LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection

Jo Van Bulck*, Daniel Moghimi†, Michael Schwarz‡, Moritz Lipp‡, Marina Minkin§, Daniel Genkin§,
Yuval Yarom¶, Berk Sunar†, Daniel Gruss‡, and Frank Piessens*

*imec-DistriNet, KU Leuven   †Worcester Polytechnic Institute   ‡Graz University of Technology
§University of Michigan   ¶University of Adelaide and Data61

Abstract—The recent Spectre attack first showed how to inject incorrect branch targets into a victim domain by poisoning microarchitectural branch prediction history. In this paper, we generalize injection-based methodologies to the memory hierarchy by directly injecting incorrect, attacker-controlled values into a victim's transient execution. We propose *Load Value Injection (LVI)* as an innovative technique to reversely exploit Meltdown-type microarchitectural data leakage. LVI abuses that faulting or assisted loads, executed by a legitimate victim program, may transiently use dummy values or poisoned data from various microarchitectural buffers, before eventually being re-issued by the processor. We show how LVI gadgets allow to expose victim secrets and hijack transient control flow. We practically demonstrate LVI in several proof-of-concept attacks against Intel SGX enclaves, and we discuss implications for traditional user process and kernel isolation.

State-of-the-art Meltdown and Spectre defenses, including widespread silicon-level and microcode mitigations, are orthogonal to our novel LVI techniques. LVI drastically widens the spectrum of incorrect transient paths. Fully mitigating our attacks requires serializing the processor pipeline with `lfence` instructions after possibly *every* memory load. Additionally and even worse, due to implicit loads, certain instructions have to be blacklisted, including the ubiquitous x86 `ret` instruction. Intel plans compiler and assembler-based full mitigations that will allow at least SGX enclave programs to remain secure on LVI-vulnerable systems. Depending on the application and optimization strategy, we observe extensive overheads of factor 2 to 19 for prototype implementations of the full mitigation.

I. INTRODUCTION

Recent research on transient-execution attacks has been characterized by a sharp split between on the one hand Spectre-type misspeculation attacks, and on the other hand, Meltdown-type data extraction attacks. The first category, Spectre-type attacks [4, 23, 38, 39, 44], trick a victim into transiently diverting from its intended execution path. Particularly, by poisoning the processor's branch predictor machinery, Spectre adversaries steer the victim's transient execution to gadget code snippets, which inadvertently expose secrets through the shared microarchitectural state. Importantly, Spectre gadgets execute entirely *within* the victim domain and can hence only leak architecturally accessible data.

The second category consists of Meltdown-type attacks [9, 42, 53, 57, 61, 67, 70], which target architecturally inaccessible data by exploiting illegal data flow from faulting or assisted instructions. Particularly, on vulnerable processors, the results of unauthorized loads are still forwarded to subsequent transient operations, which may encode the data before an exception is eventually raised. Over the past year, delayed exception handling and microcode assists have been shown to transiently expose data from various microarchitectural elements (*i.e.*, L1D cache [42, 61], FPU register file [57], line-fill buffer [42, 53, 67], store buffer [9], and load ports [29, 67]). Unlike Spectre-type attacks, a Meltdown attacker in one security domain can directly exfiltrate architecturally inaccessible data belonging to another domain (e.g., kernel memory). Consequently, existing Meltdown mitigations focus on restricting the attacker's point of view, e.g., placing victim data out of reach [20], flushing buffers after victim execution [25, 29], or zeroing unauthorized data flow directly at the silicon level [28].

Given the widespread deployment of Meltdown countermeasures, including changes in operating systems and CPUs, we ask the following fundamental questions in this paper:

Can Meltdown-type effects only be used for leakage or also for injection? Would current hardware and software defenses suffice to fully eradicate Meltdown-type threats based on illegal data flow from faulting or assisted instructions?

A. Our Results and Contributions

In this paper, we introduce an innovative class of Load Value Injection (LVI) attack techniques. Our key contribution is to recognize that, under certain adversarial conditions, unintended microarchitectural leakage can also be inverted to *inject* incorrect data into the victim's transient execution. Being essentially a "reverse Meltdown"-type attack, LVI abuses that a faulting or assisted load instruction executed within a victim domain does not always yield the expected result, but may instead transiently forward dummy values or (attacker-controlled) data from various microarchitectural buffers. We consider attackers that can either directly or indirectly induce page faults or microcode assists during victim execution. LVI provides such attackers with a primitive to force a *legitimate* victim execution to transiently compute on "poisoned" data (e.g., pointers, array indices) before the CPU eventually detects the fault condition and discards the pending architectural state changes. Much like in Spectre attacks, LVI relies on "confused deputy" code gadgets surrounding the faulting or assisted load in the victim to hijack transient control flow and disclose information. We are the first to combine Meltdown-style microarchitectural data leakage with Spectre-style code

TABLE I. Characterization of known side-channel and transient-execution attacks in terms of targeted microarchitectural predictor or data buffer (vertical axis) vs. leakage- or injection-based methodology (horizontal axis). The LVI attack plane, first explored in this paper, is indicated on the lower right and applies an injection-based methodology known from Spectre attacks (upper right) to reversely exploit Meltdown-type data leakage (lower left).

μ-Arch Buffer / Methodology		Leakage	Injection
Prediction history	PHT	BranchScope [15], Bluethunder [24]	Spectre-PHT [38]
	BTB	SBPA [1], BranchShadow [40]	Spectre-BTB [38]
	RSB	Hyper-Channel [8]	Spectre-RSB [39, 44]
	STL	—	Spectre-STL [23]
Program data	L1D	Meltdown [42]	LVI-NULL
	L1D	Foreshadow [61]	LVI-L1D
	FPU	LazyFP [57]	LVI-FPU
	SB	Fallout [9]	LVI-SB
	LFB/LP	ZombieLoad [53], RIDL [67]	LVI-LFB/LP

gadget abuse to compose a novel type of transient load value injection attacks.

Table I summarizes how Spectre [38] first applied an injection-based methodology to invert prior branch prediction side-channel attacks, whereas LVI similarly shows that recent Meltdown-type microarchitectural data leakage can be reversely exploited. Looking at Table I, it becomes apparent that Spectre-style injection attacks have so far only been applied to auxiliary history-based branch prediction and dependency prediction buffers that accumulate program metadata to steer the victim's transient execution indirectly. Our techniques, on the other hand, intervene much more directly in the victim's transient data stream by injecting erroneous load values straight from the CPU's memory hierarchy, *i.e.*, intermediate load and store buffers and caches.

These fundamentally different microarchitectural behaviors (*i.e.*, misprediction vs. illegal data flow) also entail that LVI requires defenses that are orthogonal and complementary to existing Spectre mitigations. Indeed, we show that some of our exploits can transiently redirect conditional branches, even *after* the CPU's speculation machinery correctly predicted the architectural branch outcome. Furthermore, since LVI attacks proceed entirely *within* the victim domain, they remain intrinsically immune to widely deployed software and microcode Meltdown mitigations that flush microarchitectural resources after victim execution [25, 29]. Disturbingly, our analysis reveals that even state-of-the-art hardened Intel CPUs [28], with silicon changes that zero out illegal data flow from faulting or assisted instructions, do not fully eradicate LVI-based threats.

Our findings challenge prior views that, unlike Spectre, Meltdown-type threats could be eradicated straightforwardly at the operating system or hardware levels [10, 18, 22, 45, 72]. Instead, we conclude that potentially every illegal data flow in the microarchitecture can be inverted as an injection source to purposefully disrupt the victim's transient behavior. This observation has profound consequences for reasoning about secure code. We argue that depending on the attacker's capabilities, ultimately, *every* load operation in the victim may potentially serve as an exploitable LVI gadget. This is in sharp contrast to prior Spectre-type effects that are contained around clear-cut (branch) misprediction locations.

Successfully exploiting LVI requires the ability to induce page faults or microcode assists during victim execution. We show that this requirement can be most easily met in Intel SGX environments, where we develop several proof-of-concept attacks that abuse dangerous real-world gadgets to arbitrarily divert transient control flow in the enclave. We furthermore mount a novel transient fault attack on AES-NI to extract full cryptographic keys from a victim enclave. While LVI attacks in non-SGX environments are generally much harder to mount, we consider none of the adversarial conditions for LVI to be unique to Intel SGX. We explore consequences for traditional process isolation by showing that, given a suitable LVI gadget and a faulting or assisted load in the kernel, arbitrary supervisor memory may leak to user space. We also show that the same vector could be exploited in a cross-process LVI attack.

Underlining the impact and the practical challenges arising from our findings, Intel plans to mitigate LVI by extensive revisions at the compiler and assembler levels to allow at least compilation of SGX enclaves to remain secure on LVI-vulnerable systems. Particularly, fully mitigating LVI requires introducing `lfence` instructions to serialize the processor pipeline after possibly *every* memory load operation. Additionally, certain instructions featuring implicit loads, including the pervasive x86 `ret` instruction, should be blacklisted and emulated with equivalent serialized instruction sequences. We observe extensive performance overheads of factor 2 to 19 for our evaluation of prototype compiler mitigations, depending on the application and whether `lfences` were inserted by an optimized compiler pass or through a naive post-compilation assembler approach.

In summary, our main contributions are as follows:

- We show that Meltdown-type data leakage can be inverted into a Spectre-like Load Value Injection (LVI) primitive. LVI transiently hijacks data flow, and thus control flow.
- We present an extensible taxonomy of LVI-based attacks.
- We show the insufficiency of silicon changes in the latest generation of acclaimed Meltdown-resistant Intel CPUs
- We develop practical proof-of-concept exploits against Intel SGX enclaves, and we discuss implications for traditional kernel and process isolation in the presence of suitable LVI gadgets and faulting or assisted loads.
- We evaluate compiler mitigations and show that a full mitigation incurs a runtime overhead of factor 2 to 19.

B. Responsible Disclosure and Impact

We responsibly disclosed LVI to Intel on April 4, 2019. We also described the non-Intel-specific parts to ARM and IBM. To develop and deploy appropriate countermeasures, Intel insisted on a long embargo period for LVI, namely, until March 10, 2020 (CVE-2020-0551, Intel-SA-00334). Intel considers LVI particularly severe for SGX and provides a compiler and assembler-based full mitigation for enclave programs, described and evaluated in Section IX. Intel furthermore acknowledged

P	RW	US	WT	UC	A	D	S	G	Physical Page Number	Rsvd.	XD

Fig. 1. Overview of an x86 page-table entry and attributes that may trigger architectural page fault exceptions (red bold) or microcode assists (green italic). Attributes that are periodically cleared by some OS kernels are underlined; all other fields can only be modified by privileged attackers.

Fig. 2. Overview of the memory hierarchy in modern x86 microarchitectures.

that LVI may in principle be exploited in non-SGX user-to-kernel or process-to-process environments and suggested addressing by manually patching any such exploitable gadgets upon discovery.

We also contacted Microsoft, who acknowledged the relevance when paging out kernel memory and continues to investigate the applicability of LVI to the Windows kernel. Microsoft likewise suggested addressing non-SGX scenarios by manually patching any exploitable gadgets upon discovery.

II. BACKGROUND

A. CPU Microarchitecture

In a complex instruction set architecture (ISA) such as Intel x86 [31] instructions are decoded into RISC-like *micro-ops*. The CPU executes micro-ops from the reorder buffer out of order when their operands become available but retires micro-ops in order. Modern CPUs perform history-based speculation to predict branches and data dependencies ahead of time. While the CPU implements the most common fast-path logic directly in hardware, certain corner cases are handled by issuing a *microcode assist* [13, 17]. In such a corner case, the CPU flags the corresponding micro-op to be re-issued later as a microcode routine. When encountering exceptions, misspeculations, or microcode assists, the CPU pipeline is flushed, and any outstanding micro-op results are discarded from the reorder buffer. This rollback ensures that the results of unintended *transient instructions*, which were wrongly executed ahead of time, are never visible at the architectural level.

Address translation: Modern CPUs use virtual addresses to isolate concurrently running tasks. A multi-level page-table hierarchy is set up by the operating system (OS) or hypervisor to translate virtual to physical addresses. The lower 12 address bits are the index into a 4 KB page, while higher address bits index a series of page-table entries (PTEs) that ultimately yield the corresponding physical page number (PPN). Figure 1 overviews the layout of an Intel x86 PTE [13, 32]. Apart from the physical page number, PTEs also specify permission bits to indicate whether the page is present, accessible to user space, writable, or executable.

The translation lookaside buffer (TLB) caches recent address translations. Upon a TLB miss, the CPU's page-miss handler performs a page-table walk and updates the TLB. The CPU's TLB miss handler circuitry is optimized for the fast path, and delegates more complex operations, e.g., setting of "accessed" and "dirty" PTE bits, using microcode assists [17]. Depending on the permission bits, a page fault (#PF) may be raised to abort the memory operation and redirect control to the OS.
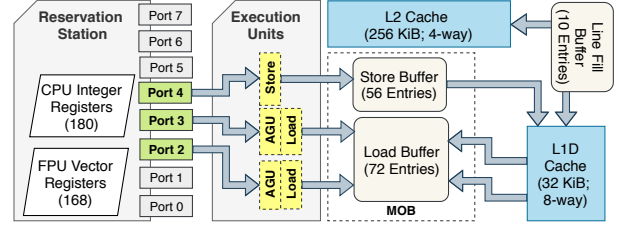
Memory hierarchy: Superscalar CPUs consist of multiple physical cores connected through a bus interconnect to the memory controller. As the main memory is relatively slow, the CPU uses a complex memory subsystem (cf. Figure 2), including various caches and buffers. On Intel CPUs, the L1 cache is the fastest and smallest, closest to the CPU, and split into a separate unit for data (L1D) and instructions (L1I). L1D is usually a 32 KB 8-way set-associative cache. It is virtually-indexed and physically-tagged, such that lookups can proceed in parallel to address translation. A cache line is 64 bytes, which also defines the granularity of memory transactions (load and store) through the cache hierarchy. To handle various sized memory operations, L1D is connected to a memory-order buffer (MOB), which is interfaced with the CPU's register files and execution units through dedicated load ports (LPs).

The MOB includes a store buffer (SB) and load buffer (LB), plus various dependency prediction and resolution circuits to safeguard correct ordering of memory operations. The SB keeps track of outstanding store data and addresses to commit stores in order, without stalling the pipeline. When a load entry in LB is predicted to not depend on any prior store, it is executed out of order. If a store-to-load (STL) dependency is detected, the SB forwards the stored data to the dependent load. However, if the dependency of a load and preceding stores is not predicted correctly, these optimizations may lead to situations where the load consumes either stale data from the cache or wrong data from the SB while the CPU reissues the load to obtain the correct data. These optimizations within the MOB can undermine security [9, 23, 35].

Upon on L1D cache miss, data is fetched from higher levels in the memory hierarchy via the line-fill buffer (LFB), which keeps track of outstanding load and store requests without blocking the L1D cache. The LFB retrieves data from the next cache levels or main memory and afterward updates the corresponding cache line in L1D. An "LFB hit" occurs if the CPU has a cache miss for data in a cache line that is in the LFB. Furthermore, uncacheable memory and non-temporal stores bypass the cache hierarchy using the LFB.

B. Intel SGX

Intel Software Guard Extensions (SGX) [13] provides processor-level isolation and attestation for secure "enclaves" in the presence of an untrusted OS. Enclaves are contained in the virtual address space of a conventional user-space process, and virtual-to-physical address mappings are left under explicit

control of untrusted system software. To protect against active address remapping attackers [13], SGX maintains a shadow entry for every valid enclave page in the enclave page-cache map (EPCM) containing amongst others the expected virtual address. Valid address mappings are cached in the TLB, which is flushed upon enclave entry, and a special EPCM page fault is generated when encountering an illegal virtual-to-physical mapping (cf. Appendix A).

However, previous work showed that Intel SGX root attackers can mount high-resolution, low-noise side-channel attacks through the cache [7, 46, 52], branch predictors [15, 24, 40], page-table accesses [63, 65, 71], or interrupt timing [64]. In response to recent transient-execution attacks [11, 53, 61, 67], which can extract enclave secrets from side-channel resistant software, Intel released microcode updates which flush microarchitectural buffers on every enclave entry and exit [25, 29].

C. Transient-Execution Attacks

Modern processors safeguard architectural consistency by discarding the results of any outstanding transient instructions when flushing the pipeline. However, recent research on transient-execution attacks [38, 42, 61] revealed that these unintended transient computations may leave secret-dependent traces in the CPU's microarchitectural state, which can be subsequently recovered through side-channel analysis. Following a recent classification [10], we refer to attacks exploiting misprediction [23, 37, 38, 39, 44] as Spectre-type, and attacks exploiting transient execution after a fault or microcode assist [9, 42, 53, 57, 61, 67] as Meltdown-type.

Meltdown-type attacks extract unauthorized program data across architectural isolation boundaries. Over the past years, faulting loads with different exception types and microcode assists have been demonstrated to leak secrets from intermediate microarchitectural buffers in the memory hierarchy: the L1 data cache [42, 61, 70], the line-fill buffer and load ports [53, 67], the FPU register file [57], and the store buffer [9, 51].

A perpendicular line of Spectre-type attacks, on the other hand, aims to steer transient execution in the victim domain by poisoning various microarchitectural predictors. Spectre attacks are limited by the depth of the transient-execution window, which is ultimately bounded by the size of the reorder buffer [68]. Most Spectre variants [38, 39, 44] hijack the victim's transient control flow by mistraining shared branch prediction history buffers prior to entering the victim domain. Yet, not all Spectre attacks depend on branch history, e.g., in Spectre-STL [23] the processor's memory disambiguation predictor incorrectly speculates that a load does not depend on a prior store, allowing the load to transiently execute with a stale outdated value. Spectre-STL has for instance been abused to hijack the victim's transient control flow in case the stale value is a function pointer or indirect branch target controlled by a previous attacker input [68].

III. LOAD VALUE INJECTION

Table I summarizes the existing transient-execution attack landscape. The Spectre family of attacks (upper right) con-
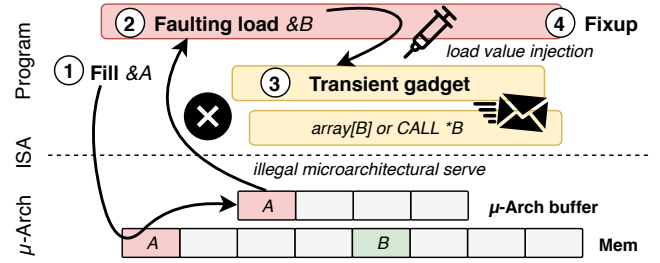

Fig. 3. Phases in a Load Value Injection (LVI) attack: (1) a microarchitectural buffer is filled with value A; (2) the victim executes a faulting or assisted load to retrieve value B which is incorrectly served from the microarchitectural buffer; (3) the injected value A is forwarded to transient instructions following the faulting or assisted load, which may now perform unintended operations depending on the available gadgets; (4) the CPU flushes the faulting or assisted load together with all other transient instructions.

tributed an injection-based methodology to invert prior prediction history side-channels (upper left) by abusing confused-deputy code gadgets within the victim domain. At the same time, Meltdown-type attacks (lower left) demonstrated cross-domain data leakage. The LVI attack plane (lower right) remains unexplored until now. In this paper, we adopt an injection-based methodology known from Spectre attacks to reversely exploit Meltdown-type microarchitectural data leakage. LVI brings a significant change in the threat model, similar to switching from branch history side-channels to Spectre-type attacks. Crucially, LVI has the potential to replace the outcome of *any* victim load, including implicit load micro-ops like in the x86 `ret` instruction, with attacker-controlled data. This is in sharp contrast to Spectre-type attacks, which can only replace the outcomes of branches and store-to-load dependencies by poisoning execution metadata accumulated in various microarchitectural predictors.

A. Attack Overview

We now outline how LVI can hijack the result of a trusted memory load operation, under the assumption that attackers can provoke page faults or microcode assists for (arbitrary) load operations in the victim domain. The attacker's goal is to force a victim to transiently compute on unintended data, other than the expected value in trusted memory. Injecting such unexpected load values forces a victim to transiently execute gadget code immediately following the faulting or assisted load instruction with unintended operands.

Figure 3 overviews how LVI exploitation can be abstractly broken down into four phases.

1) In the first phase, the microarchitecture is optionally prepared in the desired state by filling a hidden buffer with an (attacker-controlled) value A.
2) The victim then executes a load micro-op to fetch a trusted value B. However, in case this instruction suffers a page fault or microcode assist, the CPU may erroneously serve the load request from the microarchitectural buffer. This results in incorrect forwarding of value A to dependent transient micro-ops following the faulting or assisted load. At this point, the attacker has succeeded in tricking the

victim into transiently computing on the injected value A instead of the trusted value B.

3) These unintended transient computations may subsequently expose victim secrets through microarchitectural state changes. Depending on the specific "gadget" code surrounding the original load operation, LVI may either encode secrets directly or serve as a transient control or data flow redirection primitive to facilitate second-stage gadget abuse, e.g., when B is a trusted code or data pointer.

4) The architectural results of gadget computations are eventually discarded at the retirement of the faulting or assisted load instruction. However, secret-dependent traces may have been left in the CPU's microarchitectural state, which can be subsequently recovered through side-channel analysis.

B. A Toy Example

Listing 1 provides a toy LVI gadget to illustrate how faulting loads in a victim domain may trigger incorrect transient forwarding. Our example gadget bears a high resemblance to known Spectre gadgets but notably does *not* rely on branch misprediction or memory disambiguation. Furthermore, our gadget executes entirely within the victim domain and is hence not affected by widely deployed microcode mitigations that flush microarchitectural buffers on context switch. Regardless of the prevalence of this specific toy gadget, it serves as an initial example which is easy to understand and illustrates the power of LVI as a generic attack primitive.

Following the general outline of Figure 3, the gadget code in Listing 1 first copies a 64-bit value `untrusted_arg` provided by the attacker into trusted memory (e.g., onto the stack) at line 2. In the example, the argument copy is further not used, and this store operation merely serves to bring some attacker-controlled value into some microarchitectural buffer. Subsequently, in the second phase of the attack, a pointer-to-pointer `trusted_ptr` (e.g., a pointer in a dynamically allocated struct) is dereferenced at line 3. We assume that, upon the first-level pointer dereference, the victim suffers a page fault or microcode assist. The faulting load causes the processor to incorrectly forward the attacker's value `untrusted_arg` that was previously brought into the store buffer by the completely unrelated store at line 2, like in a Meltdown-type attack [9]. At this point, the attacker has succeeded in replacing the architecturally intended value at address `*trusted_ptr` with her own chosen value. In the third phase of the attack, the gadget code transiently uses `untrusted_arg` as the base address for a second-level pointer dereference and uses the result as an index in a lookup table. Similar to a Spectre gadget [38], the lookup in `array` serves as the sending end of a cache-based side-channel, allowing to encode arbitrary memory locations within the victim's address space.

Figure 4 illustrates how in the final phase of the attack, after the fault has been handled and the load has been re-issued allowing the victim to complete, adversaries can abuse access timings to the probing array to reconstruct secrets from the victim's transient execution. Notably, the timing diagram showcases two clear drops: one dip corresponds to
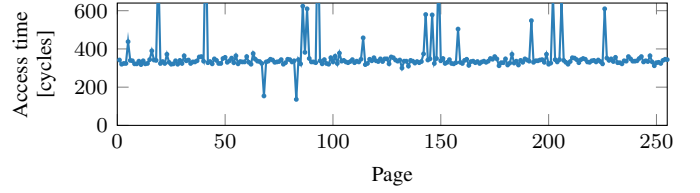

Fig. 4. Access times to the probing array after the execution of Listing 1. The dip at 68 (`'D'`) is the transmission specified by the victim's architectural program semantics. The dip at 83 (`'S'`) is the victim secret at the address `untrusted_arg` injected by the attacker.

```
1  void call_victim(size_t untrusted_arg) {
2      *arg_copy = untrusted_arg;
3      array[**trusted_ptr * 4096];
4  }
```

Listing 1: An LVI toy gadget for leaking arbitrary data from a victim domain.

the architecturally intended value that was processed after the faulting load got successfully re-issued, while the second dip corresponds to the victim secret at the address chosen by the attacker. This toy example hence serves as a clear illustration of the danger of incorrect transient forwarding following a faulting load in a victim domain. We elaborate further on attacker assumptions and gadget requirements for different LVI variants in Sections IV and VI respectively.

C. Difference with Spectre-type Attacks

While LVI adopts a gadget-based exploitation methodology known from Spectre-type attacks, both attack families exploit fundamentally different microarchitectural behaviors (*i.e.*, incorrect transient forwarding vs. misprediction). We explain below how LVI is different from and requires orthogonal mitigations to known Spectre variants.

a) LVI vs. branch prediction: Most Spectre variants [10, 38, 39, 44] transiently hijack branch outcomes in a victim process by poisoning various microarchitectural branch prediction history buffers. On recent and updated systems, these buffers are typically not simultaneously shared anymore and flushed on context switch. Furthermore, to foil mistraining strategies within a victim domain, hardened compilers insert explicit `lfence` barriers after potentially mispredicted branches.

In contrast, LVI allows to hijack the result of *any* victim load micro-op, not just branch targets. By directly injecting incorrect values from the memory hierarchy, LVI allows data-only attacks as well as control-flow redirection in the transient domain. Essentially, LVI and Spectre exploit different subsequent phases of the victim's transient execution: while Spectre hijacks control flow *before* the architectural branch outcome is known, LVI-based control-flow redirection manifests only *after* the victim attempts to fetch the branch-target address from application memory. LVI does not rely on mistraining of any (branch) predictor, and hence, applies even to CPUs without exploitable prediction elements, and to systems protected with up-to-date microcode and compiler mitigations.

b) LVI vs. speculative store bypass: Spectre-STL [23] exploits the memory disambiguation predictor, which may speculatively issue a load even before all prior store addresses are known. That is, in case a load is mispredicted to not depend on a prior store, the store is incorrectly not forwarded and the load transiently executes with a stale outdated value.

Crucially, while Spectre-STL is strictly limited to injecting stale values for loads that closely follow a store to the exact same address, LVI has the potential to replace the result of *any* victim load with unrelated and possibly attacker-controlled data. LVI therefore drastically widens the spectrum of incorrect transient paths. As an example, the code in Listing 1 is not in any way exposed to Spectre-STL since the store and load operations are to different addresses, but this gadget can still be exploited with LVI in case the load suffers a page fault or microcode assist. Consequently, LVI is also not affected by Spectre-STL mitigations, which disable the memory disambiguation predictor in microcode or hardware.

c) LVI vs. value prediction: While value prediction has already been proposed more than two decades ago [41, 69], commercial CPUs do not implement it yet due to complexity concerns [49]. As long as no commercial CPU supports value speculation, Spectre-type value misprediction attacks are purely theoretical. In LVI, there is no mistraining of any (value) predictor, and hence, it applies to today's CPUs already.

IV. ATTACKER MODEL AND ASSUMPTIONS

We focus on software adversaries who want to disclose secrets from an isolated victim domain, e.g., the OS kernel, another process, or an SGX enclave. For SGX, we assume an attacker with root privileges, *i.e.*, the OS is under control of the attacker [13]. Successful LVI attacks require carefully crafted adversarial conditions. In particular, we identify the following three requirements for LVI exploitability:

a) Incorrect transient forwarding: As with any fault injection attack, LVI requires some form of exploitable incorrect behavior. We exploit that faulting or assisted loads do not always yield the expected architectural result, but may transiently serve dummy values or poisoned data from various microarchitectural buffers. There are many instances of incorrect transient forwarding in modern CPUs [9, 10, 42, 53, 57, 61, 67]. In this work, we show that such incorrect transient forwarding is *not* limited to cross-domain data leakage. We are the first to show cross-domain data injection *and* identify dummy 0x00 values as an exploitable incorrect transient forwarding source, thereby widening the scope of LVI even to microarchitectures that were previously considered Meltdown-resistant.

b) Faulting or assisted loads: LVI requires firstly the ability to (directly or indirectly) provoke architectural exceptions or microcode assists for legitimate loads executed by the victim. This includes *implicit* load micro-ops as part of larger ISA instructions, e.g., popping the return address from the stack in the x86 ret instruction. Privileged SGX attackers can straightforwardly provoke page faults for enclave memory loads by modifying untrusted page tables, as demonstrated by prior research [65, 71]. Even unprivileged attackers can induce demand paging non-present faults by abusing the OS interface to unmap targeted victim pages through legacy interfaces or contention of the shared page cache [19]. Finally, more recent works showed that Meltdown-type effects are *not* limited to architectural exceptions, but also exist for assisted loads [9, 53, 67]. In case a microcode assist is required, the load micro-op does not architecturally commit, but may still transiently forward incorrect values before being re-issued as a microcode routine. Microcode assists occur in a wide variety of conditions, including subnormal floating point numbers and setting of "accessed" and "dirty" PTE bits [13, 29].

c) Code gadgets: A final yet crucial requirement for LVI is the presence of a suitable code gadget that allows to hijack the victim's transient execution and encode unintended secrets in the microarchitectural state. In practice, this requirement comes down to identifying a load operation in the victim code that can be faulting or assisted, followed by an instruction sequence that redirects control or data flow based on the loaded value (e.g., a pointer, or array index). We find that there are many different types of gadgets which mostly consist of only a few ubiquitously used instructions. We provide practical instances of such exploitable gadgets in Section VI.

V. BUILDING BLOCKS OF THE ATTACK

We compose transient fault-injection attacks using the three building blocks described in the previous section and Figure 3.

A. Phase $\mathcal{P}1$: Microarchitectural Poisoning

The main challenge in the first phase is to prepare the CPU's microarchitectural state such that a (controlled) incorrect transient forwarding happens for the faulting load in the second stage. We later classify LVI variants based on the microarchitectural buffer that forwards the incorrect data. Depending on the variant, it suffices in this phase to fill a particular buffer (cf. Section II-A: L1D, LFB, SB, LP) with a chosen value at a chosen location. This is not always a requirement, as we also consider a special LVI-NULL variant that abuses incorrect forwarding of 0x00 dummy values which are often returned when faulting loads miss the cache, or on Meltdown-resistant microarchitectures [28]. Such null values are "hard wired" in the CPU, and the poisoning phase can hence be entirely omitted for LVI-NULL attacks.

In a straightforward scenario, the shared microarchitectural buffer can be poisoned directly from within the attacker context. This scenario assumes, however, that said buffer is *not* explicitly overwritten or flushed when switching from the attacker to the victim domain, which is often not anymore the case with recent software and microcode mitigations [25, 29]. Alternatively, for buffers competitively shared among logical CPUs, LVI attackers can resort to concurrent poisoning from a co-resident hyperthread running in parallel to the victim [53, 61, 67].

Finally, in the most versatile LVI scenario, the attack runs *entirely* within the victim domain without placing any assumptions on prior attacker execution or co-residence. We abuse appropriate "fill gadgets" preceding the faulting load within the victim execution. As explored in Section VI, LVI

variants may impose more or fewer restrictions on suitable fill gadget candidates. The most generically exploitable fill gadget loads or stores attacker-controlled data from or to an attacker-chosen location, without introducing any architectural security problem. This is a common case if attacker and victim share an address space (enclave, user-kernel boundary, sandbox) and exchange arguments or return values via pointer passing.

B. Phase $\mathcal{P}2$: Provoking Faulting or Assisted Loads

In the second and principal LVI phase, the victim executes a faulting or assisted load micro-op triggering incorrect transient forwarding. The crucial challenge here is to provoke a fault or assist for a *legitimate* and trusted load executed by the victim.

a) Intel SGX: When targeting Intel SGX enclaves, privileged adversaries can straightforwardly manipulate PTEs in the untrusted OS to provoke page-fault exceptions [71] or microcode assists [9, 53]. Even user-space SGX attackers can indirectly revoke permissions for enclave code and data pages through the unprivileged `mprotect` system call [61]. Alternatively, if the targeted LVI gadget requires a more precise temporal granularity, privileged SGX attackers can leverage a single-stepping interrupt attack framework like SGX-Step [63] to manipulate PTEs and revoke enclave-page permissions precisely at instruction-level granularity.

b) Generalization to other environments.: In the more general case of unprivileged cross-process, cross-VM, or sandboxed attackers, we investigated exploitation via memory contention. Depending on the underlying OS or hypervisor implementation and configuration, an attacker can forcefully evict selected virtual memory pages belonging to the victim via legacy interfaces or by increasing physical memory utilization [19]. The "present" bit of the associated PTE is cleared (cf. Figure 1), and the next victim access faults. On Windows, this can even affect the kernel heap due to demand paging [50].

Furthermore, prior research has shown that the page-replacement algorithm on Windows periodically clears "accessed" and "dirty" PTE bits [53]. Hence, unprivileged attackers can simply wait until the OS clears the accessed bit on the victim PTE. Upon the next access to that page, the CPU's page-miss handler circuitry prematurely aborts the victim's load micro-op to issue a microcode assist for re-setting the accessed bit on the victim PTE [13, 53]. Finally, even without any OS intervention, a victim program may expose certain load gadget instructions that always require a microcode assist (e.g., split-cacheline accesses which have been abused to leak data from load ports [66, 67]).

C. Phase $\mathcal{P}3$: Gadget-Based Secret Transmission

The key challenge in the third LVI phase is to identify an exploitable code "gadget" exhibiting incorrect transient behavior over poisoned data forwarded from a faulting load micro-op in the previous phase. In contrast to all prior Meltdown-type attacks, LVI attackers do *not* control the instructions surrounding the faulting load as the load runs entirely in the victim domain. We, therefore, propose a gadget-oriented exploitation methodology closely mirroring the classification from the Spectre world [10, 38].

a) Disclosure gadget: A first type of gadget, akin Spectre-PHT-style information disclosure, encodes victim secrets in the instructions immediately following the faulting load (cf. Listing 1). The gadget encodes secrets in conditional control flow or data accesses. Importantly, however, this gadget does *not* need to be secret-dependent. Hence, LVI can even target side-channel resistant constant-time code [16]. That is, at the architectural level, the victim code only dereferences known, non-confidential values when evaluating branch conditions or array indices. At the microarchitectural level, however, the faulting load in the second LVI phase causes the known value to be transiently replaced. As a result of this "transient remapping" primitive, the gadget instructions may now inadvertently leak secret values that were brought into the targeted microarchitectural buffer during prior victim execution.

b) Control-flow hijack gadget: A second and more powerful type of LVI gadgets, mirroring Spectre-BTB-style branch-target injection, exploits indirect branches in the victim code. In this case, the attacker's goal is not to disclose forwarded values, but instead to abuse them as a transient control-flow hijacking primitive. That is, when dereferencing a function pointer (`call`, `jmp`) or loading a return address from the stack (`ret`), the faulting load micro-op in the victim code may incorrectly pick up attacker-controlled values from the poisoned microarchitectural buffer. This essentially enables the attacker to arbitrarily redirect the victim's transient control flow to selected second-stage code gadgets found in the victim address space. Adopting established techniques from jump-oriented [5] and return-oriented programming (ROP) [56], second-stage gadgets can further be chained together to compose arbitrary transient instruction sequences. Akin traditional memory-safety exploits, attackers may also leverage "stack pivoting" techniques to transiently point the victim stack to an attacker-controlled memory region.

Although they share similar goals and exploitation methodologies, LVI-based control-flow hijacking should be regarded as a *complementary* threat compared to Spectre-style branch-target injection. Indeed, LVI only manifests *after* the victim attempts to fetch the architectural branch target, whereas Spectre abuses speculative execution *before* the actual branch outcome is determined. Hence, the CPU may first (correctly or incorrectly) predict transient control flow based on the history accumulated in the BTB and RSB, until the victim execution later attempts to verify the speculation by comparing the actual branch-target address loaded from application memory. At this point, LVI kicks in since the faulting load micro-op yields an incorrect attacker-controlled value and erroneously redirects the transient instruction stream to a poisoned branch-target address.

LVI-based control-flow hijack gadgets can be as little as a single x86 `ret` instruction, making this case extremely dangerous. As explained in Section IX, fully mitigating LVI requires blacklisting all indirect branch instructions and emulating them with equivalent serialized instruction sequences.

c) Widening the transient window: A final challenge is that, unlike traditional fault-injection attacks that cause persistent bit flips at the architectural level [36, 47, 59], LVI

attackers can only disturb victim computations for a limited time interval before the CPU eventually catches up, detects the fault, and aborts transient execution. This implies that there is only a limited "transient window" in which the victim inadvertently computes on the poisoned load values, and all required gadget instructions need to complete within this window to transmit secrets. The transient window is ultimately bounded by the size of the processor's reorder buffer [68].

Naturally, widening the transient window is a requirement common to all transient-execution attacks. Therefore, we can leverage techniques known from prior Spectre attacks [11, 39, 44]. Common techniques include, e.g., flushing selected victim addresses or PTEs from the CPU cache.

    d) Summary: To summarize, we construct LVI attacks with the three phases $\mathcal{P}1$ (poisoning), $\mathcal{P}2$ (provoking injection), $\mathcal{P}3$ (transmission). For each of the phases, we have different instantiations, based on the specific environment, hardware, and attacker capabilities. We now discuss gadgets in Section VI and, subsequently, practical LVI attacks on SGX in Section VII.

VI. LVI Taxonomy and Gadget Exploitation

We want to emphasize that LVI represents an entirely new *class* of attack techniques. Building on the (extended) transient-execution attack taxonomy by Canella et al. [10], we propose an unambiguous naming scheme and multi-level classification tree to reason about and distinguish LVI variants in Appendix B.

In the following, we overview the leaves of our classification tree by introducing the main LVI variants exploiting different microarchitectural injection sources (cf. Table I). Given the particular relevance of LVI to Intel SGX, we especially focus on enclave adversaries but also include a discussion on gadget requirements and potential applicability to other environments.

A. LVI-L1D: L1 Data Cache Injection

In this section, we contribute an innovative "reverse Fore-shadow" injection-based exploitation methodology for SGX attackers. Essentially, LVI-L1D can best be regarded as a *transient page-remapping* primitive allowing to arbitrarily replace the outcome of *any* legitimate enclave load value (e.g., a return address on the stack) with any data currently residing in the L1D cache and sharing the same virtual page offset.

    a) Microarchitectural poisoning: An "L1 terminal fault" (L1TF) occurs when the CPU prematurely early-outs address translation when a PTE has the present bit cleared or a reserved bit set [61, 70]. A special type of L1TF may also occur for SGX EPCM page faults (cf. Appendix A) if the untrusted PTE contains a rogue physical page number [25, 61]. In our LVI-L1D attack, the root attacker replaces the PPN field in the targeted untrusted PTE, before entering or resuming the victim enclave. If the enclave dereferences the targeted location, SGX raises an EPCM page fault. However, before the fault is architecturally raised, the poisoned PPN is sent to the L1D cache. If a cache hit occurs at the rogue physical address (composed of the poisoned PPN and the page offset specified by the load operation), illegal values are "injected" into the victim's transient data stream.
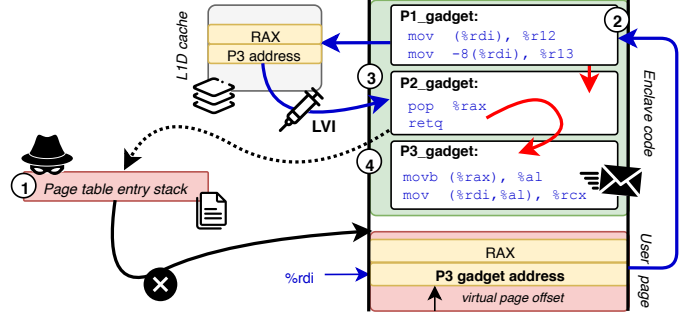

Fig. 5. Transient control-flow hijacking using LVI-L1D: (1) the enclave's stack PTE is remapped to a user page outside the enclave; (2) a $\mathcal{P}1$ gadget inside the enclave loads attacker-controlled data into L1D; (3) a $\mathcal{P}2$ gadget pops trusted data (return address) from the enclave stack, leading to faulting loads which are transiently served with poisoned data from L1D; (4) the enclave's transient execution continues at an attacker-chosen $\mathcal{P}3$ gadget encoding arbitrary secrets in the microarchitectural CPU state.

    b) Gadget requirements: LVI-L1D works on processors vulnerable to Foreshadow, but with patched microcode, *i.e.*, not on more recent silicon-resistant CPUs [25]. The $\mathcal{P}1$ gadget, a load or store, brings secrets or attacker-controlled data into the L1D cache. The $\mathcal{P}2$ gadget is a faulting or assisted memory load. The $\mathcal{P}3$ gadget creates a side-channel from the transient domain, or it redirects control flow based on the injected data (e.g., x86 `call` or `ret`), ultimately also leading to the execution of an attacker-chosen $\mathcal{P}3$ gadget. The addresses in both memory operations must have the same page offset (*i.e.*, lowest 12 virtual address bits). This is not a limiting factor since L1D can hold 32 KiB of data, allowing the three gadgets ($\mathcal{P}1$, $\mathcal{P}2$, $\mathcal{P}3$) to be far apart in the enclaved execution. Similar to architectural memory-safety SGX attacks [62], we found that high degrees of attacker control are often provided by enclave entry and exit code gadgets copying user data to or from chosen addresses outside the enclave.

Current microcode flushes L1D on enclave entry and exit, and hyperthreading is recommended to be disabled [25]. We empirically confirmed that if hyperthreading is enabled, no $\mathcal{P}1$ gadget is required and that on outdated microcode, L1D can trivially be poisoned before enclave entry.

    c) Gadget exploitation: Figure 5 illustrates LVI-L1D hijacking return control flow in a minimal enclave. First, the attacker uses a page fault controlled-channel [71] or SGX-Step [63] to precisely advance the enclaved execution to right before the desired $\mathcal{P}1$ gadget. Next, the attacker sets up the malicious memory mapping ① by changing the PPN of the enclave stack page to a user-controlled page. The enclave then executes a $\mathcal{P}1$ gadget ② accessing the user page and loading attacker-controlled data into the L1D cache (e.g., when invoking `memcpy` to copy parameters into the enclave). Next, the enclave executes the $\mathcal{P}2$ gadget ③ which pops some data plus a return address from the enclave stack. For address resolution, the CPU first walks the untrusted page tables leading to the rogue PPN to be forwarded to L1D. Since the prior $\mathcal{P}1$ gadget ensured that data is indeed present in L1D at the required address, a cache hit occurs, and the poisoned data (including the return

address) is served to the dependent transient micro-ops. Now, execution transiently continues at the attacker-chosen $\mathcal{P}3$ gadget ④ residing at an arbitrary location inside the enclave. The $\mathcal{P}3$ gadget encodes arbitrary secrets into the microarchitectural state before the CPU resolves the EPCM memory accesses, unrolls transient execution, and raises a page fault.

Note that for clarity, we focused on hijacking `ret` control flow in the above example, but we also demonstrated successful LVI attacks for `jmp` and `call` indirect control-flow instructions. We observe that large or repeated $\mathcal{P}1$ loads enable attackers to setup a fake "transient stack" in L1D to repeatedly inject illegal values for consecutive enclave stack loads (`pop-ret` sequences). Much like in architectural ROP code re-use attacks [56], we experimentally confirmed that attackers may chain together multiple $\mathcal{P}3$ gadgets to compose arbitrary transient computations. LVI attackers are only limited by the size of the transient window (cf. Section V-C).

d) Applicability to non-SGX environments: We carefully considered whether cross-process or virtual machine Foreshadow variants [70] may also be reversely exploited through an injection-based LVI methodology. However, we concluded that these variants are already properly prevented by the recommended PTE inversion [12] countermeasure, which has been widely deployed in all major OSs (cf. Appendix B).

B. LVI-SB, LVI-LFB, and LVI-LP: Buffer and Port Injection

LVI-SB applies an injection-based methodology to reversely exploit store buffer leakage. The recent Fallout [9] attack revealed how faulting or assisted loads can pick up SB data if the page offset of the load (least-significant 12 virtual address bits) matches with that of a recent outstanding store. Similarly, LVI-LFB and LVI-LP inject from the line-fill buffer and load ports, respectively, which were exploited for data leakage in the recent RIDL [67] and ZombieLoad [53] attacks.

a) Gadget requirements: In response to Fallout, RIDL, and ZombieLoad, recent Intel microcode updates now overwrite SB, LFB, and LP entries on every enclave and process context switch [29]. Hence, to reversely exploit SB, LFP, or LP leakage, we first require a $\mathcal{P}1$ gadget to bring interesting data (e.g., secrets or attacker-controlled addresses) into the appropriate buffer. Next, we need a $\mathcal{P}2$ gadget consisting of a trusted load operation which can be faulted or assisted, followed by a $\mathcal{P}3$ gadget creating a side-channel for data transmission or control flow redirection. For LVI-SB, we further require that the store and load addresses in $\mathcal{P}1$ and $\mathcal{P}2$ share the same page offset and are sufficiently close, such that the injected data in $\mathcal{P}1$ has not yet been drained from the store buffer. Alternatively, for LVI-LFB and LVI-LP, attackers may resort to injecting poisoned data from a sibling logical core, as LFB and LP are competitively shared between hyperthreads [29, 53].

b) Gadget exploitation: We found that LVI-SB can be a particularly powerful primitive, given the prevalence of store operations closely followed by a return or indirect call. We illustrate this point in Listing 2 with trusted proxy bridge code that is automatically generated by Intel's `edger8r` tool of the official SGX-SDK [30]. The `edger8r`-generated bridge

```
1  ; %rbx: user-controlled argument ptr (outside enclave)
2  sgx_my_sum_bridge:
3   ...
4   call my_sum          ; compute 0x10(%rbx) + 0x8(%rbx)
5   mov %rax,(%rbx)       ; P1: store sum to user address
6   xor %eax,%eax
7   pop %rbx
8   ret                   ; P2: load from trusted stack
```

Listing 2: Intel `edger8r`-generated code snippet with LVI-SB gadget.

code is responsible for transparently verifying and copying user arguments to and from enclave memory. The omitted code verifies that the untrusted argument pointer, which is also used to pass the result, lies outside the enclave [62].

An attacker can interrupt the enclave after line 4, clear the supervisor or accessed bit for the enclave stack, and resume the enclave. As the `edger8r` bridge code solely verifies that the attacker-provided argument pointer lies outside the enclave, it provides the attacker with full control over the lower 12 bits of the store address ($\mathcal{P}1$). When the enclave code returns at line 8, the control flow is redirected to the attacker-injected location, as the faulting or assisted `ret` ($\mathcal{P}2$) incorrectly picks up the value from the SB (which in this case is the sum of two attacker-provided arguments). Similar to LVI-L1D (Figure 5), an attacker can encode arbitrary enclave secrets by chaining together one or more $\mathcal{P}3$ gadgets in the victim enclave code.

Finally, note that LVI is *not* limited to control flow redirection as secrets may also be encoded directly in the data flow through a combined $\mathcal{P}2$-$\mathcal{P}3$ gadget (e.g., by means of a double-pointer dereference as illustrated in the toy example of Listing 1).

c) Applicability to non-SGX environments: Importantly, in contrast to LVI-L1D above, SB, LFB, and LP leakage does *not* necessarily require adversarial manipulation of PTEs, or rely on microarchitectural conditions that are specific to Intel SGX. Hence, given a suitable fault or assist primitive plus the required victim code gadgets, LVI-SB, LVI-LFB, and LVI-LP may be relevant for other contexts as well (cf. Section VIII).

C. LVI-NULL: 0x00 Dummy Injection

A highly interesting special case is LVI-NULL, which is based on the observation that known Meltdown-type attacks [42, 61] commonly report a strong bias to the value zero for faulting loads. We experimentally confirmed that the latest generation of acclaimed Meltdown-resistant Intel CPUs (`RDCL_NO` [28] from Whiskey Lake onwards) merely zero-out the results of faulting load micro-ops while still passing a dummy `0x00` value to dependent transient instructions. While this nulling strategy indeed suffices to prevent Meltdown-type data leakage, we show that the ability to inject zero values in the victim's transient data stream can be dangerously exploitable. Hence, LVI-NULL reveals a fundamental shortcoming in current silicon-level mitigations, and ultimately requires more extensive changes in the way the CPU pipeline is organized.

a) Gadget requirements: Unlike the other LVI variants, LVI-NULL does not rely on any microarchitectural buffer to inject poisoned data, but instead directly abuses dummy `0x00` values injected from the CPU's silicon circuitry in the $\mathcal{P}1$ phase.
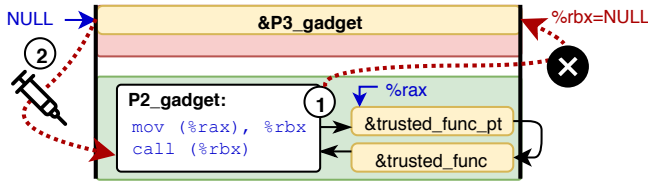
Fig. 6. Transient control-flow hijacking using LVI-NULL: (1) a $\mathcal{P}2$ gadget inside the enclave dereferences a function pointer-to-pointer, leading to a faulting load which forwards the dummy value null; (2) the following indirect call transiently dereferences the attacker-controlled null page outside the enclave, causing execution to continue at an attacker-chosen $\mathcal{P}3$ gadget address.

```
1  asm_oret: ; (linux-sgx/sdk/trts/linux/trts_pic.S#L454)
2  ...
3  mov     0x58(%rsp),%rbp       ; %rbp <- NULL
4  ...
5  mov     %rbp,%rsp             ; %rsp <- NULL
6  pop     %rbp                  ; %rbp <- *(NULL)
7  ret                           ; %rip <- *(NULL+8)
```

Listing 3: LVI-NULL stack hijack gadget in Intel SGX-SDK.

The $\mathcal{P}2$ gadget consists of a trusted load operation that can be faulted or assisted, followed by a $\mathcal{P}3$ gadget which, when operating on the unexpected value null, creates a side-channel for secret transmission or control-flow redirection.

In some scenarios, transiently replacing a trusted load micro-op with the unexpected value zero may directly lead to information disclosure, as explored in the AES-NI case study of Section VII-B. Moreover, LVI-NULL is especially dangerous in the common case of indirect pointer dereferences.

b) Gadget exploitation: While transiently computing on zero values might at first seem rather innocent, we make the key insight that zero can be regarded as a *valid* virtual address and that SGX root attackers can trivially map an arbitrary memory page at virtual address null. Using this technique, we contribute an innovative *transient null-pointer dereference* primitive that allows to hijack the result of *any* indirect pointer dereference in the victim enclave's transient domain.

We first consider the case of a data pointer stored in trusted memory, e.g., as a local variable on the stack. After revoking access rights on the respective enclave memory page, loading the pointer forces its value to zero, causing any following dereferences in the transient domain to read attacker-controlled data via the null page. This serves as a powerful "transient pointer-value hijacking" primitive to inject arbitrary data in a victim enclaved execution, which can be subsequently used in a $\mathcal{P}3$ gadget to disclose secrets or redirect control flow.

Figure 6 illustrates how the above technique can furthermore be exploited to arbitrarily hijack transient control flow in the case of function pointer-to-pointer dereferences, e.g., a function pointer in a heap object. The first dereference yields zero, and the actual function address is thereafter retrieved via the attacker-controlled null page. For the simpler case of single-level function pointers, we experimentally found that transient control flow cannot be directly redirected to the zero address outside the enclave, which is in line with architectural restrictions imposed by Intel SGX [13]. However, adversaries might load the relocatable enclave image at virtual address null. We, therefore, recommend that the first page is marked as non-executable or that a short infinite loop is included at the base of every enclave image to effectively "trap" any transient control flow redirections to virtual address null.

Finally, a special case is loading a stack pointer. Listing 3 shows a trusted code snippet from the Intel SGX-SDK [30] to restore the enclave execution context when returning from an untrusted function.[1] An attacker can interrupt the victim code right before line 3, and revoke access rights on the trusted stack page used by the enclave entry code. After resuming the enclave, the victim then page faults at line 3. However, the transient execution first continues with a zeroed %rbp register, which eventually gets written to the %rsp stack pointer register at line 5. Crucially, at this point, all subsequent pop and ret transient instructions dereference the attacker-controlled memory page mapped at virtual address null. This stack pointer zeroing primitive essentially allows LVI-NULL attackers to setup an arbitrary fake transient "shadow stack" at address null. We experimentally validated that this technique can furthermore be abused to mount a full transient ROP [56] attack by chaining together multiple subsequent pop-ret gadgets.

c) Applicability to non-SGX environments: LVI-NULL does not exploit any microarchitectural properties that are specific to Intel SGX, and may apply to other environments as well. However, we note that exploitation may be hindered by various architectural and software-level defensive measures that are in place to harden against well-known architectural null pointer dereference bugs. Some Linux distributions do not allow to map virtual address zero in user space. Furthermore, recent x86 SMAP and SMEP architectural features further prohibit respectively user-space data and code pointer dereferences in kernel mode. SMAP and SMEP have been shown to also hold in the microarchitectural transient domain [10, 26].

VII. LVI CASE STUDIES ON INTEL SGX

A. Gadget in Intel's Quoting Enclave

In this section, we show that exploitable LVI gadgets may occur in real-world software. We analyze Intel's trusted quoting enclave (QE), which has been widely studied in previous transient-execution research [11, 53, 61] to dismantle remote attestation guarantees in the Intel SGX ecosystem. As a result, the QE trusted codebase has been thoroughly vetted and hardened against all known Meltdown-type and Spectre-type attacks by manually inserting lfence instructions after potentially mispredicted branches, as well as flushing leaky microarchitectural buffers on every enclave entry and exit.

a) Gadget description: We started from the observation that most LVI variants first require a $\mathcal{P}1$ load-store gadget with an attacker-controlled address and data, followed by a faulting or assisted $\mathcal{P}2$ load that picks up the poisoned data. Similar to the edger8r gadget discussed in Section VI-B, we

---

[1] Note that we also found similar, potentially exploitable gadgets in the rsp-rbp function epilogues emitted by popular compilers such as gcc.

```
1  __intel_avx_rep_memcpy: ; libirc_2.4/efi2/libirc.a
2  ...                     ; P1: store to user address
3  vmovups %xmm0,-0x10(%rdi,%rcx,1)
4  ...
5  pop     %r12            ; P2: load from trusted stack
6  ret
```

Listing 4: LVI gadget in SGX-SDK `intel_fast_memcpy` used in QE.

therefore focused our manual code review on pointer arguments which are passed to copy input and output data via untrusted memory outside the enclave [62]. Particularly, we found that QE securely verifies that the output pointer to hold the resulting quote falls outside the enclave while leaving the base address in unprotected memory under attacker control. An Intel SGX quote is composed of various metadata fields, followed by the asymmetric signature (cf. Appendix C). After computing the signature, but before erasing the EPID private key from enclave memory, QE invokes `memcpy` to copy the corresponding quote metadata fields from trusted stack memory to the output buffer outside the enclave. Crucially, we found that as part of the last metadata fields, a 64-byte attacker-controlled `report_data` value is written to the attacker-provided output pointer.

We reverse engineered the proprietary `intel_fast_me mcpy` function used in QE and found that in this case, the quote is outputted using 128-bit vector instructions. Listing 4 provides the corresponding assembly code snippet, where the final 128-bit store at line 3 (including 12 bytes of attacker data) is closely followed by a `pop` and `ret` instruction sequence at lines 5-6 when returning from the `memcpy` invocation. This forms an exploitable LVI-SB transient control-flow hijacking gadget: the `vmovups` instruction ($\mathcal{P}1$) first fills the store buffer with user data at a user-controlled page offset aligned with the return address on the enclave stack, and closely afterwards the faulting or assisted `ret` instruction ($\mathcal{P}2$) incorrectly picks up the poisoned user data. The attacker now succeeded to redirect transient control flow to an arbitrary $\mathcal{P}3$ gadget address in the enclave code, which may subsequently lead to QE private key disclosure [11]. Note that when transiently executing the $\mathcal{P}3$ gadget, the attacker also controls the value of the `%r12` register popped at line 5 (which can be injected via the prior stores similarly to the return address). We further remark that Listing 4 is not limited to LVI-SB, since the store data may also have been committed from the store buffer to the L1 cache and subsequently picked up using LVI-L1D.

The Intel SGX-SDK [30] randomizes the 11 least significant bits of the stack pointer on enclave entry. However, as return addresses are aligned, the entropy is only 7 bits, resulting on average in a correct alignment in 1 out of every 128 enclave entries when fixing the store address in $\mathcal{P}1$.

b) Experimental results: We validate the exploitability and success rate of the above assembly code using a benchmark enclave on an i7-8650U with the latest microcode `0xb4`. We inject both the return address and the value popped into `%r12` via the store buffer. For $\mathcal{P}3$, we can use the poisoned value in `%r12` to transmit data over an address outside the enclave. We ensure that the code in Listing 4 is page aligned to interrupt the

victim enclave using a controlled-channel attack [71]. Before resuming the victim, we clear the user-accessible bit for the enclave stack. Additionally, to extend the transient window, we inserted a memory access which misses the cache before line 3.

In the first experiment, we disable stack randomization in the victim enclave to reliably quantify the success rate of the attack in the ideal case. LVI works very reliably, picking up the injected values $99\,453$ times out of $100\,000$ runs. With on average 9090 tries per second, we achieve an error-free transmission rate of $9.04\,\mathrm{kB/s}$ for our disclosure gadget.

In the second experiment, we simulate the full attack environment including stack randomization. As expected, the success rate drops by an average factor of 128. The injected return address is picked up 776 times out of $100\,000$ runs, leading to a transmission rate of $70.54\,\mathrm{B/s}$. We did not reproduce this attack against Intel's officially signed quoting enclave, as we found it especially challenging to debug the attack for production QE binaries and to locate $\mathcal{P}3$ gadgets that fit within the limited transient window without excessive TLB misses. However, we believe that our experiments showcased all the required primitives to break Intel SGX's remote attestation guarantees, as demonstrated before by SGXPectre [11] and Foreshadow [61]. In response to our findings, Intel will harden all architectural enclaves with full LVI software mitigations (cf. Section IX) so as to restore trust and initiate TCB recovery for the SGX ecosystem [27].

B. Transient Fault Attack on AES-NI

In this case study, we show that LVI-NULL can be exploited to perform a cryptographic fault attack [47, 59] on Intel's constant-time AES-NI hardware extension. We exploit that a privileged SGX attacker can induce faulty all-zero round keys into the transient data stream of a minimal AES-NI enclave. After the fault, the output of the decryption carries a faulty plaintext in the transient domain. To simplify the attack, we consider a known-ciphertext scenario and we assume a side-channel in the post-processing which allows to recover the faulty decryption output from the transient domain. Note that prior research [68] on Spectre-type attacks has shown that transient execution may fit a significant number of AES-NI decryptions (over 100 rounds on modern Intel processors).

Intel AES-NI [21] is implemented as an x86 vector extension. The `aesdec` and `aesdeclast` instructions perform one round of AES on a 128-bit register using the round key provided in the first register operand. Round keys are stored in trusted memory and, depending on the available registers and the AES-NI software implementation, the key schedule is either preloaded or consulted at the start of each round. In our case study, we assume that round keys are securely fetched from trusted enclave memory before each `aesdec` instruction.

a) Attack outline: Figure 7 illustrates the different phases in our transient fault injection attack on AES-NI:

1) We use SGX-Step [63] to precisely interrupt the victim enclave after executing only the initial round of AES.
2) The root attacker clears the user-accessible bit on the memory page containing the round keys.
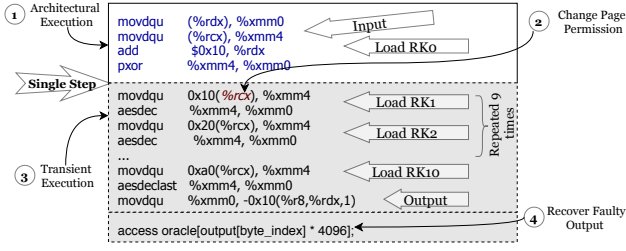
Fig. 7. Overview of the AES-NI fault attack: (1) the victim architecturally executes the initial AES round, which `xors` the input with round key 0; (2) access rights on the memory page holding the key schedule are revoked; (3) upon the next key access ($\mathcal{P}2$), the enclave suffers a page fault, causing the CPU to transiently execute the next 10 AES rounds with zeroed round keys; (4) finally the faulty output is encoded ($\mathcal{P}3$) through a cache-based side-channel.

3) The attacker resumes the enclave, leading to a page fault when loading the next round keys from trusted memory. We abuse theses faulting load as $\mathcal{P}2$ gadgets which transiently forward dummy (all-zero) round keys to the remaining `aesdec` instructions. Note that we do not need a $\mathcal{P}1$ gadget, as the CPU itself is responsible for zero-injection.
4) Finally, we use a $\mathcal{P}3$ disclosure gadget after the decryption.

By forcing all but the first AES round key to zero, our attack essentially causes the victim enclave to compute a round-reduced AES in the transient domain. To recover the first round key, and hence the full AES key, the attacker can simply feed the faulty output plaintext recovered from the transient domain to an inverse AES function with all keys set to zero. This results in an output that holds the secret AES first round key, `xor`-ed with the (known) ciphertext.

b) Experimental results: We run the attack for 100 different AES keys on a Core i9-9900K with `RDCL_NO` and the latest microcode `0xae`. For each experiment, we run the attack to recover 10 candidates for each byte of the faulty output. On average, each recovered key candidate matches the expected faulty output 83% of the time. Using majority vote for the 10 candidates, we recover the correct output for an average of 15.61 out of 16 bytes of the AES block, indicating that the output matches the attack model with 97% accuracy. The attack takes on average $25.94\,\text{s}$ (including enclave creation time) and requires $246\,707$ executions of the AES function.

For post-processing, we modified an AES implementation to zero out the round keys after the first round. We successfully recovered the secret round-zero key using any of the recovered faulty plaintext outputs to the inverse encryption function.

VIII. LVI IN OTHER CONTEXTS

A. User-to-Kernel

The main challenge in a user-to-kernel LVI attack scenario is to provoke faulting or assisted loads during kernel execution. As any application, the kernel may encounter page faults or microcode assists, e.g., due to demand paging via the extended page tables setup by the hypervisor, or when swapping out supervisor heap memory pages in the Windows kernel [50]. We do not investigate the more straightforward scenario where the kernel encounters a page fault when accessing a user-space

address, as in this case the user already architecturally controls the value read by the kernel.

a) Experimental setup: We focus on exploiting LVI-SB via microcode assists for setting the accessed bit in supervisor PTEs. In our case study, we execute the $\mathcal{P}1$ poisoning phase directly in user space by abusing that current microcode mitigations only flush the store buffer on kernel exit to prevent leakage [9, 29]. As the store buffer is not drained on kernel entry, it can be filled with attacker-chosen values by writing to arbitrary user-accessible addresses before performing the system call. Note that, alternatively, the store buffer could also be filled during kernel execution by abusing a selected $\mathcal{P}1$ gadget, similar to our SGX attacks.

In the $\mathcal{P}2$ phase, the attacker needs to trigger a faulting or assisted load micro-op in the kernel. In our proof-of-concept, we assume that the targeted supervisor page is swappable, as is the case for Windows kernel heap objects [50], but to the best of our knowledge not for the Linux kernel. In order to repeatedly execute the same experiment and assess the overall success rate, we simulate the workings of the page-replacement algorithm by means of a small kernel module, which artificially clears the accessed bit on the targeted kernel page.

As we only want to demonstrate the building blocks of the attack, we did not actively look for real-world gadgets in the kernel. For our evaluation, we manually added a simple $\mathcal{P}3$ disclosure gadget, which, similar to a Spectre gadget, indexes a shared memory region based on a previously loaded value as follows: `array[(*kernel_pt) * 4096]`. In case the trusted load on `kernel_pt` requires a microcode assist, the value written by the user-space attacker will be transiently injected from the store buffer and subsequently encoded into the CPU cache.

b) Experimental results: We evaluated LVI-SB on an Intel Core i7-8650U with Linux kernel 5.0. On average, 1 out of every 7739 ($n = 100\,000$) assisted loads in the kernel use the injected value from the store buffer instead of the architecturally correct value. For our non-optimized proof-of-concept, this results on average in a successfully injected value into the kernel execution every $6.5\,\text{s}$. One of the reasons for this low success rate is the context switch between $\mathcal{P}1$ and $\mathcal{P}2$, which reduces the probability that the attacker's value is still outstanding in the store buffer [9]. We verified this by evaluating the injection rate without a context switch, *i.e.*, if the store buffer is poisoned via a suitable $\mathcal{P}1$ gadget in the kernel. In this case, on average, 1 out of every 8 ($n = 100\,000$) assisted loads in the kernel use the injected value.

B. Cross-Process

We now demonstrate how LVI-LFB may inject poisoned data from a concurrently running attacker process.

a) Experimental setup: For the poisoning phase $\mathcal{P}1$, we assume that the attacker and the victim are co-located on the same physical CPU core [53, 61, 67]. The attacker directly poisons the line-fill buffer by writing or reading values to or from the memory subsystem. To ensure that the values travel through the fill buffer, the attacker simply flushes the accessed

values using the unprivileged `cflflush` instruction. In case hyperthreading is disabled, the adversary would have to find a suitable $\mathcal{P}1$ gadget that processes untrusted, attacker-controlled arguments in the victim code, similar to our SGX attacks.

In our proof-of-concept, the victim application loads a value from a trusted shared-memory location, e.g., a shared library. As shown by Schwarz et al. [53], Windows periodically clears the PTE accessed bit, which may cause microcode assists for trusted loads in the victim process. The attacker flushes the targeted shared-memory location from the cache, again using `clflush`, to ensure that the victim's assisted load $\mathcal{P}2$ forwards incorrect values from the line-fill buffer [53, 67] instead of the trusted shared-memory content.

b) Experimental results: We evaluated the success rate of the attack on an Intel i7-8650U with Linux kernel 5.0. We used the same software construct as in the kernel attack for the transmission phase $\mathcal{P}3$. Both attacker and victim run on the same physical core but different logical cores. On average, 1 out of 101 ($n = 100\,000$) assisted loads uses the value injected by the attacker, resulting in an injection probability of nearly $1\,\%$. With on average 1122 tries per second, we achieve a transmission rate of $11.11\,\mathrm{B/s}$ for our disclosure gadget.

IX. DISCUSSION AND MITIGATIONS

In this section, we discuss both long-term silicon mitigations to rule out LVI at the processor design level, as well as compiler-based software workarounds that need to be deployed on the short-term to mitigate LVI on existing systems.

A. Eradicating LVI at the Hardware Design Level

The root cause of LVI needs to be ultimately addressed through silicon-level design changes in future processors. Particularly, to rule out LVI, the hardware has to ensure that no illegal data flows from faulting or assisted load micro-ops exist at the microarchitectural level. That is, no transient computations depending on a faulting or assisted instruction are allowed. We believe this is already the behavior in certain ARM and AMD processors, where a faulting load does not forward any data [2]. Notably, we showed in Section VI-C that it does *not* suffice to merely zero out the forwarded value, as is the case in the latest generation of acclaimed Meltdown-resistant Intel processors enumerating `RDCL_NO` [28].

B. A Generic Software Workaround

Silicon-level design changes take considerable time, and at least for SGX enclaves a short-term solution is needed to mitigate LVI on current, widely deployed systems. In contrast to previous Meltdown-type attacks, merely flushing microarchitectural buffers before or after victim execution is *not* sufficient to defend against our novel, gadget-based LVI attack techniques. Instead, we propose a software-based mitigation approach which inserts explicit `lfence` speculation barriers to serialize the processor pipeline after every vulnerable load instruction. The `lfence` instruction is guaranteed by Intel to halt transient execution until all prior instructions have completed [28]. Hence, inserting an `lfence` after every

TABLE II. Indirect branch instruction emulations needed to prevent LVI and whether or not they require a scratch register which can be clobbered.

Instruction	Possible Emulation	Clobber
`ret`	`pop %reg; lfence; jmp *%reg`	✓
`ret`	`not (%rsp); not (%rsp); lfence; ret`	✗
`jmp (mem)`	`mov (mem),%reg; lfence; jmp *%reg`	✓
`call (mem)`	`mov (mem),%reg; lfence; call *%reg`	✓

potentially faulting or assisted load micro-op guarantees that the value forwarded from the load operation is not an injected value but the architecturally correct one. Relating to the general attack scheme of Figure 3, we introduce an `lfence` instruction in between phases $\mathcal{P}2$ and $\mathcal{P}3$ to inhibit any incorrect transient forwarding by the processor. Crucially, in contrast to existing Spectre-PHT compiler mitigations [10, 28] which only insert `lfence` barriers after potentially mispredicted conditional jump instructions, fully mitigating LVI requires stalling the processor pipeline after potentially *every* explicit as well as implicit memory-load operation.

Explicit memory loads, *i.e.*, instructions with a memory address as input parameter, can be protected straightforwardly. A compiler, or even a binary rewriter [14], can add an `lfence` instruction to ensure that any dependent operations can only be executed after the load instruction has successfully retired. However, some x86 instructions also include *implicit* memory load micro-ops which cannot be mitigated in this way. For instance, indirect branches and the `ret` instruction load an address from the stack and immediately redirect control flow to the loaded, possibly injected value. As the faulting or assisted load micro-op in this case forms part of a larger ISA-level instruction, there is no possibility to add an `lfence` barrier between the memory load ($\mathcal{P}2$) and the control-flow redirection ($\mathcal{P}3$). Table II shows how indirect branch instructions have to be blacklisted and emulated through an equivalent sequence of two or more instructions, including an `lfence` after the formerly implicit memory load. Notably, as some of these emulation sequences clobber scratch registers, LVI mitigations for indirect branches cannot be trivially implemented using binary rewriting techniques and should preferably be implemented in the compiler back-end, before the register allocation stage.

a) Evaluation of our prototype solution: We initially implemented a prototypical compiler mitigation using LLVM [43] (8.3.0) and applied it to a recent OpenSSL [48] version (1.1.1d) with default configuration. We chose OpenSSL as it serves as the base of the official Intel SGX-SSL library [33] allowing to approximate the expected performance impact of the proposed mitigations. Our proof-of-concept mitigation tool allows to augment the building process of arbitrary C code by first instrumenting the compiler to emit LLVM intermediate code, adding the necessary `lfence` instructions after every explicit memory load, and finally proceeding to compile the modified file to an executable. Our prototype tool cannot mitigate loads which are not visible at the LLVM intermediate representation, e.g., the x86 back-end may introduce loads for registers spilled onto the stack after register allocation.
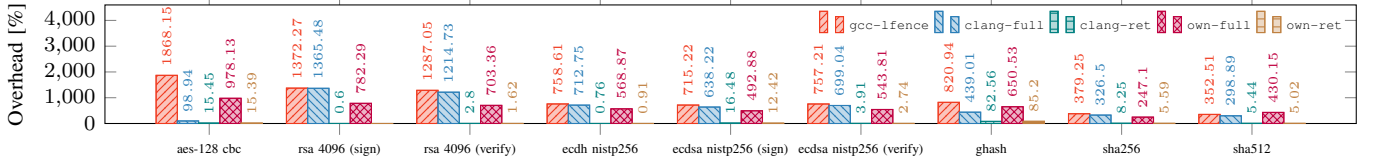
Fig. 8. Performance overhead of our LLVM-based prototype (fence loads + ret vs. ret-only) and Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for OpenSSL on an Intel i7-6700K CPU.

To deal with assembly source files, our tool introduces an `lfence` after every `mov` operating on memory addresses. Our prototype does not mitigate all types of indirect branches, but can optionally replace `ret` instructions with the proposed emulation code, where `%r11` is used as a caller-save register that can be clobbered.

To measure the performance impact of the introduced `lfence` instructions and the `ret` emulation, we recorded the average throughput ($n = 10$) of various cryptographic primitives using OpenSSL's `speed` tool on an isolated core on an Intel i7-6700K. As shown in Figure 8, the performance overhead reaches from a minimum of 0.91% for a partial mitigation which only rewrites `ret` instructions to a maximum of 978.13% for the full mitigation including `ret` emulation and load serialization. Note that for real-world deployment, the placement of `lfence` instructions should be evaluated for completeness and more optimized than in our prototype implementation. Still, our evaluation serves as an approximation of the expected performance impact of the proposed mitigations.

b) Evaluation of Intel's proposed mitigations: To further evaluate the overheads of more mature, production-quality implementations, we were provided with access to Intel's current compiler-based mitigation infrastructure. Hardening of existing code bases is facilitated by a generic post-compilation script that uses regular expressions to insert an `lfence` after every x86 instruction that has a load micro-op. Working exclusively at the assembly level, the script is inherently compiler-agnostic and can hence only make use of indirect branch emulation instruction sequences that do not clobber registers. In general, it is therefore recommended to first decompose indirect branches from memory using existing Spectre-BTB mitigations [60]. As not all code respects calling conventions, `ret` instructions are by default replaced with a clobber-free emulation sequence which first tests the return address, before serializing the processor pipeline and issuing the `ret` (cf. Table II). We want to note that this emulation sequence still allows privileged LVI adversaries to provoke a fault or assist on the return address when leveraging a single-stepping framework like SGX-Step [63] to precisely interrupt and resume the victim enclave after the `lfence` and before the final `ret`. However, we expect that in such a case the length of the transient window would be severely restricted as `eresume` appears to be a serializing instruction itself [32]. Furthermore, as recent microcode flushes microarchitectural buffers on enclave entry, the poisoning phase would be limited to LVI-NULL. Any inadvertent transient control-flow redirections to
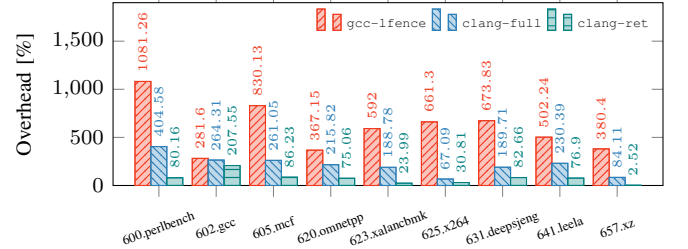

Fig. 9. Performance overhead of Intel's mitigations for non-optimized assembler `gcc` (fence loads + ret) and optimized `clang` (fence loads + indirect branch + ret vs. ret-only) for SPEC2017 on an Intel i9-9900K CPU.

virtual address null can be mitigated by marking the first enclave page as non-executable (cf. Section VI-C).

Intel furthermore developed an optimized LVI mitigation pass for LLVM-based compilers. The pass operates at the LLVM intermediate representation and uses a constraint solver from integer programming to optimally insert `lfence` instructions along all paths in the control-flow graph from a load ($\mathcal{P}2$) to a transmission ($\mathcal{P}3$) gadget [3, 34]. As the pass operates at the LLVM intermediate representation, any additional loads introduced by the x86 back-end are not mitigated. We expect such implicit loads from e.g., registers that were previously spilled onto the stack to be difficult to exploit in practice, but we leave further security evaluation of the mitigations as future work. The pass also replaces indirect branches, and `ret` instructions are eliminated in an additional machine pass using a caller-save clobber register.

Figure 8 provides the OpenSSL evaluation for the Intel mitigations ($n = 10$). The unoptimized `gcc` post-compilation full mitigation assembly script for fencing all loads and `ret` instructions clearly incurs the highest overheads from 352.51% to 1868.15%, which is slightly worse than our own (incomplete) LLVM-based prototype. For the OpenSSL experiments, Intel's optimized `clang` LLVM mitigation pass for fencing loads, conditional branches, and `ret` instructions generally reduces overheads within the same order of magnitude, but more significantly in the AES case. Lastly, in line with our own prototype evaluation, smaller overheads from 2.52% to 86.23% are expected for a partial mitigation strategy which patches only `ret` instructions while leaving other loads and indirect branches potentially exposed to LVI attackers.

Finally, to assess expected overheads in larger and more varied applications, we evaluated Intel's mitigations on the SPEC2017 `intspeed` benchmark suite. Figure 9 provides

the results as executed on an isolated core on a i9-9900K CPU, running Linux 4.18.0 with Ubuntu 18.10 ($n = 3$).[2] One clear trend is that Intel's optimized LLVM mitigation pass outperforms the naive post-compilation assembly script.

X. Outlook and Future Work

We believe that our work presents interesting opportunities for developing more efficient compiler mitigations and software hardening techniques for current, widely deployed systems.

A. Implications for Transient-Execution Attacks and Defenses

LVI again illustrates the constant race between attackers and defenders. With LVI, we introduced an advanced attack technique that bypasses existing software and hardware defenses. While potentially harder to exploit than previous Meltdown-type attacks, LVI shows that Meltdown-type incorrect transient forwarding effects are not as easy to fix as expected [10, 42, 72]. The main insight with LVI is that transient-execution attacks, as well as side-channel attacks, have to be considered from two viewpoints: observing and injecting data. It is not sufficient to only mitigate data leakage direction, as it was done so far, and the injection angle also needs to be considered. Hence, in addition to flushing microarchitectural buffers on context switch [25, 29], additional mitigations are required. We believe that our work has a substantial influence on future transient-execution attacks as new discoveries of Meltdown-type effects now need to be studied in both directions.

Although the most realistic LVI attack scenarios are secure enclaves such as Intel SGX, we demonstrated that none of the ingredients for LVI are unique to SGX and other environments can possibly be attacked similarly. We encourage future attack research to further investigate improved LVI gadget discovery and exploitation techniques in non-SGX settings, e.g., cross-process and sandboxed environments [38, 44].

An important insight for silicon mitigations is that merely zeroing out unintended data flow is insufficient to protect against LVI adversaries. At the compiler level, we expect that advanced static analysis techniques may further improve the extensive performance overheads of current `lfence`-based mitigations (cf. Section IX-B). Particularly, for non-control-flow hijacking gadgets, it would be desirable to serialize only those loads that are closely followed by an exploitable $\mathcal{P}3$ gadget for side-channel transmission.

B. Raising the Bar for LVI Exploitation

While not completely eliminated, our analysis in Section VI and Appendix B revealed that the LVI attack surface may be greatly reduced by certain system-level software measures in non-SGX environments. For instance, the correct sanitization of user-space pointers and the use of x86 SMAP and SMEP features in commodity OS kernels may greatly reduce the possible LVI gadget space. Furthermore, we found that certain software mitigations, which were deployed to prevent Meltdown-type data leakages, also unintentionally thwart their LVI counterparts,

e.g., eager FPU switching [57] and PTE inversion [12]. LVI can also be inhibited by preventing victim loads from triggering exceptions and microcode assists. However, this may come with significant changes in system software, as e.g., PTE accessed and dirty bits must not be cleared anymore, and kernel pages must not be swapped anymore. Although such changes are possible for the OS, they are not possible for SGX, as the attacker is in control of the page tables.

As described in Section IX-B, Intel SGX enclaves require extensive compiler mitigations to fully defend against LVI. However, we also advocate architectural changes in the SGX design which may further help raising the bar for LVI exploitation. LVI is for instance facilitated by the fact that SGX enclaves share certain microarchitectural elements, such as the cache, with their host application [13, 46, 52]. Furthermore, enclaves can directly operate on untrusted memory locations passed as pointers in the shared address space [55, 62]. As a generic software hardening measure, we suggest that pointer sanitization logic [62] further restricts the attacker's control over page offset address bits for unprotected input and output buffers. To inhibit transient null-pointer dereferences in LVI-NULL exploits, we propose that microcode marks the memory page at virtual address zero as uncacheable [6, 54, 58]. Similarly, LVI-L1D could be somewhat restricted by terminating the enclave or disabling SGX altogether upon detecting a rogue PPN in the EPCM microcode checks, which can only indicate a malicious or buggy OS.

XI. Conclusion

We presented Load Value Injection (LVI), a novel class of attack techniques allowing the direct injection of attacker data into a victim's transient data stream. LVI complements the transient-execution research landscape by turning around Meltdown-type data leakage into data injection. Our findings challenge prior views that, unlike Spectre, Meltdown threats could be eradicated straightforwardly at the operating system or hardware levels and ultimately show that future Meltdown-type attack research must also consider the injection angle.

Our proof-of-concept attacks against Intel SGX enclaves and other environments show that LVI gadgets exist and may be exploited. Existing Meltdown and Spectre defenses are orthogonal to and do not impede our novel attack techniques, such that LVI necessitates drastic changes at the compiler level. Fully mitigating LVI requires including `lfences` after possibly *every* memory load, as well as blacklisting indirect jumps, including the ubiquitous x86 `ret` instruction. We observe extensive slowdowns of factor 2 to 19 for our prototype evaluation of this countermeasure. LVI demands research on more efficient and forward-looking mitigations on both the hardware and software levels.

---

[2] Note that we had to exclude the `648.exchange2_s` benchmark program as it is written in Fortran and hence not supported by the mitigation tools.

REFERENCES

[1] O. Acıiçmez, c. K. Koç, and J.-p. Seifert, "On the Power of Simple Branch Prediction Analysis," in *AsiaCCS*, 2007.

[2] AMD, "Speculation Behavior in AMD Micro-Architectures," 2019.

[3] J. Bender, M. Lesani, and J. Palsberg, "Declarative fence insertion," in *ACM SIGPLAN Notices*, vol. 50, no. 10.   ACM, 2015, pp. 367–385.

[4] A. Bhattacharyya, A. Sandulescu, M. Neugschwandtner, A. Sorniotti, B. Falsafi, M. Payer, and A. Kurmus, "SMoTherSpectre: exploiting speculative execution through port contention," in *CCS*, 2019.

[5] T. K. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang, "Jump-oriented programming: a new class of code-reuse attack," in *AsiaCCS*, 2011.

[6] D. D. Boggs, R. Segelken, M. Cornaby, N. Fortino, S. Chaudhry, D. Khartikov, A. Mooley, N. Tuck, and G. Vreugdenhil, "Memory type which is cacheable yet inaccessible by speculative instructions," 2019, uS Patent App. 16/022,274.

[7] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *WOOT*, 2017.

[8] Y. Bulygin, "Cpu side-channels vs. virtualization malware: The good, the bad, or the ugly," *ToorCon*, 2008.

[9] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom, "Fallout: Leaking Data on Meltdown-resistant CPUs," in *CCS*, 2019.

[10] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss, "A Systematic Evaluation of Transient Execution Attacks and Defenses," in *USENIX Security Symposium*, 2019.

[11] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution," in *EuroS&P*, 2019.

[12] J. Corbet, "Meltdown strikes back: the L1 terminal fault vulnerability," 2018. [Online]. Available: https://lwn.net/Articles/762570/

[13] V. Costan and S. Devadas, "Intel SGX Explained," *Cryptology ePrint Archive, Report 2016/086*, 2016.

[14] S. Dinesh, N. Burow, D. Xu, and M. Payer, "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization," in *S&P*, 2020.

[15] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "BranchScope: A New Side-Channel Attack on Directional Branch Predictor," in *ASPLOS*, 2018.

[16] Q. Ge, Y. Yarom, D. Cock, and G. Heiser, "A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware," *Journal of Cryptographic Engineering*, 2016.

[17] A. Glew, G. Hinton, and H. Akkary, "Method and apparatus for performing page table walks in a microprocessor capable of processing speculative instructions," 1997, uS Patent 5,680,565.

[18] D. Gruss, D. Hansen, and B. Gregg, "Kernel Isolation: From an Academic Idea to an Efficient Patch for Every Computer," *USENIX ;login*, 2018.

[19] D. Gruss, E. Kraft, T. Tiwari, M. Schwarz, A. Trachtenberg, J. Hennessey, A. Ionescu, and A. Fogh, "Page Cache Attacks," in *CCS*, 2019.

[20] D. Gruss, M. Lipp, M. Schwarz, R. Fellner, C. Maurice, and S. Mangard, "KASLR is Dead: Long Live KASLR," in *ESSoS*, 2017.

[21] S. Gueron, "Intel Advanced Encryption Standard (Intel AES) Instructions Set – Rev 3.01," 2012.

[22] M. D. Hill, J. Masters, P. Ranganathan, P. Turner, and J. L. Hennessy, "On the Spectre and Meltdown Processor Security Vulnerabilities," *IEEE Micro*, vol. 39, no. 2, pp. 9–19, 2019.

[23] J. Horn, "speculative execution, variant 4: speculative store bypass," 2018.

[24] T. Huo, X. Meng, W. Wang, C. Hao, P. Zhao, J. Zhai, and M. Li, "Bluethunder: A 2-level directional predictor based side-channel attack against sgx," *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pp. 321–347, 2020.

[25] Intel, "Deep Dive: Intel Analysis of L1 Terminal Fault," 2018.

[26] ——, "Intel Analysis of Speculative Execution Side Channels ," 2018, revision 4.0.

[27] ——, "Intel SGX Trusted Computing Base (TCB) Recovery," 2018. [Online]. Available: https://software.intel.com/sites/default/files/managed/01/7b/Intel-SGX-Trusted-Computing-Base-Recovery.pdf

[28] ——, "Speculative Execution Side Channel Mitigations," 2018, revision 3.0.

[29] ——, "Deep Dive: Intel Analysis of Microarchitectural Data Sampling," 2019.

[30] ——, "Get Started with the SDK," 2019. [Online]. Available: https://software.intel.com/en-us/sgx/sdk

[31] Intel, "Intel 64 and IA-32 Architectures Optimization Reference Manual," 2019.

[32] Intel, "Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide," 2019.

[33] ——, "Intel® Software Guard Extensions SSL," 2019. [Online]. Available: https://github.com/intel/intel-sgx-ssl

[34] ——, "Load Value Injection," 2020, white paper accompanying Intel-SA-00334. [Online]. Available: https://software.intel.com/security-software-guidance/

[35] S. Islam, A. Moghimi, I. Bruhns, M. Krebbel, B. Gulmezoglu, T. Eisenbarth, and B. Sunar, "SPOILER: Speculative load hazards boost rowhammer and cache attacks," in *USENIX Security Symposium*, 2019.

[36] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors," in *ISCA*, 2014.

[37] V. Kiriansky and C. Waldspurger, "Speculative Buffer Overflows: Attacks and Defenses," *arXiv:1807.03757*, 2018.

[38] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *S&P*, 2019.

[39] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre Returns! Speculation Attacks using the Return Stack Buffer," in *WOOT*, 2018.

[40] S. Lee, M. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *USENIX Security Symposium*, 2017.

[41] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen, "Value locality and load value prediction," *ACM SIGPLAN Notices*, vol. 31, no. 9, pp. 138–147, 1996.

[42] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading Kernel Memory from User Space," in *USENIX Security Symposium*, 2018.

[43] LLVM, "The LLVM Compiler Infrastructure," 2019. [Online]. Available: https://llvm.org

[44] G. Maisuradze and C. Rossow, "ret2spec: Speculative Execution Using Return Stack Buffers," in *CCS*, 2018.

[45] J. Masters, "Thoughts on NetSpectre," 2018. [Online]. Available: https://www.redhat.com/en/blog/thoughts-netspectre

[46] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How SGX amplifies the power of cache attacks," in *CHES*, 2017.

[47] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *S&P*, 2020.

[48] OpenSSL, "OpenSSL: The Open Source toolkit for SSL/TLS," 2019. [Online]. Available: http://www.openssl.org

[49] L. Orosa, R. Azevedo, and O. Mutlu, "AVPP: Address-first value-next predictor with value prefetching for improving the efficiency of load value

prediction," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, no. 4, p. 49, 2018.

[50] M. Russinovich, "Pushing the Limits of Windows: Paged and Nonpaged Pool," 2009. [Online]. Available: https://blogs.technet.microsoft.com/markrussinovich/2009/03/10/pushing-the-limits-of-windows-paged-and-nonpaged-pool/

[51] M. Schwarz, C. Canella, L. Giner, and D. Gruss, "Store-to-Leak Forwarding: Leaking Data on Meltdown-resistant CPUs," *arXiv:1905.05725*, 2019.

[52] M. Schwarz, D. Gruss, S. Weiser, C. Maurice, and S. Mangard, "Malware Guard Extension: Using SGX to Conceal Cache Attacks," in *DIMVA*, 2017.

[53] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-Privilege-Boundary Data Sampling," in *CCS*, 2019.

[54] M. Schwarz, R. Schilling, F. Kargl, M. Lipp, C. Canella, and D. Gruss, "ConTExT: Leakage-Free Transient Execution," *arXiv:1905.09100*, 2019.

[55] M. Schwarz, S. Weiser, and D. Gruss, "Practical Enclave Malware with Intel SGX," in *DIMVA*, 2019.

[56] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS*, 2007.

[57] J. Stecklina and T. Prescher, "LazyFP: Leaking FPU Register State using Microarchitectural Side-Channels," *arXiv:1806.07480*, 2018.

[58] K. Sun, R. Branco, and K. Hu, "A New Memory Type Against Speculative Side Channel Attacks," 2019.

[59] A. Tang, S. Sethumadhavan, and S. Stolfo, "CLKSCREW: Exposing the perils of security-oblivious energy management," in *USENIX Security Symposium*, 2017.

[60] P. Turner, "Retpoline: a software construct for preventing branch-target-injection," 2018.

[61] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium*, 2018.

[62] J. Van Bulck, D. Oswald, E. Marin, A. Aldoseri, F. Garcia, and F. Piessens, "A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes," in *CCS*, 2019.

[63] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control," in *Workshop on System Software for Trusted Execution*, 2017.

[64] ——, "Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic," in *CCS*, 2018.

[65] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *USENIX Security Symposium*, 2017.

[66] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "Addendum to RIDL: Rogue in-flight data load," 2019.

[67] ——, "RIDL: Rogue in-flight data load," in *S&P*, 2019.

[68] J. Wampler, I. Martiny, and E. Wustrow, "Exspectre: Hiding malware in speculative execution." in *NDSS*, 2019.

[69] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *MICRO*, 1997.

[70] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution," 2018.

[71] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *S&P*, 2015.

[72] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. W. Fletcher, and J. Torrellas, "InvisiSpec: Making Speculative Execution Invisible in the Cache Hierarchy," in *MICRO*, 2018.

APPENDIX A
INTEL SGX PAGE TABLE WALKS

For completeness, Figure 10 summarizes the additional access control checks enforced by Intel SGX to verify the outcome of the untrusted address translation process [13, 65].
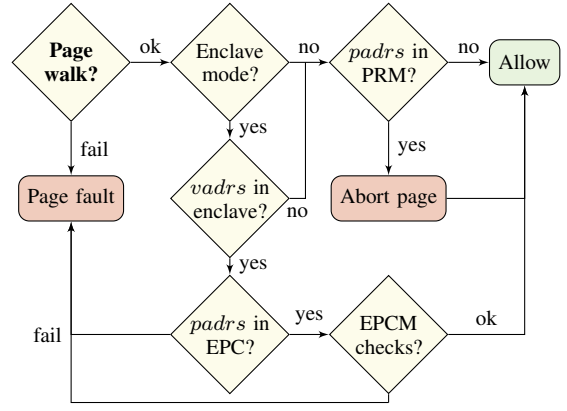

Fig. 10. Access control checks (page faults) in the SGX page table walk for a virtual address *vadrs* that maps to a physical address *padrs*.
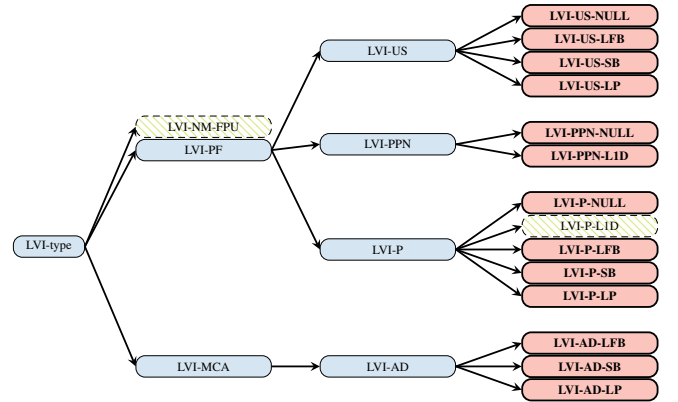

Fig. 11. Extensible LVI classification tree (generated using https://transient.fail/) with possible attack variants (red, bold), and neutralized variants that are already prevented by current software and microcode mitigations (green, dashed).

APPENDIX B
LVI CLASSIFICATION TREE

In this appendix, we propose an unambiguous naming scheme to reason about and distinguish LVI variants, following the (extended) transient-execution attack classification tree by Canella et al. [10]. Particularly, in a first level, we distinguish the *fault or assist type* triggering the transient execution, and at a second level we specify the *microarchitectural buffer* which is used as the injection source. Figure 11 shows the resulting two-level LVI classification tree. Note that, much like in the perpendicular Spectre class of attacks [10], not all CPUs from all vendors might be susceptible to all of these variants.

a) Applicability to Intel SGX: We remark that some of the fault types that may trigger LVI in Figure 11 are specific to Intel SGX's root attacker model. Particularly, LVI-US generates supervisor-mode page faults by clearing the user-accessible bit in the untrusted page table entry mapping a trusted enclave memory location. The user-accessible bit can only be modified by root attackers that control the untrusted OS, and hence does not apply in a user-to-kernel or user-to-user LVI scenario. Furthermore, LVI-PPN generates SGX-specific EPCM page

faults by supplying a rogue physical page number in a page-table entry mapping trusted enclave memory (cf. Section VI-A). This variant is specific to Intel SGX's EPCM memory access control model.

Finally, as explored in Section VIII, LVI-P and LVI-AD are not specific to Intel SGX, and might apply to traditional kernel and process isolation as well.

b) Neutralized variants: Interestingly, as part of our analysis, we found that some LVI variants are in principle feasible on unpatched systems, but are already properly prevented as an unintended side-effect of software mitigations that have been widely deployed in response to Meltdown-type cross-domain leakage attacks.

We considered whether virtual machine or OS process Foreshadow variants [70] may also be reversely exploited through an injection-based LVI methodology, but we concluded that no additional mitigations are required. In the case of virtual machines, the untrusted kernel can only provoke non-present page faults (and hence LVI-P-L1D injection) for less-privileged applications, and never for more privileged hypervisor software. Alternatively, we find that cross-process LVI-P-L1D is possible in demand-paging scenarios when the kernel does not properly invalidate the PPN field when unmapping a victim page and assigning the underlying physical memory to another process. The next page dereference in the victim process provokes a page fault leading to the L1TF condition and causing the L1D cache to inject potentially poisoned data from the attacker process into the victim's transient data stream. However, while this attack is indeed feasible on unpatched systems, we found that it is already properly prevented by the recommended PTE inversion [12] countermeasure which has been widely deployed in all major operating systems in response to Foreshadow.

Secondly, we considered that some processors transiently compute on unauthorized values from the FPU register file before delivering a device-not-available exception (#NM) [57]. This may be abused in a "reverse LazyFP" LVI-NM-FPU attack to inject attacker-controlled FPU register contents into a victim application's transient data stream. However, we concluded that no additional mitigations are required for this variant as all major operating systems inhibit the #NM trigger completely by unconditionally applying the recommended eager FPU switching mitigation. Likewise, Intel confirmed that for every enclave (re-)entry SGX catches and signals the #NM exception before any enclave code can run.

Finally, we concluded that the original Meltdown [42] attack to read (cached) kernel memory from user space cannot be inverted into an LVI-L1D equivalent. The reasoning here is that the user-accessible page-table entry attribute is only enforced in privilege ring 3, and a benign victim process would never dereference kernel memory.

APPENDIX C
INTEL SGX QUOTE LAYOUT

We first provide the C data structure layout representing a quote in Listing 5. Note that the report_data field in the sgx_report_body_t; is part of the (untrusted) input

```c
typedef struct _sgx_report_data_t {
    uint8_t              d[64];
} sgx_report_data_t;

typedef struct _report_body_t {
    ...
    /* (320) Data provided by the user */
    sgx_report_data_t       report_data;
} sgx_report_body_t;

typedef struct _quote_t {
    uint16_t            version;        /* 0   */
    uint16_t            sign_type;      /* 2   */
    sgx_epid_group_id_t epid_group_id;  /* 4   */
    sgx_isv_svn_t       qe_svn;         /* 8   */
    sgx_isv_svn_t       pce_svn;        /* 10  */
    uint32_t            xeid;           /* 12  */
    sgx_basename_t      basename;       /* 16  */
    sgx_report_body_t   report_body;    /* 48  */
    uint32_t            signature_len;  /* 432 */
    uint8_t             signature[];    /* 436 */
} sgx_quote_t;
```

Listing 5: https://github.com/intel/linux-sgx/blob/master/common/inc/sgx_quote.h#L87

```c
/* emp_quote: Untrusted pointer to quote output
 *            buffer outside enclave.
 * quote_body: sgx_quote_t holding quote metadata
 *            (without the actual signature).
 */
ret = qe_epid_sign(...
                   emp_quote,   /* fill in signature */
                   &quote_body, /* fill in metadata */
                   (uint32_t)sign_size);
...

/* now copy sgx_quote_t metadata (including user-
   provided report_data) into untrusted output buffer*/
memcpy(emp_quote, &quote_body, sizeof(sgx_quote_t));

/* now erase enclave secrets (EPID private key) */
CLEANUP:
  if(p_epid_context)
      epid_member_delete(&p_epid_context);
  return ret;
}
```

Listing 6: https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.cpp#L1139

provided as part of the QE invocation. The only requirement on this data is that it needs to have a valid SGX report checksum, and hence needs to be filled in by a genuine enclave running on the target system (but this can also be for instance an attacker-controlled debug enclave).

Furthermore, Listing 7 provides the get_quote entry point in Intel SGX-SDK Enclave Definition Language (EDL) specification. Note that the quote data structure holding the asymmetric cryptographic signature is relatively big, and hence is not transparently cloned into enclave memory. Instead this pointer is declared as user_check and explicitly verified to lie outside the enclave in the QE implementation, allowing to directly read from and write to this pointer from the trusted enclave code.

Listing 6 finally provides the C code fragment including the memcpy invocation discussed in Section VII-A.

TABLE III. Number of `lfence`s inserted by different compiler and assembler mitigations for the OpenSSL and SPEC benchmarks (cf. Figures 8 and 9).

Benchmark	Unoptimized assembler (Intel)		Optimized compiler (Intel)			Unoptimized LLVM intermediate (ours)	
	gcc-plain	gcc-lfence	clang-plain	clang-full	clang-ret	load+ret	ret-only
OpenSSL (`libcrypto.a`)	0	73 998	0	24 710	5608	39 368	5119
OpenSSL (`libssl.a`)	0	15 034	0	5248	1615	10 228	1415
600.perlbench	0	104 475	0	32 764	2584	-	-
602.gcc	10	458 799	1	148 069	17 198	-	-
605.mcf	0	1191	0	266	44	-	-
620.omnetpp	0	78 968	0	36 940	5578	-	-
623.xalancbmk	2	252 080	0	110 353	10 750	-	-
625.x264	0	31 748	0	5582	528	-	-
631.deepsjeng	0	4315	0	545	118	-	-
641.leela	0	8997	0	1669	340	-	-
657.xz	0	7820	0	1534	419	-	-

```
1  public uint32_t get_quote(
2    [size = blob_size, in, out] uint8_t *p_blob,
3    uint32_t blob_size,
4    [in] const sgx_report_t *p_report,
5    sgx_quote_sign_type_t quote_type,
6    [in] const sgx_spid_t *p_spid,
7    [in] const sgx_quote_nonce_t *p_nonce,
8    // SigRL is big, so we cannot copy it into EPC
9    [user_check] const uint8_t *p_sig_rl,
10   uint32_t sig_rl_size,
11   [out] sgx_report_t *qe_report,
12   // Quote is big, we should output it in piece meal.
13   [user_check] uint8_t *p_quote,
14   uint32_t quote_size, sgx_isv_svn_t pce_isvnsvn);
```

Listing 7: https://github.com/intel/linux-sgx/blob/master/psw/ae/qe/quoting_enclave.edl#L43

APPENDIX D
LFENCE COUNTS FOR COMPILER MITIGATIONS

Table III additionally provides the number of `lfence` instructions inserted by the various compiler and assembler mitigations introduced in Section IX-B for the OpenSSL and SPEC2017 benchmarks.