Off-Limits: Abusing Legacy x86 Memory Segmentation to Spy on Enclaved Execution

Jago Gyselinck, Jo Van Bulck, Frank Piessens, and Raoul Strackx

imec-DistriNet, KU Leuven, Celestijnenlaan 200A, B-3001 Belgium

Abstract. Enclaved execution environments, such as Intel SGX, enable secure, hardware-enforced isolated execution of critical application components without having to trust the underlying operating system or hypervisor. A recent line of research, however, explores innovative controlled-channel attacks mounted by untrusted system software to partially compromise the confidentiality of enclave programs. Apart from exploiting relatively well-known side-channels like the CPU cache and branch predictor, these attacks have so far focused on tracking side-effects from enclaved address translations via the paging unit.
This paper shows, however, that for 32-bit SGX enclaves the unacclaimed x86 segmentation unit can be abused as a novel controlled-channel to reveal enclaved memory accesses at a page-level granularity, and in restricted circumstances even at a very precise byte-level granularity. While the x86 paging unit has been extensively studied from both an attack as well as a defense perspective, we are the first to show that address translation side-channels are not limited to paging. Our findings furthermore confirm that largely abandoned legacy x86 processor features, included for backwards compatibility, suggest new and unexpected side-channels.

Keywords: Intel SGX, Controlled-channel, x86, Paging, Segmentation

1 Introduction

Most popular operating systems and virtual machine managers have now been around for multiple decades. During this period, a steady stream of critical vulnerabilities has been found in their expansive code bases. These vulnerabilities continue to be problematic for any application that wishes to do secure computations on such a platform. In order to shield applications from potentially malicious or compromised system software, a significant research effort has recently been put into creating Protected Module Architectures (PMAs) [17,23,7,18]. These architectures offer isolated execution for security sensitive application components, while leaving the underlying system software explicitly untrusted. With the introduction of its Software Guard Extensions (SGX) [18,1,6], Intel brought their implementation of a PMA to the mass consumer market. Conceived as an extension to the x86 instruction set architecture, SGX provides strong trusted computing guarantees with a minimal Trusted Computing Base (TCB), which is limited to the protected module or *enclave*, and the processor package.

Recent research [28,22,26,16,27,19,25] has shown, however, that the combination of SGX's strong adversary model and reduced TCB allows a privileged attacker to create high resolution, low-noise *controlled-channels* that leak information about the enclave's internal state. More specifically, enclave programs still rely on the untrusted operating system to manage shared platform resources such as CPU time or memory. Within SGX's adversary model, an attacker may attempt to leverage control over these resources to infer enclave secrets. Notably, Xu et al. [28] first showed how to recover rich information such as full images and text from a single enclaved execution by carefully restricting enclave page access rights and observing the resulting page fault sequences. Since their seminal work, more conventional side-channels such as the processor cache [11,19,2] and branch prediction unit [16] have also been improved in the context of SGX.

Considering that innovative page fault attacks [28,22] only recently became relevant in a kernel-level PMA adversary model, they have received considerable attention from the research community. A good level of understanding of page table attack surface has since been built up by (*i*) exploring stealthy attack variants [26,27] that abuse other side-effects of the page table walk, (*ii*) developing software-based defense mechanisms [21,5,24,4] for off-the-shelf SGX processors, and (*iii*) designing fortified PMAs [7,9] that rule out these attacks at the hardware level. This paper shows, however, that enclaved memory accesses in 32-bit mode not only leak through page tables, but also through the largely overlooked x86 memory segmentation unit. A feature that is for the most part disabled on 64-bit systems, but regains relevance when considering 32-bit enclaves. We advance the understanding of address translation controlled-channels by showing that under certain assumptions, attackers can leverage control over segment limits to infer byte-granular memory access patterns from an enclaved execution. Furthermore, our findings illustrate that the backwards compatibility requirement of modern x86 processors suggests new and unexpected side-channels stemming from largely abandoned legacy features. In summary, the contributions of this paper are:

- We show how for 32-bit enclaves the x86 segmentation unit can be abused as a novel, noise-free side-channel to reveal precise byte-granular control flow and instruction sizes in the first megabyte of a victim enclave.
- We explain how for the remainder of the enclave address space, segmentation attacks can infer memory accesses at a conventional page-level granularity.
- We implement our attacks and practically demonstrate their enhanced precision by defeating a recently proposed branch obfuscation defense.
- We reveal an undocumented Intel microcode update that silently blocks our attacks without updating the processor's security version number. Only the very recent Spectre CPU updates can adequately prevent our attacks.

2 Background

We first present Intel SGX and our attacker model, before introducing the necessary background on x86 memory organization.

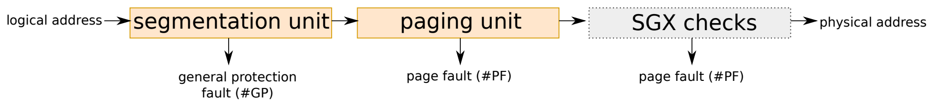


Fig. 1: On x86 logical addresses pass through the segmentation and paging units respectively. The resulting physical address is additionally sanitized by SGX.

2.1 Intel SGX and Adversary Model

Recent Intel processors include an architectural extension called the Software Guard Extensions (SGX) [18,1,12] which bring strong, processor-enforced confidentiality and integrity guarantees for protected software modules called *enclaves*. SGX enclaves live inside a conventional OS process, and span a contiguous virtual address range (`ELRANGE`) for their protected code and data. The processor's memory access control logic takes care to block any access to `ELRANGE` from outside the corresponding enclave, regardless of the current CPU privilege level. Furthermore, to prevent active memory mapping attacks [6] performed by a kernel-level attacker in control of page table mappings, the processor verifies that every physical enclave address is accessed via the expected virtual address.

SGX includes several new x86 instructions to switch the processor in and out of enclave mode. `EENTER` allows to transfer control to a specific point in the enclave, while its counterpart `EEXIT` returns control flow back to untrusted memory. In case of an interrupt or fault during enclaved execution, an Asynchronous Enclave eXit (AEX) occurs. Much like leaving an enclave, AEX saves the enclave's state to later be resumed, while again clearing any processor state that may leak information. After the reason for the interrupt has been serviced, the enclave can be resumed using the `ERESUME` instruction.

SGX considers even the kernel as potentially malicious. Our attacks assume a less powerful attacker; we show that user-level capabilities suffice to control the segmentation unit. However, as will be indicated in our attack descriptions, we make use of a secondary framework to execute our attacks. While these are often interchangeable, some require a more privileged attacker. In case such a secondary framework is chosen, the attacker model should be upgraded accordingly. In general, we also assume the attacker has access to the enclave's object code, unless explicitly stated otherwise.

At the system level, we focus exclusively on 32-bit enclaves, for segmentation is practically disabled in 64-bit mode. Furthermore, as discussed in more detail in Section 5, we assume the processor runs one of the vulnerable microcode versions listed in Appendix A.

2.2 x86 Memory Management

To enable SGX enclaves to be easily integrated in legacy applications, they live in the same address space. Unfortunately this implies that the architectural complexities of x86 memory organization play a crucial role in assessing SGX's isolation properties [6]. Memory management in the IA-32 architecture [12] proceeds

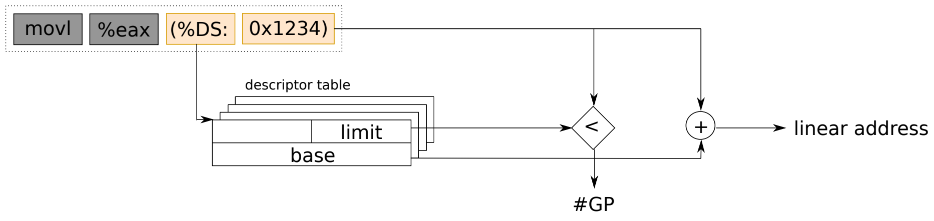


Fig. 2: The segmentation unit checks that each referenced segment adheres to the segment's limitations described in the descriptor table

via two distinct hardware components, visualized in Fig. 1. Application software uses *logical* addresses, which are first passed through a dedicated segmentation unit to yield *linear* addresses as an input to the paging unit. SGX finally enforces some (limited) additional checks on the resulting *physical* addresses.

The Segmentation Unit. Segmentation serves as a way to divide the logical address space into segments. The x86 hardware provides 6 *segment registers* (`%CS`, `%DS`, `%SS`, `%ES`, `%FS` and `%GS`) to directly reference segments. Each machine instruction either explicitly references a segment register, or one is implied. Pushing or popping data from the stack, for example, always references the stack segment (`%SS`). Similarly, the instruction pointer (`%eip`) is always relative to the code segment (`%CS`). Move instructions to memory always imply the data segment (`%DS`). Segments `%FS` and `%GS` are typically used for thread-local storage.

Figure 2 displays how the segmentation unit operates during the execution of an instruction. For each segment referenced, its *segment descriptor* is located in the Local Descriptor Table (LDT) or Global Descriptor Table (GDT). Each descriptor records the base (linear) address of the segment, its limit and the associated access rights (i.e., read, write, execute). When the instruction does not violate the access rights to the segment, and the logical address remains within the segment limits, the linear address is calculated by adding the segment base to the logical address. Otherwise a General Protection fault (#GP) is issued.

In 32-bit mode segment descriptors measure only 64 bits in size. This is too limited to store 32-bit base and limit addresses, plus other attributes (e.g., access rights). To resolve this issue, only a 20-bit limit field is used in combination with a special *granularity* bit. When this bit is clear, limits can be specified up to $2^{20}-1$ (1 MiB) at byte granularity. Otherwise the limit field is interpreted at 4 KiB granularity, allowing the logical address space to reach $(2^{20}-1)*4096$. As the 12 least significant bits of this limit are not checked, the full 2^{32} address space can be accessed [12, §5.3].

Over time, segmentation has evolved to become more and more obsolete. In 64-bit mode, the processor ignores the segment descriptor registers for `%DS`, `%SS` and `%ES`, limit checks are no longer performed and the base of `%CS` is always treated as zero. [12, §3.4]

The Paging Unit. After the segmentation unit translated a logical address to a linear one, the paging unit translates it in turn to a physical address. It does so by dividing the linear address space in fixed memory regions called pages. The base address of each referenced page is located in an in-memory page table structure maintained by the operating system. After the page table walk, the processor obtains the physical page base address, plus the associated access rights and other attributes (e.g., whether the page is present or has been accessed before).

3 Segmentation-based attacks

Intel SGX enclaves execute in the same logical address space as their host process. Just like logical addresses used in the untrusted (legacy) part of a process pass through the segmentation and paging unit, so do the addresses referenced during SGX enclave execution. These address translation units are under complete control of the potentially malicious kernel. To prevent an attacker from mistranslating enclave addresses, Intel SGX applies additional checks as a final step (see Fig. 1). During enclave creation, the processor records for every enclave page the logical address they should be loaded at and to which physical address they should be translated to. The kernel is still in control over all memory allocation decisions. She can for example decide to evict enclave pages from main memory, but the hardware will check whether the memory translation units have been set up correctly.

Unfortunately, SGX's untrusted page table design also opens up powerful controlled-channel attacks. Early page fault-driven attacks [28,22] and more recently improved fault-less page table-based attacks [26,27] show that paging mechanisms can be abused by an attacker to leak enclave memory accesses at page-level granularity. When these memory accesses are secret-dependent, they may reveal sensitive information. To the best of our knowledge, academic research has only looked into leveraging the paging unit. For an SGX-capable processor in 32-bit mode however, the segmentation unit also interposes on every enclaved address translation.

3.1 Interaction Between Segmentation and SGX

The Intel Programming Reference Manual [13] states that "enclaves abide by all the segmentation policies set up by the OS", but several sanity checks on the segmentation policy have been put in place. For example, it is enforced that the `%CS`, `%DS`, `%SS` and `%ES` registers point to segment descriptors which have their base address set to zero, as any other value could maliciously change the interpretation of enclaved code. Trusted in-enclave segment selectors and descriptors for the `%FS` and `%GS` segments are saved and replaced on enclave entry to facilitate access to the enclave's thread local storage. This means that the `%FS` and `%GS` segments are immune to the attacks described in this paper, for any modifications made by an attacker will not propagate to enclaved execution.

```
void vote(enum candidate c) {
   if (c == candidate_a)
      handle_candidate_a();
   else
      handle_candidate_b();
   handle_total_votes();
   return;
}

void handle_candidate_a() {
   candidate_a_votes++;
}

void handle_candidate_b() {
   candidate_b_votes++;
}

void handle_total_votes() {
   total_votes++;
}
```

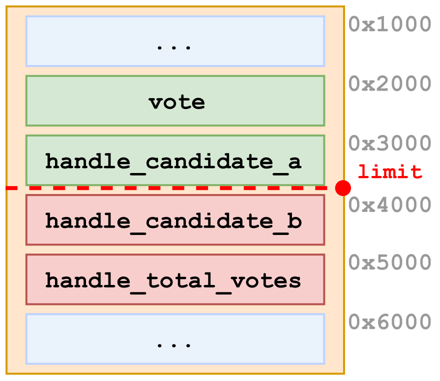


Fig. 3: Example enclave with secret dependent control transfer.

We make the crucial observation that, while the Intel SGX hardware forces segment descriptor base addresses to be `0x0`, their limit is still under the control of an attacker. Reducing the limit of a segment, will cause a general protection fault whenever an attempt is made to cross it. In Section 2.2 we have discussed that segment limits can be specified at byte-granularity up to the 1 MiB boundary. Limits past this bound can only be specified at 4 KiB granularity.

3.2 Attack #1: Page Granular Attacks

We will first explore the possibilities when the granularity flag is set. As said before, this will allow to leak information about memory accesses at page granularity. When compared to the earlier explored page fault-driven attacks, there is one fundamental difference. Restricting page access rights makes small chunks of memory inaccessible for the processor, while leaving others completely unaffected. Segmentation presents a rougher, binary condition; either a memory location is within the segment or it is not. In other words, moving the segment limit not only influences a single page, but all pages that are now above the segment limit.

It is this difference that presents an interesting challenge. We present a running example code snippet in Fig. 3 with some sample enclave code that represents a simple voting mechanism. We assume the vote being cast is secret and that the attacker wishes to derive its value. For simplicity we assume all functions of interest are aligned on their own pages. To illustrate the problem, imagine the vote function being executed, with the segment limit taken as indicated on Fig. 3. Now assume that the attacker observes a general protection fault. Clearly, this may occur when the vote function was called for candidate B, as the handler for that candidate is outside of the segment. However, a second possibility also exists where the vote is for candidate A. Here, control will be passed back to the vote function, which in turn calls the total vote handler, causing a general protection fault as well. The two general protection faults will be identical to an attacker, who is now unable to derive any information. To solve this, we combine the segmentation unit with a secondary framework. In most of our examples, we use the page-fault side-channel as an extra layer of information for simplicity of illustration. This side-channel then functions as an oracle to indicate to the attacker whether the memory access has passed the segmentation stage. The exact same can also be achieved by monitoring the page accessed bit [26,27]. Alternatively, we can make sure the enclave takes just one step, for which a single stepping interrupt framework such as SGX-Step [25] can be used.

Since this first attack has the same granularity as the original page fault driven attacks, it would not be useful in this context to use that same side-channel as the secondary framework. At the same time, replicating previous page table-based attacks results [28,22,26,27] without using the paging unit demonstrates that state-of-the-art defenses that move the page tables into enclave memory [8,9] may not suffice for 32-bit enclaves. Because of this, we illustrate how we can replicate page fault-driven attacks using solely the segmentation unit and SGX-Step, without the need to alter page table entries.

Reconsider the running example of Fig. 3, where we wish to extract which candidate was voted for. Initially, we set the limit of `%CS` at `0x3000`, making pages of both candidate handlers inaccessible. The attacker is then guaranteed to observe a general protection fault when the enclave is single stepped until one of the handlers is called. At this point, the attacker can move the segment limit to also include the handler for candidate A (limit at `0x4000`). When the enclave is resumed, the single-stepping framework makes sure at most one instruction is executed, after which two situations can be distinguished:

1. No fault is observed, which indicates that the vote was for candidate A. Control is successfully passed to the handler for that candidate, which is located within the segment.
2. A second general protection fault is observed. This indicates that the vote was for candidate B, as only a call to this function crosses the segment boundary.

3.3 Precise Byte Granular Attacks

In this section, we present the most fine-grained attacks that are possible using the segmentation unit. Keep in mind that these are also the attacks with the most limitations. Again, they are applicable to 32-bit enclaves only, where the region of interest to the attacker is located within the first megabyte of the victim enclave's memory layout.

The segmentation and paging unit are closely integrated. While conceptually they can be regarded as executing one after the other at the architectural

Table 1: Segmentation plus paging configurations and whether they generate a General Protection fault (#GP) or Page Fault (#PF).

eip $\leq$ limit	**page access rights**	**(eip + inst size) $\leq$ limit**	**Fault type**
✗	-	-	#GP$_1$
✓	✓	✗	#GP$_2$
✓	✗	-	#PF

level (see Fig. 1), we found this to be inaccurate at the microarchitectural level. We will show that by carefully setting segment limits and page rights, detailed information about the control flow and even instruction sizes leak to an attacker.

Combining the Segmentation and Paging Units. Only when an instruction is completely contained within the limits of the code segment, it may execute. When the instruction falls outside the code segment's limit, a #GP is generated. An interesting edge case occurs, however, when a multi-byte instruction starts within the code segment, but passes its boundaries. In that case, the fault thrown depends on the paging unit: only when the page the instruction is located on has execute permissions, a #GP is thrown. Otherwise, the paging unit generates a Page Fault (#PF). This behavior is summarized in Table 1.

We conclude that the segmentation and paging units verify access rights and limits in parallel at the microarchitectural level. We suspect that the exact outcome may differ between different processor generations and models, but always found stable outcomes on a single machine.

Attack #2: Inferring Instruction Sizes. Previous enclaved execution side-channel attacks [28,26,16,19,2] rely on static analysis of the victim enclave's source code. In some cases however, the object code of the enclave may not be available to the attacker or it may be randomized on enclave load [20]. If so, it may be of interest to the attacker to learn as much as possible about the instructions that are being executed [15]. For example, when code is randomized, this information may reveal the location of crucial functions by comparing the leaked outline to the non-randomized object code. To this end, we contribute a novel approach to infer enclaved instruction sizes by leveraging the segmentation and paging units and applying the techniques mentioned before.

To infer instruction sizes, we retake the idea of having two layers of information: the segmentation and the paging unit. An intuitive approach would be to take the segment limit at the start of an instruction, while revoking the access rights to the underlying page. Surely, this leads to a first general protection fault, as the instruction falls outside of the segment. In consecutive steps, we may gradually increase the segment limit with a single byte, until we observe a page fault. This would imply that the whole instruction is now within the segment, thus also revealing the instruction size. However, as explained above,

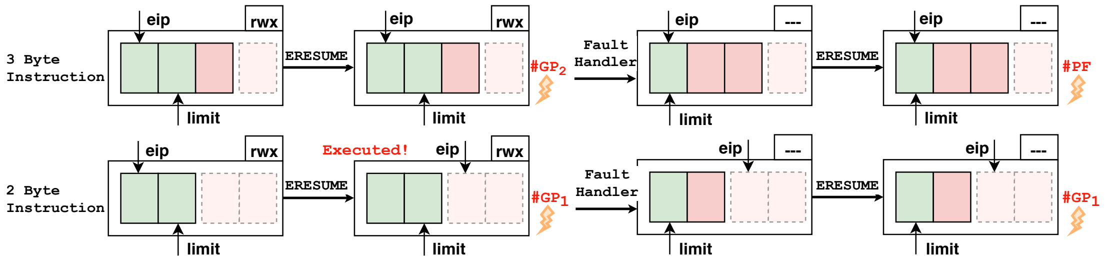


Fig. 4: Fault sequence attack to infer instruction size (three vs. two bytes).

the x86 segmentation and paging units work in parallel at the hardware level. We experimentally confirmed that including the first byte of an instruction into the segment is enough to activate the paging unit. As a result, as long as the underlying page access rights are revoked, a page fault will be reported regardless of instruction size.

Our practical attack therefore combines information leakage from both the paging and segmentation units. Figure 4 illustrates how an attacker can distinguish an exemplary three-byte instruction from a two-byte one. Initially, after interrupting the enclave before the instruction of interest, we set the code segment limit to include two bytes of the instruction about to be executed, and assign read/write/execute permissions to the underlying page. Next, the enclave is continued through the `ERESUME` instruction, and we observe a general protection fault #GP$_1$ or #GP$_2$, depending on whether the code segment limit violation was caused by either the current or the next instruction.[1] At this point, however, the attacker has no way of distinguishing #GP$_1$ and #GP$_2$, as both show up as identical general protection faults raised by the segmentation unit.

To overcome this challenge, we introduce the notion of *fault sequence attacks* as a novel generalization of the page fault sequences originally presented by Xu et al. [28]. That is, before resuming the victim enclave a second time, we configure the code segment limit to include the first byte of the instruction of interest and revoke access rights to the underlying page. According to Table 1, we now only observe a #GP when the secret in-enclave instruction pointer falls outside of the code segment. In case the enclaved instruction was larger than two bytes, on the other hand, the instruction pointer was not incremented and a #PF will be observed since the first byte of the instruction is included in the code segment. As such, our approach observes the *combined sequence* of general protection and page faults to infer the secret in-enclave instruction pointer.

[1] Note that we assume here that the next instruction is located immediately after the current one in memory. We explain in the next section how segmentation-based attacks can infer secret target addresses in case of jump instructions.

```
void foo(unsigned int secret) {
   if (secret)
      asm __volatile__("nop");
   else
      asm __volatile__("nop");
}
```

```
1  push   %ebp
2  mov    %esp,%ebp
3  cmp    $0x0,0x8(%ebp)
4  je     7 <foo+0xc>
5  nop
6  jmp    8 <foo+0xd>
7  nop
8  pop    %ebp
9  ret
```

Fig. 5: Using byte-granular segment limits, we can infer very precise control flow.

Attack #3: Inferring Branch Target Addresses. When we are able to set segmentation limits with a byte-level granularity, we can infer much more fine-grained control flow than page-unit-based attacks [28,22,26,27]. Consider the C code of Figure 5 and its translation to assembly. Even though the condition of `secret` results in the execution of only a few different instructions, we are able to infer which branch is taken and thus the boolean value of `secret`.

An attacker could first interrupt enclave execution by retracting the access rights of the page on which the "foo" function is located. Next, the page access rights can be restored, while lowering the segment limit to exclude any instruction past line 4 in the assembly listing. Placing the segment limit at this address excludes both control flow branches while just including the `je` (i.e., "jump equal") instruction.

Regardless of the value of `secret`, a general protection fault will occur when the enclave is resumed. When `secret` evaluates to false, the `cmp` (i.e., "compare") instruction on line 3 will have set the equal flag. Executing the `je` instruction on line 4 will then result in a #GP fault as the jump destination crosses the segment boundary. When the enclave is resumed after the fault is handled, another attempt will be made to execute the `je` instruction.

Alternatively, the secret evaluates to true. In that case the jump will not be taken. A general protection fault is issued as line 5's `nop` (i.e., "no-operation") instruction is (completely) located past the code segment's bounds.

To distinguish the two cases, we again rely on the paging unit and leverage the differences in microarchitectural behavior when an instruction is located in- or outside of the code segment on a non-executable page. Specifically, we revoke access rights to the underlying page, while leaving the segment limit untouched. When the enclave is resumed, two cases can occur:

– **General protection fault is issued:** This implies that the instruction must be located past the limits of the code segment. Hence, the enclave attempted to execute the `nop` instruction on line 5. This could only occur when `secret` was `true` and the `cmp` instruction cleared the equal flag.
– **Page fault is issued:** This is similar to non-branching instructions that are located on a non-executable page within the code segment. Conditional jump instructions will also lead to a #PF when they are located within the code

segment, even when their target points outside the code segment. Hence, we can derive that the `je` instruction attempts to continue execution at its specified target. This implies that the `cmp` instruction cleared the equal flag, and thus `secret` was `false`.

The above mechanism only works when targeting forward jump instructions. With backward jumps, execution will branch within the segment and another approach is required. We discuss this in more detail in the following section.

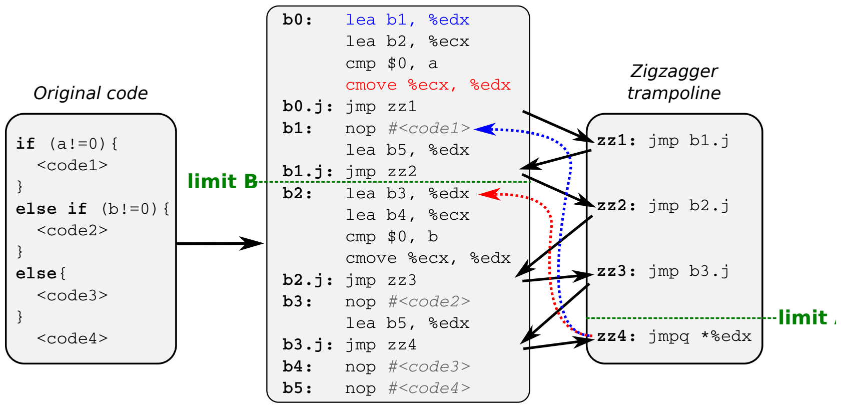
4 A Practical End-to-End Attack Scenario

In this section, we present a practical attack scenario that exploits the increased attack surface stemming from the x86 segmentation unit. Specifically, we show how the ability to infer precise byte-granular control flow information (attack variant `#3`) defeats state-of-the-art branch prediction hardening techniques.

Recent research on *branch shadowing* attacks [16] demonstrated that fine-grained enclave-private control flow leaks through the CPU-internal branch target buffer. This work also included a compile-time defense scheme called Zigzagger. The key idea, illustrated in Fig. 6, is to obfuscate secret-dependent target addresses via an oblivious `cmove` (i.e., "conditional move") instruction,[2] followed by a tight trampoline sequence of unconditional jumps that ends with a single indirect branch instruction. Zigzagger's security argument relies on the observation that (*i*) the branch shadowing attack in itself cannot directly infer the target address of the indirect branch at `zz4`, plus (*ii*) recognizing the unconditional jumps `zz1` to `zz3` becomes considerably more challenging when rapidly jumping back and forth between the instrumented code and the trampoline. Previous research on precise interrupt-driven attacks [25] has shown that condition (*ii*) is insufficient for an SGX attacker that can reliably single-step enclaved execution. To date, however, no practical attack demonstration against Zigzagger-instrumented code has been presented. We show that, when the hardened code lives in the first megabyte of a 32-bit victim enclave, condition (*i*) additionally does not hold, for general protection faults deterministically reveal the secret-dependent indirect branch target address.

We attack the Zigzagger defense by combining our segmentation attacks with SGX-Step [25]. We first revoke access rights for the page on which the Zigzagger code is located. This provides us with a starting point where we can set up our attack. Initially, we want the instrumented code to execute up to the secret dependent jump in `zz4`. As the Zigzagger trampoline is located above the instrumented code in memory, we can achieve this by lowering the code segment limit to exclude `zz4` (limit A in Fig. 6). Once page access rights have been restored, the enclave is resumed, after which a general protection fault is observed when execution reaches `zz4`. At this point a secret dependent jump is about to be made. Note that with segmentation alone, determining which branch will be

[2] The `cmove` instruction packs a condition and move into a single instruction. The move is only performed when the equal flag in the processor's status register is set.



```
if (a!=0){
  <code1>
}
else if (b!=0){
  <code2>
}
else{
  <code3>
}
  <code4>
```

```
b0:   lea b1, %edx
      lea b2, %ecx
      cmp $0, a
      cmove %ecx, %edx
b0.j: jmp zz1
b1:   nop #<code1>
      lea b5, %edx
b1.j: jmp zz2
b2:   lea b3, %edx
      lea b4, %ecx
      cmp $0, b
      cmove %ecx, %edx
b2.j: jmp zz3
b3:   nop #<code2>
      lea b5, %edx
b3.j: jmp zz4
b4:   nop #<code3>
b5:   nop #<code4>
```

```
zz1: jmp b1.j
zz2: jmp b2.j
zz3: jmp b3.j
zz4: jmpq *%edx
```

Fig. 6: Example code snippet [16] protected by Zigzagger. The secret branch address in `edx` is obfuscated with `cmov` and a tight `jmp` trampoline sequence.

taken is not possible, as lowering the segment limit to exclude any of the two secret dependent branches also excludes the jump instruction. This is where we require a secondary framework. By using SGX-Step's precise interrupt capabilities we can make sure that if we reset the segment limit and resume the enclave, at most one instruction is executed. The interrupt can also arrive early, however, after which a zero-step is taken meaning no enclaved instructions are executed. Because of this, the attacker should verify on the next interrupt whether the jump in `zz4` has executed. To do this, we revoke access rights to the page on which the Zigzagger code is located, as well as lowering the segment limit to exclude `zz4` (limit A in Fig. 6). Next, two types of faults can occur:

- **#PF**: The current instruction is within the segment, as we can pass the segmentation stage. A page fault occurs because the access rights for the underlying page have been revoked.
- **#GP**: `zz4` is still being executed, the indirect branch instruction is outside of the code segment, causing a general protection fault. This indicates that the interrupt arrived too early, causing a zero-step. In this case, we can simply retry the single-stepping process above.

Once it has been established that the jump has been executed, we can execute a final test to see which one of the branches has been jumped to. We keep page access rights revoked, but lower the segment limit to now also exclude all Zigzagger code from `b2` on (limit B in Fig. 6). When the enclave is resumed, again two types of faults may occur, following the same pattern as above:

- **#PF**: The current instruction is within the code segment. Execution is at `b1`, also indicating that `a` is not equal to 0.
- **#GP**: The current instruction is now outside of the code segment. This indicates that execution is at `b2` and `a` is equal to 0.

To evaluate our attack, we create an experimental setup where the enclaved Zigzagger code is executed 1000 subsequent times, with random values for the secret `a`. Our attack was able to correctly infer the secret branch target address in the vast majority (98%) of those runs. For the other runs, our 32-bit SGX-Step port did not interrupt the victim enclave early enough. We are confident, however, that our 32-bit port could be further fine-tuned to uphold the guarantee that no more than one instruction is executed before an interrupt. This would eliminate misses of the attack window to achieve a 100% success rate, at the expense of more interrupts arriving too early.

5 Discussion and Mitigations

Our work shows that for 32-bit enclaves, the attack surface from address translation is *not* limited to paging, but also encompasses the often overlooked x86 segmentation unit. This finding may have profound consequences for state-of-the-art defenses [8,9] that move page table memory out of reach of an attacker. Indeed, we showed that page-granular access patterns can be revealed without altering page table entries (attack variant `#1`). Moreover, we demonstrated that memory accesses in the first megabyte of a 32-bit enclave are additionally vulnerable to very precise byte-granular segmentation-based attacks. We showed how this ability (variant `#3`) can be abused to directly circumvent innovative control flow obfuscation hardening techniques [16], and can be leveraged to infer instruction sizes (variant `#2`). The latter may in turn break fine-grained, in-enclave address space layout randomization techniques [20].

Our attacks are restricted to 32-bit enclaves only, as x86 processors practically disable segmentation in 64-bit mode. At this point in time, it is hard to estimate how wide-spread 32-bit enclaves are, or eventually will be. SGX is still a developing technology and only time will tell whether people wish to enclave their legacy 32-bit software. While this assuredly limits the applicability of segmentation-based attacks, it also confirms an important hypothesis. Namely, that supporting 32-bit enclave software in the interest of backwards compatibility may introduce unexpected security vulnerabilities – as has been suggested before [6]. Exploring such legacy aspects could furthermore bring valuable insights for the design and verification of novel hardware-software PMA co-designs [7,10]. As such, we encourage further research to explore the additional attack surface stemming from enclave interaction with legacy x86 features.

While developing our attack framework, we found that recent Intel microcode updates silently address segmentation-based attacks against 32-bit enclaves. Remarkedly, we could not find any official Intel reference that documents this behavior, and can only hypothesize on the extra security checks. Specifically, we found that the patched `EENTER`/`ERESUME` instructions now immediately fault whenever any of the segment limits fall within `ELRANGE`. While this effectively prevents all attack variants `#1` to `#3`, we confirmed that the current solution still leaves (limited) segmentation-based attack surface. That is, an adversary can still detect the *use* of a particular segment by setting the segment limit to exclude the enclave base address, and observing a general protection fault whenever the segment is accessed during the enclaved execution. Since `%CS/DS` are always referenced on enclave entry, and `%FS/GS` are loaded from a trusted in-enclave data structure, only the use of `%SS/ES` can be established in this manner.

We had to fall back to manual testing to identify vulnerable microcode versions. Our results are summarized in Appendix A. As a crucial observation, however, we found that the relevant microcode updates do *not* increase the CPU Security Version Number (CPUSVN), which reflects the processor's TCB measurement for local and remote enclave attestations [1]. Importantly, since SGX's attacker model assumes a potentially malicious kernel, microcode revisions that do not increase CPUSVN can be silently rolled back without alerting the victim enclave or remote stakeholder. Only the very recent Spectre [3,14] microcode patches increase CPUSVN and adequately prevent our attacks. Our findings therefore provide additional evidence that (32-bit) enclave attestations with a pre-Spectre CPUSVN should be considered untrustworthy.

6 Conclusion

Recent research on Intel SGX side-channel attacks has focused on the paging unit, caches and branch target buffer. In this paper we have looked into a previously unexplored hardware component: the segmentation unit. We found that for 32-bit enclaves, segmentation-based attacks may reveal security sensitive information. By combining microarchitectural behavior originating from the interplay between the IA-32 segmentation and paging unit, our generalized notion of fault sequence attacks can infer very detailed information. When a 32-bit enclave uses the first 1 MiB of its address space, fine-grained control flow plus instruction sizes can be leaked to an attacker. We furthermore showed how segmentation-based attacks additionally reveal memory accesses past the 1 MiB boundary at a conventional page-level granularity.

We found that Intel has silently patched segmentation-based enclave attack surface, but without updating the CPUSVN number. This implies that kernel-level attackers are able to rollback the microcode revisions unnoticed, until SGX remote attestation schemes reject attestation reports of processors with old microcode revisions. Only with the very recent microcode patches that address the Spectre attacks, will the CPUSVN number be increased and exploitation of the segmentation unit be adequately prevented.

Responsible Disclosure and Availability

We responsibly disclosed our results to Intel and a microcode patch has been distributed. To ensure the reproducibility of our results, and to encourage future research that explores 32-bit enclave vulnerabilities, we have made the full source code of our segmentation attack framework, 32-bit SGX-Step port, and SGX SDK runtime modifications publicly available.[3]

[3] `https://distrinet.cs.kuleuven.be/software/off-limits/`

Acknowledgements

This work was partially supported by the Research Fund KU Leuven. Jo Van Bulck and Raoul Strackx are supported by a grant of the Research Foundation – Flanders (FWO).

References

1. Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, volume 13. ACM New York, NY, USA, 2013.
2. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, 2017. USENIX Association.
3. Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre attacks: Leaking enclave secrets via speculative execution. *arXiv preprint arXiv:1802.09085*, 2018.
4. Guoxing Chen, Wenhao Wang, Tianyu Chen, Sanchuan Chen, Yinqian Zhang, XiaoFeng Wang, Ten-Hwang Lai, and Dongdai Lin. Racing in hyperspace: Closing hyper-threading side channels on sgx with contrived data races. In *Security and Privacy (SP), 2018 IEEE Symposium on*. IEEE, 2018.
5. Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the 2017 Asia Conference on Computer and Communications Security*, Asia CCS '17, pages 7–18. ACM, 2017.
6. Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.
7. Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium*, pages 857–874. USENIX Association, 2016.
8. Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 857–874, Austin, TX, 2016. USENIX Association.
9. D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 190–202, Dec 2014.
10. Andrew Ferraiuolo, Andrew Baumann, Chris Hawblitzel, and Bryan Parno. Komodo: Using verification to disentangle secure-enclave hardware from software. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017.
11. Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on Intel SGX. In *Proceedings of the 10th European Workshop on Systems Security (EuroSec'17)*, 2017.
12. Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developers Manual*, 2017.
13. Intel Corporation. *Intel® Software Guard Extensions Programming Reference*, 2017.

14. Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. *ArXiv e-prints*, January 2018.
15. Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 523–539. USENIX Association, 2017.
16. 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 *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, 2017. USENIX Association.
17. Pieter Maene, Johannes Gotzfried, Ruan De Clercq, Tilo Muller, Felix Freiling, and Ingrid Verbauwhede. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Transactions on Computers*, 2017.
18. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2Nd International Workshop on Hardware and Architectural Support for Security and Privacy*, HASP '13, pages 10:1–10:1, New York, NY, USA, 2013. ACM.
19. Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware guard extension: Using sgx to conceal cache attacks. In *14th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA'17, July 2017.
20. Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.
21. Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *24th Annual Network and Distributed System Security Symposium (NDSS)*, 2017.
22. Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (ASIA CCS)*, pages 317–328. ACM, 2016.
23. Raoul Strackx, Job Noorman, Ingrid Verbauwhede, Bart Preneel, and Frank Piessens. Protected software module architectures. In *ISSE 2013 Securing Electronic Business Processes*, pages 241–251. Springer, 2013.
24. Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending sgx enclaves against page-table-based side-channel attacks. *arXiv preprint arXiv:1712.08519*, December 2017.
25. Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, pages 4:1–4:6. ACM, 2017.
26. 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 *Proceedings of the 26th USENIX Security Symposium.* USENIX Association, 2017.

27. Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2421–2434, New York, NY, USA, 2017. ACM.
28. Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

A Vulnerable Microcode Versions

Only very recently, Intel provided microcode revisions to foil our segmentation-based attacks. We tested the following microcode revisions for our Skylake machine:

version	release date	CPUSVN	vulnerable
0x1E	unknown	`020202ffffff00000000000000000000`	Yes
0x2E	unknown	`020202ffffff00000000000000000000`	Yes
0x9E	unknown	`020202ffffff00000000000000000000`	Yes
0x4A	unknown	`020202ffffff00000000000000000000`	Yes
0x8A	unknown	`020202ffffff00000000000000000000`	Yes
0xBA	April 9th, 2017	`020202ffffff00000000000000000000`	No
0xC2	November 16th, 2017	02**07**02ffffff00000000000000000000	No