SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control

Jo Van Bulck
imec-DistriNet, KU Leuven
jo.vanbulck@cs.kuleuven.be

Frank Piessens
imec-DistriNet, KU Leuven
frank.piessens@cs.kuleuven.be

Raoul Strackx
imec-DistriNet, KU Leuven
raoul.strackx@cs.kuleuven.be

Abstract

Protected module architectures such as Intel SGX hold the promise of protecting sensitive computations from a potentially compromised operating system. Recent research convincingly demonstrated, however, that SGX's strengthened adversary model also gives rise to to a new class of powerful, low-noise side-channel attacks leveraging first-rate control over hardware. These attacks commonly rely on frequent enclave preemptions to obtain fine-grained side-channel observations. A maximal temporal resolution is achieved when the victim state is measured after every instruction. Current state-of-the-art enclave execution control schemes, however, do not generally achieve such instruction-level granularity.

This paper presents SGX-Step, an open-source Linux kernel framework that allows an untrusted host process to configure APIC timer interrupts and track page table entries directly from user space. We contribute and evaluate an improved approach to single-step enclaved execution at instruction-level granularity, and we show how SGX-Step enables several new or improved attacks. Finally, we discuss its implications for the design of effective defense mechanisms.

CCS Concepts • **Security and privacy → Side-channel analysis and countermeasures**;

Keywords Intel SGX, Controlled-Channel, Interrupt

1 Introduction

Today's computing platforms rely on privileged system software to separate applications, and to govern the interactions between them. Commodity monolithic Operating System (OS) kernels, however, consist of millions of lines of code written in unsafe languages, exposed to both logical bugs and low-level software vulnerabilities. In response to these concerns, the past years have seen a significant research effort [3, 6, 9] on Protected Module Architectures (PMAs) that support isolated execution of security-sensitive application components or *enclaves* with a minimal Trusted Computing Base (TCB). These proposals have in common that they enforce security primitives directly in hardware, or in a small hypervisor, so as to prevent the untrusted OS from accessing enclaved code or data directly, while still leaving it in charge of shared platform resources such as system memory or CPU time. With the arrival of Intel's Software Guard eXtensions (SGX) [6, 7], such strong hardware-enforced trusted computing guarantees are now available on mainstream consumer devices.

Recent research demonstrated, however, that the increased capabilities of a privileged PMA attacker allow her to construct high-resolution, low-noise channels to spy on enclaved execution. Specifically, the past months have seen a steady stream of kernel-level SGX attacks exploiting information leakage from page tables [13, 15], CPU caches [4, 10], or branch prediction units [8]. These attacks commonly exploit the OS's control over timer devices to gain fine-grained side-channel observations from frequent enclave preemptions. As such, the precision at which one can interrupt a victim enclave, determines the temporal resolution of the attack.

This paper shows that enclaved execution can be reliably monitored at a *maximal* temporal resolution (i.e., instruction per instruction). Specifically, we present and evaluate SGX-Step, which is the first framework of its kind to achieve true single-stepping for arbitrary enclave programs. We furthermore lower the bar for enclave preemption attacks considerably by exporting user space memory mappings for the local APIC timer device and enclave page tables. As part of our evaluation, we defeat a recently proposed branch prediction defense [8], demonstrating SGX-Step's enhanced precision over previous proposals. Summarized, we make the following contributions:

- We show that enclaved execution can be precisely single-stepped using a novel APIC timer manipulation.
- We implement SGX-Step as an open-source[1] Linux kernel driver and runtime library, and explain how it improves the temporal resolution of existing attacks.
- We evaluate our approach on two different SGX processors, and provide evidence that SGX-Step enables new attacks that were previously deemed infeasible.

SysTEX'17, October 2017, Shanghai, China

ACM ISBN 978-1-4503-5097-6/17/10…$15.00
https://doi.org/10.1145/3152701.3152706

[1] https://github.com/jovanbulck/sgx-step

2 Background and Related Work

2.1 Attacker Model and Intel SGX

Ongoing concerns on protecting sensitive data from software running at higher privilege levels have led to the Software Guard eXtensions (SGX) [6, 7] included in recent Intel x86 processors. SGX enables hardware-enforced isolation and attestation of security-critical code in *enclaves*, embedded in the virtual address space of a conventional OS process. Legacy page tables are left under explicit control of the untrusted OS, but the processor's Memory Management Unit (MMU) enforces that enclave-private memory can never be directly accessed from outside. Hardware-level cryptography furthermore allows the untrusted OS to initialize enclaves, and swap in/out protected pages to untrusted storage.

Enclave code is restricted to user mode, and has access to all its protected pages, as well as to the unprotected part of the application's address space. Dedicated CPU instructions switch the processor in or out of *enclave mode.* EENTER transfers control from the unprotected application context to a predetermined location inside the enclave, and the EEXIT instruction can be used to exit an enclave programmatically. Alternatively, in case of a fault or external interrupt, the processor executes an Asynchronous Enclave Exit (AEX) procedure that saves the execution context securely in a preallocated State Save Area (SSA) inside the enclave, and replaces the CPU registers with a synthetic state to avoid direct information leakage to the untrusted Interrupt Service Routine (ISR). The AEX procedure also takes care of pushing a predetermined Asynchronous Exit Pointer (AEP) on the unprotected call stack, so as to allow the OS interrupt handler to return transparently to unprotected trampoline code outside the enclave. From this point, an interrupted enclave can be continued by means of the ERESUME instruction.

To aid enclave development, SGX differentiates between *debug* and *production* enclaves, where private memory of the former is accessible from outside via special ring-0 EDBGRD/EDBGWR instructions. Debug operations are ignored for production enclaves, however, such that they are provided with strong isolation of code and data memory. SGX furthermore includes measures against obvious interference with production enclaves. Specifically, in enclave mode, the processor ignores performance counters, hardware breakpoints, and the single-step trap flag (RFLAGS.TF).

2.2 Enclave Preemption Attacks

Given SGX's strong adversary model, several recent studies have looked into its side-channel attack surface. Given the scope of this paper, we focus exclusively on attacks that preempt the enclaved execution, but it is worth noting that some recent L1 cache attacks [1, 11] can be mounted from a co-resident logical processor, without interrupting the victim enclave. Enclave preemption attacks on the other hand either leverage page faults or interrupts to inspect enclave behavior.

Fault-Driven Attacks. Seminal work by Xu et al. [15] first showed how carefully revoking access rights on enclave pages and observing the associated page faults, allows an adversarial OS to extract large amounts of sensitive data (full text, and images) from SGX enclaves. Subsequent work [14] has leveraged page faults as an enclave execution control technique to more easily exploit thread synchronization bugs in enclaves. Since page faults are triggered deterministically by the hardware, fault-driven attacks generally suffer from very little to no noise. A fundamental limitation of this channel, however, concerns the relatively coarse-grained (4 KB) granularity at which page faults reveal memory accesses. Moreover, in order for the enclaved execution to continue, access rights on the faulting pages should be restored.

Interrupt-Driven Attacks. More recent SGX attacks improve over the spatial resolution of the page fault channel by exploiting information leakage at a cache line granularity. Not all, but a significant fraction of these attacks suspend the victim enclave to obtain precise side-channel observations. Earlier proposals such as CacheZoom [10] rely on a rather coarse-grained kernel patch to interrupt the victim enclave more frequently. More recent work by Hähnel et al. [4] significantly improves the temporal resolution of enclave cache attacks by directly configuring the local APIC timer in kernel space. While their approach approximately interrupts enclaved execution every three instructions, true single-stepping is not achieved, since *(i)* they focus on instructions with memory operands only, and *(ii)* the approach was implemented and evaluated in a software simulator, leaving intricate microarchitectural interactions with real SGX hardware fundamentally unclear.

Recent research [8] on *branch shadowing* attacks demonstrated that enclave-private control flow can be inferred by abusing cache collisions in the CPU-internal Branch Target Buffer (BTB). Such attacks critically rely on the periodic interleaved execution of the victim enclave with carefully aligned spy shadow code. Lee et al. [8] employ a kernel patch to achieve a relatively coarse-grained enclave interrupt granularity of about 50 instructions, which can be further improved to about 5 instructions by disabling the CPU cache hierarchy entirely (CR0.CD). Note however that disabling caching of course also invalidates aforementioned CPU cache attacks.

Finally, our own previous work [13] on stealthy page table-based attacks relies on frequent enclave preemptions to measure page table access patterns. This work also introduced a highly accurate PTE FLUSH+FLUSH technique, where a concurrent spy thread running on another logical core continuously monitors a specific page table entry, and sends an inter-processor interrupt upon detecting an access. Note that this approach is distinct from single-stepping in that the enclaved execution is only preempted when a specific trigger page was accessed, whereas SGX-Step interrupts *each* instruction sequentially.

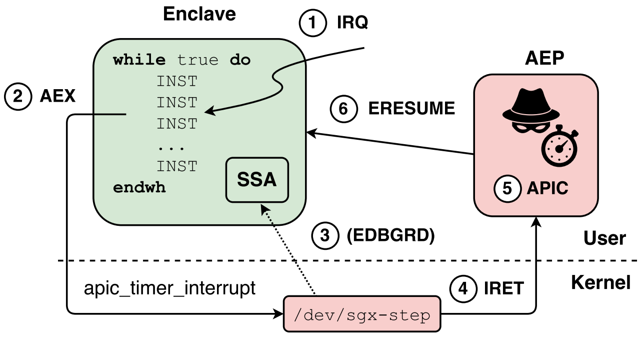


Figure 1. Framework for single-stepping SGX enclaves.

3 Design and Implementation

Our single-stepping objective is to execute an enclave one instruction at a time. Note that advanced x86 hardware debug assistance features such as the single-step trap flag (RFLAGS.TF) or hardware breakpoints are explicitly suppressed in enclave mode [6]. Our implementation therefore leverages the OS’s control over hardware timer devices to *emulate* this behavior with frequent enclave interrupts.

APIC Timer Configuration. Every Intel processor comes with a local Advanced Programmable Interrupt Controller (APIC) [6] to configure and deliver interrupts destined for that core. The APIC also contains a timer that can be operated in one of three modes. In one-shot or periodic mode, the timer is configured through memory-mapped I/O registers. Specifically, by writing into an initial-count register, an internal current-count register can be initialized. The local APIC decrements the current-count at the CPU’s bus frequency, divided by the value specified in the divide-configuration register, and generates an interrupt whenever the current-count reaches zero. In one-shot mode a single interrupt is generated, whereas in periodic mode the initial-count is automatically copied back into the current-count register. Alternatively, in TSC-deadline mode, an interrupt is generated when the CPU’s internal timestamp counter reaches the absolute value specified in a dedicated model-specific register. This mode is substantially more precise, since the timestamp counter operates at the processor’s nominal frequency, instead of the much slower external bus frequency. The Skylake CPUs used in the evaluation for instance run at a base frequency of 2.5 GHz and 3.4 GHz, whereas the fixed external bus frequency is only 100 MHz (25/34 times slower).

To facilitate APIC configuration, SGX-Step comes with a runtime library that creates user space virtual memory mappings for the physical APIC memory I/O configuration registers. By writing into the exported memory locations, the untrusted host process can easily configure the APIC timer one-shot/periodic interrupt source or trigger inter-processor interrupts directly from user space. Figure 1 summarizes the sequence of hardware and software steps when interrupting and resuming an SGX enclave through our framework. ① The local APIC timer interrupt arrives within an enclaved instruction. ② The processor executes the AEX procedure that securely stores execution context in the enclave’s SSA frame, initializes CPU registers, and vectors to the kernel-level interrupt handler. ③ Our `/dev/sgx-step` loadable kernel module registered itself in the APIC event call back list to make sure it is called on every timer interrupt. At this point, any attack-specific, kernel-level spy code can easily be plugged in. Furthermore, to enable precise evaluation of our approach on attacker-controlled debug enclaves, SGX-Step can *optionally* be instrumented to retrieve the stored instruction pointer from the interrupted enclave’s SSA frame using the EDBGRD instruction. ④ The kernel returns to the user space AEP trampoline. We modified the untrusted runtime of the official SGX SDK to allow easy registration of a custom AEP stub. ⑤ At this point, any attack-specific user mode spy code can again easily be run, before the single-stepping adversary configures the APIC timer for the next interrupt, just before executing ⑥ ERESUME.

Timer Interval Prediction. With our framework in place, the only remaining challenge is to establish a suitable, platform specific timer interval so as to interrupt the first instruction executed by the enclave after ERESUME. The timer interrupt should not systematically arrive too soon, within the monolithic ERESUME instruction, as then no progress would be made (i.e., *zero-step*). Alternatively, should the interrupt arrive too late after completion of ERESUME, more than one instruction would be executed (i.e., *multi-step*). The single-stepping adversary is therefore required to accurately predict the duration between the moment the timer is configured and completion of ERESUME. Naturally, due to modern processor optimizations, execution time prediction becomes increasingly difficult the more code is actually executed in the timer interval. In this respect, previous enclave preemption attempts [4, 8, 10] all configure the APIC timer in kernel space, whereas enclaves have to be resumed in user mode. Consequently, these approaches suffer from significant timer jitter stemming from the considerable amount of code and a privilege level switch in the interrupt return path.

An important contribution of our framework therefore is that we drastically cut the amount of code in the timer interval path by directly configuring the APIC timer from user space. As a result, SGX-Step reduces the timer configuration challenge to prediction of ERESUME execution time, which we found to be relatively deterministic on our evaluation platforms. Our user-space APIC timer trick only works for the aforementioned single-shot or periodic timer modes, however, since TSC deadline configuration requires the privileged WRMSR instruction. We thus improve timer interval predictability at the cost of a lower timer frequency. Note that this inconvenience can be overcome, however, for instance by executing a deterministic amount of NOP instructions between timer configuration and ERESUME.

In our experimental setup, we operate the APIC timer in one-shot mode with division 2. As explained above, timer configuration depends on CPU frequency, and hence remains inherently platform-specific. We established suitable timer intervals for both our evaluation platforms through an empirical approach that leverages SGX-Step to retrieve the interrupted instruction pointer from an attacker-controlled debug calibration enclave. We leave exploration of fully automated timer configuration approaches as future work.

Monitoring Page Table Entries. Single-stepping enclaved execution incurs a substantial slowdown, and is often only desired for some specific functions of interest. SGX-Step therefore allows an adversary to initiate single-stepping mode after a specific code or data page has been accessed, using enclave preemption from either page faults [15] or a dedicated spy thread [13]. Specifically, analogous to the APIC configuration trick above, SGX-Step establishes user space virtual memory mappings for the *unprotected* physical memory containing the victim enclave's Page Table Entries (PTEs). By manipulating PTEs directly from user space, an adversary can provoke page faults ("present" bit), or gain insight in enclave memory usage ("accessed" and "dirty" attributes).

4 Evaluation

We evaluate the effectiveness of SGX-Step on both a mid-end laptop and a higher-end desktop CPU. We first provide microbenchmarks, and afterwards demonstrate the enhanced attack potential of SGX-Step in two scenarios that are not exploitable with current, state-of-the-art techniques.

All experiments were conducted on real, off-the-shelf SGX hardware. Our first evaluation platform is a commodity Dell Inspiron 13 7359 laptop running a generic Linux 4.2.0 kernel on a Skylake dual-core Intel i7-6500U CPU with a base frequency of 2.5 GHz. Our Dell Optiplex 7040 desktop, on the other hand, features a generic Linux 4.4.0 kernel and a Skylake quad-core i7-6700 processor running at 3.4 GHz. Like previous SGX preemption attacks [4, 8, 10, 13] and conformant to our attacker model, we disabled TurboBoost plus dynamic frequency scaling (C-States, SpeedStep), and affinitized the victim enclave thread to a specific logical core to increase predictability on both machines.

4.1 Single-Stepping Microbenchmark

Our objective is to reliably single-step *arbitrary* enclave programs, including inexpensive instructions without memory operands. To evaluate how accurately SGX-Step realizes such true single-stepping, we constructed a challenging microbenchmark experiment featuring a test enclave with a long slide of successive NOP instructions. At the microarchitectural level, a 1-byte NOP is the lowest cost instruction, consuming only a single micro-op without memory or register dependencies [5]. As such, many NOPs can be executed in a limited time window, and even a relatively small amount of jitter on timer interrupt arrivals can lead to the execution of multiple NOPs in the benchmark enclave. Hence, we argue that an approach that reliably single-steps a NOP instruction slide, can easily single-step arbitrary instructions as well.

Table 1. Interrupts categorized according to the number of instructions executed in the victim enclave (i.e., zero-step, single-step, or multi-step). When laptop/desktop experimental results differ, we present the laptop measurements first.

Experiment	0-Step	1-Step	> 1	1-Step Ratio
NOP	2,083 / 1,617	100,000	0	97.96 / 98.41%
strlen	8,829 / 4,982	460,000	0	98.12 / 98.93%
Zigzagger	5,739 / 2,872	210,000	0	97.34 / 98.65%

Our benchmark enclave executes a slide of 100,000 successive NOP instructions. As part of the experiment, we instructed the SGX-Step driver to retrieve the instruction pointer from the state save area of the interrupted debug enclave using the EDBGRD instruction, so as to infer the exact number of instructions executed in between two successive enclave interrupts.[2] In the evaluation on our laptop/desktop platforms, we measured a total of respectively 102,083 and 101,617 interrupts for the instruction slide. We confirmed that exactly 2,083/1,617 out of these did not change the enclave instruction pointer (i.e., zero-step), whereas the remaining 100,000 interrupts caused a *single* increment of the enclave instruction pointer. We thus conclude that SGX-Step was able to reliably single-step all 100,000 NOPs, without ever executing more than one NOP at a time. A small fraction of interrupts (2.04% on the laptop and 1.59% for the desktop) actually arrived too early, within the ERESUME instruction. These interrupts are superfluous, but rather harmless as they do not result in enclaved code being executed.

4.2 Precise Enclave Execution Control Attacks

Determining String Length. Previous work [4] explored the temporal resolution limits of the page fault channel, discussed in Section 2.2. That is, since an attacker needs to restore access rights on faulting pages in order to guarantee progress, fault-driven attacks cannot infer information from enclaved functions that access a single code and data page. As an example of such a function, consider the elementary `strlen` implementation in Fig. 2. Assuming the compiler uses a CPU register for the loop counter, the entire loop easily fits within a single code page, and every iteration accesses only one data page (containing the string). As such, progress

[2] Note that EDBGRD only serves evaluation purposes, to establish the number of instructions executed in the benchmark enclave, and would not be used in real attacks against production enclaves.

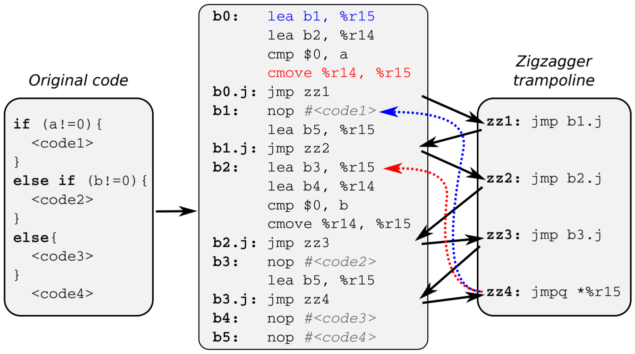
```
size_t strlen (char *str)
{
  char *s;

  for (s = str; *s; ++s);
  return (s - str);
}
```

```
     mov    %rdi,%rax
  1: cmpb   $0x0,(%rax)
     je     2f
     inc    %rax
     jmp    1b
  2: sub    %rdi,%rax
     retq
```

Figure 2. Example of secret-dependent data accesses in a tight loop (source code and compiled assembly form).

can only be made if both the strlen code page and secret string data page are accessible. That is, the length of the secret string cannot be inferred from page fault sequences. Previous research [13] has shown, however, that page accesses can be observed without page faults, for instance by querying the PTE “accessed” bit after interrupting the enclave. We thus leverage SGX-Step to single-step the tight strlen execution loop, each time recording/clearing the “accessed” bit of the PTE referencing the string being processed. Note that accurate single-stepping results themselves also allow the string length to be inferred from the number of interrupts (i.e., instructions executed by the victim enclave).

The right hand side of Fig. 2 provides the assembly version of the strlen C source code on the left. We explicitly compiled the code with optimizations set for size (-Os) to ensure a very compact loop with only 4 assembly instructions and a single memory operand. Note that precisely single-stepping this loop is considerably more challenging than the case without optimizations (totalling 5 instructions and 3 memory operands). In our experimental setup, we single-stepped a benchmark enclave that processed the string "SysTEX 2017" (11 characters) 10,000 successive times. On every interrupt, just before resuming the enclave, we queried the PTE “accessed” bit from the user space AEP trampoline handler. We correctly recognized the string length for all 10,000 strlen invocations. Additionally, we analyzed the full enclave instruction pointer trace, retrieved with EDBGRD, to categorize interrupts according to the amount of instructions executed in the victim enclave. The results are in Table 1. A first important finding, in line with our microbenchmark observations, is that SGX-Step reliably single-stepped all 460,000 instructions on both the laptop and desktop processors, and without ever executing more than one instruction per interrupt. Only a relatively small fraction of the total number of interrupts ($\leq$ 1.88%) arrived within ERESUME and did not result in an enclaved instruction being executed. These zero-step observations can be easily filtered out, as we confirmed that they never falsely triggered the “accessed” bit of the string PTE.



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

```
b0:   lea b1, %r15
      lea b2, %r14
      cmp $0, a
      cmove %r14, %r15
b0.j: jmp zz1
b1:   nop #<code1>
      lea b5, %r15
b1.j: jmp zz2
b2:   lea b3, %r15
      lea b4, %r14
      cmp $0, b
      cmove %r14, %r15
b2.j: jmp zz3
b3:   nop #<code2>
      lea b5, %r15
b3.j: jmp zz4
b4:   nop #<code3>
b5:   nop #<code4>
```

```
zz1: jmp b1.j

zz2: jmp b2.j

zz3: jmp b3.j

zz4: jmpq *%r15
```

Figure 3. Example code snippet protected by Zigzagger. The final target address in r15 is obfuscated with CMOV and a tight trampoline sequence of JMP instructions (from [8]).

Defeating Zigzagger. Section 2.2 introduced *branch shadowing attacks* that rely on frequent enclave preemptions to execute shadow probing code for inferring enclave-private control flow via targeted BTB cache collisions. This recent work [8] also includes a compile-time defense scheme called Zigzagger. The key idea, illustrated in Fig. 3, is to obfuscate the target address of a conditional jump via a CMOV instruction, followed by a tight trampoline sequence of unconditional jumps that ends with a single indirect branch instruction. By rapidly jumping back and forth between the instrumented code and the trampoline, Zigzagger makes recognizing the current branch instruction considerably more challenging. Its security argument states that “since all of the unconditional branches are executed almost simultaneously in sequence, recognizing the current instruction pointer is difficult” [8]. Moreover, the branch shadowing attack in itself cannot directly infer the secret target address of the indirect branch at zz4. We show, however, that even Zigzagger-instrumented code can be reliably single-stepped. Specifically, an attacker leveraging SGX-Step can reliably probe *each* intermediate unconditional trampoline jump (i.e., zz1 to zz3). Observe that after branching to the secret target address, execution continues at one of only two possible target addresses, and eventually lands on the Zigzagger trampoline at either zz2 or zz3. As such, a single-stepping adversary can infer the secret if condition, after detecting execution of the indirect branch at zz4, by probing the unconditional zz2 jump – which is only executed for the first code block.

We evaluate a proof-of-concept Zigzagger attack by repeatedly single-stepping the hardened assembly code[3] from Fig. 3. Specifically, we single-stepped a benchmark enclave that executes the 21-instruction code snippet 10,000 successive times, and afterwards analyzed the EDBGRD instruction pointer trace to establish the number of instructions executed on every interrupt. In line with our previous findings, Table 1 shows that SGX-Step *never* executes more than one instruction in the victim enclave per interrupt, allowing precise execution of the shadow code on both evaluation platforms.

[3] Since Zigzagger and the attack code from [8] were not made public, we repeat the example assembly code snippet from that paper here. For the same reason, we could not launch the actual branch shadowing attack, only showing its feasibility with our single-stepping results.

Hence, these benchmarks can be considered clear evidence that SGX-Step enables *new* attacks, previously deemed infeasible. The superfluous zero-step interrupt fractions (2.66% for the laptop and only 1.35% on the desktop) also keep on par with previous observations, and do not impede a real attack since the BTB cache remains unaffected by the victim.

5 Discussion

Attack Resolution and Implications. We showed that enclaves can be reliably interrupted one instruction at a time. In this, SGX-Step improves significantly over related state-of-the-art enclave preemption schemes that only approximate such instruction-level granularity after either disabling the CPU cache [8], or focussing exclusively on instructions with memory operands in a simulator [4]. From a practical perspective, SGX-Step furthermore lowers the bar for precise enclave preemption attacks from user space.

These findings have profound consequences for the design of effective defenses. Specifically, compiler-based techniques are fundamentally insufficient when they rely on (partial) *atomic behavior* of the instruction stream, as effectively demonstrated for the Zigzagger [8] branch obfuscation technique above. Our precise `strlen` attack furthermore highlights the inadequacy of defenses that focus on "aligning specific code and data blocks to exist entirely within a single page", as still officially recommended by Intel [7].

Detecting Suspicious Interrupts. Heuristic compiler defenses, on the other hand, could focus on detecting high interrupt rates as an artefact of an ongoing attack. Importantly, in contrast to enhanced PMA designs such as Sanctum [3], SGX enclaves are explicitly left *interrupt-unaware*, since they ought to be resumed through a dedicated ERESUME instruction. However, a contemporary line of research [2, 12] leverages x86 Transactional Synchronization eXtensions (TSX) to detect page faults or interrupts in enclave mode.

T-SGX [12] protects against page fault-based attacks by wrapping each basic block in a TSX transaction, and aborting the enclave after counting too many consecutive transaction aborts. Déjà Vu [2] instruments an enclave program to detect frequent preemptions through a reliable in-enclave reference clock thread that uses TSX to ensure it cannot be silently stopped by an untrusted OS. Both solutions would recognize the frequent interrupt rates generated by SGX-Step, but also suffer from several important limitations, however. First, an SGX-enabled processor (e.g., the laptop we used in our experiments) is not always shipped with TSX extensions, ruling out this defense for critical infrastructural software such as Intel's Launch and Quoting Enclaves. Second, TSX defenses incur a significant run-time performance overhead [2, 12]. Third, these defenses cannot offer full protection as they rely on heuristics to recognize suspicious interrupt rates, which could also be caused by repeated cache conflicts or benign interrupts under heavy system load.

6 Conclusion

Our work shows that enclaved execution can be accurately single-stepped one instruction at a time. We demonstrated SGX-Step's improved temporal resolution over state-of-the-art preemption schemes in two challenging attack scenarios, highlighting the need for adequate defense mechanisms.

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

References

[1] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. 2017. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT '17)*. USENIX Association.

[2] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. 2017. 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)*. ACM, 7–18.

[3] Victor Costan, Ilia Lebedev, and Srinivas Devadas. 2016. Sanctum: Minimal hardware extensions for strong software isolation. In *Proceedings of the 25th USENIX Security Symposium*. USENIX Association.

[4] Marcus Hähnel, Weidong Cui, and Marcus Peinado. 2017. High-resolution side channels for untrusted operating systems. In *2017 USENIX Annual Technical Conference (ATC '17)*. USENIX Association.

[5] Intel Corporation. 2016. *Intel 64 and IA-32 architectures optimization reference manual.* Reference no. 248966-033.

[6] Intel Corporation. 2017. *Intel 64 and IA-32 architectures software developer's manual.* Reference no. 325462-062US.

[7] Intel Corporation. 2017. *Intel software guard extensions developer guide.* software.intel.com/en-us/documentation/sgx-developer-guide.

[8] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. 2017. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association.

[9] P. Maene, J. Götzfried, R. de Clercq, T. Müller, F. Freiling, and I. Verbauwhede. 2017. Hardware-based trusted computing architectures for isolation and attestation. *IEEE Trans. Comput.* 99 (2017).

[10] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX amplifies the power of cache attacks. In *Conference on Cryptographic Hardware and Embedded Systems (CHES '17)*.

[11] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware guard extension: Using SGX to conceal cache attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA '17)*.

[12] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*. San Diego, CA.

[13] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*. USENIX Association.

[14] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security (ESORICS '16)*. Springer.

[15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. 2015. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy*. IEEE, 640–656.