Repurposing Segmentation as a Practical LVI-NULL Mitigation in SGX

Lukas Giner
Graz University of Technology

Andreas Kogler
Graz University of Technology

Claudio Canella
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

Daniel Gruss
Graz University of Technology

Abstract

Load Value Injection (LVI) uses Meltdown-type data flows in Spectre-like confused-deputy attacks. LVI has been demonstrated in practical attacks on Intel SGX enclaves, and consequently, mitigations were deployed that incur tremendous overheads of factor 2 to 19. However, as we discover, on fixed hardware LVI-NULL leakage is still present. Hence, to mitigate LVI-NULL in SGX enclaves on LVI-fixed CPUs, the expensive mitigations would still be necessary.

In this paper, we propose a lightweight mitigation focused on LVI-NULL in SGX, LVI-NULLify. We systematically analyze and categorize LVI-NULL variants. Our analysis reveals that previously proposed mitigations targeting LVI-NULL are not effective. Our novel mitigation addresses this problem by repurposing segmentation, a fast legacy hardware mechanism that x86 already uses for every memory operation. LVI-NULLify consists of a modified SGX-SDK and a compiler extension which put the enclave in control of LVI-NULL-exploitable memory locations. We evaluate LVI-NULLify on the LVI-fixed Comet Lake CPU and observe a performance overhead below 10% for the worst case, which is substantially lower than previous defenses with a prohibitive overhead of 1220% in the worst case. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves against LVI-NULL today.

1 Introduction

Transient-execution attacks, *i.e.*, Meltdown [41], Spectre [34], or ZombieLoad [52], are powerful microarchitectural attacks for leaking sensitive data. These attacks are commonly classified into Spectre-type and Meltdown-type attacks [8]. Spectre-type attacks [18,33,34,36,42] exploit that the transient instructions following a wrongly predicted branch are not committed but still leave traces in the microarchitectural state.

With Load Value Injection (LVI), Van Bulck et al. [60] presented a new type of transient-execution attacks related to Meltdown-type attacks. Meltdown-type attacks trigger a faulting load in the attacker domain to transiently consume its value, circumventing permission checks. LVI causes the fault in the victim domain, making the victim transiently consume a value from the attacker, *i.e.*, LVI transiently injects data into a victim.

Recent processors mitigate Meltdown-type attacks in silicon [27], e.g., Comet Lake processors have no known Meltdown-type vulnerability. As hardware defenses for Meltdown-type attacks in general also mitigate the corresponding LVI attacks, attackers cannot inject arbitrary data into the victim domain. However, on several microarchitectures, the hardware defense, instead of returning a value from the victim domain, only zeroes out the value [7, 60]. While this prevents data leakage, it can be used as a side channel to detect whether an address is valid, e.g., to break KASLR [7]. Even worse, this remains exploitable in an LVI attack variant, namely LVI-NULL [60]. With LVI-NULL, the attacker can still inject '0' values into the victim domain.

The LVI paper showed the dangers of LVI-NULL in an AES-NI attack, but the proposed defenses for LVI-NULL have not been thoroughly evaluated [60]. While the software workarounds for LVI also prevent LVI-NULL, they are costly, and Intel suggests that developers "should determine the level of software hardening that their environment requires, based on risk analysis and an evaluation of the performance impacts of mitigation" [29]. Potentially, every load instruction can suffer from a fault, requiring memory fences for such instructions [60]. This also includes replacing certain instructions, e.g., the return instruction, with sequences of other instructions [29, 60]. The worst-case overhead for these mitigations on real-world workloads is between factor 2 and 19 [38, 60]. This raises the question whether these prohibitively expensive defenses are still required on processors with hardware mitigations against LVI just to defend against LVI-NULL.

In this paper, we propose a lightweight mitigation tailored to LVI-NULL in SGX. Our mitigation, LVI-NULLify, is built on a systematic analysis of LVI-NULL variants, yielding new insights on the attack building blocks of each variant. In particular, we identify that four out of six variants rely on pointer

redirection to the null page. Based on our analysis and experimental validation, we discover that LVI-NULL mitigations proposed by Van Bulck et al. [60] are not effective.

The idea of LVI-NULLify is to offset all memory accesses performed within the SGX enclave relative to the start of the enclave's memory region. To implement this property, LVI-NULLify repurposes segmentation. Segmentation is a fast legacy hardware mechanism on x86 that is used during address translation for every memory operation. The first part of LVI-NULLify is a compiler extension, which generates only segment-relative data loads. Consequently, any '0' injection only loads data from the start of the enclave's memory region, which is under full control of the SGX enclave. The second part of LVI-NULLify is a modified SGX-SDK that maintains interoperability with the untrusted userspace program.

The security of LVI-NULLify relies on special preparation of the enclave's memory region, mitigating transient injection of arbitrary values. LVI-NULLify marks the first pages in the enclave's memory region as non-executable. Transiently executing non-executable memory leads to an immediate stall, preventing any attack. LVI-NULLify also marks these pages as non-readable. We empirically validated that this immediately stalls the load and dependent instructions.

In our evaluation, we show that LVI-NULLify is extremely lightweight, with runtime overheads below 10% in the worst case. This is substantially faster than previous defenses against LVI with a prohibitive overhead of 1220% in the worst case in our tests. The memory overhead of LVI-NULLify is around 21.5% on the code size due to the generation of instruction sequences that explicitly use segmentation. We illustrate that our mitigation is a practical solution to protect SGX enclaves on hardware vulnerable to LVI-NULL but not LVI.

To summarize, we make the following contributions:

1. We systematically analyze and categorize LVI-NULL variants, revealing common attack requirements, and insufficiencies of previously proposed defenses.
2. We propose, LVI-NULLify, a novel lightweight defense against LVI-NULL in SGX, repurposing segmentation in a peculiar fashion.[1]
3. We evaluate the security and performance of LVI-NULLify. We demonstrate that SGX enclaves on the LVI-fixed Comet Lake CPU are only secure with our defense. We observe a performance overhead below 10%.

Outline. Section 2 provides background. Section 3 details our threat model. Section 4 systematically analyzes LVI-NULL variants. Section 5 presents the design and implementation of LVI-NULLify. Section 6 evaluates its security and performance. Section 7 discusses limitations. Section 8 concludes.

[1] We open source LVI-NULLify and provide an anonymized code repository to the reviewers: https://github.com/lvi-nullify/LVI-NULLify.

2 Background

2.1 Transient-Execution Attacks

Transient-execution attacks [8] are a new class of attacks that exploit so-called transient instructions, *i.e.*, instructions that are executed but never retired, to leak sensitive data. Kocher et al. [34] introduced the first sub-class with Spectre, while Lipp et al. [41] introduced the second with Meltdown. While Spectre attacks exploit control- or data-flow predictions made by the hardware, Meltdown exploits the deferred permission check when accessing memory from a different security domain. This deferred permission check allows the out-of-order execution to encode the normally inaccessible data in the cache from where the attacker then extracts it. Subsequent work showed additional variants in both sub-classes [4, 8, 18, 33, 42, 44, 52, 53, 57, 69]. Additional work has summarized the state-of-the-art of both transient-execution attacks [5, 8, 70] and defenses [6, 8].

2.2 Load Value Injection

Load Value Injection (LVI) turns Meltdown around by exploiting faults in the victim [60]. Thus, instead of leaking values, LVI injects values into the transient execution of the faulting victim. For LVI, the attacker prepares a microarchitectural buffer, e.g., the store buffer or L1, by filling it with the values that should be injected into the victim. Then, the victim has to suffer a fault or a microcode assist when fetching data from memory to transiently use the values injected by the attacker. The execution of subsequent instructions with the injected value is then exploited to either encode secrets in the microarchitecture or hijack the control or data flow. Similar to Spectre, LVI requires the gadget to be in the victim and has to additionally induce a fault or assist in the victim.

For unmitigated processors, the state-of-the-art solution for LVI is to insert `lfence` instructions after memory loads [29]. These fences ensure that faulting loads retire before the next instruction, effectively stopping all variants of LVI. However, this type of software mitigation comes with a performance penalty between factor 2 and 19 [38, 60].

2.2.1 LVI-NULL

Starting with the Cascade Lake microarchitecture, Intel processors include in-silicon mitigations against Meltdown, Foreshadow, and MDS attacks, including LVI [27]. These mitigations prevent non-zero value injections through all currently known buffers. However, this mitigation only prevents the attacker from injecting attacker-controlled data. Instead of stalling, faulting loads still transiently forward '0' to dependent instructions [7, 60]. Hence, by inducing a fault in the victim domain, an attacker injects the constant value '0' into the transient execution of the victim. This variant of LVI is called LVI-NULL. Even injecting '0' can be exploited to great

effect, e.g., to transiently inject round keys consisting entirely of '0' into AES-NI computations [60].

The Comet Lake series represents Intel's latest SGX-enabled generation available for both mobile and desktop workstation models that is affected by LVI-NULL [27]. Ice Lake processors based on the Sunny Cove architecture appear to be unaffected by LVI-NULL [27].

2.3 Intel SGX

To provide processor-level isolation and attestation for secure enclaves, Intel developed Software Guard Extensions (SGX) [12]. By design, SGX assumes that only the processor is trustworthy. Hence, an attacker can have full control of the operating system while still being within the threat model.

When a secure enclave is run, it is placed in the virtual address space of an untrusted user-space process. While the operating system is untrusted, it is still responsible for maintaining the virtual-to-physical address mappings. Naturally, this would make the enclave vulnerable to address remapping attacks [12]. To prevent these, SGX maintains its own shadow entry in the Enclave Page Cache Map (EPCM) containing the expected virtual address and the permission bits (R-W-X) for each valid enclave page. In case an illegal virtual-to-physical mapping is encountered, an EPCM page fault is raised.

Although side-channel attacks are not in scope of the SGX threat model, previous work showed that powerful side-channel attacks can be mounted against SGX. A root attacker can still mount low-noise side-channel attacks through the cache [2,43,50], page-table accesses [62,64,71], interrupt timing [63], or branch predictors [14, 19, 40]. SGX is also vulnerable to transient-execution attacks [9, 52, 59, 65] and Intel has released microcode updates to protect against them [23, 25].

2.4 Virtual Memory and Segmentation

In modern systems, virtual address spaces are used as an abstraction and to isolate processes. Hence, they are natively supported by the hardware. Each process works in its own, largely non-overlapping, virtual address space and cannot unintentionally interfere with the memory of another process. The used virtual addresses need to be translated to the corresponding physical addresses using a multi-level page translation table. The location of the table for the current process is indicated by a dedicated register and is switched by the operating system upon a context switch.

Another concept besides paging is segmentation. The idea is to have a set of segments for different uses, e.g., code, data, stack. While older processors used segmentation to enable the use of more physical memory, newer ones mainly use it as a protection mechanism.

Segments are configured via segment descriptors that are located in memory and are then used in conjunction with paging. Each segment descriptor has a base address and a limit. During the address translation, the CPU adds the base address to the segmented virtual address, yielding a non-segmented virtual address. Some instructions use segments implicitly (e.g., `push` and `pop` with the stack segment), and code fetches are implicitly performed via the code segment. Data segments can be used explicitly with memory referencing instructions.

On modern systems, paging has completely replaced segmentation for virtual address translation. Consequently, processor manufacturers removed this feature in the 64-bit long mode (IA-32e) for all segments but `fs` and `gs`. All but these two segments are now required to have a base of 0 and the maximum possible size. The segments `fs` and `gs` still support base and limit as they are broadly used to implement thread local storage for user threads and core local storage in operating systems. Hence, to use the base and limit feature of segmentation on 64-bit systems, user-level software has to use instructions that use `fs` or `gs`, and the operating system has to set up `fs` or `gs` with a base and a limit.

2.5 Object Relocations

Relocations are an essential part of the ELF file format [1, 13]. If a symbol is referenced inside an object file, the linker or the dynamic loader has to resolve the symbol's address and replace all the occurrences of this reference with the real symbol address. The relocation type specifies how this address should be calculated and which symbol is referenced.

SGX enclaves behave similarly to dynamic libraries and can be loaded on arbitrary addresses inside the main program's virtual address space. Therefore, enclaves and dynamic libraries need a mechanism to adjust addresses inside the image to point to the desired position in the address space. The most common way to achieve this is by using relative addressing, where all the absolute addresses inside the library are calculated over the instruction pointer. This type of relocation can be resolved during linking of the dynamic library.

In contrast to relative addressing, dynamic libraries also support absolute addressing where the dynamic loader resolves the addresses after the base address where the image is loaded is known. Here, the loader replaces placeholders inside the dynamic library with the real symbol address.

3 Threat Model

Hardware. For our mitigation, we assume a current or future Intel processor with SGX that mitigates LVI in hardware but does not prevent LVI-NULL, such as, e.g., the Comet Lake microarchitecture. We assume that there are no Meltdown-type transient-execution attacks [41, 52, 59, 65] that directly leak data from enclaves. Moreover, hardware vulnerabilities such as Rowhammer [16, 30, 32] or undervolting [31, 45, 47] are out of scope. We also assume that Spectre-type attacks [8, 9, 34] are either mitigated in hardware, firmware, or software. Additionally, we assume hyper-

threading to be disabled. The Intel SGX Attestation Service indicates whether hyperthreading is enabled, so the verifying party can enforce its status.

Software. We assume a privileged attacker that is explicitly within the scope of the Intel SGX threat model. For the enclave, we assume that it is not vulnerable to traditional side-channel attacks, such as cache attacks [2,43,50] or controlled-channel attacks [71]. We assume that an attacker can start the enclave as often as required and thus rely on precise execution control, such as single- or zero-stepping [62]. Bugs in the enclave, e.g., synchronization problems [49,66], or missing validations on the ABI or API level [61], are out of scope.

We consider only 64-bit enclaves, since enclaves can be (cf. Section 2.4) attacked via 32-bit segmentation [17], but not via 64-bit segmentation due to differences in the behavior.

Takeaway: Our mitigation targets 64-bit SGX enclaves on CPUs vulnerable to LVI-NULL, but not vulnerable to LVI.

4 Detailed Investigation of LVI-NULL

In this section, we first investigate the prevalence and impact of different LVI-NULL scenarios, and their applicability to SGX. We then examine the overhead and efficacy of current and proposed mitigations.

4.1 LVI-NULL Categorization

We distinguish control-flow and data-flow attacks (cf. Figure 1).

Control-flow Attacks. In control-flow attacks, the instruction pointer is transiently redirected in a way that serves the attacker. Again, we distinguish two cases: direct code redirection (❶) *to* the null page, or indirect redirection to arbitrary locations (❷ and ❸) *via* the null page. Direct redirection (❶) is achieved by faulting the load that reads the call target, thus injecting '0' and redirecting code execution to the null page. In contrast, arbitrary redirection allows code execution anywhere in memory if the null page is attacker-controlled, e.g., for Intel SGX enclaves (cf. Section 2.3). It applies to indirect jumps (❷), which load their targets from memory via at least one indirection. Faulting the second to last load causes the jump target to be loaded from an offset in the null page, which allows arbitrary redirection. A special case of indirect redirection are sequences like `pop rsp; ret`, which load the stack pointer from memory and then return. This allows an attacker to set up a transient stack (❸) on the null page by faulting the stack pointer load, and perform a well understood ROP [46,55] attack from there.

Data-flow Attacks. Data-flow attacks inject data into the victim's execution. We distinguish between direct (❺) and indirect (❻) loads, which allow the injection of either '0' or arbitrary values. Van Bulck et al. [60] showed that injecting '0' into the hardware AES-NI key schedule leaks the full key. A special case are binary branches and switch statements (❹), which can be compiled as jump tables. Here, data-flow manipulation changes the control flow, but only to the available branches.

Limitations. While LVI-NULL attacks are possible to execute from user space, several significant limitations apply. First, user-space attackers cannot manipulate page tables directly. This prevents these attackers from arbitrarily causing assists or faults on targeted loads. Secondly, most operating systems do not allow mapping of the null page by default. Both Linux and Windows require privileged access to map it, limiting the user-space attack surface to two cases (❹ and ❺). The exploitability of direct load '0' injection (❺) depends on the targeted algorithm, and is thus best mitigated by developers themselves. Manipulating regular branches with '0' injections (❹) is similar to Spectre variants and can be mitigated the same way.

4.2 Control-flow Injection

C/C++ compilers, such as GCC and Clang, commonly emit code patterns containing jump instructions whose target depends on an address or value loaded from memory. In our analysis, we found 3 categories of such jumps that are potentially susceptible to LVI-NULL.

Case 1: Virtual Function Calls in C++ (❶ and ❷) When objects in C++ call a virtual function, it is not known at compile time which function is being called. To solve this, each object has its own table (vtable), which contains the location of its virtual functions. Because the location of a dynamically allocated object itself (and thus its vtable) is also not known at compile time, calling a virtual function requires at least 2 loads. This creates 2 possible points of injection. First, the attacker may inject '0' when the target is read from the vtable. This load may be generated by an indirect call instruction or a `mov` before a direct call, and can transiently redirect execution to the null page (❶). Secondly, the attacker can inject '0' one load earlier, *i.e.*, when the address of the vtable is read. This causes the null page to act as the vtable, allowing transient redirection of execution to any location (❷). As the offset in the vtable is known at compile time, it is compiled to an immediate value that cannot be manipulated by LVI-NULL.

Exploitable in SGX enclaves? Very likely.
Indirect function calls (❷) occur frequently and are almost always immediately exploitable, as they allow redirection to any suitable gadget.

Case 2: Global Offset Table (❶) Another potentially interesting case is the global offset table (GOT), which enables programs to use functions in dynamically linked libraries. Unlike vtables, the GOT is always at a known location, and so

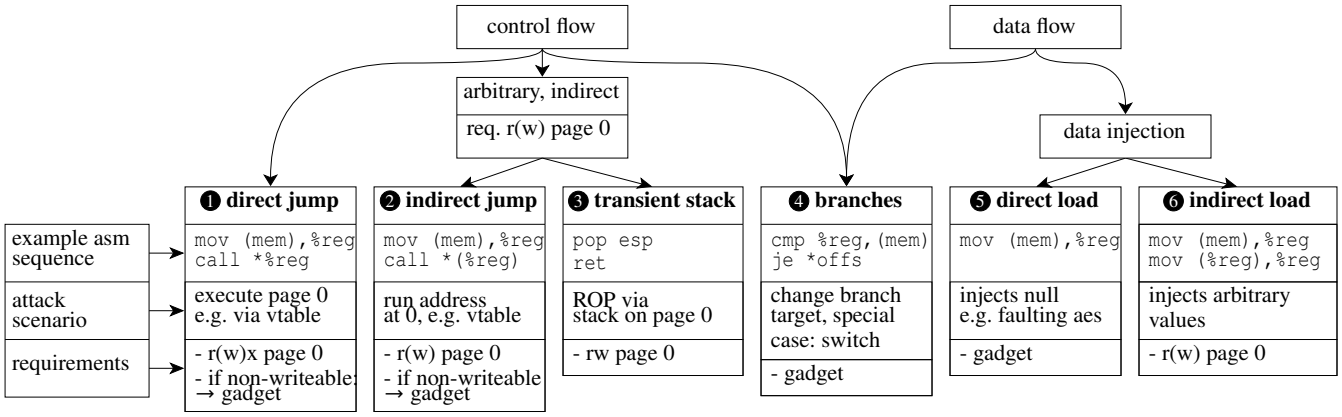


Figure 1: Categorization of LVI-NULL into control-flow and data-flow attacks. Subcategories list the different attack vectors and example assembly sequences for each (in AT&T syntax), the attack scenario, and their requirements.

only the call target is loaded. After the initial dynamic relocation of the symbols in the GOT, this only creates the potential to transiently redirect execution to address 0x0.

Exploitable in SGX enclaves? No.
While direct function calls (❶) are frequent, they are not exploitable in SGX.

Case 3: Switch Statements (❹) For certain switch cases, compilers generate a jump table, first loading the variable in question, and then looking up the corresponding jump target. This only applies to switch variables loaded from a single memory location, and not derived calculations. In position-independent code (PIC), this creates 2 attack points: injecting '0' into the variable itself, or injecting '0' when the jump target is calculated. The former transiently leads the switch into the '0' case, executing code there as if the variable were '0'. The latter causes the program to jump into the data section instead, as both GCC and Clang load offsets relative to the jump table. These offsets are likely not valid code, and furthermore, as Canella et al. [8] showed, the executable bit is respected in transient execution, so this injection is not exploitable here. When compiled as non-relocatable (no-pic), execution can again be redirected to the first case. Additionally, it can now be redirected to address 0x0, as the jump table contains absolute addresses, which can be zeroed on load (❶).

Exploitable in SGX enclaves? Unlikely.
Similiar to Spectre-PHT [8], exploitability is highly dependent on the specific case, but case-0-injections can be prevented reliably (see Section 5).

These are the three cases of commonly used code we found to enable control-flow injection. We expect there are more cases in other code patterns, compilers, or languages. However, for SGX, LVI-NULLify copes with all types of control-flow injection, as all control-flow injections rely on the null page. Table 1 explores the prevalence of such gadgets in standard SGX code.

4.3 Data Injection

Data injection gadgets are simply direct or indirect loads from memory, and as such, they are ubiquitous in all programs. Van Bulck et al. [60] have shown that in some cases, even '0' injections can be exploited to great effect. However, in cases where data injection does not lead to changes in control flow, it depends entirely on the algorithm at hand whether it can be exploited. As a direct '0' injection (❺) cannot be mitigated by software changes short of adding a load-serializing instruction after all potentially problematic loads, we do not consider this case. Instead, we leave it to the authors of software to guard their critical computations, such as cryptography, with this possibility in mind. However, in Section 5, we propose a way to prevent arbitrary data injection via indirect loads (❻).

Exploitable in SGX enclaves? Likely.
The danger of transient data injection depends on the targeted algorithm, but arbitrary value injection provides high flexibility for exploitation.

4.4 Applying LVI-NULL Variants in SGX

Of the attack vectors presented in Section 4.1 (cf. Figure 1), 4 out of 6 require at least read access to the null page. Variants ❹ and ❺ have no particular requirements and apply in any case. Most modern operating systems do not map the null page by default and typically require root privileges to do so [11]. Since the purpose of SGX enclaves is to protect against malicious or compromised operating systems, their threat model currently allows attackers to use the null page as they see fit.

Gadget/File	QE	LE	PCE	PVE	trts	tsdc	tcxx	tcmalloc

`mov (mem), %reg`
`mov (%reg), reg`
`call *reg`
	216	0	123	216	0	0	0	4

`mov (mem), %reg` `call *(%reg)`	0	45	0	0	0	0	19	3
`mov (mem), %rsp`	0	2	0	0	1	0	1	0

`mov (mem), %reg`
`mov %reg, %rsp`
`ret`
	1	1	1	1	1	0	0	0

`mov (mem), %reg1`
`mov %reg1, %reg2`
`mov %reg2, %rsp`
`ret`
	0	0	0	0	0	0	0	0

`pop %reg`
`mov %reg, %rsp`
`ret`
	0	0	0	0	0	0	0	0

Table 1: Number of control-flow gadgets found in Intel's prebuilt (quoting, launch, provisioning) enclaves and SDK libraries. Search was limited to instruction sequences with fewer than 10 separating instructions.

From within an enclave, all memory of the user-space process is available for reading and writing according to its page-table entries, as it would be to the process itself. This implies that variants ❷, ❸, and ❻ apply fully if the null page is writable or with limitations, if it is not. Variant ❶, however, requires the null page to be executable. Van Bulck et al. [60] experimentally found that code outside of enclave memory is not executable from within an enclave, even during transient execution. This was later confirmed by Intel [29], and we have reproduced this result as well. It follows that the only way to execute instructions at address 0x0 is to load the enclave itself starting at the null page. Since the Intel SDK does not build enclaves with execute permissions on this page [29], we consider variant ❶ *not exploitable in SGX* enclaves.

To evaluate the prevalence of assembly sequences that allow LVI-NULL types ❷ and ❸, we search several prebuilt- and SDK-generated binaries for a limited selection of exploitable assembly patterns. As Table 1 shows, indirect calls (❷) are plentiful in these binaries, though they are currently mitigated by `lfence` instructions. We find that there are even some gadgets for variant ❸. An especially interesting observation is that the original transient stack gadget, as described by Van Bulck et al. [60], is still present in unmitigated form in the prebuilt launch enclave for Linux provided by Intel as of SDK release 2.11.

An analysis of how some code patterns generate vulnerable instruction sequences is shown in Section 4.2.

Takeaway: ❷, ❸, ❹, ❺, and ❻ are all feasible in SGX. ❶ is feasible, but mitigated by default.

4.5 Current and Proposed Mitigations

In this section, we discuss the two main types of mitigations against LVI and LVI-NULL for SGX.

4.5.1 Memory Fences

The officially suggested mitigation against LVI is to stop transient execution before it can be exploited. Similar to the mitigations for Spectre [24], Intel also suggests to use memory fences for aborting transient execution [29]. As it is infeasible to add memory fences manually, these memory fences are supposed to be emitted by the compiler. With the publication of LVI [60], Intel has provided 2 levels of mitigation [28, 29], and Google engineer Zola Bridges another [3, 37]:

Control-Flow Mitigation. This mitigation replaces `ret`, `call`, and `jmp` instructions by fenced alternatives. It protects transient control-flow redirection at the cost of effectively disabling all control-flow predictors. However, it does not generally protect against value injection and only prevents these special cases. Compilation options: `-mlvi-cfi`

SESES. "Speculative Execution Side Effect Suppression" aims to prevent more than just LVI by adding an `lfence` instruction before every instruction that operates on memory. This approach fully mitigates LVI, LVI-NULL, and other transient execution attacks. Compilation options: `-mseses`

Optimized Cut. In addition to CFI, this mitigation for LVI (which we call "optimized cut") tries to separate loads from potential transmit gadgets by analyzing the control-flow graph of applications. Hence, the compiler can insert far fewer `lfence` instructions than SESES while still providing the same security guarantees w.r.t. LVI. Compilation options: `-mlvi-hardening -mllvm -x86-lvi-load-opt-plugin=OptimizeCut.so -x86-experimental-lvi-inline-asm-hardening`

While these three levels of mitigation differ in the amount of `lfence` instructions (cf. Table 2), they all incur heavy performance penalties in the range of factor 2 to 19 [38, 60].

4.5.2 Page Table Protections

Van Bulck et al. [60] also proposed specific mitigations for LVI-NULL. To prevent execution of the null page (❶), they suggest marking the first page in an enclave as non-executable or placing an infinite loop at the base of the enclave image.

As described in Section 4.4, marking a page non-executable indeed prevents execution in the transient domain. Experiments on our i5-10210U show that this holds even if the OS marks a page as executable after loading the enclave. Read, write, and execute permissions are also stored with the expected virtual address in the protected Enclave Page Cache Map (EPCM) entries. Our experiments suggest that in transient execution, the CPU considers the permission bits of both the page table and the EPCM and applies whichever is less permissive. As this prevents ❶, an infinite loop or similar is not necessary.

To stop transient null pointer dereferences (❷, ❸, and ❻), Van Bulck et al. [60] suggest marking the null page as uncacheable. This has also been proposed as a possible mitigation for Spectre attacks [51], as uncachable memory cannot

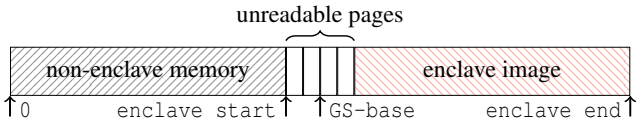


Figure 2: Memory layout of enclaves protected with LVI-NULLify.

be read during transient execution. Any transient access to uncachable memory simply stalls [51]. While this would indeed prevent loads from this page, the OS can simply change these flags at any time, as they are not protected by SGX. We verified that in contrast to the read, write, and execute permissions, the memory type is not enforced by SGX and can be manipulated to mount an attack. Additionally, injection via the shared line-fill buffer is possible on some architectures [44,51,52,65] if hyperthreading is enabled.

Takeaway: Current mitigations are too costly or are insufficient.
All LVI-NULL variants are preventable by LVI mitigations, but incur substantial performance degradation. Other proposed mitigations are only partially effective.

5 LVI-NULLify

Previous mitigations have been designed primarily for LVI, not LVI-NULL. Hence, they mitigate attacks that are already mitigated more efficiently in current and future Intel processors, e.g., the recent Comet Lake microarchitecture. Following the analysis of Section 4.5, we can see that these previous mitigations either have a substantial performance overhead or are limited to only certain variants of LVI-NULL. This motivates the need for a defense that is more tailored to LVI-NULL. In this section, we present LVI-NULLify, our mitigation for LVI-NULL affected hardware that achieves a better balance between performance cost and remaining attack surface than previous LVI mitigations. The worst-case overhead on our LVI mitigated Comet Lake is only $\approx 9\%$, our older LVI-vulnerable Coffee Lake-R reaches a maximum of 36% overhead.

5.1 LVI-NULLify Design

LVI-NULLify aims to prevent all LVI-NULL variants in our threat model (see Section 3). This includes variants ❶, ❷, ❸, and ❻. Though we categorize switch expressions as a subclass of branches (❹) in Section 4.1, we briefly describe a mitigation in Section 5.1.4 that is also applicable outside of SGX. Since all other variants involve the null page, the central feature of LVI-NULLify must be to control either its contents or accessibility. Therefore, we devise a way to effectively *move the LVI-NULL target page into the enclave*, even if address 0x0 is not in the enclave's linear address space.

SGX does not currently offer any control over pages outside of the enclave memory range. Hence, loading an enclave anywhere but page null means giving up control of the null page. As multiple enclaves can and often need to be loaded simultaneously, only loading an enclave if it is mapped at address 0x0 is not a practical option. Our solution is to *offset every memory load in the enclave* such that any pointers that are loaded from memory are added to an immutable constant. For this constant, we use the virtual base address of the enclave image. The resulting memory layout is shown in Figure 2. Even if an address load faults and transiently returns '0', the resulting load address is still within the enclave. This puts the control over the pages targeted by LVI-NULL into the hands of the enclave, regardless of where an attacker maps it.

5.1.1 Using Segmentation

To offset loads on commodity Intel CPUs, we rely on segmentation. In 64-bit mode, segments typically have to start at address 0x0. However, the GS and FS segment registers are an exception and can have non-zero base addresses. Conveniently, the `EENTER` and `ERESUME` instructions automatically set these base addresses to the enclave base address plus a developer-controlled, positive offset. As the GS and FS registers are set on each entry, and the offsets are stored in enclave memory in the thread control structures (TCS), these values are inaccessible to the OS. They are also part of the enclave's attestation, preventing manipulation at load time. The Intel SGX-SDK sets both registers to the same value, creating an unnecessary redundancy. We can thus repurpose one of the two segment registers, in our case GS, for LVI-NULLify.

Since address calculation with segment bases is an integral part of x86 hardware, there is no noticeable slowdown for GS-relative loads, as we experimentally verify in Section 6. We set the GS base to the beginning of the enclave, such that data loads in our enclave are now relative to the beginning of the enclave. This means that we essentially build a non-relocatable object (no-pic) that gains its position independence by adding GS base to all addresses.

Applying this mitigation to generic SGX enclaves requires the modification of 3 components: the compiler, the Intel SGX-SDK, and the Intel Platform Software (PSW). The compiler has to emit GS-relative loads for all memory-load instructions. We discuss how this is implemented in LLVM in Section 5.2. Section 5.3 then details the custom relocations that are necessary after compilation. Changing all load instructions also implies that all addresses inside the enclave are invalid outside and vice versa. Hence, this necessitates an automatic pointer conversion from the trusted runtime system (tRTS) to the untrusted runtime system (uRTS). Additionally, the tRTS itself needs to be built with our compiler modifi-

cation. We detail the necessary modifications to the SDK in Section 5.3.2. As the GS offset is in part calculated by the PSW, we need to enable it to distinguish between enclaves mitigated by LVI-NULLify and unmitigated enclaves. The changes to the PSW are also described in Section 5.3.2.

5.1.2 First Enclave Pages

With GS-relative addressing, most transiently faulting indirect loads read from the first page of the enclave. For this reason, it is critical that reading from it does not return values that can be used in an attack. To prevent variant ❶ of LVI-NULL, which requires code execution, simply making this page non-executable is sufficient. Additionally, marking the first page non-readable in the EPCM entry prevents all variants that load from this page (❷, ❸, and ❻). Transient accesses to such pages simply stall, as we have experimentally confirmed on several CPUs, including our i5-10210U Comet Lake. This stops all further dependent accesses. However, the first page in an enclave may contain data that the dynamic loader inside the enclave needs to access. We, therefore, shift the enclave image and prepend empty pages that are neither readable nor executable. The amount of such pages that are needed depends on the enclave program; loads with offsets, where the base address can be zeroed, may transiently load more than 1 page from the GS base address or even below it. Negative offsets could, therefore, lead to loads that escape the enclave. Fortunately, the size of all immediate offsets is known at compile time. We can determine a safe amount of empty pages to map before and after the GS base after compilation. An example with 2 pages on each side of the GS base address is shown in Figure 2.

The residual attack surface to this approach are dynamic arrays. When neither position nor size are known at compile time, loads can take the form *base* + *offset*, where both base and offset are loaded from memory. If an attacker faults the base, the transient load target is potentially anywhere from the enclave beginning up to the size of the dynamically allocated structure.

5.1.3 Alternative Approaches

There are other, less peculiar mechanisms to prevent faulting loads from reaching the null page. We want to examine two candidates here, as their shortcomings are not immediately apparent.

First, we could simply add the base address to all loads by converting every load to use complex addressing. For this, we cannot load the base address from memory, as this load could again be zeroed. A solution would be to always keep the base address in a CPU register, but this reduces the number of registers available to the compiler. This can incur significant performance impacts as less efficient code can be generated. Alternatively, we could use memory fences for these loads, which would again result in large performance overheads. Additionally, any load that already uses complex-addressing needs an additional offset computation beforehand.

The second approach would be to employ immediate addresses by using the dynamic loader to relocate all symbols. An undesirable side effect of this method is the need to set the text section of the enclave to writable and executable. As enclave pages cannot currently change access permissions at runtime, this would weaken the enclave's protection against traditional exploits that might otherwise not be exploitable.As x64 does not generally support 64-bit displacement in complex addressing [21], this is also not a possibility. Only `AL`, `AX`, `EAX`, and `RAX` may be the source or target of immediate 64bit-addressed memory loads, which introduces the need for more intermediary registers [26].

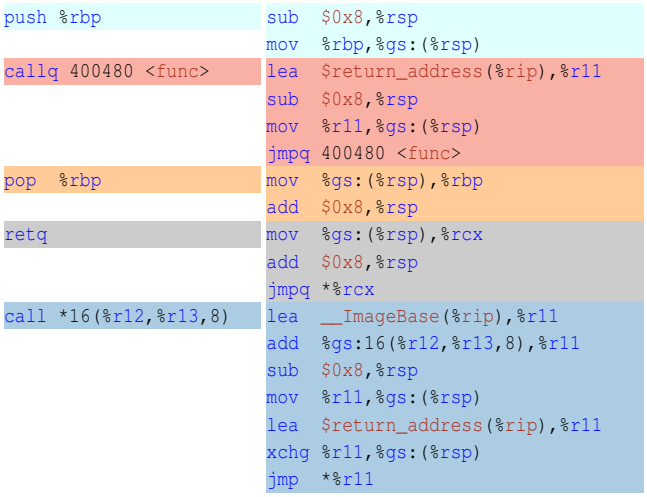
5.1.4 Mitigating Switch Statements

Regular branches (❹) can be mitigated with Spectre-PHT mitigations, e.g., speculative load hardening. Switch statements present a special case of branches. On processors with hardware mitigations for branch target injection, e.g., single thread indirect branch predictors (STIBP) [22], such as the Comet Lake series, switch statements are protected from cross-hyperthread manipulation through the branch target buffer. However, they are still vulnerable to LVI-NULL when the compiler generates a jump table (see Section 4.2, case 3). We can mitigate the case of single-variable conditions (conditions that only depend on one in-memory variable) by targeted insertion of `lfence` instructions. This is described in more detail in Section 5.2. It is the only variant we mitigate that can also be exploited outside of SGX, and this mitigation can also be applied alone.

5.2 Compiler Changes

We developed an open-source implementation of the compiler-part of LVI-NULLify that is based on the LLVM compiler framework [39]. It handles the insertion of fences in switch statements (cf. Section 4.2) and ensures that every load is relative to the GS segment (cf. Section 5). Our modification consists of two new passes, one module pass (*i.e.*, a transformation pass) that works on the LLVM intermediate representation (IR) and one machine function pass in the x86 backend. The mitigation for switches is purely done in the pass on the IR, while the GS-relative addressing is implemented in the backend pass. For our modification, we added 1024 lines of code to the LLVM code base (24 in 10 existing files, 1000 in 3 new files).

Switches. When compiling an application with optimizations enabled, LLVM already tries to optimize the performance of switches [35, 68]. These optimizations are implemented as a transformation pass that iterates over all functions in the translation unit. If a switch is encountered, the pass tries to



(a) unmodified (b) modified

Figure 3: Figure 3a shows the unmodified assembly instructions containing implicit loads while Figure 3b shows the instruction sequence with which they get replaced by our modified compiler.

apply these optimizations. Unfortunately, this transformation pass is only executed when the application is compiled with at least optimization level 1. Hence, we cannot use it and need to implement a new pass.

We extend LLVM with a new module pass (*-flvi-null*) that iterates over all functions in the translation unit and searches for a switch within the basic blocks that comprise the function. If such a switch is found, the mitigation is applied. If the switch already contains a '0' case, the compiler simply modifies the case such that the first instruction within the basic block is a fence instruction. Otherwise, LVI-NULL falls back to the default case of the switch. Either one can lead to exploitable behavior. To prevent this, the compiler inserts a new '0' case that contains a fence instruction and then performs an unconditional branch to the default case. This new '0' case is only ever executed if the default case is supposed to handle zero values, or if an attack takes place. This significantly reduces the performance impact.

By considering these two cases, the compiler part of LVI-NULLify can mitigate LVI-NULL targeting switches. This mitigation is not specific to SGX and does not require segmentation, hence this can also be used in non-enclave applications. While this mitigation on its own is not sufficient to fully prevent LVI-NULL, as it only mitigates a subset of variant ❹, it is a necessary building block for LVI-NULLify.

GS-Relative Addressing of Loads. As discussed in Section 5, every load, explicit or implicit, has to be relative to the GS segment. Thus, in case of an LVI-NULL attack, the control flow is re-directed to a location that is controlled by the SGX enclave. To achieve this, we add a new machine function pass in the x86 backend of LLVM.

In the machine function pass, we iterate over each instruction of a given machine function, and replace explicit loads, implicit loads from pushing to and popping from the stack, as well as calls, jumps and returns, as all of these loads are also exploitable by LVI-NULL (❶, ❷, ❸, and ❻). Hence, the transformation pass replaces each such instruction with an equivalent sequence of instructions that use GS-relative addressing where necessary. Figure 3 shows some cases that we consider for implicit loads and how our modified compiler mitigates them.

As jump and call instructions cannot use the GS segment, and the CS segment cannot be changed (see Section 2.4), we manually convert the relative call address of, e.g., an indirect call, to an absolute address by adding the image base. To protect this pointer conversion from LVI-NULL, we use a rip-relative `lea` instruction to calculate the image base instead of loading the address from memory. Hence, we ensure that all pointers inside the enclave are relative to GS regardless of their data representation.

As our pass is run as the last pass before the actual code is emitted, no additional load can appear that is not GS-relatively addressed. With these compiler modifications, and in combination with the further LVI-NULLify components, we successfully mitigate LVI-NULL, as we show in Section 6.

5.3 Relocation and SGX-SDK Changes

In addition to the compiler changes (cf. Section 5.2), we require additional changes in the relocations of the object files (cf. Section 2.5) and changes in the SGX-SDK and SGX-PSW to realize the GS-relative addressing. We discuss these changes in this section.

5.3.1 Relocation Types

The generated instructions from our compiler pass do not use the instruction pointer for relative addressing, and therefore, the compiler emits absolute relocations. Since the enclave image is signed by the author of the enclave and the signature is verified during the enclave initialization, absolute address relocations cannot be resolved before verifying the signature [12]. Additionally, enclave memory cannot be modified from outside, so each enclave contains an ELF loader to resolve any relocations during the initialization phase. A disadvantage of this is that the page flags cannot be changed after the enclave is instantiated. Therefore, pages containing absolute relocations must remain writable at runtime, even if they contain text sections. Pages that are writable *and* executable are a traditional security concern. While the enclave signer issues an error message if it finds such text absolute relocations, this error can be suppressed. Therefore, keeping the original page flags without making pages writable is a design goal of LVI-NULLify.

Since we do not need absolute address relocations when using the GS segment to specify the image base, we fully replace these absolute relocations. We build an additional tool to change the relocation types directly in the object files after compilation. LVI-NULLify does not enforce additional requirements on the build environment by reusing the existing relocation types instead of implementing a new one for GS-relative addressing.

The compiler extension emits `R_X86_64_x` absolute relocations as part of the code generation. We then iterate over the generated ELF object file and exchange these absolute relocations with the `R_X86_64_COPY` relocation type. The copy relocation type fills in the relocation destination with the symbol's offset from the image base, exactly what is needed for the GS-relative addressing. The copy relocation's addend is set to the difference between the GS-base and the real enclave base to implement the additional pages in front of the enclave (see Section 5.1.2 and Figure 2), *i.e.*, shifting the address of the relocated symbol.

5.3.2 SGX-SDK and SGX-PSW

For LVI-NULLify to work, some changes to the SGX-SDK and the SGX-PSW are required. The changes detailed here are made mostly to the enclave-loading mechanism and for automated pointer conversion between enclave and host application.

SGX-PSW. The SGX-PSW is used globally for loading all enclaves. For LVI-NULLify, the PSW has to set the GS segment in the thread-control-structure template (see Section 5.1.1). This template is used for creating threads inside the enclave, and is used for the attestation process. Hence, the same changes are also required in the SGX signer. We ensure backward compatibility with enclaves that are not protected by LVI-NULLify by indicating the use of LVI-NULLify in the enclave signature structure. Hence, the PSW only modifies the GS segment if LVI-NULLify was used for building the enclave.

SGX-SDK. Most changes in the SDK affect the ECALL and OCALL interface. With LVI-NULLify, the enclave and the host application basically operate in different virtual address spaces. Hence, the ECALL and OCALL interface have to apply pointer conversion. On enclave entry, the GS segment is automatically set by the `ENCLU` instruction. We modify the `enclave_entry` function to convert the stack pointer, the base pointer, and the pointer to the structure used to pass additional data into the enclave. Some minor changes also adapt the elf loader inside the enclave for GS-relative addressing, as some of the supported relocation types refer to the absolute enclave base.

In the SGX-SDK, the *edger8r* application is responsible for parsing the enclave interface definition file and generating the trusted and untrusted part of the enclave interface. To ensure the enclave can use pointers passed to an ECALL implementation without manual modification of the code, we modified the *edger8r* application for the code generation of the trusted enclave API. As this parser already has all the information about functions and their parameter types, it can automatically generate code for converting pointers from absolute pointer addresses to GS-relative addresses. Thus, all the default cases of passing pointers into an ECALL or OCALL are handled automatically.

The enclave definition language also allows the definition of data structures for ECALLs and OCALLs. These structures can be automatically copied into the enclave memory, but nested structures or structures containing additional data over pointers must be copied by hand from the enclave developer [20]. Hence, we also leave pointer conversion for such data types to the enclave developer.

6 Evaluation

6.1 Security Evaluation

For the security evaluation, we first perform a theoretical analysis of all variants in the context of our mitigation. Additionally, we also evaluate our own proofs of concept demonstrating LVI-NULL (see Appendix C). All experiments are run on an Intel Core i5-10210U Comet Lake that is vulnerable to LVI-NULL but not to LVI. In all of our experiments, LVI-NULLify successfully prevents all targeted variants of LVI-NULL.

Variant ❶, direct jumps. If the target of a jump instruction, such as `call` or `jmp`, is loaded from memory, it can be zeroed using LVI-NULL. As a result, transient execution continues at address 0x0, which is either outside the enclave, and therefore not executable in the context of SGX, or on the first page of the enclave, which is also not executable as ensured by the SDK. This behavior stays the same with LVI-NULLify, and is thus *not exploitable*.

Variant ❷, indirect jumps. When the first of the two loads in an indirect jump is zeroed, the jump target is read from address 0x0, plus potentially an offset used in the indirect-jump instruction. This is, e.g., the case for an entry in a vtable (cf. Section 4.2). Without LVI-NULLify, this address points to the virtual address 0x0+offset. This address can be outside of the enclave and thus under attacker control.

With LVI-NULLify, the load is performed with GS base, which ensures that the address is inside the enclave. As a number of pages (that depend on the largest such offset in the enclave) directly after GS base are non-readable, the address load stalls, and no jump occurs. Since function offsets can generally be determined at compile time, the required number of buffer pages can be reported by the compiler. An example would be finding the maximum number of entries in a vtable. This does not consider programs that use 'manually' constructed jump tables. While in rare cases, a dynamic offset could be large enough to reach beyond the allocated

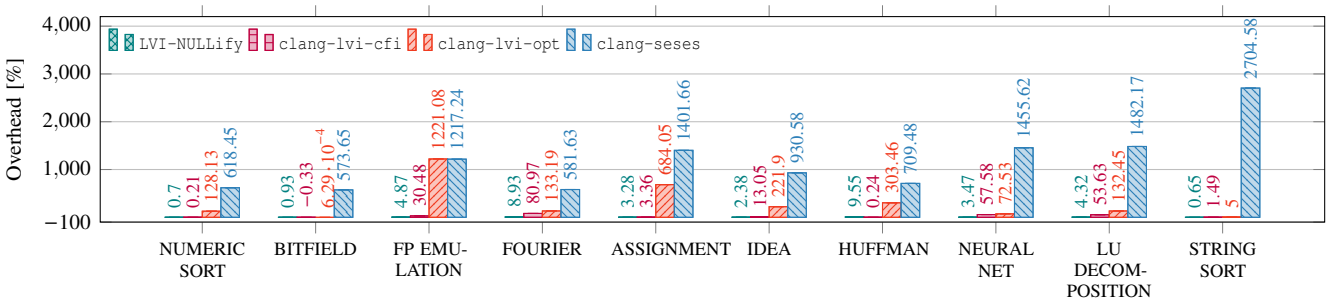


Figure 4: Mean runtime overhead in sgx-nbench [58] on our i5-10210U@1.6GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. N=50, standard deviations vs unmitigated mean plotted, but too small to be visible.

buffer pages, e.g., when manually constructing dynamic jump tables, most cases are prevented, and the remaining rely on very specific circumstances. Hence, *we consider this variant mitigated*.

Variant ❸, transient stack. As shown by Van Bulck et al. [60], function epilogues that load the value of the stack pointer from memory and return can be exploited to transiently use the null page as the stack by zeroing the load. This attack allows for arbitrary code redirection using transient return-oriented programming. LVI-NULLify replaces the return instruction (`ret`) with a GS-relative load and jump (cf. Figure 3). As a result, mounting an LVI-NULL attack moves the transient stack to a non-readable page within the enclave. *This prevents the transient stack attack*, as long as the enclave developer does not actively try to circumvent that, e.g., by loading the stack pointer with an indirect load instruction using a very large offset.

Variant ❹, branches. For regular branches, LVI-NULL behaves very similar to Spectre-PHT and are thus *out of scope* for LVI-NULLify. Developers can mitigate them with speculative load hardening if they choose. As discussed in Section 4.1, switch constructs represent a special case of branches, as they can be implemented as a jump table. Without mitigations, execution can be redirected to case '0', which may also be the default case. LVI-NULLify places an `lfence` instruction in the affected case, thereby mitigating it. As the deciding variable may depend on more than one memory load, *we consider this variant only partially mitigated*.

Variant ❺, direct load. All data loads are still susceptible to direct '0' injection with our mitigation. Thus, an attacker can use LVI-NULL for data-only attacks, e.g., as shown for AES-NI [60]. Since exploitability highly depends on the victim algorithm, mitigation is left to the enclave developer. Cryptographic libraries need to consider different side channels in their implementation already. Variant ❺ becomes one more issue on this list. We therefore consider it *out of scope* for LVI-NULLify.

Variant ❻, indirect load. When values are loaded indirectly, *i.e.*, by loading the target address from memory, arbitrary values can be injected when the first load is zeroed. Similar to ❷, values are loaded from address 0x0 with a possible offset. With LVI-NULLify, the now GS-relative load ensures that this load is inside the enclave's address space. If the offset falls within the non-readable pages at the beginning of the enclave, arbitrary value loading is prevented. Again, offsets are dependent on the program, and most can be statically determined at compile time, which allows adjusting the number of buffer pages accordingly in the compiler. Dynamic arrays of unknown size may still produce transient loads that reach into the enclave memory itself. In these cases, non-zero data injection may still occur. Because most cases are prevented, and the remaining rely on very specific circumstances, *we consider this variant mostly mitigated.*

All told, our analysis suggests that LVI-NULLify prevents the majority of LVI-NULL variants and cases at a significantly lower performance impact than Intel's optimized-cut solution, not to mention SESES. Additionally to this reasoning, we also evaluated our claims with proof-of-concept implementations of the attack variants. Where our proofs of concept were successful without LVI-NULLify, enabling it prevents leakage in all cases.

We have no indication whether the discussed remaining vulnerabilities occur in real-world code. However, C and C++ grant developers vast freedom to implement features in non-standard ways (e.g. manual jump tables for ❷ that our compiler extension is unaware of) which we would not catch and, thus, not mitigate. Therefore, our mitigation bridges the gap between the very expensive optimized-cut mitigation and the less secure control-flow mitigation. When enclaves are not subject to one of the described caveats, our mitigation provides the same level of security as the optimized-cut mitigation at much lower performance cost.

6.2 Performance Evaluation

For the performance evaluation, we first investigate the number of emitted instructions, *i.e.*, the number of `lfence`s and GS-relative loads, of LVI-NULLify and Intel's control-flow and optimized-cut mitigations as well as SESES(cf. Section 4.5 and Appendix B). Our expection is that the number of `lfence` instructions has a direct and significant impact on the performance while GS-relative loads provide better

performance. We substantiate this by benchmarking SGX applications with all of the above mentioned mitigations.

In line with previous work [10, 15, 54, 56, 67, 72], we evaluate the performance of our mitigation based on SGX benchmarks written in C/C++, the *nbench* adaptation for SGX [15, 58] and *SGXBENCH* [48]. As our mitigation is highly specialized for the SGX environment, we can only benchmark SGX enclaves, preventing us from measuring the compiler-introduced overhead for regular benchmarks, such as the SPEC benchmarking suite.

In our setup, we build enclaves with clang 11 and optimization level *O3*. As the optimized-cut mitigation by default does not mitigate the enclave entry assembly, it was compiled with the experimental mitigation for assembly to provide a better comparison with our mitigation in SGXBENCH, which measures enclave-entry performance. We evaluate on an Intel Core i5-10210U (1.6GHz, Comet Lake) and an Intel Core i9-9900K (3.5GHz, Coffee Lake-R). While the i9-9900K is also affected by LVI, it serves as a reference for workstation performance, compared to the mobile Comet Lake. Moreover, this CPU has also been used by Phoronix [38] to benchmark the overhead of Intel's LVI mitigations. We provide these results in Appendix A. All experiments were run on isolated cores with fixed frequencies to reduce the variance of the measured values.

6.2.1 Analysis of Emitted Instructions

Table 2 shows the result for our evaluation of emitted instructions for the two benchmarks as well as three libraries that are essential components of SGX. The SESES mitigation issues the largest amount of `lfences`, *i.e.*, more than 29 700 for libsgx_tstdc.a, which is to be expected as it simply fences every memory read and write that it encounters. Intel's optimized-cut mitigation improves upon this by removing more than 23 000 `lfences`. The control-flow mitigation further reduces this number, down to 1400 `lfences`, but at the cost of reduced security as it does not mitigate all loads. None of these three mitigations issue a significant number of GS-relative loads, *i.e.*, 6 at most. Contrary to the other mitigations, LVI-NULLify issues the lowest amount of `lfence` instructions but the highest amount of GS-relative loads. This change in behavior significantly improves the performance, as our subsequent performance evaluation of the benchmarks shows.

Naturally, due to LVI-NULLify replacing certain instructions with a longer sequence of secure instructions (cf. Figure 3), we expect the binaries that LVI-NULLify generates to be larger than for the Intel mitigations. As Table 2 shows, this is indeed true: in the worst case, we see an increase of 21.5% over the unmitigated baseline.

Software	LVI-NULLify LFENCE / GS / KB	control-flow LFENCE / GS / KB	optimized cut LFENCE / GS / KB	SESES LFENCE / GS / KB
nbench	37 / 11433 / 224(+19%)	319 / 6 / 192(+2%)	3289 / 6 / 200(+7%)	13780 / 6 / 233(+24%)
sgxbench	55 / 4323 / 113(+22%)	231 / 6 / 93(+0%)	1274 / 6 / 97(+4%)	5229 / 6 / 109(+18%)
libsgx_trts.a	4 / 1483 / 109(+10%)	105 / 6 / 102(+3%)	591 / 6 / 104(+5%)	1872 / 6 / 108(+9%)
libsgx_tstdc.a	0 / 23356 / 1322(+3%)	1400 / 0 / 1367(+7%)	6188 / 0 / 1383(+8%)	29754 / 0 / 1454(+13%)
libsgx_tcxx.a	1 / 14916 / 799(+10%)	812 / 0 / 722(-1%)	3353 / 0 / 730(+0%)	17818 / 0 / 775(+6%)

Table 2: We show the number of `lfence` and GS-relative instructions the different mitigation techniques insert and the overall file size in kB (and its change to baseline) for a selection of software, including benchmarks and SGX components.

6.2.2 nbench

The relative performance overhead shown in Figure 4 clearly demonstrates that the strong LVI-NULL mitigation provided by LVI-NULLify comes in at or even below the cost of Intel's control-flow-mitigation, which only covers variants ❷ and ❸. Table 3 contains the benchmark's raw results in iterations per second. We also see that some of the tests, like `String Sort` and `Bitfield`, operate almost entirely on registers, s.t. the overheads do not represent the differences of the mitigations very well. Memory heavier benchmarks like `FP Emulation`, on the other hand, clearly demonstrate the advantage of our mitigation vs. Intel's optimized-cut mitigation. Here we achieve an overhead reduction of 1216 percentage points. As this overhead is more in line with the original results by Van Bulck et al. [60] and Phoronix [38], we consider this to better represent the difference between the mitigations. We also note some benchmarks where LVI-NULLify performs better than the unmitigated reference. We consider this an artifact of cache alignment or similar effects specific to this benchmark and not representative of our mitigation.

6.2.3 SGXBENCH

When compiling the SGXBENCH [48] suite, we found that some loads in the benchmarks are not fenced by Intel's optimized-cut mitigation. For benchmarks that copy memory, this makes the comparison to our mitigation rather uninteresting, as tests show very similar performance. The results for a selection of benchmarks are listed in Appendix A. Two of the benchmarks still provide a useful comparison, *einit/edestroy* and *empty ocall*. They show that at $\approx 0.17\%$ and $\approx 3.3\%$ lower performance, respectively, our mitigation does not introduce any significant slowdown for this basic enclave functionality.

7 Discussion and Limitations

Hardware and Microcode Changes. Ultimately, LVI and LVI-NULL have to be mitigated in silicon, as we can already see from CPUs that are not affected by any LVI variant. However, as it is infeasible to replace all affected CPUs, an intermediate solution compatible with affected CPUs is necessary. Van Bulck et al. [60] suggested the possibility of a

Test/Mitigation	none (σ)	LVI-NULLify (σ)	control-flow (σ)	optimized cut (σ)	SESES (σ)
NUMERIC SORT	723.69 (0.181)	718.67 (0.093)	722.15 (0.061)	317.23 (0.018)	100.73 (0.008)
STRING SORT	70.46 (0.003)	70.01 (0.005)	69.43 (0.003)	67.11 (0.002)	2.51 (0.000)
BITFIELD	316 550 164 (165 589)	313 635 987 (102 467)	317 587 729 (228 815)	316 548 172 (411 084)	46 990 509 (1326)
FP EMULATION	30.17 (0.002)	28.77 (0.009)	23.12 (0.002)	2.28 (0.000)	2.29 (0.000)
FOURIER	23 851.98 (7.853)	21 896.10 (12.773)	13 180.42 (3.819)	10 228.70 (2.137)	3499.27 (0.086)
ASSIGNMENT	41.72 (0.013)	40.39 (0.004)	40.36 (0.003)	5.32 (0.000)	2.78 (0.000)
IDEA	7257.17 (0.529)	7088.14 (0.759)	6419.30 (20.861)	2254.47 (0.311)	704.18 (0.060)
HUFFMAN	2335.15 (0.314)	2131.65 (1.249)	2329.53 (0.474)	578.78 (0.061)	288.48 (0.012)
NEURAL NET	66.20 (0.027)	63.98 (0.056)	42.01 (0.024)	38.37 (0.004)	4.26 (0.000)
LU DECOMP	1467.54 (0.780)	1406.82 (0.351)	955.22 (0.434)	631.33 (0.129)	92.75 (0.003)

Table 3: Average performance in sgx-nbench [58] on i5-10210U@1.6GHz of our GS mitigation vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. Iterations/s, higher is better. N=50

microcode update that simply marks the null page uncachable. However, we identified several problems with this approach.

First, transient loads from an uncachable page can pick up values from the line-fill buffer [41, 52]. With hyperthreading enabled, clearing the line-fill buffer on enclave entry and exit is then also not sufficient.

Second, we experimentally verified that the operating system can change the memory type of enclave pages. Hence, a malicious operating system could change the memory type of the null page to cachable. Only if there is a method to lock entries in the TLB, SGX could ensure that the TLB entry for the null page stays in the TLB, preventing the operating system from changing the memory type.

Hence, we conclude that microcode mitigations are not as simple as assumed. The fact that there is no microcode update for any CPU to prevent LVI-NULL also indicates that microcode mitigations might not be possible.

Limitations. While LVI-NULLify conceptually prevents most variants of LVI-NULL, our technical implementation is currently limited by a few factors. Some of these limitations can be solved using additional engineering effort, while others can be solved directly by the enclave developer.

Most limitations are due to our proof-of-concept compiler transformation pass. The transformation pass currently uses a machine function pass to apply LVI-NULLify. However, as assembly is not handled by this machine function pass, we currently cannot directly patch inline assembly or assembly files automatically.

The remaining limitations are due to the pointer conversion between enclave and host application. While all the cases where the enclave developer adheres to best practice and the strict interface definitions are supported, there are corner cases that cannot be supported in an automated way, e.g., if the pointer is hidden behind an unknown type and reinterpreted by the developer.

8 Conclusion

In this paper, we presented a novel, lightweight defense against LVI-NULL in SGX. Based on a systematic analysis of LVI-NULL variants, we identified the attack requirements and discovered that previous mitigations targeting LVI-NULL are not effective. Our mitigation, LVI-NULLify, addresses this problem by repurposing segmentation to offset every load during enclave execution. LVI-NULLify consists of a modified SGX-SDK and a compiler extension that we open source. We evaluated LVI-NULLify on LVI-fixed CPUs and observed a performance overhead below 10% for the worst case, which is substantially lower than previous defenses. We conclude that LVI-NULLify is a practical solution to protect SGX enclaves on processors that remain susceptible to LVI-NULL.

Acknowledgments

We want to thank the anonymous reviewers and especially our shepherd, Fangfei Liu, for their comments and suggestions. We also want to thank Aikata Aikata for her support with hardware procurement. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 681402). Additional funding was provided by generous gifts from Intel, Amazon and ARM. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding parties.

References

[1] Daniel Pierre Bovet. Special sections in Linux binaries, January 2013. URL: https://lwn.net/Articles/531148/.

[2] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[3] Zola Bridges. LLVM SESES pass for LVI, 2020. URL: https://reviews.llvm.org/D75939.

[4] Claudio Canella, Daniel Genkin, Lukas Giner, Daniel Gruss, Moritz Lipp, Marina Minkin, Daniel Moghimi, Frank Piessens, Michael Schwarz, Berk Sunar, Jo Van Bulck, and Yuval Yarom. Fallout: Leaking Data on Meltdown-resistant CPUs. In *CCS*, 2019.

[5] Claudio Canella, Khaled N. Khasawneh, and Daniel Gruss. The Evolution of Transient-Execution Attacks. In *GLSVLSI*, 2020.

[6] Claudio Canella, Sai Manoj Pudukotai Dinakarrao, Daniel Gruss, and Khaled N. Khasawneh. Evolution of Defenses against Transient-Execution Attacks. In *GLSVLSI*, 2020.

[7] Claudio Canella, Michael Schwarz, Martin Haubenwallner, Martin Schwarzl, and Daniel Gruss. KASLR: Break It, Fix It, Repeat. In *AsiaCCS*, 2020.

[8] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium*, 2019. Extended classification tree and PoCs at https://transient.fail/.

[9] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *EuroS&P*, 2019.

[10] Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: closing hyperthreading side channels on sgx with contrived data races. In *S&P*, 2018.

[11] Adam Chester. Exploiting Windows 10 Kernel Drivers - NULL Pointer Dereference, 2018.

[12] Victor Costan and Srinivas Devadas. Intel SGX Explained. *Cryptology ePrint Archive, Report 2016/086*, 2016.

[13] David Drysdale. How programs get run: ELF binaries, 2015. URL: https://lwn.net/Articles/631631/.

[14] Dmitry Evtyushkin, Ryan Riley, Nael CSE Abu-Ghazaleh, ECE, and Dmitry Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *ASPLOS*, 2018.

[15] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 2017.

[16] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O'Connell, Wolfgang Schoechl, and Yuval Yarom. Another Flip in the Wall of Rowhammer Defenses. In *S&P*, 2018.

[17] Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx. Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution. In *ESSoS*, 2018.

[18] Jann Horn. speculative execution, variant 4: speculative store bypass, 2018.

[19] Tianlin Huo, Xiaoni Meng, Wenhao Wang, Chunliang Hao, Pei Zhao, Jian Zhai, and Mingshu Li. Bluethunder: A 2-level Directional Predictor Based Side-Channel Attack against SGX. In *CHES*, 2020.

[20] Intel. Intel Software Guard Extensions SDK for Linux OS Developer Reference, May 2016. Rev 1.5.

[21] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture, 2016.

[22] Intel. Branch Target Injection / CVE-2017-5715 / INTEL-SA-00088, 2018. URL: https://software.intel.com/security-software-guidance/advisory-guidance/branch-target-injection.

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

[24] Intel. Speculative Execution Side Channel Mitigations, 2018. Revision 3.0.

[25] Intel. Deep Dive: Intel Analysis of Microarchitectural Data Sampling, 2019.

[26] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z, 2019.

[27] Intel. Affected Processors: Transient Execution Attacks, 2020. URL: https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model.

[28] Intel. An Optimized Mitigation Approach for Load Value Injection, 2020. URL: https://software.intel.com/security-software-guidance/best-practices/optimized-mitigation-approach-load-value-injection.

[29] Intel. Load Value Injection, 2020. URL: https://software.intel.com/content/www/us/en/develop/articles/software-security-

guidance/technical-documentation/load-value-injection.html.

[30] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. SGX-Bomb: Locking Down the Processor via Rowhammer Attack. In *SysTEX*, 2017.

[31] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 Processor Integrity from Software. In *USENIX Security Symposium*, 2020.

[32] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors. In *ISCA*, 2014.

[33] Vladimir Kiriansky and Carl Waldspurger. Speculative Buffer Overflows: Attacks and Defenses. *arXiv:1807.03757*, 2018.

[34] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *S&P*, 2019.

[35] Anton Korobeynikov. Improving switch lowering for the llvm compiler system. In *SYRCoSE*, May 2007.

[36] Esmaeil Mohammadian Koruyeh, Khaled Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *WOOT*, 2018.

[37] Michael Larabel. Google Engineer Shows "SESES" For Mitigating LVI + Side-Channel Attacks, 2020. URL: https://www.phoronix.com/scan.php?page=news_item&px=LLVM-SESES-Mitigating-LVI-More.

[38] Michael Larabel. The Brutal Performance Impact From Mitigating The LVI Vulnerability, 2020. URL: https://www.phoronix.com/scan.php?page=article&item=lvi-attack-perf.

[39] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *IEEE / ACM International Symposium on Code Generation and Optimization – CGO*, 2004.

[40] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, 2017.

[41] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium*, 2018.

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

[43] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In *CHES*, 2017.

[44] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural Data Leakage via Automated Attack Synthesis. In *USENIX Security Symposium*, 2020.

[45] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based Fault Injection Attacks against Intel SGX. In *S&P*, 2020.

[46] Nergal. The advanced return-into-lib(c) explits: PaX case study, 2001.

[47] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. VoltJockey: Breaking SGX by Software-Controlled Voltage-Induced Hardware Faults. In *Asian-HOST*, 2019.

[48] Raul Quinonez. SGXBENCH framework for benchmarking SGX enclaves, 2018. URL: https://github.com/sgxbench/sgxbench.

[49] Michael Schwarz, Daniel Gruss, Moritz Lipp, Clémentine Maurice, Thomas Schuster, Anders Fogh, and Stefan Mangard. Automated Detection, Exploitation, and Elimination of Double-Fetch Bugs using Modern CPU Features. In *AsiaCCS*, 2018.

[50] Michael Schwarz, Daniel Gruss, Samuel Weiser, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, 2017.

[51] Michael Schwarz, Moritz Lipp, Claudio Canella, Robert Schilling, Florian Kargl, and Daniel Gruss. ConTExT: A Generic Approach for Mitigating Spectre. In *NDSS*, 2020.

[52] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. ZombieLoad: Cross-Privilege-Boundary Data Sampling. In *CCS*, 2019.

[53] Michael Schwarz, Martin Schwarzl, Moritz Lipp, Jon Masters, and Daniel Gruss. NetSpectre: Read Arbitrary Memory over Network. In *ESORICS*, 2019.

[54] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[55] Hovav Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *CCS*, 2007.

[56] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *NDSS*, 2017.

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

[58] utds3lab. Adaptation of nbench-byte-2.2.3 for Intel SGX, 2017. URL: https://github.com/utds3lab/sgx-nbench.

[59] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*, 2018.

[60] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yarom Yuval, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *S&P*, 2020.

[61] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio Garcia, and Frank Piessens. A Tale of Two Worlds: Assessing the Vulnerability of Enclave Shielding Runtimes. In *CCS*, 2019.

[62] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Workshop on System Software for Trusted Execution*, 2017.

[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying Microarchitectural Timing Leaks in Rudimentary CPU Interrupt Logic. In *CCS*, 2018.

[64] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets Without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium*, 2017.

[65] Stephan van Schaik, Alyssa Milburn, Sebastian Österlund, Pietro Frigo, Giorgi Maisuradze, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. RIDL: Rogue In-flight Data Load. In *S&P*, 2019.

[66] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. AsyncShock: Exploiting Synchronisation Bugs in Intel SGX Enclaves. In *ESORICS*, 2016.

[67] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating enclave malware via confinement. In *RAID*, 2019.

[68] Hans Wennborg. The recent switch lowering improvements, October 2015. URL: http://llvm.org/devmtg/2015-10/slides/Wennborg-SwitchLowering.pdf.

[69] Yuan Xiao, Yinqian Zhang, and Radu Teodorescu. SPEECHMINER: A Framework for Investigating and Measuring Speculative Execution Vulnerabilities. In *NDSS*, 2020.

[70] Wenjie Xiong and Jakub Szefer. Survey of Transient Execution Attacks. *arXiv:2005.13435*, 2020.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *S&P*, 2015.

[72] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. MPTEE: Bringing Flexible and Efficient Memory Protection to Intel SGX. In *EuroSys*, 2020.

A Benchmarking results

In addition to our LVI-NULL-only affected Comet Lake CPU, we also provide benchmark overheads for the older i9-9900K Coffee Lake CPU in Figure 5. We can see that while their are some differences, the relative performances between the mitigations is roughly the same on this desktop CPU as it is on the mobile i5-10210U.

Table 4 shows the execution times for various SGXBENCH benchmarks on our Comet Lake i5-10210U.

B Sample Compilation Options for Mitigations

Control-flow Mitigation:

```
clang-lvi-cfi -mlvi-cfi
-Iclang-lvi-cfi/sgxsdk/include
-Iclang-lvi-cfi/sgxsdk/include/tlibc -fpic -O3
-nostdinc -fvisibility=hidden -fstack-protector
```

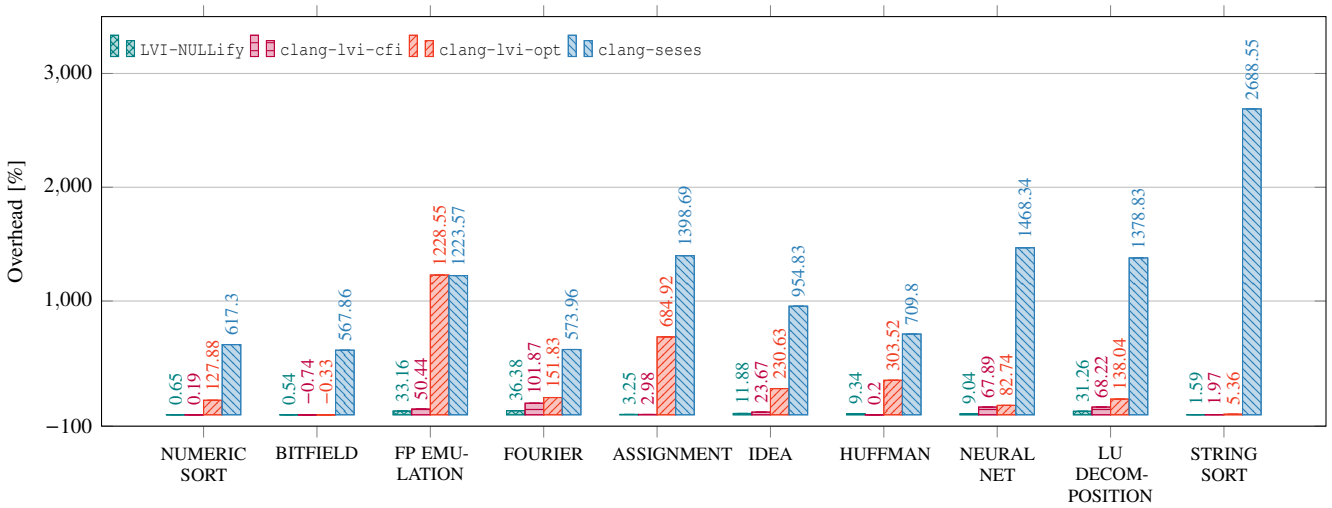


Figure 5: Mean performance overhead in sgx-nbench [58] on our i9-9900K@3.5GHz of LVI-NULLify vs. Intel's control-flow and optimized-cut mitigations as well as SESES. Clang 11 was used for all tests. N=50, standard deviations w.r.t. baseline mean are plotted, but too small to be visible.

Test/Mitigation	none (σ)	LVI-NULLify (σ)	control-flow (σ)	optimized cut (σ)	SESES (σ)
empty function	37072.3 (2655.0)	37660.3 (1878.0)	37349.9 (1832.2)	39015.5 (1701.7)	42612.8 (3325.2)
empty ocall	14496.6 (1151.5)	14980.8 (1096.3)	14735.5 (1146.6)	16052.1 (1080.8)	15995.6 (995.6)
ocall in/out	15651.8 (835.7)	16433.0 (809.6)	16236.2 (702.1)	17510.5 (819.4)	23309.8 (1023.8)
encrypted read	14884.4 (756.5)	15228.5 (843.2)	14984.3 (758.9)	16429.6 (799.8)	21868.8 (966.6)
encrypted write	14720.8 (799.7)	15279.9 (827.8)	14989.7 (740.8)	16507.6 (779.2)	21866.2 (967.0)
einit/edestroy	141845200.9 (793624.2)	142087389.7 (849561.9)	142354110.9 (841980.8)	142506399.6 (798824.8)	142649039.2 (827876.1)

Table 4: Runtime of the SGXBENCH benchmarks on an i5-10210U@1.6GHz in cycles. Lower is better. N=1000000 for all except eint/edestroy where N=1000

```
-fpic -c Enclave.c -o Enclave.o
```

SESES Mitigation:

```
clang-lvi-seses -mseses
-Iclang-lvi-seses/sgxsdk/include
-Iclang-lvi-seses/sgxsdk/include/tlibc
-fpic -O3 -nostdinc -fvisibility=hidden
-fstack-protector -fpic -c Enclave.c -o
Enclave.o
```

Optimized-Cut Mitigation:

```
clang-lvi-opt -mlvi-hardening -mllvm
-x86-lvi-load-opt-plugin=OptimizeCut.so -mllvm
-x86-experimental-lvi-inline-asm-hardening
-Iclang-lvi-opt/sgxsdk/include
-Iclang-lvi-opt/sgxsdk/include/tlibc -fpic -O3
-nostdinc -fvisibility=hidden -fstack-protector
-fpic -c Enclave.c -o Enclave.o
```

C LVI-NULL POC Implementation Details

In addition to LVI-NULLify, the relevant proofs of concept can also be found in our repository at https://github.com/lvi-nullify/LVI-NULLify.

For attacks on SGX, an attacker would typically use a framework like SGX-Step [62] to interfere with a victim enclave at more or less precise points. For our POCs however, we can use a more cooperative approach, which simplifies the code and imitates a very strong attacker. Right before vulnerable loads in our victim, we OCALL to the attacker who then removes the *accessed* bit from our target page. This reliably causes 0 to be injected into the next loads from this page, triggering our LVI-NULL attacks. We can then measure rates of leakage via a transmission gadget; in our case an access to a page outside the enclave.

When we compile with LVI-NULLify, we see that all leakage is completely prevented.

D Artifact Appendix

D.1 Abstract

The public repository[2] contains all the code necessary to reproduce the data for all performance graphs/tables in the paper, as well as PoCs to demonstrate that the mitigation works. This includes patches and build instructions for LLVM11, the Intel SGX SDK and PSW, as well as the benchmarks. The artifact requires SGX to evaluate, and is easiest to run on Ubuntu 18.04 or 20.04.

D.2 Artifact check-list (meta-information)

- **Program:** Adapted versions of `nbench` and `sgxbench` are downloaded & installed via included scripts.
- **Compilation:** Requires a modified Clang 11, install & download script is included.
- **Transformations:** A tool to fix up relocations is included (relocator).
- **Run-time environment:** Needs a native Linux installation that supports SGX, Ubuntu 18.04 or 20.04 are strongly recommended. Build scripts need internet access at several points. Requires root for installation and evaluation. PoCs require the PTEditor kernel module.
- **Hardware:** Intel CPU with SGX support, needs to be vulnerable to LVI-Null for PoC tests (affected CPUs).

  The PoCs need a kernel module, which means either self-signing or disabling secure boot. This may require physical access to the machine.
- **Run-time state:** As this artifact includes performance benchmarks, a stable CPU frequency and isolated cores are recommended.
- **Execution:** For ideal testing, the system should have isolated cores, fixed frequency, and not much other activity.
- **Metrics:** Benchmarks report cycle count or iterations/s, PoCs report leakage percentage.
- **Output:** Benchmark outputs are .csv tables with performance, an included spreadsheet can convert to a graph similar to the paper.
- **Experiments:** Installation scripts are included and described here and in READMEs.
- **How much disk space required (approximately)?:** 4-5GB
- **How much time is needed to prepare workflow (approximately)?:** 2-3h
- **How much time is needed to complete experiments (approximately)?:** 3-6h, depends on hardware
- **Publicly available?:** `https://github.com/lvi-nullify/LVI-NULLify/`
- **Code licenses (if publicly available)?:** zlib

[2] `https://github.com/lvi-nullify/LVI-NULLify`

D.3 Description

D.3.1 How to access

Clone `https://github.com/lvi-nullify/LVI-NULLify/releases/tag/ae` and follow the README.md from there.

D.3.2 Hardware dependencies

As this is a mitigation for Intel SGX, SGX support is a hard requirement. To fully evaluate the PoCs, and not just mitigation performance, the CPU also needs to be vulnerable to LVI. You can check if your CPU is vulnerable here: `https://software.intel.com/content/www/us/en/develop/topics/software-security-guidance/processors-affected-consolidated-product-cpu-model.html`

D.3.3 Software dependencies

We strongly recommend Ubuntu 18.04 or 20.04 as these are officialy supported by Intel, and all our tools were tested on them.

Beyond standard compilation tools (ninja, cmake etc) our PoCs require the PTEditor kernel module [3]. Other requirements are listed in the README files at the appropriate points.

D.4 Installation

Follow the detailed README in the top-level directory to set up our modified clang compiler and relocator and install the SGX driver as well as our modified SGX SDK and PSW.

Once that is done, you can already test your installation with the PoCs by following the README file in the POC directory.

With a working PSW and driver, you can follow the README in the benchmarks directory to download and build the benchmarks.

D.5 Experiment workflow

After building the benchmarks, follow along in the README to start all or a subset of them. An important aspect to keeping benchmarks comparable is to fix the CPU's frequency to a sustainable level, and idealy run them on an isolated core.

PoCs can be run according to the README in the POC folder.

D.6 Evaluation and expected results

The main results in our paper are contained in Figure 4/Table 3. These are the performance overheads of our LVI-NUll mitigation compared to other, similar mitigations. The second, more implicit result is the efficacy of LVI-NULLify.

For the benchmarks, the absolute performance overheads vary significantly between different machines and architectures (compare Figure 4 and Figure 5), but the relative differences should be roughly similar. That is: LVI-Nullify should be the fastest mitigation, or at least very close to Intel CFI, typically followed, with some distance, by Intel's optimized-cut mitigation.

For the PoCs, starting once without and once with mitigation should produce qualitatively similar results to the examples shown in the README. That means, for the 3 PoCs where LVI-Nullify is

[3] `https://github.com/misc0110/PTEditor/`

effective, leakage rate should drop to zero, or a level that is comparable to the noise-catching output "other". While absolute leakage rates before applying the mitigation may differ significantly from system to system, they should be clearly differentiable from "other".

The respective READMEs for benchmarks and PoCs detail how to reproduce these results.

D.7 Experiment customization

Attack PoCs need a cache miss threshold, which is automatically determined. If this doesn't work, it can be set manually in the corresponding *App.cpp* file. All PoCs include a conf.h file, in which the character that should be leaked can be changed if desired.

Both benchmark run-scripts contain a variable called "isolated_core" that sets the core on which they should be run on. Set this to an isolated core, if available.

sgx-nbench contains a parameter to change the number of iterations in the file, see the benchmarking README.

D.8 Methodology

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-badging
- http://cTuning.org/ae/submission-20201122.html
- http://cTuning.org/ae/reviewing-20201122.html