

# The Evolution of Transient-Execution Attacks

Claudio Canella  
Graz University of Technology  
claudio.canella@iaik.tugraz.at

Khaled N. Khasawneh  
George Mason University  
kkhasawn@gmu.edu

Daniel Gruss  
Graz University of Technology  
daniel.gruss@iaik.tugraz.at

## ABSTRACT

Historically, non-architectural state was considered non-observable. Side-channel attacks, in particular on caches, already showed that this is not entirely correct and meta-information, such as the cache state, can be extracted. Transient-execution attacks emerged when multiple groups discovered the exploitability of speculative execution and, simultaneously, the exploitability of deferred permission checks in modern out-of-order processors. These attacks are called transient as they exploit that the processor first executes operations that are then reverted as if they were never executed. However, on the microarchitectural level, these operations and their effects can be observed. While side-channel attacks enable and exploit direct access to meta-data from other security domains, transient-execution attacks enable and exploit direct access to actual data from other security domains. In this paper, we show how the transient-execution landscape evolved since the initial discoveries. We show that the understanding and systematic view of the field has advanced and now facilitate the discovery of new attack variants.

## KEYWORDS

transient execution, Meltdown, Spectre, LVI

### ACM Reference Format:

Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. 2020. The Evolution of Transient-Execution Attacks. In *Proceedings of the Great Lakes Symposium on VLSI 2020 (GLSVLSI '20)*, September 7–9, 2020, Virtual Event, China. ACM, New York, NY, USA, 6 pages. <https://doi.org/10.1145/3386263.3407583>

## 1 INTRODUCTION

Modern processors are designed to comply with a defined interface, the instruction-set architecture, allowing to run the same binaries on different processors. Hence, processors can optimize performance and efficiency as long as the behavior is architecturally defined. One optimization that improves performance substantially is caching. Caching is entirely transparent to software, but it introduces a significant speed up for memory accesses and execution by storing data likely to be accessed in the near future in small internal CPU memories. Another optimization is branch prediction. The processor guesses which direction of a branch is taken and thus does not have to wait with the execution of further instructions until the branch is resolved. If the branch predictor correctly predicted the outcome, a significant performance gain is possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

GLSVLSI '20, September 7–9, 2020, Virtual Event, China

© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.

ACM ISBN 978-1-4503-7944-1/20/09...\$15.00

<https://doi.org/10.1145/3386263.3407583>

From a security perspective, non-architectural state was long considered non-observable as there is no attacker-accessible interface to check what is in the cache. Kocher [23] brought up the idea that caches could be used in side-channel attacks. Such attacks measure the latency of memory accesses to infer what is in the cache, yielding an interface that makes non-architectural state visible to an attacker. However, the information extracted here is only meta-information, *i.e.*, whether the data was retrieved from the cache or main memory. While this enables powerful cryptographic [28] and non-cryptographic [12] attacks, both the security community and the architecture community deemed it most reasonable to design code in a way such that secrets do not influence meta-information, *e.g.*, no secret-dependent memory accesses [3].

In 2018, Spectre [22] and Meltdown [25] were disclosed to the public. They exploit transient execution, the execution of operations which should not be executed, to leave microarchitectural traces. The processor runs into transient execution when a branch is mispredicted, due to lazy exception handling, or other microarchitectural constellations requiring at least a partial pipeline flush or stall. Initially, it was expected that more Spectre variants would be found, but Meltdown was perceived as a one-off vulnerability.

With Foreshadow [38, 41], it became clear that further Meltdown-like effects exist. Foreshadow extended the user-to-kernel attack, to a generic attack able to leak data from any physical memory location. In some sense, it foreshadowed the further development in this field, with many different Meltdown-type attacks discovered by now [4, 32, 40].

While the field of transient-execution attacks has grown ever since its initial discovery, so has the field of potential countermeasures. Concurrent work has analyzed and highlighted different approaches from academia and industry in mitigating Spectre-, Meltdown-, and LVI-type attacks [5].

In this paper, we show how the transient-execution landscape evolved since these initial discoveries. We introduce a 6-phase generalization of transient-execution attacks. We describe the currently known types of Spectre attacks, exploiting microarchitectural mispredictions. In contrast to previous work [7], we then provide a novel systematic view on Meltdown attacks that emphasizes similarities between Meltdown-type attacks rather than differences. We show that the understanding and systematic view of the field has advanced since the 2018 disclosure. We conclude that the discovery of new attack variants increasingly builds on this systematic view.

**Outline.** Section 2 provides technical background on speculative and out-of-order execution, cache attacks, and store-to-load forwarding. Section 3 introduces transient-execution attacks and structures them in 6 attack phases. Section 4 describes state of the art on Spectre-type attacks, Meltdown-type attacks based on the similarities we identified, and LVI attacks as inverted Meltdown-type attacks. Section 5 concludes our paper.

## 2 BACKGROUND

In this section, we provide background on speculative and out-of-order execution, cache attacks in general as well as on store-to-load forwarding.

### 2.1 Speculative and Out-of-Order Execution

As CPUs need to outperform the previous generation of CPUs, vendors include many different forms of optimizations. Two such optimizations are speculative and out-of-order execution.

**Out-of-Order Execution.** CPUs parallelize more and more parts of the execution of instructions. In the pipeline, the fetch, decode, execute, and write-back stages handle a multitude of operations simultaneously. This parallelization allows the CPU to execute instructions out-of-order as soon as their operands are available, while instructions that precede them in the instruction stream are still waiting for their operands. Even though the instructions are executed out-of-order, they must still retire (commit) in-order as in the instruction stream to ensure a correct architectural state and enable precise interrupts. This design goes back to Tomasulo’s algorithm [37].

**Speculative Execution.** Speculative execution is an optimization that tries to improve the performance of the non-linear instruction stream of an application. The CPU contains a branch prediction unit (BPU) that tries to predict, based on past behavior, what the most likely direction of control flow is in the instruction stream. The BPU typically consists of several structures, each designed to predict the behavior of different control-flow mechanisms. We refer to Canella et al. [7] for a more detailed description of the individual structures that comprise the BPU.

### 2.2 Cache Attacks

Cache attacks have become a widely used building block in microarchitectural attacks, especially for transient-execution attacks. Cache attacks exploit the timing difference between accessing data that resides in the cache, e.g., data was used recently, and data in main memory. Hence, due to the cache being closer to the CPU, its latency in responding to a memory request is far lower than a main memory access. While there are a variety of cache attacks such as Flush+Reload [13, 44], Prime+Probe [28, 29], Flush+Flush [10], or Evict+Time [3, 28], all of them share the same three stages.

- (1) The attacker brings the microarchitecture into a pre-defined state.
- (2) The attacker lets the victim perform an operation.
- (3) The attacker infers information on victim behavior based on microarchitectural state changes.

Covert channels are a special case of cache attacks. In such a scenario, the attacker controls both the sender and the receiver, enabling them to bypass restrictions on the architectural level.

### 2.3 Store-to-Load Forwarding

For store-to-load forwarding, two components are used, namely the memory disambiguation predictor and the store buffer. The idea for store-to-load forwarding is to forward the data from a store that is in the store buffer and has not yet been written back to the cache to a load that uses the same address. There are four different

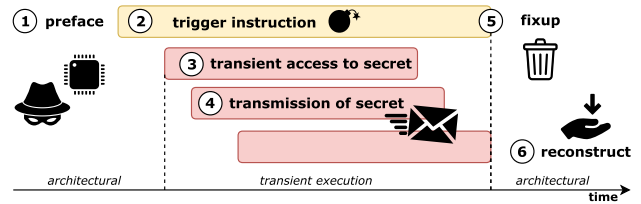


Figure 1: **High-level overview of a transient-execution attack in 6 phases: (1) preparation, (2) transient-execution trigger, (3) access secret, (4) encode secret, (5) architectural revert, (6) decode secret.**

cases that need to be considered for store-to-load forwarding with respect to the load address:

- **True positive match.** A store buffer entry is matched with the same full physical address. In this case, the data is forwarded, which is the correct behavior.
- **True negative match.** No store buffer entry matches, and there is no store buffer entry with the same full physical address. No data is forwarded, which is the correct behavior.
- **False negative match.** No store buffer entry matches, although there is one with the exact same full physical address. The load retrieves stale data, e.g., from the L1 cache, and has to be reissued later on.
- **False positive match.** A store buffer entry is matched, but the full physical address check at the end fails. In this case, data is first transiently forwarded, but the load has to be reissued later on.

## 3 BASIC IDEA OF TRANSIENT-EXECUTION ATTACKS

With transient execution, we describe the execution of instructions that are executed but whose results are never committed to the architectural level. We refer to these instructions as transient instructions [7, 22, 25]. Nevertheless, these transient instructions leave traces behind in the microarchitecture by changing its state, e.g., loading data into the cache. The literature has so far identified two different causes of transient execution, namely speculative and out-of-order execution [7, 22, 25]. With speculative execution, it is possible to execute transient instructions due to a wrong prediction of a branch or data dependency. Out-of-order execution can lead to transient execution due to a previous instruction in the pipeline triggering an exception. In that case, the next instructions in the pipeline may still be executed. Both cases have one thing in common: the operations and the architectural state have to be reverted, including flushing the pipeline.

While transient-execution attacks so far have relied on side channels, *i.e.*, predominantly the cache, for the transmission, we note that they themselves are not side channels, contradicting previous literature [14].

Canella et al. [7] proposed 5 distinct phases for a transient-execution attack while Xiong and Szefer [43] propose 3 phases. The phases by Xion and Szefer [43] are similar to the ones by Canella [7] with the three middle phases merged together. We extend the proposal by Canella et al. [7] with one additional distinct

Phase 3 that we will describe in the following. Our extended phases are shown in Figure 1. Note that each phase may be either performed indirectly by the victim, *i.e.*, attacker provides the necessary trigger to the victim, or directly by the attacker.

**Phase 1.** This phase is the preparation phase of the attack and consists of two parts. In the first, the attacker prepares the microarchitecture in a way that allows entering transient execution during which the secret data is accessed. This includes ensuring that the transient window, *i.e.*, the time between the execution of the first transient instruction and the last one, is large enough. In the second, the attacker prepares the microarchitectural transmission channel so that the data can be recovered.

**Phase 2.** The attacker triggers transient execution using a trigger instruction. For Spectre-type attacks, this could be any branch in the victim domain. For Meltdown-type or LVI-type attacks, this could be any form of an aborting instruction, *i.e.*, an instruction that triggers a fault, an assist, or an interrupt. In the case of Spectre-type and LVI-type attacks, the trigger instruction is in the victim domain. For Meltdown-type attacks, it is in the attacker domain.

**Phase 3.** We split Phase 3 from the proposal by Canella et al. [7] into two phases, Phase 3 and Phase 4. In Phase 3, the transient instructions are executed either in the victim or the attacker domain. Typically, these instructions access data of interest to the attacker, *i.e.*, a secret such as a password. As a final step in this phase, the attacker prepares the data for transmission through a microarchitectural covert channel such as the cache.

**Phase 4.** This phase is the last in the transient domain. In this phase, the attacker encodes the previously prepared data in the microarchitecture. So far, the covert channel that has been used predominantly in the published literature is the cache [7].

**Phase 5.** In this phase, the CPU realizes the mistake, *i.e.*, the misprediction or that an exception has been raised. The architectural changes are reverted and the pipeline is flushed. The CPU then continues with the execution of the correct instructions, *i.e.*, a number of instructions may have to be reissued. In the case of Spectre, the CPU executes the correct side of the branch; in Meltdown, it executes the exception handler. Nevertheless, in this phase, the information the attacker wants to obtain is already encoded in the microarchitecture where not all changes are reverted.

**Phase 6.** In the recovery phase, the attacker extracts the encoded data from the microarchitectural level to the architectural level. As the data is usually encoded in the cache, the attacker uses cache side channels, such as Flush+Reload, to perform this task.

## 4 TRANSIENT-EXECUTION ATTACKS

In this section, we discuss the known transient-execution attacks in more detail. We first focus on attacks that belong to the Spectre class of attacks, then move on to the Meltdown class of attacks, and finally discuss LVI-type attacks.

### 4.1 Spectre-type Attacks

Spectre-type attacks so far exclusively rely on exploiting transient execution following a misprediction. Misprediction occurs whenever the CPU predicts to take the wrong path based on past behavior. To make this prediction, the BPU consists of multiple predictors, each designed to predict a different type of branch. The Pattern

History Table (PHT) is used to predict the outcome of a direct branch, such as an `if`-statement. Typically, the PHT is indexed by some bits of a virtual address and contains a saturating counter per entry [9]. The Branch Target Buffer (BTB) is used to predict the target of a branch following an indirect branch. For that, the BTB stores the previous jump target for the current address. The Return Stack Buffer (RSB) is used to improve the performance of return statements. Every call stores the return address at the top of the RSB, and each return pops the top of it. The last predictor that has been exploited so far is the memory disambiguator that is involved in store-to-load forwarding. This predictor tries to predict whether a load operation overlaps with a previous store operation [20].

Canella et al. [7] systematically analyzed Spectre-type attacks and how each unit is shared within a processor core. Their analysis showed various potential strategies for mistraining the individual branch predictors. Same-address-space strategies mistrain within the same address space using either the target address or a congruent address that refers to the same entry in the predictors. They also showed that the same two strategies can be exploited in a cross-address-space scenario if the predictor is shared between hyperthreads. Their proposed classification also indicates for each Spectre variant which predictor is exploited.

Spectre-type attacks require different types of gadgets, *i.e.*, code sequences, to execute. These types, as well as different detection techniques, have been analyzed by Canella et al. [7]; hence we refer the reader to their paper for further information.

**Spectre-PHT.** Spectre-PHT was one of the first two Spectre variants discovered by Kocher et al. [22]. Here, the PHT is exploited by mistraining it repeatedly to take a certain path following, *e.g.*, a bounds check. Afterward, the attacker can supply a value that would cause the CPU to execute the other path of the bounds check, but due to the prediction, it still executes the mistrained path. Schwarz et al. [33] have demonstrated that such an attack is also possible via the network.

**Spectre-BTB.** Spectre-BTB is the second variant discovered by Kocher et al. [22]. As the name indicates, it exploits the behavior of the BTB to force the CPU to predict a wrong jump target. With Spectre-BTB, the attacker can redirect control-flow to virtually any address within the victim domain. Similarly to return-oriented programming (ROP) [34], the attacker can chain multiple jump targets together to achieve arbitrary transient execution as far as the transient window allows. Straight-line speculation vulnerabilities [2] exploiting exception generating instructions, unconditional direct and indirect branches also fall into this category.

**Spectre-RSB.** Horn [15] and Kocher et al. [22] first mentioned the possibility of exploiting the RSB, but the first attack was demonstrated by Koruyeh et al. [24] and Maisuradze and Rossow [27]. In a Spectre-RSB attack, the attacker poisons the RSB with malicious return destinations that the victim program pops from the stack upon a return, redirecting control flow to a leak gadget. Straight-line speculation vulnerabilities [2] exploiting exception or function returns fall into this category as well.

**Spectre-STL.** Horn [16] demonstrated that loads can transiently obtain stale values. The reason for that is that the memory disambiguator predicts that a current load does not depend on any previous store. The load operation is then scheduled before the preceding store and obtains the old value from the cache. In its

essence, Spectre-STL exploits a false negative match in the store buffer.

## 4.2 Meltdown-type Attacks

Canella et al. [7] proposed a canonical naming scheme for Meltdown-type attacks. In the first layer, they distinguish the type of the fault or the assist that causes transient execution. In the following layers, they distinguish different reasons for the fault or assist and from which buffer they leak from. Consequently, every Meltdown-type attack has a precise canonical name.

In this paper, we want to highlight the similarities between the different Meltdown-type attacks. Based on our analysis of the attacks, we divide Meltdown-type attacks into the following three groups based on their microarchitectural behavior:

- **Deferred Permission Check.** Meltdown-type attacks in this category show architecturally correct behavior but with a lack of permission checks, e.g., Meltdown-US [25]. These Meltdown-type attacks perform operations that, from the CPU's perspective, would be valid and meaningful at a different permission level. For instance, attempting to access a kernel address is valid and meaningful for kernel code.
- **Incorrect Use of Intermediate Values.** Some Meltdown-type attacks exploit the usage of intermediate values to retrieve data, e.g., Foreshadow [38, 41]. The behavior exploited in these attacks is always either not valid or not meaningful, regardless of the permission level. For instance, the architecture defines that a non-present page-table entry may contain any data. Interpreting this data, e.g., as a physical address, is always incorrect.
- **Use-After-Free.** Newer Meltdown-type attacks show behavior that is similar to use-after-free vulnerabilities, causing the usage of stale values, e.g., ZombieLoad [32], RIDL [40], Fallout [4].

**4.2.1 Deferred Permission Check.** The idea behind Meltdown-type attacks in this category is that data is forwarded despite the permission check failing in parallel. For each load  $\mu$ OP that is added to the re-order buffer, a new entry in the load buffer is reserved. When the load  $\mu$ OP is scheduled, the load buffer is updated with information about the new load, *i.e.*, register number and virtual address information. Simultaneous to the lookup in the memory hierarchy, the TLB is checked to retrieve the corresponding physical address. Once the page-table information has been retrieved, the permissions are checked. As the permission check fails, the processor raises a fault but still updates the physical page number in the load buffer. At this point in time, the data is ready to be forwarded to the register and can be used in subsequent transient instructions.

**Attacks.** Meltdown-US, the original Meltdown [25], exploits faults following a check on the user-space-accessible bit and allows leaking kernel data. Canella et al. [6] demonstrate such an attack on 32-bit Linux systems using JavaScript and also provide the first detailed analysis of Intel's hardware Meltdown mitigation. Canella et al. [7] used Meltdown-PK to bypass read and write isolation provided by Intel's memory protection keys. Furthermore, they showed that hardware instructions for bounds checking provided on both IA-32 and IA-64 can be bypassed with Meltdown-BND and

Meltdown-MPX. Meltdown-BND was also the first demonstration of Meltdown-type effects on AMD processors. Other attacks demonstrated that data can be leaked from registers. The first, Meltdown-GP [1, 17, 18] can be used to leak data from privileged system registers. With the second, Stecklina and Prescher [35] showed that registers that can be switched between kernel and user mode can be exploited. In their paper, they targeted FPU and SIMD registers. Xiao et al. [42] developed a framework to study transient-execution attacks, discovering a new Meltdown variant on AMD processors that bypasses segment limits.

Two cases of store-to-load forwarding, namely the true positive and the true negative match, also belong in this attack category. For a true positive match, Kiriansky and Waldspurger [21] observed that the writable bit is transiently ignored when performing store-to-load forwarding. Schwarz et al. [31] and Canella et al. [6, 7] demonstrated that this also happens for other checks, e.g., the user-space-accessible bit. For a true negative match, Schwarz et al. [31] and Canella et al. [6, 7] showed that the information about the negative case can be abused in combination with the true positive case for distinguishing valid addresses from invalid ones, enabling KASLR breaks.

**4.2.2 Incorrect Use of Intermediate Values.** This category mainly consists of Meltdown-P attacks [7, 38, 41]. Meltdown-P attacks follow the same behavior as a Meltdown-US attack, but instead of exploiting the user-accessible bit it exploits the present bit. At the point where the TLB is queried for the valid translation, the lookup fails as the page is indicated as not being present. To perform the page-table walk, a microcode assist is issued [32]. As the attacker runs within a virtual machine, two such walks have to be issued as the guest virtual address is first translated into a guest physical which is then translated into a host physical address. When the information of the guest page table is checked, none of the information is valid as the present bit is not set, causing the processor to raise a fault. Nevertheless, the physical address field in the load buffer is filled out with the information from the page table, *i.e.*, the guest physical address. Similar to Meltdown-US, the data is ready to be forwarded to the register, and the processor matches the guest's physical address to the cache line tag of the retrieved data. This allows an attacker to read arbitrary host physical memory by exploiting non-present page-table entries and a transient access to them.

**Attacks.** Van Bulck et al. [38] have demonstrated this effect in the context of transiently forwarding data from SGX-protected cache lines. Weisse et al. [41] have extended their work by demonstrating the effect in a non-SGX environment, such as the aforementioned virtual machine scenario.

The two remaining cases of store-to-load forwarding, namely the false negative and the false positive match, are also in this category. For a false negative match, Cauligi et al. [8] describe a theoretical attack that combines Meltdown- and Spectre-type effects. In such an attack, the memory disambiguator predicts a dependency, allowing the load to continue. The store buffer then only finds a partial match and returns the incorrectly matched data. Canella et al. [4] exploited a false-positive match to read recent writes from SGX enclaves or the kernel.

**4.2.3 Use-After-Free.** Use-after-free vulnerabilities are not only common in software [36]. We discuss the basic idea behind attacks in this category based on a ZombieLoad v1 attack. While the steps are the same as in the previous attacks, the suspected reason for the erroneous matching of the secret data is the stale physical address from the load buffer. In a ZombieLoad v1 attack, the store buffer, line-fill buffer, L1 data cache are looked up with the virtual address. Due to a cache line conflict, the L1 data cache lookup fails, leading to an abort and a subsequent re-issuing of the load. Nevertheless, the current load, a zombie load, continues and returns data. The stale physical page number (*i.e.*, a use after free) is used to match the tag of the previously retrieved data. If they match, the data is forwarded to the register and can be leaked by subsequent instructions.

**Attacks.** The first description of such an attack was given by Lipp et al. [25] when they demonstrated a Meltdown-US attack on uncached and uncacheable memory. They showed that leaking from uncacheable memory is only possible if an architectural access to the same memory location occurs, *i.e.*, a hyperthread performs a legal load. According to Lipp et al. [11, 25], the leakage originates in the line-fill buffer. Subsequent publications further investigate this buffer as a leakage source [19, 26, 30, 32, 40]. On processors that do not leak from L1 directly anymore, Van Schaik et al. [40] and Schwarz et al. [32] showed that there is still leakage possible via so called microarchitectural data sampling (MDS) attacks. For instance, Schwarz et al. [32] demonstrated different variants to leak data from the line-fill buffer due to aborted loads continuing their execution. Additionally, they showed that one of these variants even affects Intel Cascade Lake CPUs that contain fixes for Meltdown and MDS attacks. Finally, Schwarz et al. [32] also demonstrated that the initial mitigation of using the `verw` instruction to overwrite buffers does not fully work.

### 4.3 LVI-Type Attacks

Spectre turned previously known attacks on branch prediction around, not exploiting the branch prediction to retrieve data from a previous execution, but instead injecting a transient control-flow change into a victim to leak data. LVI (Load Value Injection) [39] does the same with Meltdown, *i.e.*, instead of exploiting Meltdown-type effects to retrieve data from a victim domain, it uses them to transiently inject data into the victim. Contrary to Meltdown-type attacks, LVI attacks cannot always control when a fault and other conditions occur as they occur within the victim domain. In that sense, LVI is more similar to Spectre-type attacks where certain gadgets within the victim are required. For LVI, the complexity of gadgets is much lower as a single memory access can already be a LVI gadget. The same is true for a single indirect call, jump, or return.

In principle, an LVI attack attempts to obtain data from the victim domain that the victim can legally access. All three of the previously identified types of Meltdown-type effects can be used for LVI attacks, *i.e.*, deferred permission checks, incorrect use of intermediate values, and use-after-free. While all three types can be exploited, the deferred permission check type of attacks is more realistic in the threat model of SGX because they require repeated illegal behavior of the victim domain. Table 1 shows the different

Attack	Requirement
LVI-US	victim repeatedly accesses a kernel address
LVI-RW	illegal write to read-only memory
LVI-PK	illegal access to a PKU-protected memory location
LVI-MPX, LVI-BND	illegal out-of-bounds access
LVI-GP	memory access triggering a general protection fault
LVI-NM	FPU register access with FPU being unavailable
LVI on segment limits	memory access outside segment limit

Table 1: **The different LVI attacks relying on a deferred permission check and what event they require to enter transient execution.**

attack types relying on the deferred permission check and what they require to trigger the attack.

In a normal scenario, faults are handled by the operating system, preventing them from happening again. Hence, a more realistic scenario for an LVI attack in a normal context is to use misprediction to suppress the fault. Using misspeculation, the attacker can suppress the fault as previous work has already outlined [7, 22, 25], but in this case, a Spectre-type attack might already be sufficient to extract the targeted data from the victim. In this scenario, LVI may only be of use if the Spectre-type attack does not allow enough control of the victim, which LVI might be able to supply.

In the SGX scenario, one type of attack that is particularly relevant is the incorrect use of intermediate values. By inverting Meltdown-P to obtain LVI-P, the attacker can transiently inject data from a chosen physical address into the victim’s execution by unmapping a page and loading data into the L1 data cache. For the LVI-NUL case, the attacker can inject a NULL value instead, *e.g.*, on processors with first-generation mitigations. Both cases have been demonstrated by Van Bulck et al. [39]. For LVI-P-NUL, the attacker transiently injects a NULL value when accessing the stack to read the return address from virtual address 0, redirecting control flow to the location that is contained at this location. In LVI-P-L1D, fake return addresses are injected to re-direct control flow to an attacker chosen location.

For non-SGX-based LVI attacks, use-after-free type of attacks have been shown to be the most effective [39]. In these scenarios, microarchitectural assists have been used to trigger the use-after-free situation in the line-fill buffer. The store buffer false positive match is one case that can occur easily, but the requirements on gadgets are much higher than in the LFB case. For this attack, the attacker has to place a matching store in the store buffer, which is statically partitioned. Hence, the attacker has to inject the store on the same hyperthread before a context switch or via a gadget within the victim that stores to the attacker-tweakable address before reading from an unrelated, but partially matching, address. If the store is successfully injected, the store buffer false-positively matches a store, *i.e.*, without a full physical address match. As in the case of ZombieLoad, the load in the victim continues but is ultimately squashed. Nevertheless, the wrong data is transiently forwarded to the dependent operations.

## 5 CONCLUSION

For a long time, the non-architectural state was considered to be unobservable. Side-channel attacks have demonstrated that this is not entirely true as meta-information can be extracted from the

microarchitectural state. To optimize the performance of modern processors, vendors included various optimizations such as speculative and out-of-order execution. Transient-execution attacks exploit these features as their effect on the microarchitectural state is not reverted even in the case that the result was unrolled. Contrary to side-channel attacks, transient-execution attacks allow direct access to the actual data from another security domain, not just meta-data. In this paper, we showed how the transient-execution landscape has evolved since the initial discoveries. We showed that the understanding and systematic view of the field has advanced and now facilitate the discovery of new attack variants.

## ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union’s Horizon 2020 research and innovation program (grant agreement No 681402). This work has been supported by the Austrian Research Promotion Agency (FFG) via the project ESPRESSO, which is funded by the province of Styria and the Business Promotion Agencies of Styria and Carinthia. Additional funding was provided by generous gifts from ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

## REFERENCES

- [1] ARM. 2018. Cache Speculation Side-channels. Version 2.4.
- [2] ARM. 2020. Straight-line Speculation. Version 1.0.
- [3] Daniel J. Bernstein. 2005. Cache-Timing Attacks on AES. <http://cr.ypt.to/antiforgery/cachetiming-20050414.pdf>
- [4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. 2019. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*.
- [5] Claudio Canella, Sai Manoj Pudukotai Dinakararao, Daniel Gruss, and Khaled N. Khasawneh. 2020. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*.
- [6] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. 2020. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*.
- [7] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtuyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*. Extended classification tree and PoCs at <https://transient.fail/>.
- [8] Sunjay Cauligi, Craig Disselkoen, Klaus v Gleissenthall, Deian Stefan, Tamara Rezk, and Gilles Barthe. 2019. Towards Constant-Time Foundations for the New Spectre Era. *arXiv:1910.01755* (2019).
- [9] Agner Fog. 2016. The microarchitecture of Intel, AMD and VIA CPUs: An optimization guide for assembly programmers and compiler makers.
- [10] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. In *DIMVA*.
- [11] Daniel Gruss, Michael Schwarz, and Moritz Lipp. 2018. Meltdown: Basics, Details, Consequences. In *BlackHat USA*.
- [12] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. 2015. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium*.
- [13] David Gullasch, Endre Bangerter, and Stephan Krenn. 2011. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *S&P*.
- [14] Mark D Hill, Jon Masters, Parthasarathy Ranganathan, Paul Turner, and John L Hennessy. 2019. On the Spectre and Meltdown Processor Security Vulnerabilities. *IEEE Micro* 39, 2 (2019), 9–19.
- [15] Jann Horn. 2018. Reading privileged memory with a side-channel.
- [16] Jann Horn. 2018. speculative execution, variant 4: speculative store bypass.
- [17] Intel. 2018. Intel Analysis of Speculative Execution Side Channels. Revision 4.0.
- [18] Intel. 2018. Q2 2018 Speculative Execution Side Channel Update.
- [19] Intel. 2019. Intel-SA-00233 Microarchitectural Data Sampling Advisory. <https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00233.html>
- [20] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Kriebel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *USENIX Security Symposium*.
- [21] Vladimir Kiriansky and Carl Waldspurger. 2018. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757* (2018).
- [22] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. 2019. Spectre Attacks: Exploiting Speculative Execution. In *S&P*.
- [23] Paul C. Kocher. 1996. Timing Attacks on Implementations of Diffe-Hellman, RSA, DSS, and Other Systems. In *CRYPTO*.
- [24] Esmail Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. 2018. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*.
- [25] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. 2018. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*.
- [26] Giorgi Maisuradze. 2019. *Assessing the Security of Hardware-Assisted Isolation Techniques*. Ph.D. Dissertation, Saarland University.
- [27] G. Maisuradze and C. Rossow. 2018. ret2spec: Speculative Execution Using Return Stack Buffers. In *CCS*.
- [28] Dag Arne Osvik, Adi Shamir, and Eran Tromer. 2006. Cache Attacks and Countermeasures: the Case of AES. In *CT-RSA*.
- [29] Colin Percival. 2005. Cache missing for fun and profit. In *BSDCan*.
- [30] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2021. CrossTalk: Speculative Data Leaks Across Cores Are Real. In *S&P*.
- [31] Michael Schwarz, Claudio Canella, Lukas Giner, and Daniel Gruss. 2019. Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs. *arXiv:1905.05725* (2019).
- [32] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. 2019. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*.
- [33] Michael Schwarz, Martin Schwarzl, Moritz Lipp, and Daniel Gruss. 2019. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*.
- [34] Hovav Shacham. 2007. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*.
- [35] Julian Stecklina and Thomas Prescher. 2018. LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels. *arXiv:1806.07480* (2018).
- [36] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. 2013. SoK: Eternal War in Memory. In *S&P*.
- [37] Robert M Tomasulo. 1967. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development* 11, 1 (1967), 25–33.
- [38] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasicki, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*.
- [39] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. 2020. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*.
- [40] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. 2019. RIDL: Rogue In-flight Data Load. In *S&P*.
- [41] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasicki, 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. <https://foreshadowattack.eu/foreshadow-NG.pdf>
- [42] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. 2020. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*.
- [43] Wenjie Xiong and Jakub Szefer. 2020. Survey of Transient Execution Attacks. *arXiv:2005.13435* (2020).
- [44] Yuval Yarom and Katrina Falkner. 2014. Flush+Reload: a High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium*.