AVX2 unit by default. In step 2, if the sender then uses the AVX2 unit, it will power on the unit for at least 1 ms. In step 3, the receiver can measure whether the AVX2 unit is powered on by measuring the time of using AVX2 unit. In



Cache covert channel:

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if (offset < arr1_len) {
    sec = arr1[offset];
    value2 = arr2[sec*c];}
```

AVX-based covert channel:

```
struct array *arr1 = ...;
struct array *arr2 = ...;
unsigned long offset = ...;
if(offset < arr1_len){
  if(arr1[offset])
     _mm256_instruction();}
```

Fig. 7. Example disclosure gadgets for different covert channels.

this way, the sender encodes the message into the state of the AVX2 unit, as shown in Figure 6. Other examples are the covert channels leveraging cache LRU states [15, 62, 129].

5.5 Disclosure Gadget

The covert channel is used in the disclosure gadget to transfer the secret to be accessible to the attacker architecturally. The disclosure gadget usually contains two steps: (1) load the secret into a register and (2) encode the secret into a covert channel. As shown in Figure 7, the disclosure gadget code depends on the covert channel used. For covert channels in the memory hierarchy (e.g., cache side channel), it will consist of memory access whose address depends on the secret value. For AVX-based covert channels, the disclosure gadget encodes the secret by using (or not using) AVX instruction.

5.6 Metrics for Covert Channels

We propose the following metrics to compare different covert channels:

- **Level of Sharing**: This metric indicates how the sender and the receiver should co-locate. As shown in Table 4, some of the covert channels only exist when the sender and the receiver share the same physical core. Other attacks exist when the sender and the receiver share the same chip or even the same motherboard.
- **Bandwidth**: This metric measures how fast the channel is. The faster the channel, the faster the attacker can transfer the secret. Table 4 compared the bandwidth of different covert channels. Usually, the bandwidth is measured in a real system considering the noise from activities by other software and the operating system.
- **Time Resolution of the Receiver**: As shown in Figures 5 and 6, the receiver needs to measure and differentiate different states. For a timing channel, the time resolution of the receiver's clock decides whether the receiver can observe the difference between the sender sending 0 or 1. The last column of Table 4 shows the timing difference between states. Some channels, such as cache L1, require a very high-resolution clock to differentiate 5 cycles from 15 cycles, while the LLC covert channel only needs to differentiate 500 cycles from 800 cycles, and the receiver only needs a coarse-grained clock.
- **Retention Time**: This metric measures how long the channel can keep the secret. In some of the covert channels (volatile channels in Section 5.3), no state is changed, e.g., the channel leveraging port contention [2]. The retention time of such channels is zero, and the receiver must measure the channel concurrently when the sender is sending information. Other covert channels (persistent channels in Section 5.4) leverage state change in microarchitecture, and the retention time depends on how long the state will stay; for example, the AVX2 unit will be powered off after about 1 ms. If the receiver does not measure the state in time, he or she will obtain no information. For other states, such as register, cache,

Table 5. Taxonomy of the Existing Transient Execution Attacks

	Cause of Transient Execution					
Covert Channel	PHT	BTB	RSB	STL	LFB	Exception
Execution Ports	□	[10]	□	□	□	□
L1 Cache Ports	□	□	□	□	□	□
Memory Bus	□	□	□	□	□	□
AVX2 unit	[104]	□	□	□	□	□
FP div unit	[37]	□	□	□	□	[37]
PHT	[22]	□	□	□	□	□
TLB	□	□	□	□	□	□
L1, L2 (tag, LRU)	[64]	[21, 64]	[65, 78]	[77, 84]	[102, 115]	[23, 63, 72, 107, 113, 114, 123]
LLC (tag, LRU)	□	□	□	□	□	□
Cache Coherence	[111]	□	□	□	□	[111]
Cache Directory	□	□	□	□	□	□
DRAM row buffer	□	□	□	□	□	□
Other Channel	□	□	□	□	□	□

□ shows attacks that are possible but not demonstrated yet.

and so forth, the retention time depends on the usage of the unit and when the unit will be used by another user.

5.7 Comparison of Covert Channels

Table 4 lists different micro-architectural covert channels. The existence of a covert channel depends on whether the unit is shared in the attack setting. For example, AVX2 units, TLB, and the L1/L2 caches are shared among programs using the same physical core. Therefore, a covert channel can be built among hyper-threads and threads sharing a logical core in a time-sliced setting. The LLC, cache coherence states, and DRAM are shared among different cores on the chip, and therefore, a covert channel can be built between different cores or different chips.

Some covert channels may use more than one component listed in Table 4. For example, in the cache hierarchy, there could be multiple levels of caches shared among the sender and the receiver. In the Flush+Reload cache covert channel, the receiver can use the *clflush* instruction to flush a cache line from all the caches, and the sender may load the cache line into L1/L2 of that core or the shared LLC. If the sender and the receiver are in the same core, then the receiver will reload the data from L1. If the sender and the receiver are in different cores and only sharing the LLC, the receiver will reload the data from the LLC. Therefore, even with the same covert channel protocol, the location of the covert channel depends on the actual setting of the sender and the receiver.

As shown in Table 4, the channels in caches have relatively high bandwidth (~1 MBits/s), which allows the attacker to launch efficient attacks. Covert channels in AVX and TLB are slower but enough for practical attacks.

6 EXISTING TRANSIENT EXECUTION ATTACKS

As we have discussed, the transient execution attacks contain two parts: triggering transient execution to obtain data that is otherwise not accessible (discussed in Section 4) and transferring the data via a covert channel (discussed in Section 5). Table 5 shows the attacks that are demonstrated in different academic publications and CVEs. For demonstrating different speculation primitives, researchers usually use the covert channel in caches (row L1, L2 in Table 5). This is because the

Table 6. Known Transient Execution Attacks on Different Platforms

Cause of Transient Execution		Intel	AMD [4, 16]	Arm [5, 16]	RISC-V[41]
Control Flow	PHT (V1)	☒	☒	☒	☒
	BTB (V2)	☒	☒	☒	☒
	RSB (V5)	☒	□	☒	□
Address Speculation	STL (V4,MDS)	☒	□	☒	□
	LFB (MDS)	☒	□	□	☒
Exception	PF-US (V3)	☒	□	☒	□
	PF-P (L1TF)	☒	□	□	□
	PF-RW (V1.2)	☒	□	☒	□
	NM (LazyFP)	☒	□	□	□
	GP (V3a)	☒	□	☒	□
	Other	☒	☒	☒	□

☒ indicates that an attack of the type on the platform; □ indicates that there is no known attack.

cache Flush+Reload covert channel is simple and efficient. For demonstrating different covert channels used in transient execution attacks, researchers usually use PHT (Spectre V1). This is because Spectre V1 is easy to demonstrate. Note that every entry in the table can become an attack. For mitigations, each entry of the table should be mitigated; either mitigate all the covert channels or prevent accessing the secret data in transient execution.

If the victim executes transiently, the victim will encode the secret into the channel, and the behavior cannot be analyzed from the software semantics without a hardware model of how the hardware executes the software and how prediction mechanisms behave (and what micro-architectural states they change). If the attacker executes transiently, the micro-architecture propagates data that is not allowed to propagate at the ISA level (propagation is not visible at the ISA level but can be reconstructed through covert channels that observe the changes in the micro-architecture states). To formally model and detect the behavior, a new micro-architectural model, including the transient behavior, should be used [20, 46, 47, 49, 82].

6.1 Feasibility of Existing Attacks

Attacks leveraging mis-speculation require the attacker to mis-train the prediction unit in the setup phase to let the victim execute gadgets speculatively (Section 3.3.3 and Section 4.8). To be able to mis-train, the attacker either needs to control part of the victim's execution to generate the desired history for prediction or needs to co-locate with the victim on the same core. MDS attacks also require the attacker and the victim to share the same address speculation unit. As shown in Table 3 on the level of sharing required for the setup phase, the prediction unit is shared only within a physical core, and for some prediction unit, they may not even be shared between hyper-threads. In practice, it is not trivial to co-locate the setup phase and the transient execution phase on the same core.

When a covert channel across processes is required, the sharing of hardware is needed (Table 1 on the attack scenarios and Section 5.7), which requires the co-location between the transient execution phase and the decoding phase. Furthermore, for a specific attack implementation, only one disclosure primitive is used, and the attack can be mitigated by blocking the covert channel.

6.2 Attacks on Different Commercial Platforms

Most of the existing studies focus on Intel processors, Table 6lists the known attacks on processors from different venders, such as AMD [4, 16], Arm [5, 16], and RISC-V [41]. As shown in the table,

Table 7. Comparison of Different Mitigation Schemes in Micro-architecture

Mitigation Schemes	Performance Overhead
Fence after each branch	88% [132]
Stop propagating all data	30–55% [8]; 21% [7]; 20–51% [122]; 8.5% [139]; 4.19% [138]
Stop propagating all data to cache changes	7.7% [110], 13% [68]
Stop propagating all data to Flush+Reload	7% [68]
Stop propagating all tagged secret data	71% for security-critical applications, < 1% for real-world workloads [38, 100]
Partitioned cache	1–15% [62]
Stop (Undo) speculative change in caches	7.6% [132]; 11% [98]; 4% [1]; 5.1% [97]; 2–6% [61]; 8.3% [125];

Spectre attack and its variants using branch prediction are found on all the platforms; this is because branch speculation is fundamental in modern processors. Other types of transient execution depend on the micro-architecture implementation of speculation units and show different results on different platforms.

7 MITIGATIONS IN MICRO-ARCHITECTURE DESIGN

Given the severity of transient execution attacks, numerous mitigations have been proposed. In this section, we focus on micro-architectural mitigations to attacks that occur when the victim executes transiently under wrong control flow prediction. As shown in Table 6, such attacks that leverage control flow prediction are more fundamental and affect all modern computer architectures. Attacks that leverage address speculation and exceptions are implementation dependent, and we consider them as implementation bugs. They can be fixed, although the performance penalty is unknown for now. We focus on the possible future micro-architectural designs that are safe against control flow prediction. Thus, software mitigation schemes, such as [17, 18, 88], and software vulnerability detection schemes [89, 116, 117] are not discussed in detail.

7.1 Mitigating Transient Execution

The simplest mitigation is to stop any transient execution. However, it will come with a huge performance overhead; e.g., adding a fence after each branch to stop branch prediction causes 88% performance loss [132].

7.1.1 Mitigating the Triggering of Transient Execution. To mitigate attacks where the victim executes transiently, one solution is to limit the attackers' ability to mis-train the prediction units to prevent the disclosure gadget to be executed transiently (the first metric in Section 4.6). The prediction units (e.g., PHT, BTB, RSB, STL) should not be shared among different users. This can be achieved by static partition for concurrent users and flushing the state during context switches. For example, there are ISA extensions for controlling and stopping indirect branch predictions [3, 54]. In [110], a decode-level branch predictor isolation technique is proposed, where a special micro-op that clears the branch predictor states will be executed when the security domain switches. In [143], it is proposed to use a thread-private random number to encode the branch prediction table, to build isolation between threads in the branch predictor. However, for both proposals, if the attacker can train the prediction unit by executing victim code with certain input (e.g., always provide valid input in Spectre V1), isolation is not enough.

There is also mitigation in software to stop speculation by making the potential secret data depend on the result of the branch condition leveraging data dependency, e.g., masking the data with the branch condition [17, 88], because current processors do not speculate on data. However, this solution requires identifying all control flow dependency and all disclosure gadgets, to figure out all possible control flow that could lead to the execution of the disclosure gadgets, and to patch each of them. It is a challenge to identify all (current and future) disclosure gadgets, because disclosure gadgets may vary due to the encoding used for different covert channels, and formal methods that model the micro-architecture behavior are required [46, 47].

7.1.2 Mitigating Transient Execution of the Disclosure Gadget. To mitigate leaking secrets during the transient execution attacks, one way is to prevent the transient execution of the disclosure gadget, i.e., to stop loading of secrets in transient execution or stop propagating the secret to younger instructions in the disclosure gadget transiently. For attacks where the attacker executes transiently, it means stopping propagating secret data to the younger instructions. For attacks where the victim executes transiently, however, the logic may not know which data is secret. To mitigate the attacks, secret data should be tagged with metadata as in secure architecture designs, which will be discussed in Section 7.1.3.

Another solution is that data cannot be propagated speculatively, and thus, cannot be sent to covert channels speculatively, which can potentially prevent transient execution attacks with any covert channel. In *Context-Sensitive Fencing* [110], fences will be injected at the decoder level to stop speculative data propagation if there are potential Spectre attacks. In *NDA* [122], a set of propagation policies are designed for defending the attacks leveraging different types of transient executions (e.g., transient execution due to branch prediction or all transient execution), showing the trade-off between security and performance. Similarly, in *SpecShield* [7, 8], different propagation policies are designed and evaluated. In *Conditional Speculation* [68], the authors propose a defense scheme targeting covert channels in the memory system and propose an architecture where data cannot be transiently propagated to instructions that lead to changes in the memory system showing 13% performance overhead. To reduce performance overhead of the defense, they further change the design to only target Flush+Reload cache side channels, resulting in performance overhead of 7%. Furthermore, in *STT* [139], a dynamic information flow tracking-based micro-architecture is proposed to stop the propagation of speculative data to covert channels but reduce the performance overhead by waking up instructions as early as possible. **Speculative data-oblivious (SDO)** execution [138] is based on STT. To reduce performance overhead, SDO introduces new predictions that do not depend on operands (holding data potentially depending on speculative data). Specifically, speculative data-oblivious loads are designed to allow safe speculative load. The overhead to defend Spectre attacks is moderate, e.g., 7.7% reported in *Context-Sensitive Fencing* [110], 21% reported in *SpecShield* [7], 20 ∼ 51% (113% for defending all transient execution attacks) reported in *NDA* [122], 8.5% for branch speculation (14.5% for all transient execution) in *STT* [139], and 4.19% for branch speculation (10.05% for all transient execution) in *STT+SDO* [138].

InvarSpec [144] uses a combined compiler and hardware scheme. The compiler analyzes the control flow and data flow of the program to identify a set of safe instructions for each instruction. Here, for an instruction i, safe instructions are older instructions that cannot prevent i from becoming speculation invariant, even if they are not committed. Thus, instruction i can execute safely without waiting for safe instructions to reach the head of ROB. This method further reduces performance overhead upon other protections, e.g., reduces the overhead of InvisiSpec [132] from 15.4% to 10.9% on SPEC17. Different from STT, SpecShield, NDA, and others, the InvarSpec protects all data (including those in the register files) from leaking in transient execution.

There should be a large enough speculative window to let the disclosure gadget execute transiently for the attack to happen. The micro-architecture may be able to limit the speculation window size to prevent the encoding to the covert channel (the fourth metric in Section 4.6). However, the disclosure gadget can be very small and only contain two loads from L1 [129], which is only about 20 cycles in total. Detecting a malicious windowing gadget accurately can be challenging.

7.1.3 Mitigations in Secure Architectures. Secure architectures are designed to protect the confidentiality (or integrity) of certain data or code. Thus, secure architectures usually come with ISA extensions to identify the data or code to be protected, e.g., secret data region, and micro-architecture designs to isolate the data and code to be protected [25, 69, 108].

With knowledge about the data to be protected, hardware can further stop propagating secret data during speculation. The hardware can identify data that depends on the secret with taint checking, as proposed in [38, 64, 100, 110], and forbid tainted data to have micro-architectural side effects, or flush all the states on exit from the protected domain, to defend against persistent covert channels, and disable SMT to defend volatile covert channels. The overhead of such mitigation depends on the size of secret data to be protected. For example, as reported in *ConTExT* [100], the overhead is 71.14% for OpenSSL RSA encryption and less than 1% for real-world workloads. Similar overhead is reported in *SpectreGuard* [38]. Intel also proposes a new memory type, named **speculative-access protected memory (SAPM)** [58]. Any access to the SAPM region will cause instruction-level serialization, and speculative execution beyond the SAPM-accessing instruction will be stopped until the retirement of that instruction.

7.2 Mitigating Covert Channels

To limit the covert channels, one way is to isolate all the hardware across the sender and receiver, so any state changes caused by the sender will not be observable to the receiver. However, this is not always possible; e.g., in some attack types such as Meltdown, the attacker is both the sender and the receiver of the channel.

Another mitigation is to eliminate the sender of the covert channel in transient execution. For volatile covert channels, the mitigation is challenging. For permanent covert channels, there should not be speculative change to any micro-architectural states, or any micro-architectural state changes should be rolled back when the pipeline is squashed. Covert channels in memory systems, such as caches and TLBs, are most commonly used. Hence, most of the existing mitigations focus on cache and TLB side channels.

InvisiSpec [132] proposes the concept of "visibility point" of a load, which indicates the time when a load is safe to cause micro-architecture state changes that are visible to attackers. Before the visibility point, a load may be squashed and should not cause any micro-architecture state changes visible to the attackers. To reduce performance overhead, a "speculative buffer" is used to temporarily cache the load, without modifications in the local cache. After the "visibility point," the data will be fetched into the cache. For cache coherency, a new coherency policy is designed such that the data will be validated when stale data is potentially fetched. The gem5 [11] simulation results show a 7.6% performance loss for the SPEC 2006 benchmark [52]. Similarly, SafeSpec [60] proposes to add "shadow buffers" to caches and TLBs, so that transient changes in the caches and TLBs do not happen.

In *Muontrap* [1], "filter cache" (L0 cache) is added to each physical thread to hold speculative data. The proposed filter cache only holds data that is in Shared state, so it will not change the timing of accessing other caches. If the data is in Modified or Exclusive state in another cache and shared state in L0 is not possible without state change in the other cache, the access will be delayed until it is at the head of ROB. The cache line will be written through to L1 when the corresponding instruction commits. Different from the buffers in InvisiSpec [132] and SafeSpec [60], the filter cache is a real cache that is cleared upon a context switch, syscall, or when the execution changes security boundaries (e.g., explicit flush when exiting sandbox) to ensure isolation between security boundaries. Muontrap results in a 4% slowdown for SPEC 2006.

CleanupSpec [97] proposes to use a combination of undoing the speculative changes and secure cache designs. When mis-speculation is detected and the pipeline is squashed, the changes to the L1 cache are rolled back. For tracking the line addresses of speculative changes, 1 Kbyte storage overhead is introduced. To prevent the cross-core or multi-thread covert channel, partitioned L1 with random replacement policy and randomized L2/LLC are used. Because only a small portion of transient executions result in mis-speculations, the method shows an average slowdown of 5.1%. Similarly, *ReViCe* [61] proposes an undoing approach by adding a victim cache that stores the cache lines replaced by speculative loads. With 32-entry victim cache for L1 and 64-entry victim cache for L2, the performance overhead is 2–6%.

ReversiSpec [125] proposes a comprehensive cache coherence protocol considering speculative cache accesses. The proposed cache coherence protocol interface includes three operations: (1) speculative load, (2) merge when a speculative load is safe, and (3) purge when a speculative load is squashed. Compared to InvisiSpec [132], the speculative buffer only stores data when the data is not in the cache, and thus, less data movement will occur when a load is safe (merge). Compared to CleanupSpec [97], purge is fast as not all the changes have propagated into cache. The performance overhead is 8.3%.

Moreover, accessing speculative loads that hit in L1 cache will not cause side effects (except LRU state updates) in the memory system. Therefore, only allowing speculative L1 hits can mitigate transient execution attacks using covert channels (other than LRU) in the memory system. In *Selective Delay* [98], to improve performance, for a speculative load that misses in L1, value prediction is used. The load will fetch from deeper layers in the memory hierarchy until the load is not speculative. In their solution, 11% performance overhead is shown.

Meanwhile, many secure cache architectures are proposed to use randomization to mitigate the cache covert channels in general (not only the transient execution attacks). For example, *Random Fill cache* [74] decouples the load and the data that is filled into the cache, and thus, the cache state will no longer reflect the sender's memory access pattern. **Random Permutation (RP)***cache* [120], *Newcache cache* [75, 121], *CEASER cache* [95], and *ScatterCache* [124] randomize memory-to-cache-set mapping to mitigate contention-based occupancy-based covert channels in the cache. *Non Deterministic cache* [59] randomizes cache access delay and de-couples the relation between cache block access and cache access timing. Secure TLBs [29] are also proposed to mitigate covert channels in TLBs. But again, all the possible covert channels need to be mitigated to fully mitigate transient execution attacks. Further, *Cyclone* [48] and *PerSpectron* [85] propose micro-architecture designs to detect cache information leaks across security domains.

Another mitigation is to degrade the quality of the channel or even make the channel unusable for a practical attack. For example, many timing covert channels require the receiver to have a fine-grained clock to observe the channel (the second metric in Section 5.6). Limiting the receiver's observation will reduce the bandwidth or even mitigate the covert channel [94, 101]. Noise can also be added to the channel to reduce the bandwidth (the third metric in Section 5.6).

However, the above mitigations only cover covert channels in memory systems. To mitigate other covert channels, there are the following challenges: (1) identify all possible covert channels in micro-architecture, including future covert channels, and (2) mitigate each of the possible covert channels. Formal methods are required in this process. For example, information flow tracking, such as methods in [28, 140, 141], can be used to analyze the hardware components, where the data of transient execution could flow to. Then, it can be analyzed if each of the components could result in a permanent or transient covert channel.

7.2.1 Mitigations in Secure Architectures. With a clearly defined security domain, isolation can be designed to mitigate not only transient covert channels but also conventional covert channels. For example, to defend cache covert channels, a number of partitioned caches to different security domains are proposed, either statically [14, 26, 50, 62, 67, 73, 120, 133, 140, 141] or dynamically [31, 118]. With partition, shared resource no longer exists between the sender and the receiver, and the receiver cannot observe secret dependent behavior to decode the secret.

The above proposal assumes the hardware is isolated for each security domain. However, there is also a scenario where software outside the security domain may use the same hardware after a context switch. In the *Mi6* processor [14], cache and port partitionings are used to isolate software on different cores. Further, when there is a context switch, a security monitor flushes the architecture and micro-architecture states, which holds the information of in-flight speculation from the previously executing program. To protect the security monitor, speculation is not used in the execution of the security monitor. In OPTIMUS [90], a dynamic partitioning scheme in the granularity of the core is proposed to achieve both security and high performance.

8 OTHER ATTACKS LEVERAGING TRANSIENT EXECUTION

Another category of attacks that can be mistaken with the transient execution attacks is the covert channel attacks leveraging transient execution. Different from the transient execution attacks, where the goal of the attacker is to compromise the confidentiality of the victim's secret, these attacks have the goal to build novel covert channels leveraging the hardware units for transient execution [33, 34, 36, 56], such as the branch prediction unit, STL, and so forth.

Modern computer architectures gain performance benefits from transient execution. A correct prediction results in useful transient execution results and makes the execution faster. When a wrong prediction is made, the results of transient execution will be discarded and sometimes cause a small penalty. Therefore, there is a time difference in the execution due to transient execution, and a timing-based covert channel can be built.

As shown in Table 3, prediction units are shared between different users. The sender can train a prediction unit, and then the receiver can observe different prediction results. Practical covert channel attacks have been demonstrated by leveraging the prediction units, such as *Branchscope* attack, which uses PHT [33, 36, 139]; *Jump over ASLR*, which uses BTB [34, 122]; and *Spoiler* attack, which uses STL [56]. Other than building covert channels across processes and SGX enclaves [36], these attacks also break **Kernel address space layout randomization (KASLR)** [34] and leak the physical address mapping [56].

9 OPEN RESEARCH CHALLENGES

One research direction is to further discover new attacks. Since the detailed designs of commercial processors are not public, there might be unknown attacks. There could be more data that is exploitable in transient execution. In attacks where the victim executes transiently, as shown in Table 2, we assume the victim is executing transiently and the attacker then can access data that the victim could access architecturally. As demonstrated in Spectre V1 attack using SWAPGS instruction [12], the attacker can learn data that the victim could access architecturally even after context switch. The security boundaries that are broken in the attacks depend on the implementation and can be explored in the future. Meanwhile, there could also be new covert channels in the micro-architecture. As shown in Table 5, new combinations of transient execution and covert channel can be demonstrated for different attack scenarios.

Another research direction is to design hardware to defend the transient execution attacks. Transient execution is critical to the performance of processors and is also the root cause of the attacks. A couple of mitigation strategies are proposed and discussed in Section 7. Proposals and optimization to defend the attack with lower performance overhead are desired. The mitigations can come with different threat models to provide a trade-off between performance and security.

The transient execution attacks also affect the formal analysis of software behavior. In the past, the software analysis was based on either software semantics or an ISA level hardware model. To analyze the transient execution attacks rigorously, a formal micro-architecture level or RTL level model is required [20, 46, 82].

10 CONCLUSION

Transient execution attacks can cause critical data leakage across security boundaries in the majority of today's processors. These attacks leverage the micro-architectural states that cannot be reasoned about in the software semantics on the ISA level. This survey analyzed in detail the two main components of the attacks, transient execution and covert channels, and proposed a set of metrics to evaluate the feasibility of the attacks. A taxonomy of the attacks was presented, and the existing attacks were compared using our categorization. In the end, different mitigation schemes at the micro-architecture level were discussed. As this survey demonstrates, much future work is still needed to defend the transient execution attacks efficiently.

ACKNOWLEDGMENTS

We would like to thank anonymous reviewers for their insightful and constructive comments.

REFERENCES

[1] Sam Ainsworth and Timothy M. Jones. 2020. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. In *Annual International Symposium on Computer Architecture*. IEEE, 132–144.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *Symposium on Security and Privacy*. IEEE, 870–887.

[3] AMD. 2018. Software Techniques for Managing Speculation on AMD Processors. Retrieved May 2019 from https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf.

[4] AMD. 2020. AMD Product Security. Retrieved July 2020 from https://www.amd.com/en/corporate/product-security.

[5] Arm. 2020. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. Retrieved July 2020 from https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability.

[6] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. 2010. Acoustic side-channel attacks on printers.. In *USENIX Security Symposium*. 307–322.

[7] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. SpecShield: Shielding speculative data from microarchitectural covert channels. In *International Conference on Parallel Architectures and Compilation Techniques*. 151–164.

[8] Kristin Barber, Li Zhou, Anys Bacha, Yinqian Zhang, and Radu Teodorescu. 2019. Isolating speculative data to prevent transient execution attacks. *Computer Architecture Letters* 18, 2 (2019), 178–181.

[9] Daniel J Bernstein. 2005. Cache-timing attacks on AES.

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

[11] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.

[12] Bitdefender. 2019. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. Retrieved July 2020 from https://www.bitdefender.co.th/wp-content/uploads/gz/Bitdefender-WhitePaper-SWAPGS.pdf.

[13] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 201–215.

[14] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. 2019. Mi6: Secure enclaves in a speculative out-of-order processor. In *International Symposium on Microarchitecture*. ACM, 42–56.

[15] 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 Symposium*. 1967–1984.

[16] 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 Symposium*. 249–266.

[17] Chandler Carruth. 2018. Speculative Load Hardening (a Spectre Variant 1 Mitigation). Retrieved May 2019 from https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html.

[18] Microsoft Security Response Center. 2019. Retpoline: A Software Construct for Preventing Branch-target-injection. Retrieved October 2019 from https://support.google.com/faqs/answer/7625886.

[19] David Champagne and Ruby B. Lee. 2010. Scalable architectural support for trusted software. In *International Symposium on High Performance Computer Architecture*. 1–12.

[20] Kevin Cheang, Cameron Rasmussen, Sanjit Seshia, and Pramod Subramanyan. 2019. A formal approach to secure speculation. In *2019 IEEE 32nd Computer Security Foundations Symposium (CSF'19)*. IEEE, 288–303.

[21] 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 *European Symposium on Security and Privacy*. IEEE, 142–157.

[22] Md Hafizul Islam Chowdhuryy, Hang Liu, and Fan Yao. 2020. BranchSpec: Information leakage attacks exploiting speculative branch instruction executions. In *International Conference on Computer Design*. IEEE, 529–536.

[23] The MITRE Corporation. 2018. CVE-2018-3640. Retrieved July 2020 from https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3640.

[24] The MITRE Corporation. 2019. CVE Details. Retrieved July 2020 from https://www.cvedetails.com.

[25] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained.*IACR Cryptology ePrint Archive* 2016, 086 (2016).

[26] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security Symposium*. 857–874.

[27] Sanjeev Das, Jan Werner, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. 2019. SoK: The challenges, pitfalls, and perils of using hardware performance counters for security. In *Symposium on Security and Privacy*. 20–38.

[28] Shuwen Deng, Doğuhan Gümüşoğlu, Wenjie Xiong, Y. Serhan Gener, Onur Demir, and Jakub Szefer. 2019. SecChisel framework for security verification of secure processor architectures. In *Workshop on Hardware and Architectural Support for Security and Privacy*.

[29] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *International Symposium on Computer Architecture*. 346–259.

[30] 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 Symposium*. 51–67.

[31] Leonid Domnitser, Aamer Jaleel, Jason Loew, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Non-monopolizable caches: Low-complexity mitigation of cache side channel attacks. *Transactions on Architecture and Code Optimization* 8, 4, (2012), Article 35.

[32] Marius Evers, Po-Yung Chang, and Yale N. Patt. 1996. Using hybrid branch predictors to improve branch prediction accuracy in the presence of context switches. In *SIGARCH Computer Architecture News*, Vol. 24. ACM, 3–11.

[33] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: A feasibility study. In *Workshop on Hardware and Architectural Support for Security and Privacy*. ACM.

[34] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *International Symposium on Microarchitecture*. IEEE.

[35] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Understanding and mitigating covert channels through branch predictors. *Transactions on Architecture and Code Optimization* 13, 1 (2016), 10.

[36] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A new side-channel attack on directional branch predictor. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 693–707.

[37] Jacob Fustos, Michael Bechtel, and Heechul Yun. 2020. SpectreRewind: Leaking secrets to past instructions. In *Workshop on Attacks and Solutions in Hardware Security*. 117–126.

[38] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An efficient data-centric defense mechanism against spectre attacks. In *Annual Design Automation Conference*. 1–6.

[39] Daniel Genkin, Itamar Pipman, and Eran Tromer. 2015. Get your hands off my laptop: Physical side-channel key-extraction attacks on PCs. *Journal of Cryptographic Engineering* 5, 2 (2015), 95–112.

[40] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Annual Cryptology Conference*. Springer, 444–461.

[41] Abraham Gonzalez, Ben Korpan, Jerry Zhao, Ed Younis, and Krste Asanović. 2019. Replicating and mitigating spectre attacks on an open source RISC-V microarchitecture. In *Workshop on Computer Architecture Research with RISC-V*.

[42] 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 Symposium*. USENIX, 955–972.

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

[44] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache template attacks: Automating attacks on inclusive last-level caches. In *USENIX Security Symposium*. 897–912.

[45] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *Symposium on Security and Privacy*. IEEE, 38–55.

[46] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. 2020. SPECTECTOR: Principled detection of speculative information flows. In *Symposium on Security and Privacy*. IEEE, 160–178.

[47] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2021. Hardware/software contracts for secure speculation. In *Symposium on Security and Privacy*. IEEE.

[48] 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 *International Symposium on Microarchitecture*. ACM, 57–72.

[49] Zecheng He, Guangyuan Hu, and Ruby Lee. 2020. New models for understanding and reasoning about speculative execution attacks. *arXiv preprint arXiv:2009.07998* (2020).

[50] Zecheng He and Ruby B. Lee. 2017. How secure is your cache against side-channel attacks? In *International Symposium on Microarchitecture*. ACM, 341–353.

[51] John L. Hennessy and David A. Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier.

[52] John L. Henning. 2006. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News* 34, 4 (2006), 1–17.

[53] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical timing side channel attacks against kernel space ASLR. In *Symposium on Security and Privacy*. IEEE, 191–205.

[54] Intel. 2018. Speculative Execution Side Channel Mitigations. Retrieved May 2019 from https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf.

[55] Intel. 2019. Intel Transactional Synchronization Extensions (Intel TSX) Overview. Retrieved May 2019 from https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview.

[56] 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 Symposium*. USENIX, 621–637.

[57] Daniel A. Jiménez and Calvin Lin. 2001. Dynamic branch prediction with Perceptrons. In *International Symposium on High Performance Computer Architecture*. IEEE, 197–206.

[58] Kekai Hu, Ke Sun, and Rodrigo Branco. 2019. A New Memory Type against Speculative Side Channel Attacks. Retrieved May 2019 from https://blogs.technet.microsoft.com/srd/2018/03/15/mitigating-speculative-execution-side-channel-hardware-vulnerabilities/.

[59] Georgios Keramidas, Alexandros Antonopoulos, Dimitrios N. Serpanos, and Stefanos Kaxiras. 2008. Non deterministic caches: A simple and effective defense against side channel attacks. *Design Automation for Embedded Systems* 12, 3 (2008), 221–230.

[60] 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 *Annual Design Automation Conference*. ACM, 1–6.

[61] Sungkeun Kim, Farabi Mahmud, Jiayi Huang, Pritam Majumder, Neophytos Christou, Abdullah Muzahid, Chia-Che Tsai, and Eun Jung Kim. 2020. ReViCe: Reusing victim cache to prevent speculative cache leakage. In *Secure Development*. IEEE, 96–107.

[62] Vladimir Kiriansky, Ilia Lebedev, Saman Amarasinghe, Srinivas Devadas, and Joel Emer. 2018. DAWG: A defense against cache timing attacks in speculative execution processors. In *International Symposium on Microarchitecture*. IEEE, 974–987.

[63] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative buffer overflows: Attacks and defenses. *arXiv preprint arXiv:1807.03757* (2018).

[64] 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 *Symposium on Security and Privacy*. 1–19.

[65] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *Workshop on Offensive Technologies*. USENIX.

[66] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanović, and Dawn Song. 2020. Keystone: An open framework for architecting trusted execution environments. In *European Conference on Computer Systems*.

[67] Ruby B. Lee, Peter Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *32nd International Symposium on Computer Architecture*. 2–13.

[68] Peinan Li, Lutan Zhao, Rui Hou, Lixin Zhang, and Dan Meng. 2019. Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks. In *International Symposium on High Performance Computer Architecture*. IEEE, 264–276.

[69] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. 2000. Architectural support for copy and tamper resistant software. *SIGPLAN Notices* 35, 11 (2000), 168–177.

[70] Mikko H. Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *International Symposium on Microarchitecture*. IEEE, 226–237.

[71] Mikko H. Lipasti, Christopher B. Wilkerson, and John Paul Shen. 1996. Value locality and load value prediction. *SIGPLAN Notices* 31, 9 (1996), 138–147.

[72] 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 *27th USENIX Security Symposium*. 973–990.

[73] 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 *International Symposium on High Performance Computer Architecture*. IEEE, 406–418.

[74] Fangfei Liu and Ruby B. Lee. 2014. Random fill cache architecture. In *International Symposium on Microarchitecture*. IEEE, 203–215.

[75] Fangfei Liu, Hao Wu, Kenneth Mai, and Ruby B. Lee. 2016. Newcache: Secure cache architecture thwarting cache side-channel attacks. *Micro* 36, 5 (2016), 8–16.

[76] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-level cache side-channel attacks are practical. In *Symposium on Security and Privacy*. IEEE, 605–622.

[77] Google Project-Zero mailing list. 2018. Speculative Execution, Variant 4: Speculative Store Bypass. Retrieved May 2020 from https://bugs.chromium.org/p/project-zero/issues/detail?id=1528.

[78] Giorgi Maisuradze and Christian Rossow. 2018. ret2spec: Speculative execution using return stack buffers. In *Conference on Computer and Communications Security*. ACM, 2109–2122.

[79] Andrea Mambretti, Matthias Neugschwandtner, Alessandro Sorniotti, Engin Kirda, William Robertson, and Anil Kurmus. 2019. Speculator: A tool to analyze speculative execution attacks and mitigations. In *Annual Computer Security Applications Conference*. 747–761.

[80] Nikolay Matyunin, Jakub Szefer, Sebastian Biedermann, and Stefan Katzenbeisser. 2016. Covert channels using mobile device's magnetic field sensors. In *Asia and South Pacific Design Automation Conference*. IEEE, 525–532.

[81] Scott McFarling. 1993. *Combining Branch Predictors*. Technical Report TN-36, Digital Western Research Laboratory.

[82] Ross Mcilroy, Jaroslav Sevcik, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. 2019. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv preprint arXiv:1902.05178* (2019).

[83] Pierre Michaud, André Seznec, and Richard Uhlig. 1997. Trading conflict and capacity aliasing in conditional branch predictors. *International Symposium on Computer Architecture* 25, 2 (1997), 292–303.

[84] Marina Minkin, Daniel Moghimi, Moritz Lipp, Michael Schwarz, Jo Van Bulck, Daniel Genkin, Daniel Gruss, Frank Piessens, Berk Sunar, and Yuval Yarom. 2019. Fallout: Reading kernel writes from user space. *arXiv preprint arXiv:1905.12701* (2019).

[85] 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 *International Symposium on Microarchitecture*. IEEE, 1124–1137.

[86] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. 2018. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *Cryptographers' Track at the RSA Conference*. Springer, 21–44.

[87] Donald A. Neamen. 2012. *Semiconductor Physics and Devices: Basic Principles*. McGraw-Hill, New York.

[88] Oleksii Oleksenko, Bohdan Trach, Tobias Reiher, Mark Silberstein, and Christof Fetzer. 2018. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv preprint arXiv:1805.08506* (2018).

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

[90] Hamza Omar, Brandon D'Agostino, and Omer Khan. 2020. OPTIMUS: A security-centric dynamic hardware partitioning scheme for processors that prevent microarchitecture state attacks. *Transactions on Computers* 69, 11 (2020), 1558–1570.

[91] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: The case of AES. In *Cryptographers' Track at the RSA Conference*. Springer.

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

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

[94] Filip Pizlo. 2018. What Spectre and Meltdown Mean For WebKit. Retrieved May 2019 from https://webkit.org/blog/8048/what-spectre-and-meltdown-mean-for-webkit/.

[95] Moinuddin K. Qureshi. 2018. CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping. In *International Symposium on Microarchitecture*. IEEE, 775–787.

[96] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CROSSTALK: Speculative data leaks across cores are real. In *Symposium on Security and Privacy*. IEEE.

[97] Gururaj Saileshwar and Moinuddin K. Qureshi. 2019. CleanupSpec: An undo approach to safe speculation. In *International Symposium on Microarchitecture*. ACM, 73–86.

[98] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient invisible speculative execution through selective delay and value prediction. In *International Symposium on Computer Architecture*. ACM, 723–735.

[99] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-leak forwarding: Leaking data on meltdown-resistant CPUs. *arXiv preprint arXiv:1905.05725* (2019).

[100] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. Context: A generic approach for mitigating spectre. In *Network and Distributed System Security Symposium*.

[101] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript zero: Real JavaScript and zero side-channel attacks. In *Network and Distributed System Security Symposium*.

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

[103] Michael Schwarz, Clémentine Maurice, Daniel Gruss, and Stefan Mangard. 2017. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *International Conference on Financial Cryptography and Data Security*. Springer, 247–267.

[104] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. 2019. Netspectre: Read arbitrary memory over network. In *European Symposium on Research in Computer Security*. Springer, 279–299.

[105] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. 2010. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM* 53, 7 (2010), 89–97.

[106] Eric Sprangle, Robert S. Chappell, Mitch Alsup, and Yale N. Patt. 1997. The agree predictor: A mechanism for reducing negative branch history interference. *International Symposium on Computer Architecture* 25, 2 (1997), 284–291.

[107] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU register state using microarchitectural side-channels. *arXiv preprint arXiv:1806.07480* (2018).

[108] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2014. AEGIS: Architecture for tamper-evident and tamper-resistant processing. In *International Conference on Supercomputing*. ACM, 357–368.

[109] Jakub Szefer. 2019. Survey of microarchitectural side and covert channels, attacks, and defenses. *Journal of Hardware and Systems Security* 3, 3 (2019), 219–234.

[110] Mohammadkazem Taram, Ashish Venkat, and Dean Tullsen. 2019. Context-sensitive fencing: Securing speculative execution via microcode customization. In *International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 395–410.

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

[112] Paul Turner. 2018. Mitigating Speculative Execution Side Channel Hardware Vulnerabilities. Retrieved October 2019 from https://github.com/intelstormteam/Papers.

[113] 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 Symposium*. 991–1008.

[114] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking transient execution through microarchitectural load value injection. In *Symposium on Security and Privacy*. 1399–1417.

[115] 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 *Symposium on Security and Privacy*. IEEE, 88–105.

[116] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *Transactions on Software Engineering and Methodology* 29, 3, Article 14 (2020), 31.

[117] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead defense against spectre attacks via program analysis. *Transactions on Software Engineering* (2019).

[118] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C. Myers, and G. Edward Suh. 2016. SecDCP: Secure dynamic cache partitioning for efficient timing channel protection. In *Design Automation Conference*. IEEE.

[119] Zhenghong Wang and Ruby B. Lee. 2006. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference*. IEEE, 473–482.

[120] Zhenghong Wang and Ruby B. Lee. 2007. New cache designs for thwarting software cache-based side channel attacks. In *ACM SIGARCH Computer Architecture News*, Vol. 35. ACM, 494–505.

[121] Zhenghong Wang and Ruby B. Lee. 2008. A novel cache architecture with enhanced performance and security. In *International Symposium on Microarchitecture*. IEEE, 83–93.

[122] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing speculative execution attacks at their source. In *International Symposium on Microarchitecture*. ACM, 572–586.

[123] 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.

[124] 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 Symposium*. 675–692.

[125] You Wu and Xuehai Qian. 2020. ReversiSpec: Reversible coherence protocol for defending transient attacks. *arXiv preprint arXiv:2006.16535* (2020).

[126] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the hyper-space: High-bandwidth and reliable covert channel attacks inside the cloud. *Transactions on Networking* 23, 2 (2014), 603–615.

[127] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A framework for investigating and measuring speculative execution vulnerabilities. In *Network and Distributed System Security Symposium*.

[128] Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. 2019. Spying on temperature using DRAM. In *Design Automation and Test in Europe*. 13–18.

[129] Wenjie Xiong and Jakub Szefer. 2020. Leaking information through cache LRU states. In *International Symposium on High Performance Computer Architecture*. IEEE, 139–152.

[130] 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 *Workshop on Cloud Computing Security*. ACM, 29–40.

[131] Mengjia Yan. 2019. *Cache-based side channels: Modern attacks and defenses*. Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[132] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making speculative execution invisible in the cache hierarchy. In *International Symposium on Microarchitecture*. IEEE, 428–441.

[133] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. 2017. Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel attacks. In *International Symposium on Computer Architecture*. ACM, 347–360.

[134] 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 *Symposium on Security and Privacy*. IEEE, 888–904.

[135] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are coherence protocol states vulnerable to information leakage? In *International Symposium on High Performance Computer Architecture*. IEEE, 168–179.

[136] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *23rd USENIX Security Symposium*. 719–732.

[137] Yuval Yarom, Daniel Genkin, and Nadia Heninger. 2017. CacheBleed: A timing attack on OpenSSL constant-time RSA. *Journal of Cryptographic Engineering* 7, 2 (2017), 99–112.

[138] Jiyong Yu, Namrata Mantri, Josep Torrellas, Adam Morrison, and Christopher W. Fletcher. 2020. Speculative data-oblivious execution: Mobilizing safe prediction for safe and efficient speculative execution. In *International Symposium on Computer Architecture*. IEEE, 707–720.

[139] 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 *International Symposium on Microarchitecture*. ACM, 954–968.

[140] Danfeng Zhang, Aslan Askarov, and Andrew C. Myers. 2012. Language-based control and mitigation of timing channels. *SIGPLAN Notices* 47, 6 (2012), 99–110.

[141] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. 2015. A hardware design language for timing-sensitive information-flow security. In *ACM SIGARCH Computer Architecture News*, Vol. 43. ACM, 503–516.

[142] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2014. Cross-tenant side-channel attacks in PaaS clouds. In *Conference on Computer and Communications Security*. ACM, 990–1003.

[143] Lutan Zhao, Peinan Li, Rui Hou, Jiazhen Li, Michael C. Huang, Lixin Zhang, Xuehai Qian, and Dan Meng. 2020. A lightweight isolation mechanism for secure branch predictors. *arXiv preprint arXiv:2005.08183* (2020).

[144] Zirui Neil Zhao, Houxiang Ji, Mengjia Yan, Jiyong Yu, Christopher W. Fletcher, Adam Morrison, Darko Marinov, and Josep Torrellas. 2020. Speculation invariance (invarspec): Faster safe execution through program analysis. In *International Symposium on Microarchitecture*. IEEE, 1138–1152.

Received December 2019; revised August 2020; accepted December 2020