Survey of Transient Execution Attacks

Wenjie Xiong
Dept. of Electrical Engineering
Yale University
wenjie.xiong@yale.edu

Jakub Szefer
Dept. of Electrical Engineering
Yale University
jakub.szefer@yale.edu

ABSTRACT

Transient execution attacks, also called speculative execution attacks, have drawn much interest in the last few years as they can cause critical data leakage. Since the first disclosure of transient execution attacks in January 2018, a number of new attack types or variants have been demonstrated in 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 caused by fundamental features of modern processors that boost performance and efficiency, while covert channels are unintended channels that can be abused for information leakage, resulting from sharing of the micro-architecture 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 paper summarizes the phases of the attacks and the security boundaries that are broken by the attacks. This paper further analyzes possible causes of transient execution and different types of covert channels. This paper in addition presents metrics for comparing different aspects of the transient execution attacks (security boundaries that are broken, required control of the victim's execution, etc.) and uses them to evaluate the feasibility of the different attacks – both the existing attacks, and potential new attacks suggested by our classification used in the paper. The paper finishes by discussing the different mitigations at the micro-architecture level that have so far been proposed.

KEYWORDS

Transient Execution, Speculative Execution, Timing Channels, Covert Channels, Secure Processor Architectures

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 [50]. Some of the optimizations require aggressive speculation of the executed instructions. For example, while waiting for a conditional branch to be resolved, branch prediction will predict whether the branch will be taken or not, and the processor begins to execute code 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. The execution of the instructions down the incorrect speculated path is called the *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 it is not done so today, leading to a number of recently publicized transient execution attacks [20, 61, 69, 80, 97, 108, 110, 118] that leak data across different security boundaries in computing systems.

Today's processor designs aim to ensure the execution of a program results in architectural states as if each instruction is executed in the program order. At the Instruction Set Architecture (ISA) level, today's processors behave correctly. Unfortunately, the complicated underlying micro-architectural states, due to different optimizations, are modified during the transient execution, and the various transient execution attacks have shown that data can be leaked from the micro-architectural states. For example, timing channels can lead to information leaks that can reveal some of the micro-architectural states which are not visible at the ISA level [49, 73, 104, 137]. Especially, the micro-architectural states of a processor are today not captured by the ISA specification, and there are micro-architectural vulnerabilities that cannot be found or analyzed by only examining the processor's ISA.

Besides focusing on pure performance optimization, many processors are 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 [82], 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 [49, 73, 104, 137] – 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 or sensitive data is available during 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 data is accessed during transient execution and encoded into a covert channel, the secret data can be later extracted by the attacker from the covert channel.

A number of transient execution attack variants has been demonstrated, e.g., Spectre [2, 14, 25, 61, 62, 74, 99, 106], Meltdown [1, 20,

[1]There are also attacks using the timing difference in transient execution, e.g., [34–36, 36, 54]. 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 paper, but are listed in Section ??.

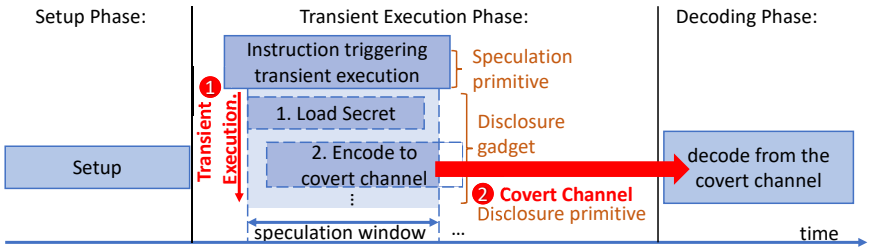

Figure 1: Phases of transient execution attacks.

60, 69], Foreshadow [108, 118], LazyFP [102], Micro-architectural Data Sampling (MDS) [80, 97, 110], Load Value Injection (LVI) [109]. These attacks have been shown to allow data leaks across different security boundaries, e.g., system privilege level, SGX enclave, sandbox, etc. 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 [3]. These attacks also affect other vendors, such as AMD or Arm, for example.

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 [12, 37, 58, 59, 95, 127] – this survey summarizes the attacks and the hardware defenses, while software-based defenses are summarized in existing work [20].

1.1 Outline and Contributions

This paper provides a survey of existing *transient execution attacks* from Jan. 2018 to July 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 summarize the security boundaries that are broken by the attacks.
- We provide a taxonomy of the existing transient execution attacks by analyzing the causes of transient execution that they leveraged, and we propose metrics to compare the feasibility of the attacks.
- We summarize and categorize the existing and potential timing-based covert channels in micro-architectures 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 TRANSIENT EXECUTION ATTACK SCENARIOS

We define transient execution attacks as 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 state change (and thus the secret). In this section, we summaries the attack scenarios, e.g., the attacker's goal, the location of the attacker, etc.

2.1 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 cross the security boundaries to obtain information related to the victim's protected data. 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., [14] demonstrates how an attacker process learns the private key when a victim OpenSSH server process is running in.
(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., [69] demonstrates an attack that allows an unprivileged application to dump kernel memory.
(3) **Virtual machine attacking another virtual machine:** The attacker and the victim resides in two different guest virtual machines, e.g., [14] shows it is possible for an attacker VM to learn the private key of 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., [61] demonstrates an attack against KVM that leaks hypervisor's memory when the attacker has full control of the OS inside a VM.

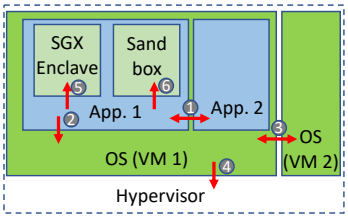

Figure 2: Security boundaries in computer systems that are broken by transient execution attacks.

(5) **Attacking the victim running inside an enclave:** The victim runs inside a security domain protected by some hardware scheme, e.g., SGX enclaves [26], XOM [66], Aegis [103], Bastion [23], Sanctum [27] or Keystone [63], and the attacker code runs outside of it, e.g., [25] 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, e.g., sandboxes in JavaScript, and the attacker code runs outside of it, as shown in [61].

All of the security boundaries listed above are broken by one or more of the existing transient execution attacks. The attacks have been shown to be able to retrieve coherent data, as well as non-coherent data. Details will be discussed in Section 3.4, especially in Table 2.

2.1.1 Attacks Targeting Coherent and Non-Coherent Data. We categorize all the data in the processor state into **coherent data** and **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 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 [2, 102] focus on attacking buffers to retrieve such non-coherent data, which in turn breaks the security boundaries.

2.2 Phases of the Attack

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

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 is performing indirect jumps to a specific address to "train" the branch predictor. The setup can be done by the attacker running some code or the attacker causing the victim to run in a predictable manner so that the micro-architectural state is set up as the attacker expects.

Transient Execution Phase: The transient execution is actually triggered in this phase, and the desired disclosure gadget executes due to the prior training in the setup phase. The piece of code that accesses and transmits secret into the covert channel is called *disclosure gadget*, following the terminology in [107]. 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, so secret data can be later decoded from the covert channel. This phase can be either executed by the victim or by the attacker.

Decoding Phase: The attacker is able to recover the data via the covert channel by running the attacker's code or by triggering 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.

2.3 Transient Execution by 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 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 to encode the secret into the covert channel.

2.3.1 Victim is Executing Transiently. If the victim is the one who executes transiently, as shown in Figure 3 (a–d), the victim is triggered to execute a disclosure gadget that encode some secret into the covert channel during transient execution, and the attacker obtains the secret by decoding the data from the covert channel. In this scenario, the attacker is assumed to be able to control or trigger the execution of the disclosure gadget in the victim's codebase. The attacker can do this by calling some victim functions with certain parameters. For example, in SGXpectre [25], the attacker can launch 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 preparing for the transient execution (i.e., 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 prediction unit to trigger the disclosure gadget in the victim code to execute transiently, as shown in Figure 3 (c,d). The second option is that the attacker triggers a setup gadget in the victim codebase to set up the transient execution, as shown in Figure 3 (a,b). For the first option, the attacker is required to share the prediction unit with the victim and to prepare some code to set up the hardware to lure the victim into desired transient execution. For the second option, the attacker is required to understand the

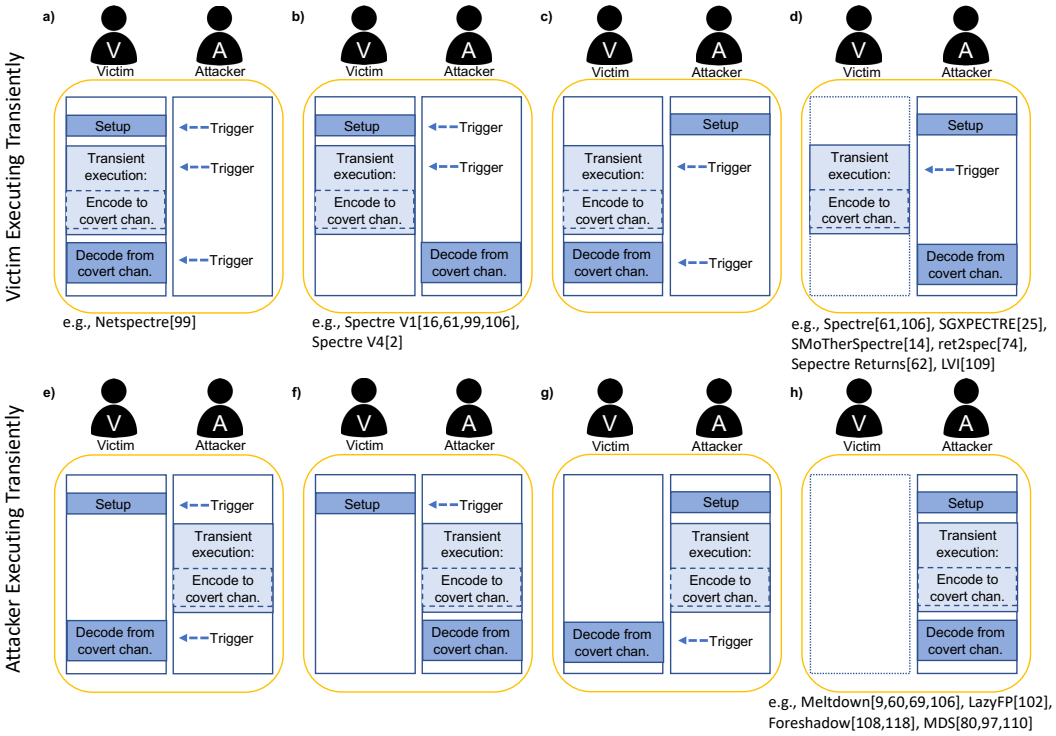

Figure 3: Possible scenarios of transient execution attacks: a-d) the attacker triggers part of the victim code to execute transiently to leak secret through the covert channel, or e-h) the attacker executes transiently to access data that she does not have permission to access and encodes it into the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

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 code, as shown in Figure 3 (b,d), or by the victim code, as shown in Figure 3 (a,c). For the second case, the attacker may directly query a decoding gadget in the victim code and leverage the results of the decoding gadget the infer information through the covert channel, or the attacker may trigger the execution of the decoding gadget and measure the time or other side effect of the execution.

2.3.2 Attacker is Executing Transiently. As shown in Figure 3 (e–h), the attacker can directly obtain the secret in transient execution. The attacker will then encode the data into a covert channel and decode it to obtain the secret in the architectural state, such as in her memory. The attacker can also launch different software threads for the setup or the decoding phases. The attacker's code shown in Figure 3 (e–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. 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 [69], the attacker code first loads protected data by its virtual address to 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 Micro-architectural Data Sampling (MDS) attacks [80, 97, 110], the attacker needs to synchronize with the victim execution to extract useful information from the non-coherent data of the victim in the buffers.

In micro-architectural implementations, transient execution allows the attacker to access more data than it is allowed in the architecture (ISA) level. Thus, this type of attack is implementation-dependent and does not work on all the CPUs, e.g., Meltdown [69], Foreshadow [108, 118], MDS [80, 97, 110], are reported to work 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 Figure 3 (e–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.

2.3.3 Feasibility of the Attack Scenarios. The required number of gadgets in the victim codebase to be triggered and required sharing in different transient execution scenarios is 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.

Table 1: Required Control of Victim 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 codebase 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 show 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 colocation between the each party's threads is possible, and do not list that here.

In most of the existing attacks, the attacker completes setup and decoding steps, as shown in Figure 3 (d,h), because they use less gadgets in the victim codebase and are more practical for the attacker. Attack scenarios (a,b) in Figure 3 are also demonstrated that have less requirement of shared hardware. In Spectre V1, since the victim disclosure gadget can be reused as the setup gadget for training the predictor, triggering victim to run the setup phase does not require additional effort for the attacker, and thus, Figure 3 (b) is also practical. 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 [99].

Scenarios (c) and (e–g) in Figure 3 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 mitigations.

3 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 the squash, not all the micro-architectural side effects are cleaned up properly, causing the possible transient execution attacks. Hence, all causes of pipeline squash are also causes of transient execution and need to be understood to know what cause transient execution attacks to occur. In this section, we first discuss all possible causes of transient execution, then we propose a set of the metrics to evaluate feasibility of the transient execution attacks.

3.1 Causes of Transient Execution

The following is an exhaustive list of possible causes of transient execution (i.e., causes of pipeline squashing).

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

(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 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 using pattern history table (PHT), and what is the target address (i.e., branch or indirect jump target) by using 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 3.6.1.

(2) **Address Speculation:** Address speculation is a prediction on 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., store-to-load (STL) forwarding in the load-store queue, line-fill buffer (LFB) in the cache. The implementation details of STL and LFB in Intel processors will be discussed in Section 3.6.2.

(3) **Value Prediction:** To further improve the performance, while the pipeline is waiting for the data to be loaded from 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 [67, 68].

Table 2: Data Leaked by the Transient Execution Attacks.

	Causes of Transient Execution			Example Attacks	Coh. Data** hypervisor	across VM	kernel data	across user app.	SGX	sandbox	Non-coh. Data**
Victim Executes Transiently	Prediction	Ctrl Flow	PHT	Spectre V1 [16, 61, 99, 106]	⊠	⊠	⊠	⊠	⊠	⊠	□
			BTB	Spectre V2 [14, 25, 61]	⊠	⊠	⊠	⊠	⊠	⊠	□
			RSB	Spectre V5 [62, 74]	⊠	⊠	⊠	⊠	⊠	⊠	□
		Addr.	STL	Spectre V4, LVI [2, 109]	⊠	⊠	⊠	⊠	⊠	⊠	⊠
			LFB	LVI [109]	⊠	⊠	⊠	⊠	⊠	⊠	⊠
		Value	no commercial implementation								
	Exception		*	LVI [109]	⊠	⊠	⊠	⊠	⊠	⊠	⊠
	Interrupts			no known attack							
	Load-to-load reordering			no known attack							
Attacker Executes Transiently	Prediction	Ctrl Flow	*	no known attack							
		Addr.	STL	Fallout [80]	□	□	□	□	□	□	⊠
			LFB	RIDL, ZombieLoad [97, 110]	□	□	□	□	□	□	⊠
		Value	no commercial implementation								
	Exception		PF-US	Meltdown (V3) [69, 106]	□	□	⊠	□	□	□	□
			PF-P	Foreshadow (L1TF) [108, 118]	⊠	⊠	⊠	⊠	⊠	□	□
			PF-RW	V1.2 [60]	□	□	□	□	□	⊠	□
			NM	LazyFP [102]	□	□	□	□	□	□	⊠
			GP	V3a [1]	□	□	⊠	□	□	□	□
	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.

Exceptions: The second possible cause for transient execution to occur are 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), etc. A list of all the exception types or permission bit violations is summarized in [20]. In addition, Xiao et al. developed a software framework to automatically explore the vulnerabilities on a variety of Intel and AMD processors [122].

Sometimes the exceptions are suppressed due to another fault, e.g., nested exceptions. For example, when using transactional memory (Intel TSX [4]), 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 [97, 110]. 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.

Interrupts: The third possible cause for 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 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.

Load-to-Load Reordering (Multi-Core): The fourth possible cause for transient execution is load-to-load reordering. Current x86 architectures use the total store order (TSO) memory model [100]. 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.

3.2 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. Misprediction is leveraged in Spectre-type attacks, e.g., [61]. Address speculation is leveraged in MDS attacks [80, 97, 110] and LVI [109].

Exceptions of loads or stores are leveraged in Meltdown attacks [69], Foreshadow attacks [108, 118], and LVI [109], etc. 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, are legal to be resumed later on, and no extra data is accessible to the attacker during the transient execution.

The sample codes of different variants are shown in Figure 4. The victim code should allow a potential mis-prediction or exception to happen. In Spectre V1 [61], to leverage PHT, a conditional branch should exist in the victim code followed by the gadget. Similarly, in Spectre V2 [61] and V5 [62, 74], the victim code should have an indirect jump (or a return from a function) that uses BTB (or RSB) for prediction of the execution path. In Spectre V4 [2], to use STL, the victim code should have a store following a load having potential address speculation. In LVI [109], a load that triggers a page fault (accessing `trusted_ptr`) will forward non-coherent data in the store buffer which is 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 [69], the attacker code should make an illegal load to cause an exception. In MDS attack [80, 97, 110], a faulty load (`value=*(new_page)`) will forward non-coherent data in the buffer.

3.3 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:

- **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 3.4.
- **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 3.5.
- **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 3.6.
- **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 3.7.

3.4 Security Boundaries that are Broken

As discussed in Section 2.1, 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 [61]. 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, SWAPGS instruction is a privileged instruction that 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 [16]. When the victim is executing transiently, the attacker can also learn the non-coherent data (for example, stale data) and also data that depends on non-coherent data (e.g., data in an address that is depended on non-coherent data). For example, in Spectre V4 [2], 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 [109], 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 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 [69], 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) [118], 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., Line Fill Buffers (LFB) or store buffer (STB)) can sometimes be accessed by the attacker in transient execution [80, 97, 110]. In addition, in CROSSTALK [91], a hardware buffer called staging buffer is discovered. The staging buffer is for some type of off-core reads, e.g., `RDRAND` instruction that requesting DRNG (Digital Random Number Generator), `CPUID` instruction that read from MachineSpecific Registers (MSRs). The staging buffer is shared across cores, and thus, the CROSSTALK paper demonstrated a cross-core attack where the victim fetch some data from RNG, and the attacker then learn the random number in the stage buffer during transient execution.

3.5 Required Control of the Victim's Execution

For the attacks leveraging mis-prediction, (mis-)training is a essential setup step 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 train the branch predictor (i.e., PHT) to execute the gadget 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 is required that 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

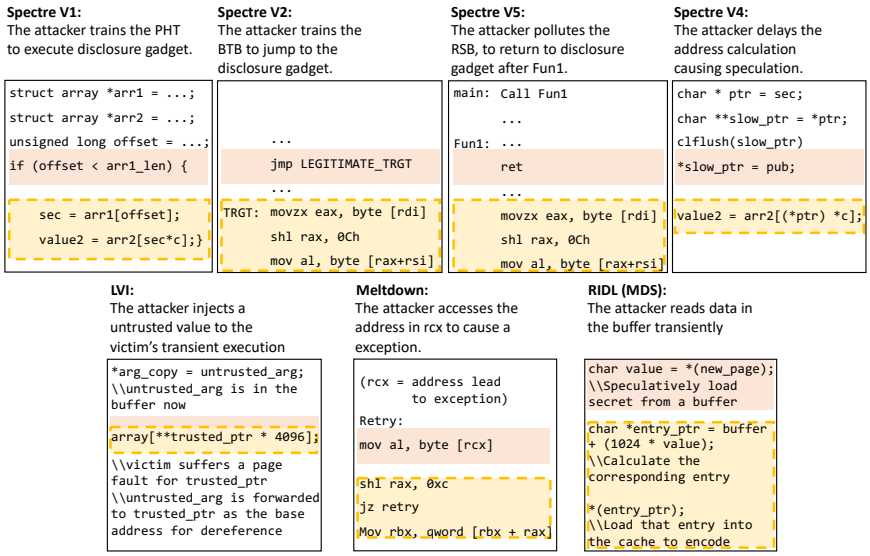

Figure 4: Example code of transient execution attacks. Code highlighted in orange triggers transient execution. Code highlighted in yellow with dashed frame is the disclosure gadget.

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.

3.6 Required Sharing during Transient Execution

As shown in Table 1, in some scenarios, the setup code and the disclosure gadget are run by different parties, e.g., Figure 3 (c-f), or in attacker's different software threads, e.g., Figure 3 (g-h). These cases require that the setup code shares the same prediction unit (entry) with the disclosure gadget. 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 as follows:

- **Same thread:** The attacker and the victim (if both of them 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 thread:** The attacker and the victim (if both of them executing) or the attacker's threads (if only the attacker is executing) are running on different logical cores (hardware threads) through simultaneous multithreading (SMT) on the same physical core.
- **Same chip, different core:** The attacker and the victim (if both of them 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 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, 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-prediction. In particular, the scenario depends on the implementation, and thus, we discuss each of the prediction units in Intel Processors in detail.

3.6.1 Control Flow Prediction: To predict the branch direction, modern branch predictors use a hybrid mechanism [33, 55, 77, 79, 101]. One major component of the branch predictor is the pattern history table (PHT). Typically, a PHT entry is indexed based on

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, 60]	f(virtual addr)	f(virtual addr)	–	–
	BTB [35, 61]	f(virtual addr)	f(virtual addr)[a]	–	–
	RSB [74]	not by address[b]	–	–	–
Addr.	STL [54, 80]	f(physical addr) [c]	–	–	–
	LFB [97, 110]	not by address	not by address	–	–
	Other[d]				
Value	no commercial impl.				

"–" 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 [35, 61].

[b] Most OSes overwrite RSBs on context switches.

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

[d] In [97], it is indicated that there could be other structures which forward data speculatively.

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 branch target buffer (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, return stack buffer (RSB) has been introduced in commercial processors. The RSB stores N most recent return addresses.

In Intel processors, the PHT and BTB[2] 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 [74]. 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 and the victim should use the same entry 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-prediction.

[2]In [35], 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 [61]. Thus, we think BTB is shared across hyper-threads in some of the processors.

3.6.2 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 store-to-load (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 [2, 94]. 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 [80]. In the end, if there is mis-prediction, the load will be marked as faulty, flushed, and reloaded again.

Another use of address speculation is in conjunction with the line-fill buffer (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 [97, 110]. Address speculation may also be used in other hardware structures in Intel processors, as indicated in [97].

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 depends on some uncached data, as in Spectre V4 [2]. 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 [110]. 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 [97]. 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.

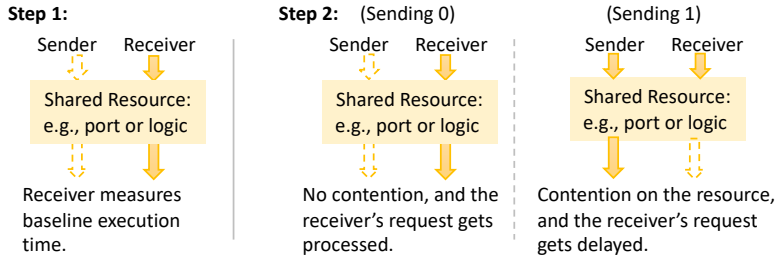

Figure 5: Steps for the sender and the receiver to transfer information through volatile covert channels. The yellow box shows the shared resource. The solid (dashed) arrow shows the shared resource is (is not) requested or used by the corresponding party.

3.6.3 Value Prediction: There is no commercial processor that implement 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.

3.7 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 predictions, 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 [75], a tool called *Speculator* is proposed to reverse engineer the micro-architecture using hardware performance counters. The results of the *Speculator* show 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 the obtaining of the result of the branch condition or the addresses by leveraging uncached loads from main memory, chains of dependent instructions, etc.

4 COVERT CHANNELS

Transient execution enables the attacker to access the secret data transiently, and a covert channel[3] [104] is required for the attacker to eventually obtain the secret data in architectural states. There is a distinction between *conventional channels* where the encoding happens in software execution path, and *transient execution channels* where the encoding phase is executed transiently. Here, we focus on covert channels that can be used in transient attacks – these can also be used as conventional covert channels.

[3]The channel is considered a covert channel, not a side channel [61], because the attacker has control over the disclosure gadget, which encodes the secret.

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 state and the receiver will observe the change to extract information, e.g., by observe the execution time.

4.1 Assumptions about Covert Channels

This survey focuses on covert channels that do not require physical presence and which only require 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 [76], acoustic signals [10, 40], etc. There are certain physical channels that can be accessed from software and not require physical presence, such as temperature [123]. However, thermal conduction is slow and the bandwidth is limited.

Any sharing of hardware resources between users could lead to a covert channel between a sender and a receiver [114]. The receiver can observe the status of the hardware with some metadata from the covert channel, such as the execution time, values of hardware performance counters (HPC), system behavior, etc.

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 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), etc. To observe the hardware states via timing, a timer is needed. In x86, *rdtscp* instruction can be used to read a high-resolution time stamp 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 [98].

The receiver can also gain information from hardware performance counters (HPCs). HPCs have information about branch prediction, cache, TLB, etc, and have been used in covert channel attacks [36]. However, HPCs must be configured in kernel mode [28], 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 [31], for example, TSX can be exploited to allow an attacker to receive an abort (call-back) if the victim process accessed a critical address.

In other cases, several covert channels are used in series. Here, for transient execution attacks, we only consider channels where the receiver can decode data architecturally. For example, in the Fetch+Bounce covert channel [94], first, the secret is encoded into

Table 4: Known Covert Channels in Micro-architecture.

	Covert Channel Type	Level of Sharing: same thread	same core, different thread	same chip, different core	same motherboard	Bandwidth	Required Time Resolution of the Receiver (CPU cycles)
Volatile Covert Channels	Execution Ports [6, 14, 114]	⊠	⊠	□	□	not given	50 vs. 80
	FP division unit [38]	⊠	⊠	□	□	~70kB/s	314 vs. 342
	L1 Cache Ports [81, 132]	⊠	⊠	□	□	not given	36 vs. 48
	Memory Bus [121]	⊠	⊠	⊠	⊠	~700 B/s	2500 vs. 8000
Persistent Covert Channels	AVX2 unit [99]	⊠	⊠	□	□	>0.02B/s	200 vs. 550
	PHT [36]	⊠	⊠	□	□	not given	65 vs. 90
	BTB [35, 117]	⊠	⊠	□	□	not given	56 vs. 65
	STL [54]	⊠	□	□	□	not given	30 vs. 300
	TLB [42, 52, 94]	⊠	⊠	□	□	~5kB/s per set	105 vs. 130[a]
	L1, L2 (tag, LRU) [59, 124, 125]	⊠	⊠	□	□	~1MB/s per cache entry	5 vs. 15[b]
	LLC (tag, LRU) [19, 73]	□	□	⊠	□	~0.7MB/s per set	500 vs. 800
	Cache Coherence [106, 130]	□	□	⊠	⊠	~1MB/s per cache entry	100 vs. 250[c]
	Cache Directory [129]	□	□	⊠	□	~0.2MB/s per slice	40 vs. 400
	DRAM row buffer [88]	□	□	⊠	⊠	~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 use L1 cache. [c] Depending on the setup, the required time resolution varies. The biggest one is shown.

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 architecturally.

4.2 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, no states are changed, e.g., sharing a port or some logic concurrently. The sender and the receiver have contention when communicating 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. The persistent covert channels will be discussed in the next subsection.

4.3 Volatile Covert Channels

In a *volatile covert channel*, there is contention for hardware 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 part of code in the same software thread that their instructions are scheduled to execute concurrently due to OoO [38]. As shown in Figure 5, 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 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 [6, 14]. There is also a covert channel leveraging the contention in the floating point division unit [38]. 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 limit number of) requests at a time. CacheBleed [132] 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 [81] 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 SMT setting.

Memory bus serves memory requests to all the cores using the main memory. In [121], 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.

4.4 Persistent Covert Channels

In a *persistent channel*, the sender and the receiver share the same micro-architectural states, e.g., registers, caches, etc. 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 different parties in the system, we further divide the persistent channels into occupancy-based and encode-based, as shown in Figure 6.

4.4.1 Occupancy-based Persistent Covert Channels. To leverage occupancy-based covert channel, the user needs to occupy the states (e.g., registers, cache, or some entries) or data to affect the execution.

- *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, 86, 87, 126, 129]. The receiver first occupies a cache set (i.e., primes). Then, the sender may use the state for her data or not, depending on the message to be sent. And in the end, the receiver reads (i.e., probes) 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 examples of the eviction-based channel are cache Evict+Time attack [13, 86], the covert channel in DRAM row buffer [88].

  Another possible contention is that the sender needs to use the same piece of data (e.g., need 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, 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 [106, 130]. 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 [129].
- *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, 131] 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 observer faster timing compared to the case when the sender does not load the shared data. There are other reuse-based attacks, such as Cache Collision attack [17] and the cache Flush+Flush attack [43].

Prediction units can also be leveraged for such covert channels due to a longer latency for mis-prediction. For example, PHT [34, 36, 134], BTB [35, 117], and STL [54] 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.

4.4.2 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 [99]. 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 be power-on the unit for at least 1 ms. In step 3, the receiver can measure whether the AVX2 unit is power-on by measuring the time of using AVXs unit. In 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 [19, 59, 124].

4.5 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 exists 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 4.3), no state is changed, e.g., the channel leveraging port contention [6]. 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 4.4) leverage state change in micro-architecture, the retention time depends on how long the state will stay, for example, AVX2 unit will be powered off after about 1ms. If the receiver does not measure the state in time, she will obtain no information. For other states, such as register, cache, etc., the retention time depends

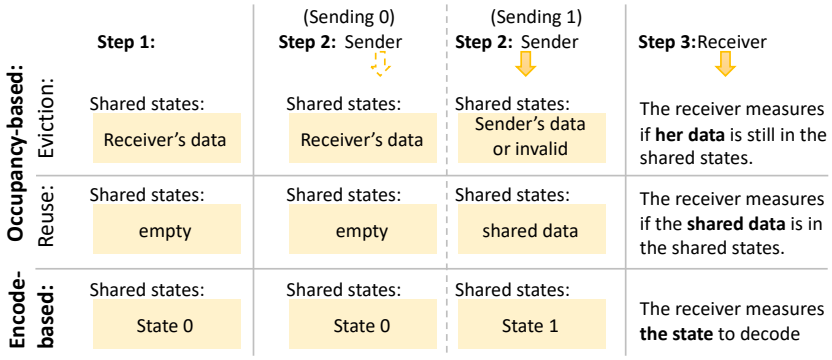

Figure 6: Steps for the sender and the receiver to transfer information through different types of persistent covert channels.

on the usage of the unit and when the unit will be used by another user.

4.6 Comparison of Covert Channels

Table 4 lists different covert channels in micro architecture. The existence of covert channel depends on whether the unit is shared in that 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.

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 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 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 (~1MBits/s), which allows the attacker to launch efficient attacks. Covert channels in AVX and TLB are slower but enough for practical attacks.

4.7 Disclosure Gadget

The covert channel is used in the disclosure gadget to transfer the secret to be accessible to the attacker architecturally. Disclosure gadget usually contains two steps: 1. load the secret into a register; 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.

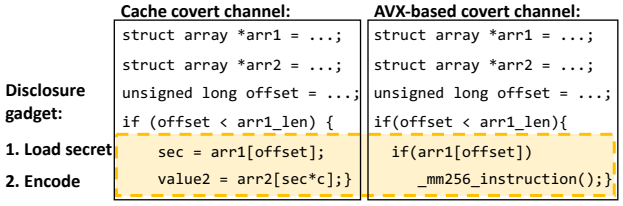

Figure 7: Example disclosure gadgets for different covert channels.

5 EXISTING TRANSIENT EXECUTION ATTACKS

The transient execution attacks contain two parts: triggering transient execution to obtain data that is otherwise not accessible (discussed in Section 3) and transferring the data via a covert channel (discussed in Section 4). 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 prediction. 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 ISA level, but can be reconstructed through cover channels which observe the changes in micro-architecture). To formally model and detect the behavior, a new micro-architectural model, including the transient behavior, should be used [24, 46, 47, 78].

5.1 Existing Transient Execution Attacks Types

To launch an attack, the attacker needs a way to cause transient execution of the victim or herself and a covert channel. Table 5 shows the attacks that are demonstrated in the publications. For demonstrating different speculation primitives, researchers usually use the covert channel in caches (row L1, L2 in Table 5). This is because the 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.

Table 5: Transient Execution Attacks Types.

	Cause of Transient Execution					
Covert Channel	PHT	BTB	RSB	STL	LFB	Exception
Execution Ports	□	[14]	□	□	□	□
L1 Cache Ports	□	□	□	□	□	□
Memory Bus	□	□	□	□	□	□
AVX2 unit	[99]	□	□	□	□	□
FP div unit	[38]	□	□	□	□	[38]
TLB	□	□	□	□	□	□
L1, L2 (tag, LRU)	[61]	[25, 61]	[62, 74]	[2, 80]	[97, 110]	[1, 60, 69, 102, 108, 109, 118]
LLC (tag, LRU)	□	□	□	□	□	□
Cache Coherence	[106]	□	□	□	□	[106]
Cache Directory	□	□	□	□	□	□
DRAM row buffer	□	□	□	□	□	□
Other Channel	□	□	□	□	□	□

□ shows attacks that are possible but not demonstrated yet.

5.2 Feasibility of Existing Attacks

5.2.1 Feasibility of the Transient Execution. As discussed in Section 2.3.3 and Section 3.5, Spectre attacks require the attacker to mis-train the prediction unit in the setup phase to let the victim execute gadget speculatively. 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, the prediction unit is shared only within a physical core, for some unit, not even share between each hyper-thread. In practice, it is not trivial to co-locate on the same core.

5.2.2 Feasibility of the Covert Channel. As shown in Table 1 and Section 4.6, in some scenarios, a covert channel across processes is required, and thus, the sharing of hardware is needed, which requires the co-location of threads. Furthermore, for a certain attack implementation, only one disclosure primitive is used, and the attack can be mitigated by blocking the covert channel.

5.3 Attacks on Different Commercial Platforms

Most of the existing studies focus on Intel processors, Table 6 lists the known attacks on processors by different venders, such as AMD [8, 20], Arm [9, 20], RISC-V [41]. As shown in the table, Spectre-type attacks using branch prediction are found on all the platforms, this is because branch speculation is fundamental in modern processors. Other transient execution depends on the micro-architecture implementation of speculation units, and show different results on different platforms.

6 MITIGATIONS OF SPECTRE-TYPE ATTACKS IN MICRO-ARCHITECTURE DESIGN

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, attacks that leveraging control flow prediction are more fundamental and affect all modern computer architectures. Attacks that leveraging address speculation and exceptions are implementation-dependent, and we consider them as implementation bugs. They can be fixed, although the performance penalty is unknown now. We focus on possible future micro-architecture designs that are safe against control flow prediction. Thus, software mitigation schemes, such as [21, 22, 83], and software vulnerability detection schemes [84, 111, 112] are not discussed in detail.

6.1 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 [127].

6.1.1 Mitigating the Trigger of Transient Execution. To mitigate Spectre-type attacks, 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 3.3). 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 flush the state during context switches. For example, there are ISA extensions for controlling and stopping indirect branch predictions [7, 53]. In [105], 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 [138], it is proposed to use 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 depends on the result of the branch condition leveraging data dependency, e.g., masking the data with the branch condition [21, 83], because current processors do not speculate on data. However, this solution requires to identify 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

Table 6: Known Transient Execution Attacks on Different Platforms.

Cause of Transient Execution		Intel	AMD [8, 20]	Arm [9, 20]	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.

identify all (current and future) disclosure gadgets, because disclosure gadgets may vary due to the encoding to different covert channels, and formal methods that model the micro-architecture behavior are required [46, 47].

6.1.2 Mitigating Transient Execution of Disclosure Gadget. To mitigate leak of secret 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 Meltdown-type and MDS-type attacks, it means to stop propagating secret data to the younger instructions. For Spectre-type attacks, 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 6.1.3.

Another solution is that data cannot be propagated speculatively, and thus, cannot be send to covert channels speculatively, which can potentially prevent transient execution attacks with any covert channel. In *Context-Sensitive Fencing* [105], fences will be injected at decoder-level to stop speculative data propagation if there are potential Spectre attacks. In *NDA* [117], a set of propagation policies are designed for defending the attacks leveraging different types of transient executions (for example, transient execution due to branch prediction or all transient execution), showing the trade-off between security and performance. Similarly, in *SpecShield* [11, 12], different propagation policies are designed and evaluated. In *Conditional Speculation* [65], 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 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 performance overhead of 7%. Furthermore, in *STT* [134], 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 [133] 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-like attacks is moderate, e.g., 7.7% in *Context-Sensitive Fencing* [105], 21% reported in *SpecShield* [11], 20 $\sim$ 51% (113% for defending all transient execution attacks) reported in *NDA* [117], and 8.5% for branch speculation (14.5% for all transient execution) in *STT* [134], 4.19% for branch speculation (10.05% for all transient execution) in *STT+SDO* [133].

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 3.3). However, the disclosure gadget can be very small that only contains two loads from L1 [124], which is only about 20 cycles in total. Detecting a malicious windowing gadget accurately can be challenging.

6.1.3 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 [26, 66, 103].

With knowledge about the data to be protected, hardware can further stop propagating secret data during speculation. The hardware can identify data that is depended on the secret with taint checking, as proposed in [37, 61, 95, 105], 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* [95], the overhead is 71.14% for OpenSSL RSA encryption and less than 1% for real-world workloads. Similar overhead is reported in *SpectreGuard* [37]. Intel also proposed a new memory type, named speculative-access protected memory (SAPM) [56]. Any access to SAPM region will cause instruction-level serialization and speculative execution beyond the SAPM-accessing instruction will be stopped until the retirement of that instruction.

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

Mitigation Schemes	Performance Overhead
Fence after each branch	88% [127]
Stop propagating all data	30–55% [12]; 21% [11]; 20–51% [117]; 8.5% [134]; 4.19% [133]
Stop propagating all data to cache changes	7.7% [105], 13% [65]
Stop propagating all data to Flush+Reload	7% [65]
Stop propagating all tagged secret data	71% for security-critical applications, < 1% for real-world workloads [37, 95]
Partitioned cache	1–15% [59]
Stop (Undo) speculative change in caches	7.6% [127]; 11% [93]; 4% [5]; 5.1% [92]; 8.3% [120]

6.2 Mitigating Covert Channels

To limit the covert channels, one way is to isolate all the hardware across the sender and receiver of the channel, so the change cannot be observable to the receiver. However, this is not always possible, e.g., in some attacks, 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 [127] proposed 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 [15] simulation results show a 7.6% performance loss for SPEC 2006 benchmark [51]. Similarly, SafeSpec [58] proposed to add “shadow buffers" to caches and TLBs, so that transient changes in the caches and TLBs does not happen.

In *Muontrap* [5], “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 shared state in L0 is not possible without causing the cache line in another cache to change state form Modified or Exclusive state, 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 [127] and SafeSpec [58], the filter cache is a real cache that is cleared upon a context switch, syscall, or when the execution change 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 [92] proposed 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 speculative changes in caches, 1Kbyte 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 results in mis-speculations, the method shows an average slowdown of 5.1%.

ReversiSpec [120] proposed a comprehensive cache coherence protocol considering speculative cache accesses. The cache coherence protocol proposed an interface including three operations: 1) speculative load, 2) merge when a speculative load is safe, 3) purge when a speculative load is squashed. Compared to InvisiSpec [127], 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 [92], purge is fast as not all the changes have propagated in to 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* [93], to improve performance, for a speculative load that miss 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* [71] 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* [115], *Newcache cache* [72, 116], *CEASER cache* [90], and *ScatterCache* [119] randomize memory-to-cache-set mapping to mitigate contention-based occupancy-based covert channels in the cache. *Non Deterministic cache* [57] randomizes cache access delay and de-couple the relation between cache block access and cache access timing. Secure TLBs [30] 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] proposed a micro-architecture 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 4.5). Limiting the receiver's observation will reduce the bandwidth or even mitigate the covert channel [89, 96]. Noise can also be added to the channel to reduce the bandwidth (the third metric in Section 4.5).

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. Formal methods are required in this process. For example, information flow tracking, such as methods in [29, 135, 136], can be used to analyze the hardware components, where the data of transient execution could flow to. Then, analyze if each of the components could result in a permanent or transient covert channel. 2. Mitigate each of the possible covert channels.

6.2.1 Mitigations in Secure Architectures. With clearly defined security domain, isolation can be designed to mitigate not only transient covert channels and also conventional covert channels. For example, to defend cache covert channels, a number of partitioned caches to different security domains are proposed, either statically [18, 27, 49, 59, 64, 70, 115, 128, 135, 136] or dynamically [32, 113]. 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 *Mi6* processor [18], caches and ports partitioning 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 [85], a dynamic partitioning scheme in the granularity of core is proposed to achieve both security and high performance.

7 CONCLUSION

This paper provide a survey of the transient execution attacks. This paper first defines the transient execution attacks and the three phases of the attacks. It then categorizes possible causes of transient executions. The security boundaries that are broken in the transient executions are discussed. It also analyzes the causes of transient execution by proposing a set of metrics and using the metrics to compare the feasibility. Furthermore, the covert channels that can be used in the attacks are categorized and compared with a new set of metrics. Combining the transient execution and the covert channels, different types of attacks are compared. In the end, possible mitigation schemes in micro-architecture designs are discussed and compared.

ACKNOWLEDGEMENTS

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

REFERENCES

[1] 2018. CVE-2018-3640. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-3640 accessed Jul. 2020.
[2] 2018. speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528 accessed May. 2019.
[3] 2019. CVE details. https://www.cvedetails.com accessed July. 2020.
[4] 2019. Intel Transactional Synchronization Extensions (Intel TSX) Overview. https://software.intel.com/en-us/cpp-compiler-developer-guide-and-reference-intel-transactional-synchronization-extensions-intel-tsx-overview accessed May. 2019.
[5] Sam Ainsworth and Timothy M Jones. 2019. Muontrap: Preventing cross-domain spectre-like attacks by capturing speculative state. *arXiv preprint arXiv:1911.08384* (2019).
[6] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. 2019. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy*. IEEE, 870–887.
[7] AMD. 2018. Software Techniques for Managing Speculation on AMD Processors. https://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf accessed May. 2019.
[8] AMD. 2020. AMD Product Security. https://www.amd.com/en/corporate/product-security accessed July. 2020.
[9] ARM. 2020. Vulnerability of Speculative Processors to Cache Timing Side-Channel Mechanism. https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability accessed July. 2020.
[10] Michael Backes, Markus Dürmuth, Sebastian Gerling, Manfred Pinkal, and Caroline Sporleder. 2010. Acoustic Side-Channel Attacks on Printers.. In *USENIX Security symposium (USENIX Security 10)*. 307–322.
[11] Kristin Barber, Anys Bacha, Li Zhou, Yinqian Zhang, and Radu Teodorescu. 2019. Specshield: Shielding speculative data from microarchitectural covert channels. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). ACM.*
[12] Kristin Barber, Li Zhou, Anys Bacha, Yinqian Zhang, and Radu Teodorescu. 2019. Isolating Speculative Data to Prevent Transient Execution Attacks. *IEEE Computer Architecture Letters* (2019).
[13] Daniel J Bernstein. 2005. Cache-timing attacks on AES. (2005).
[14] Atri Bhattacharyya, Alexandra Sandulescu, Matthias Neugschwandtner, Alessandro Sorniotti, Babak Falsafi, Mathias Payer, and Anil Kurmus. 2019. SMoTherSpectre: exploiting speculative execution through port contention. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 785–800.
[15] 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. *ACM SIGARCH Computer Architecture News* 39, 2 (2011), 1–7.
[16] Bitdefender. 2019. Bypassing KPTI Using the Speculative Behavior of the SWAPGS Instruction. https://www.bitdefender.co.th/wp-content/uploads/gz/Bitdefender-WhitePaper-SWAPGS.pdf accessed Jul. 2020.
[17] Joseph Bonneau and Ilya Mironov. 2006. Cache-collision timing attacks against AES. In *International Workshop on Cryptographic Hardware and Embedded Systems*. Springer, 201–215.
[18] Thomas Bourgeat, Ilia Lebedev, Andrew Wright, Sizhuo Zhang, Srinivas Devadas, et al. 2019. Mi6: Secure enclaves in a speculative out-of-order processor. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 42–56.
[19] Samira Briongos, Pedro Malagón, José M Moya, and Thomas Eisenbarth. 2020. RELOAD+ REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. (2020).
[20] 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 *28th USENIX Security Symposium (USENIX Security 19)*. 249–266.
[21] Chandler Carruth. 2018. Speculative Load Hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html accessed May. 2019.
[22] Microsoft Security Response Center. 2019. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886 accessed Oct. 2019.
[23] David Champagne and Ruby B Lee. 2010. Scalable architectural support for trusted software. In *2010 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 1–12.
[24] 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)*. IEEE, 288–28815.
[25] 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 *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 142–157.

[26] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016, 086 (2016), 1–118.

[27] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*. 857–874.

[28] 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 *2019 IEEE Symposium on Security and Privacy*.

[29] 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 *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*.

[30] Shuwen Deng, Wenjie Xiong, and Jakub Szefer. 2019. Secure TLBs. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

[31] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. 2017. Prime+ Abort: A Timer-Free High-Precision L3 Cache Attack using Intel TSX. In *26th USENIX Security Symposium (USENIX Security 17)*. 51–67.

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

[33] 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 *ACM SIGARCH Computer Architecture News*, Vol. 24. ACM, 3–11.

[34] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2015. Covert channels through branch predictors: a feasibility study. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 5.

[35] Dmitry Evtyushkin, Dmitry Ponomarev, and Nael Abu-Ghazaleh. 2016. Jump over ASLR: Attacking branch predictors to bypass ASLR. In *2016 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 40.

[36] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 693–707. https://doi.org/10.1145/3173162.3173204

[37] Jacob Fustos, Farzad Farshchi, and Heechul Yun. 2019. SpectreGuard: An Efficient Data-centric Defense Mechanism against Spectre Attacks.. In *Proceedings of the 56th Annual Design Automation Conference (DAC) 2019*. 61–1.

[38] Jacob Fustos and Heechul Yun. 2020. SpectreRewind: A framework for leaking secrets to past instructions. *arXiv preprint arXiv:2003.12208* (2020).

[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 *Third Workshop on Computer Architecture Research with RISC-V (CARRV 2019), Phoenix, AZ, USA*.

[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 Security 18)*. 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 (USENIX Security 15)*. 897–912.

[45] Roberto Guanciale, Hamed Nemati, Christoph Baumann, and Mads Dam. 2016. Cache storage channels: Alias-driven attacks and verified countermeasures. In *2016 IEEE 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 *2020 IEEE Symposium on Security and Privacy*. IEEE, 160–178.

[47] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. 2020. Hardware-Software Contracts for Secure Speculation. *arXiv preprint arXiv:2006.03841* (2020).

[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 *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 57–72.

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

[50] John L Hennessy and David A Patterson. 2011. *Computer architecture: a quantitative approach*. Elsevier.

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

[52] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks Against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 191–205.

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

[54] 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 *28th USENIX Security Symposium (USENIX Security 19)*. USENIX, 621–637.

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

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

[57] 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.

[58] 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 *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 60.

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

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

[61] 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 *40th IEEE Symposium on Security and Privacy*.

[62] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre returns! speculation attacks using the return stack buffer. In *12th USENIX Workshop on Offensive Technologies (WOOT 18)*.

[63] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Dawn Song, and Krste Asanovic. 2019. Keystone: A Framework for Architecting TEEs. *CoRR* abs/1907.10119 (2019). arXiv:1907.10119 http://arxiv.org/abs/1907.10119

[64] Ruby B Lee, Peter Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for protecting critical secrets in microprocessors. In *ACM SIGARCH Computer Architecture News*, Vol. 33. IEEE, 2–13.

[65] 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 *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 264–276.

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

[67] Mikko H Lipasti and John Paul Shen. 1996. Exceeding the dataflow limit via value prediction. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 226–237.

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

[69] 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 (USENIX Security 18)*.

[70] 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 *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 406–418.

[71] Fangfei Liu and Ruby B Lee. 2014. Random fill cache architecture. In *2014 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 203–215.

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

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

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

[75] 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 *Proceedings of the 35th Annual Computer Security Applications Conference*. 747–761.

[76] Nikolay Matyunin, Jakub Szefer, Sebastian Biedermann, and Stefan Katzenbeisser. 2016. Covert channels using mobile device's magnetic field sensors. In *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 525–532.

[77] Scott McFarling. 1993. *Combining branch predictors*. Technical Report. Technical Report TN-36, Digital Western Research Laboratory.

[78] 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).

[79] Pierre Michaud, André Seznec, and Richard Uhlig. 1997. Trading conflict and capacity aliasing in conditional branch predictors. In *ACM SIGARCH Computer Architecture News*, Vol. 25. ACM, 292–303.

[80] 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).

[81] 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.

[82] Donald A Neamen. 2012. *Semiconductor physics and devices: basic principles*. New York, NY: McGraw-Hill,.

[83] 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).

[84] Oleksii Oleksenko, Bohdan Trach, Mark Silberstein, and Christof Fetzer. 2020. SpecFuzz: Bringing Spectre-type vulnerabilities to the surface. In *29th USENIX Security Symposium (USENIX Security 20)*.

[85] Hamza Omar, Brandon D'Agostino, and Omer Khan. 2020. OPTIMUS: A Security-Centric Dynamic Hardware Partitioning Scheme for Processors that Prevent Microarchitecture State Attacks. *IEEE Trans. Comput.* (2020).

[86] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache attacks and countermeasures: the case of AES. In *CryptographersâĂŹ Track at the RSA Conference*. Springer, 1–20.

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

[88] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *25th USENIX Security Symposium (USENIX Security 16)*. 565–581.

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

[90] Moinuddin K Qureshi. 2018. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *2018 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 775–787.

[91] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. [n. d.]. CROSSTALK: Speculative Data Leaks Across Cores Are Real. ([n. d.]).

[92] Gururaj Saileshwar and Moinuddin K Qureshi. 2019. CleanupSpec: An Undo Approach to Safe Speculation. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 73–86.

[93] Christos Sakalis, Stefanos Kaxiras, Alberto Ros, Alexandra Jimborean, and Magnus Själander. 2019. Efficient Invisible Speculative Execution Through Selective Delay and Value Prediction. In *Proceedings of the 46th International Symposium on Computer Architecture*. ACM, 723–735.

[94] 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).

[95] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. 2020. Context: A generic approach for mitigating spectre. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS20)*.

[96] Michael Schwarz, Moritz Lipp, and Daniel Gruss. 2018. JavaScript Zero: real JavaScript and zero side-channel attacks. *Proceedings of the 25th Annual Network and Distributed System Security Symposium (NDSS18)* (2018).

[97] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. (2019), 753–768.

[98] 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.

[99] 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.

[100] 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. *Commun. ACM* 53, 7 (2010), 89–97.

[101] Eric Sprangle, Robert S Chappell, Mitch Alsup, and Yale N Patt. 1997. The agree predictor: A mechanism for reducing negative branch history interference. In *ACM SIGARCH Computer Architecture News*, Vol. 25. ACM, 284–291.

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

[103] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. 2014. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *ACM International Conference on Supercomputing 25th Anniversary Volume*. ACM, 357–368.

[104] Jakub Szefer. 2018. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security* (13 September 2018). https://doi.org/10.1007/s41635-018-0046-1

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

[106] 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).

[107] Paul Turner. 2018. Mitigating speculative execution side channel hardware vulnerabilities. https://github.com/intelstormteam/Papers accessed Oct. 2019.

[108] 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 *27th USENIX Security Symposium (USENIX Security 18)*. 991–1008.

[109] 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 *2020 IEEE Symposium on Security and Privacy*. 1399–1417.

[110] 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 *2019 IEEE Symposium on Security and Privacy*. IEEE, 88–105.

[111] Guanhua Wang, Sudipta Chattopadhyay, Arnab Kumar Biswas, Tulika Mitra, and Abhik Roychoudhury. 2020. Kleespectre: Detecting information leakage through speculative cache attacks via symbolic execution. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 3 (2020), 1–31.

[112] Guanhua Wang, Sudipta Chattopadhyay, Ivan Gotovchits, Tulika Mitra, and Abhik Roychoudhury. 2019. oo7: Low-overhead Defense against Spectre attacks via Program Analysis. *IEEE Transactions on Software Engineering* (2019).

[113] 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 (DAC), 2016 53nd ACM/EDAC/IEEE*. IEEE, 1–6.

[114] Zhenghong Wang and Ruby B Lee. 2006. Covert and side channels due to processor architecture. In *Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, 473–482.

[115] 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.

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

[117] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 572–586.

[118] 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*. Technical Report. Technical report.

[119] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. 2019. Scattercache: Thwarting cache attacks via cache set randomization. In *28th USENIX Security Symposium (USENIX Security 19)*. 675–692.

[120] You Wu and Xuehai Qian. 2020. ReversiSpec: Reversible Coherence Protocol for Defending Transient Attacks. *arXiv preprint arXiv:2006.16535* (2020).

[121] Zhenyu Wu, Zhang Xu, and Haining Wang. 2014. Whispers in the hyper-space: high-bandwidth and reliable covert channel attacks inside the cloud. *IEEE/ACM*

Transactions on Networking 23, 2 (2014), 603–615.

[122] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *Proceedings of the 27th Annual Network and Distributed System Security Symposium (NDSS20)*.

[123] Wenjie Xiong, Nikolaos Athanasios Anagnostopoulos, André Schaller, Stefan Katzenbeisser, and Jakub Szefer. 2019. Spying on Temperature using DRAM. In *Proceedings of the Design, Automation, and Test in Europe (DATE)*.

[124] Wenjie Xiong and Jakub Szefer. 2020. Leaking Information Through Cache LRU States. (2020), 139–152.

[125] 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 *Proceedings of the 3rd ACM workshop on Cloud computing security workshop*. ACM, 29–40.

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

[127] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy. In *2018 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 428–441.

[128] 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 *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 347–360.

[129] 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 *2019 IEEE Symposium on Security and Privacy*. IEEE, 888–904.

[130] Fan Yao, Milos Doroslovacki, and Guru Venkataramani. 2018. Are Coherence Protocol States Vulnerable to Information Leakage?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 168–179.

[131] Yuval Yarom and Katrina Falkner. 2014. FLUSH+ RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack.. In *USENIX Security Symposium (USENIX Security 14)*, Vol. 1. 22–25.

[132] 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.

[133] 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 *Proceedings of the International Symposium on Computer Architecture (ISCA)*. IEEE, 707–720.

[134] 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 *2019 IEEE/ACM International Symposium on Microarchitecture (MICRO)*. ACM, 954–968.

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

[136] 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.

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

[138] 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).