AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves

Scott Constable[1], Jo Van Bulck[2], Xiang Cheng[3], Yuan Xiao[1], Cedric Xing[1], Ilya Alexandrovich[1], Taesoo Kim[3], Frank Piessens[2], Mona Vij[1], and Mark Silberstein[4]

[1]*Intel Corporation,* [2]*imec-DistriNet, KU Leuven,* [3]*Georgia Institute of Technology,* [4]*Technion*

Abstract

Intel® Software Guard Extensions (Intel® SGX) supports the creation of shielded *enclaves* within unprivileged processes. While enclaves are architecturally protected against malicious system software, Intel SGX's privileged attacker model could potentially expose enclaves to new powerful side-channel attacks. In this paper, we consider hardware-software co-design countermeasures to an important class of *single-stepping* attacks that use privileged timer interrupts to precisely step through enclave execution exactly one instruction at a time, as supported, e.g., by the open-source SGX-Step framework. This is a powerful deterministic attack primitive that has been employed in a broad range of high-resolution Intel SGX attacks, but so far remains unmitigated.

We propose AEX-Notify, a flexible hardware ISA extension that makes enclaves *interrupt aware*: enclaves can register a trusted handler to be run after an interrupt or exception. AEX-Notify can be used as a building block for implementing countermeasures against different types of interrupt-based attacks in software. With our primary goal to thwart deterministic single-stepping, we first diagnose the underlying hardware behavior to determine the root cause that enables it. We then apply the learned insights to remove this root cause by building an efficient software handler and constant-time disassembler to transparently determine and atomically prefetch the working set of the next enclave application instruction.

The ISA extension we propose in this paper has been incorporated into a revised version of the Intel SGX specification.

1 Introduction

Intel® Software Guard Extensions (Intel® SGX) [16] is both the first commodity trusted execution environment (TEE) and the first TEE to become widely used in confidential computing. Intel SGX enables the construction of shielded regions of user-space memory called *enclaves*. Architectural access controls and hardware features such as memory encryption protect Intel SGX enclaves from privileged software adversaries, as well as some physical attacks. Applications range from federated learning and data analytics to generic confidential or shielded containers that can protect a variety of workloads. Importantly, the current and future generations of Intel Xeon processors improve the support for these emerging use-cases by increasing Intel SGX's protected memory capacity by up to 1500×.

At the same time, Intel SGX introduced a new, strong adversary model, which has incentivized novel offensive research. For example, researchers have found side-channel approaches that infer confidential data from enclaves without violating Intel SGX's architectural security properties. These approaches often use processor features available only to privileged software to enable or amplify various side-channel analysis techniques [2, 26, 45, 63–66, 69]. For example, Xu et al. [71] first showed how a malicious OS can manipulate an Intel SGX enclave's page tables to trigger a fault each time the enclave accesses a different 4 KiB page, exposing a noise-free trace of the code and data pages accessed by the enclave. A variety of solutions have been proposed to mitigate this class of controlled-channel attacks [18, 47, 56, 59].

A remaining unsolved problem is a class of attacks that abuse privileged hardware interfaces, such as the advanced programmable interrupt controller (APIC), to collect side-channel measurements at a very fine temporal resolution—ultimately even at a *maximal* instruction-level granularity [63]. An open-source tool called SGX-Step [63] has made this APIC-based "single-stepping" technique widely available to academic researchers. SGX-Step has, subsequently, been used in a long and ongoing line of high-resolution, interrupt-driven side-channel attacks against Intel SGX [2, 3, 7, 10, 19, 26, 30–32, 43, 45, 49–51, 54, 57, 60–62, 64]. A considerable fraction of these attacks critically relies on SGX-Step's *deterministic* single-stepping ability to sample at a perfect, instruction-level granularity. For instance, to reconstruct precise execution timings of individual enclave instructions [28, 49, 64], or to overcome the coarse-grained 4 KiB spatial resolution of prior controlled-channel attacks by observing the *exact* number of single-stepped instructions within a slightly unbalanced, intra-page conditional control

flow [2,3,36,45,62]. Some attacks have, furthermore, used a related "zero-step" technique to replay transient execution [58] or repeat power consumption measurements [43].

While side-channel attacks and their mitigations have received justifiable attention over the past several years, the study of fine-grained execution control techniques—and especially their mitigations—has been pursued with less fervor. Existing mitigation attempts fall fundamentally short in that they do not address the root cause, e.g., heuristically detecting suspicious interrupt rates [9,46,55], observing performance-monitoring counters [39], or resorting to data randomization schemes [38]. Interrupt detection heuristics are impractical in real settings where benign interrupt storms can trigger false positives. Performance-monitoring mitigations have been proven to be fragile and ineffective [31,35]. Moreover, several of these solutions are not compatible with existing hardware [39,47,59] or require custom compilers and scarcely available (deprecated) CPU extensions, such as Intel® Transactional Synchronization Extensions (Intel® TSX) [9,38,55].

In this paper, we present principled experiments that illuminate the true root cause of the hardware behavior that allows SGX-Step to single step an Intel SGX enclave. Our findings suggest that it is not possible to mitigate SGX-Step with hardware modifications alone. Hence, we propose a novel *hardware-software co-design*. The hardware component is an instruction set architecture (ISA) extension to Intel SGX called AEX-Notify, which allows enclaves to opt-in to a notification delivered by the processor whenever the enclave is interrupted or encounters an exception. The software component resides in the trusted enclave shielding runtime and consists of a constant-time disassembler and a crafted assembly stub that transparently speeds up the next instruction to be executed by the enclave application by atomically prefetching its working set, thus making the application instruction statistically unlikely to "hit" with a timer interrupt.

We use empirical evidence paired with statistical reasoning to evaluate the degree to which the mitigation prevents the adversary from deterministically single-stepping an enclave instruction stream. Furthermore, we comprehensively categorize the landscape of existing interrupt-driven Intel SGX attacks, and we conclude that our proposed defense may thwart attacks that critically rely on deterministic single-stepping, including the powerful interrupt latency [28,49,64] and interrupt counting [2, 3, 36, 45, 62] primitives, while also more generally limiting the temporal resolution of Intel SGX side-channel [10, 20, 26, 27, 30, 32, 41, 44, 57] or transient-execution [51, 54, 60, 61] attacks that employ frequent, interrupt-driven probing.

The ISA extension proposed in this paper has been incorporated into a recent revision to the Intel SGX specification [12]. We conclude the paper with a brief survey of other attacks that could conceivably be mitigated using AEX-Notify.

Contributions. In summary, our main contributions are:

- We propose a novel hardware-software co-design called AEX-Notify that allows Intel SGX enclave software to mitigate interrupt-driven attacks by deploying mitigations in software. We demonstrate the viability of this approach by applying it to completely disrupt deterministic single-stepping attacks against Intel SGX enclaves.
- We design and implement a novel *constant-time instruction decoder* that allows the mitigation to protect the vast majority of x86 instructions from malicious single-stepping or zero-stepping (e.g., 98.0% of instructions across 106 Intel SGX runtime binaries).
- We identify the hardware behavior that enables the malicious single-stepping technique used by SGX-Step. This analysis has informed the first two contributions.
- We evaluate the effectiveness of our approach through empirical observations in an attacker-favored experimental setup, paired with statistical reasoning.
- Our proposed ISA extension has been adopted by Intel.

2 Background

2.1 Intel SGX

Memory Management. An Intel SGX enclave is a hardware-protected contiguous region of virtual memory within a process. Threads outside the enclave cannot execute code within the enclave, nor can they read data from the enclave. An enclave's page tables reside outside the enclave and are managed by the untrusted OS. Hence, Intel SGX uses a protected enclave page cache map to preserve the integrity of enclave memory translations. This mechanism does not, however, prevent the OS from manipulating other paging controls, such as the present (P), accessed (A), and dirty (D) bits, which have been used to demonstrate novel side-channel attacks against Intel SGX [65,66,71]. Intel SGX also flushes translation lookaside buffer (TLB) entries for enclave addresses when entering and exiting an enclave [16]. This prevents stale virtual address translations from potentially violating Intel SGX's confidentiality and integrity properties.

Execution Control. An enclave may have one or more thread control structures (TCSs), each of which defines a fixed entry point within the enclave. A thread enters an enclave by invoking the `EENTER` instruction with the address of one of the enclave's TCSs. A thread exits the enclave by invoking the `EEXIT` instruction. The Intel SGX SDK composes `EENTER` and `EEXIT` into an abstraction called an `ecall` [33], which is similar to calling a function exported from a shared object or DLL. Inversely, the SDK also composes `EEXIT` and `EENTER` into an `ocall` abstraction that allows the enclave to call outward to untrusted functions. Because the TCS allows a single fixed entry point, the SDK's enclave entry logic must examine register arguments and/or internal state to determine whether to execute an `ecall`, an `ocall`, or to handle an exception.

Intel SGX enclaves may also encounter asynchronous exiting events, including inter-processor interrupts (IPIs), timer

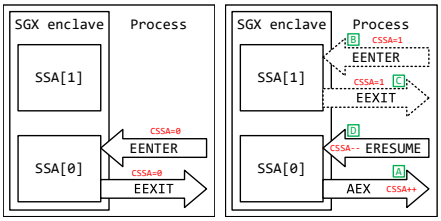

Figure 1: Intel SGX flows to call into and out of an enclave (left); flows to asynchronously exit (A) and resume (D) the enclave, and optionally handle an exception (B, C) (right).

interrupts from the APIC, and exceptions such as page faults. Any of these events trigger an asynchronous enclave exit (AEX), which saves the current processor context to a state-save area (SSA) frame within the enclave, indexed by a TCS field called `TCS.CSSA` (current SSA). After saving the context, AEX increments `TCS.CSSA`. Untrusted software can later invoke the `ERESUME` instruction, which decrements `TCS.CSSA` and restores processor context from `SSA[TCS.CSSA]`, thus allowing the thread to resume enclave execution at the point where the AEX was triggered. If the enclave must handle an exception (e.g., triggered by executing an unsupported instruction) then the untrusted runtime may instead invoke `EENTER`. The handler can determine and address the cause of the exception, e.g., by examining and modifying the contents of `SSA[0]`. The thread may then `EEXIT` the handler and `ERESUME` the enclave application. Unlike `ERESUME`, `EENTER` does not decrement `TCS.CSSA`. Therefore, a second AEX will save the processor context to `SSA[1]`, leaving intact the context saved to `SSA[0]` during the first AEX.

The interaction between these instructions and the SSA frames is depicted in Figure 1. Note that most Intel SGX runtimes use a two-stage exception handler design. The first stage executes in the `SSA[1]` context and is carefully implemented to avoid triggering an exception that would require a third SSA frame to handle. If the first stage cannot diagnose or resolve the exception, it expands the call stack, copies the contents of `SSA[0]` to the stack, and redirects `SSA[0]`'s instruction pointer to point to the second-stage exception handler. The thread "jumps" to the second stage by `EEXIT`ing the first and then invoking `ERESUME`. The second stage may query custom exception handlers registered by the enclave application. The benefit of this design is that the number of nested exceptions that can be handled is limited by the size of the call stack, instead of the number of SSA frames. Hence, most enclaves allocate only two SSA frames per thread.

Side Channels. There are numerous microarchitectural side-channel attacks that either predate TEEs or exist independently from them. For example, cache set contention attacks such as PRIME+PROBE were described eight years before Intel SGX was made public [48], and more recently Aldaya et al. [4] described a simultaneous multithreading (SMT) port contention attack across any kind of execution environments (e.g., user to user, user to kernel, user to TEE, etc.).

It is worth emphasizing that these side-channel attacks do not violate Intel SGX's architectural security properties. Intel's SGX developer reference explicitly states that: "Intel® Software Guard Extensions is not designed to handle side channel attacks or reverse engineering. It is up to the Intel® SGX developers to build enclaves that are protected against these types of attack" [33]. This type of protection can be achieved by strict adherence to constant-time programming principles, which is challenging to implement at scale and has not seen broad adoption outside of the cryptographic community. Therefore, confidential computing workloads may be susceptible to side-channel attacks that can be amplified by malicious single-stepping.

2.2 The SGX-Step Framework

Overview. SGX-Step [63] is an open-source framework that allows a privileged software adversary to accurately single-step through a production Intel SGX enclave using APIC timer interrupts. Although debug enclaves can be trivially single-stepped using the x86 trap flag, production enclaves were never intended to be single-stepped by untrusted code.

SGX-Step provides two complementary interfaces to control the execution of a victim enclave: timer interrupt-driven and page fault-driven. The former can be used to precisely advance an enclave one instruction at a time, whereas the latter can be used in controlled-channel attacks [71] to break upon selected code or data page accesses at a coarser-grained 4 KiB granularity (which we consider out-of-scope in this work). Alternatively, the page-fault interface can also be abused to cause an adversary-chosen enclave instruction to repeatedly fault without making forward progress, a technique called "zero-stepping" [54, 58, 60].

Timer configuration. In contrast to earlier, coarser-grained enclave interruption via custom kernel patches [27, 41, 44], SGX-Step drastically reduces the amount of code between arming the timer and execution of the victim enclave. Timer interval prediction is, therefore, considerably simplified by configuring the memory-mapped APIC timer register directly from user space, right before re-entering the enclave with `ERESUME`. SGX-Step by default operates the APIC timer in one-shot mode, requiring the developer to specify a suitable timer interval to reliably land the interrupt in the first enclave instruction following `ERESUME`. This platform-specific configuration parameter can be determined via a calibration tool using an attacker-controlled debug enclave.

Several studies have reported highly accurate single-stepping results with SGX-Step (cf. Section 3). In the original evaluation [63] on benchmark enclaves with several hundreds of thousands of instructions, the vast majority of SGX-Step interrupts ($> 97\%$) were found to arrive within the *first* enclave instruction after `ERESUME`, i.e., forcing the enclave to "single-step" and make exactly one instruction progress. Crucially, no

"multi-step" events were observed and *all* the remaining interrupts were found to land within ERESUME itself, i.e., causing the enclave to "zero-step" and make no architectural progress. More recent SGX-Step extensions [45, 64], furthermore, observed that such zero-step events can be trivially detected and deterministically filtered out by checking the "accessed" bit in the enclave's code page table entry (PTE), which will only ever be set by the processor when the interrupt arrived after ERESUME and at least one enclave instruction has indeed been retired. As such, after configuring a conservative timer interval that precludes multi-steps, SGX-Step achieves noiseless single-stepping at a perfect, instruction-level granularity.

3 The Danger of Single-Stepping Attacks

This section presents an overview of published attacks that critically rely on SGX-Step's ability to forcibly "single-step" a production enclave exactly one instruction at a time, and, hence, motivate the mitigation described in this paper.

Interrupt Latency. During single-stepping, interrupts are delayed until instruction retirement, and the response time to service an interrupt, hence, depends on the instruction executed in the victim enclave. Interrupts themselves thus represent a subtle source of microarchitectural leakage, allowing single-stepping adversaries to effectively split overall, start-to-end enclave execution time into a more telling sequence of *individual* instruction timings.

This insight was first demonstrated in the Nemesis [64] attack, which used SGX-Step to collect an interrupt latency trace describing the execution time for each individual enclave instruction. Nemesis showed that such interrupt latency traces may reveal several fine-grained microarchitectural properties about the interrupted enclave instruction, including opcode, operand values, page-table walks, and cache misses. Interrupt latency has, furthermore, been used to infer store buffer occupancy (store buffers are flushed on AEX) [28]. More recently, the Frontal [49] attack extended Nemesis and leveraged SGX-Step to study subtle interrupt latency variations based on the alignment of enclaved store instructions.

Both Frontal and Nemesis critically rely on deterministic single-stepping to precisely sample individual enclave instructions and correlate measurements from different executions. In other words, their accuracy and scope would have been severely hampered without SGX-Step's guarantees to step exactly one instruction at a time.

Interrupt Counting. Crucially, in sharp contrast to notoriously noisy timing channels, the ability to perfectly interrupt and, hence, count the number of instructions executed in a victim enclave allows deterministic exploitation of the slightest control flow deviations from only a single run of the victim enclave (e.g., a tight loop or one-time key generation).

Van Bulck et al. [62] built a noiseless null-byte oracle that uses SGX-Step to precisely count `strlen()` iterations in slightly non-constant-time string pointer validation logic of the Intel SGX SDK, enabling, amongst others, full AES-NI key recovery. CopyCat [45] explicitly recognized interrupt counting as a capable attack primitive that adds a deterministic temporal dimension to overcome the relatively coarse-grained 4 KiB page-level spatial granularity of prior controlled-channel attacks [71]. CopyCat employed SGX-Step to deterministically extract complete keys from several popular cryptographic libraries, as well as to defeat a state-of-the-art compiler hardening technique [29] that was explicitly aimed at withstanding SGX-Step. Aldaya et al. [2,3] similarly exploited SGX-Step's instruction-granular page-access traces to recover full keys from vetted cryptographic libraries. Kim et al. [36] identified enclave software versions by likewise counting the number of instructions between page accesses.

As with the interrupt latency measurements above, the perfect precision of deterministic single-stepping is essential for these interrupt-counting attacks to be practical; they reconstruct extremely subtle intra-page and intra-cacheline control flows that may, ultimately, deviate in only a single instruction and would not otherwise be deterministically detectable. Moreover, in practical attacks, several such intricate branches often have to be perfectly reconstructed in close succession in tight loops in a single run of the victim enclave.

High-Resolution Probing. The most general application of a single-stepping framework like SGX-Step is to amplify known side-channel attacks by collecting arbitrary side-channel samples at a *maximal* temporal resolution, i.e., ultimately after every individual enclave instruction.

Such interrupt-driven attacks have been repeatedly applied to the CPU cache side channel [10, 20, 27, 30, 44, 57], allowing to accurately probe secret-dependent data accesses in tight loops or to defeat software prefetching mitigations. Likewise, precise SGX-Step interrupt capabilities have been used to reveal high-resolution, intra-page secret-dependent control flow via the branch predictor [32, 41] or to probe instruction-granular x86 segmentation [26], alignment [62] or floating-point [6] exceptions. Finally, in the context of transient-execution [51, 54, 60, 61] or interface [7, 19] attacks, SGX-Step has also been leveraged to precisely advance an enclave until a chosen gadget of interest has been reached.

Zero-Stepping. Apart from advancing enclaved execution one instruction at a time, timer interrupts or page faults may also be abused to forcibly stall or "zero-step" a victim enclave without making architectural forward progress.

Zero-stepping was first introduced as an unlimited prefetch mechanism to reload sensitive CPU registers from SSA memory into microarchitectural buffers that can subsequently be leaked via transient execution [54, 60]. MicroScope [58] recognized that, while zero-stepping does not advance the instruction pointer architecturally, the victim enclave may still transiently execute a small number of operations following the faulting trigger instruction. These transient instructions are never architecturally committed and can, hence, be infinitely

replayed through zero-stepping, essentially allowing to collect arbitrarily many (noisy) microarchitectural resource utilization samples during only a single architectural run of the victim enclave. Such zero-stepping has been abused to considerably amplify side-channel leakage from secret-dependent data accesses in AES or port contention of arithmetic operations [58], as well as power consumption measurements [43].

Summary. Interrupts themselves leak fine-grained side-channel information through either interrupt latency, interrupt counting, or transient replaying. Exploiting this type of leakage critically depends on victim enclaves not being interrupt-aware and being forcibly executed exactly zero or one instructions at a time. Moreover, interrupts can also act as a capable attack primitive to maximize the temporal resolution of existing side channels. This allows the adversary to reliably target code patterns that may otherwise be infeasible to exploit in practice, or to defeat defenses that rely on partial atomic behavior of the instruction stream [23, 41, 44].

Deterministic single-stepping serves a fundamental building block of such interrupt-driven attacks, which have repeatedly proven both effective and difficult to mitigate in software.

4 SGX-Step Root Cause Analysis

A suitable solution to address fine-grained enclave execution control requires a precise understanding of how exactly SGX-Step succeeds in reliably interrupting the first enclave instruction following `ERESUME`, which we refer to as the *enclave application resumption point (EARP)*. While this may appear straightforward at first, configuring a timer to reliably fire an interrupt directly after the notoriously complex `ERESUME` is non-trivial, as the latter is characterized by a lengthy and non-deterministic execution time [67]. On the contrary, EARP may be *any* valid x86 instruction, including extremely lightweight instructions without memory operands that can be efficiently pipelined and only consume a single micro-op and less than one CPU cycle to execute [1].

Hence, the SGX-Step adversary is tasked with a seemingly impossible challenge: configure a (coarse-grained) timer to reliably interrupt the possibly very short EARP directly following `ERESUME`, whose execution time itself varies greatly and can take thousands of CPU cycles. We, thus, argue that the microarchitectural root cause for the apparent success of SGX-Step has so far not been sufficiently understood, which is an essential prerequisite for any effective mitigation.

Assisted Page-Table Walk. We found that the key to SGX-Step's success lies in its use of the "accessed" (A) bit. Specifically, SGX-Step always clears the A-bit in the victim enclave's page-middle directory (PMD) before arming the APIC to fire a one-shot interrupt. As explained in Section 2.2, this bit is only ever set by the processor when at least one instruction is executed by the enclave and can, hence, be used to deterministically distinguish between zero-step events where

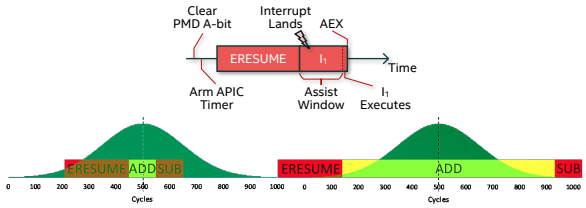

Figure 2: Adversaries can artificially prolong a short enclave instruction I_1 (e.g., `ADD`) by clearing the "accessed" (A) bit on the enclave code page. This forces the first enclave instruction following `ERESUME` to perform an expensive microcode-assisted page-table walk to set the A-bit, making it much more likely that the coarse-grained APIC timer interrupt lands during I_1 execution (i.e., single-step; detectable when $A = 1$).

the interrupt arrived too early during `ERESUME`, versus single-step events where the interrupt arrived within the EARP.

Crucially, this is not the only relevant effect of the EARP setting the PMD A-bit. The processor's page-miss handler is optimized for the common fast path and uses a much slower *microcode assist* to handle the less frequent and more complex case where a PMD or PTE needs to be modified [17, 54]. Figure 2 illustrates that this assist has the effect of prolonging the execution of the EARP instruction by opening an "assist window" of several hundreds of cycles, thus providing a convenient landing space for the coarse-grained, normally distributed APIC timer interrupt to arrive with high accuracy.

It is important to note that this microcode assist is unpreventable for existing enclaves. Recall from Section 2.1 that Intel SGX must flush enclave TLB entries on entry and exit, which is necessary to preserve Intel SGX's confidentiality and integrity. Hence, the processor must go through the page-miss handler to translate the EARP's address, and if the page-miss handler encounters an unset A-bit during that translation, then it must use a microcode assist to set the bit.

Experimental Results. We experimentally confirmed the additional latency of an assisted page table walk, which we found to be normally distributed with $\mu = 666; \sigma = 55$ cycles in case an A-bit is reset vs. $\mu = 27; \sigma = 30$ for a normal page-table walk without assist on on an Intel Ice Lake processor.

The inverse normal distribution with $\mu = 10,957$ and $\sigma = 73$ (i.e., the interrupt arrival timing distribution for the APIC one-shot mode measured in Appendix A) predicts 99.94 % single-stepping accuracy for a 500-cycle interval centered at the mean, which resembles our experimental observations and the SGX-Step authors' results [63].

5 Mitigation Objectives

We propose a hardware-software co-design that we call *AEX-Notify*. The hardware component (Section 6) is an ISA extension to Intel SGX that allows trusted enclave software to

react to interrupts. The software component (Section 7) is a carefully crafted trusted interrupt handler that determines and prefetches the interrupted application's working set, such that the EARP executes quickly and does not fault, thus obviating the prerequisites for single- and zero-stepping attacks.

Security Notion. Our explicit security goal is to *thwart the ability for privileged adversaries to deterministically single-step the instructions executed in a victim enclave*. With our defense in place, the adversary capabilities are considerably reduced to either coarser-grained 4 KiB page-fault attacks, or non-deterministic noisy microarchitectural attacks. Hence, our defense offers increased protection to a class of enclave programs that nowadays can be deterministically exploited via perfect, instruction-granular single-stepping, including through interrupt-latency measurements and interrupt counting, and may additionally hamper other types of interrupt-driven attacks discussed in Section 3.

Adversary Model. We adopt Intel SGX's standard privileged adversary model. That is, the adversary may arbitrarily control system settings, repeat enclave execution, run code concurrently on any other logical processor(s), including the victim enclave thread's sibling SMT thread, etc. Of particular importance is that the adversary may use the APIC to deliver interrupts at an interval chosen prior to entering an enclave, or issue an APIC IPI from another logical processor. We assume that the OS is malicious and can observe and manipulate the enclave's page tables within the constraints imposed by Intel SGX's architecture. This assumption has several crucial implications for our mitigation goals, as the adversary can deterministically *(i)* observe all enclave memory accesses at a 4 KiB spatial granularity; *(ii)* distinguish read/write/execute accesses on the same 4 KiB enclave page (through respectively the A, D, and XD PTE attributes); and *(iii)* observe the time of the first access to a page at instruction-granular temporal resolution (e.g., via a thread monitoring the enclave's PTEs from another logical processor [65]).

Enclave Assumptions. In addition, we make the following reasonable assumptions about the enclave program:

- It follows the standard System-V ABI for x86-64, e.g., it respects the 128-byte red zone, and will not allocate data 128 or more bytes beyond `RSP`. Furthermore, after enabling AEX-Notify, `RSP` should point to a secure, in-enclave stack [19], and we assume the higher-order page-address bits of `RSP` are known to the adversary (e.g., through observing page faults [71] on the enclave stack).
- The code is free of memory-safety bugs [40] and bugs that may cause the enclave to execute illegal instructions. Furthermore, the enclave is free of concurrency bugs [68], i.e., we assume that accesses to data shared among threads are properly synchronized.
- Enclave code and data pages are properly separated and the position of `RET` (`0XC3`) bytes within the code pages of the binary layout is known to the attacker.

Mitigation Objectives. We set the following explicit goals:

G1 *Obfuscated forward progress.* The mitigation must prevent the adversary from reliably advancing the enclave application by a single instruction (single-step) or repeatedly faulting on that instruction (zero-step); it must also prevent the adversary from detecting whether or how much instruction-granular, intra-page forward progress has been made after each enclave re-entry.

G2 *Bounded leakage.* The information leaked by the mitigation must be no greater than the information leaked by the enclave application without the mitigation.

G3 *Software compatibility.* The mitigation may not change the enclave application's computational semantics.

G4 *Practicality.* The enclave must: *(a)* incur little runtime overhead, especially in benign execution environments where interrupts are relatively infrequent; *(b)* be deployable to legacy hardware; *(c)* be compatible with ABI–compliant enclave binaries, without requiring custom recompilation; and *(d)* not interfere with host software.

In-Scope Attacks. In terms of the interrupt-driven attacks presented in Section 3, our explicit security objective is only to thwart the precise class of deterministic, perfect single-stepping attacks. This includes the versatile and fine-grained interrupt latency [28, 49, 64] and counting [2, 3, 36, 45, 62] primitives. However, our proposed mitigation does not aim to completely rule out attacks for which the underlying side channels are exploited via more generic high-resolution probing [41, 44, 60], and thus themselves do not critically rely on single-stepping interrupts. Our mitigation will certainly limit the temporal resolution of such attacks, however, and, hence, may raise the bar for exploiting them in practice, although we do not claim any specific quantifiable improvement.

With regard to zero-stepping attacks [43, 58], the particular software prefetching mitigation presented in this work only aims to prevent arbitrarily repeating a target enclave application instruction, thus blocking replay of important signals such as execution port or power usage, but does not explicitly rule out replaying the side effects of memory accesses performed by the mitigation itself.

Finally note that our proposed mitigation does not protect against interrupting enclaves and observing application code and data page accesses at a coarse-grained 4 KiB spatial resolution. In contrast to the fine-grained, instruction-granular interrupt-driven attacks we consider in this work, such controlled-channel attacks have received ample attention [18, 47, 56, 59] from the research community.

6 The AEX-Notify ISA Extension

In this section, we describe minimal ISA changes to make enclaves interrupt aware. Our proposed ISA primitive has been adopted by Intel and incorporated as an extension to the Intel SGX specification [12]. It is an attestable enclave

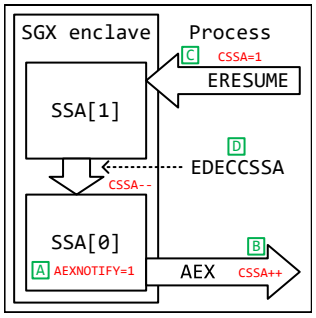

Figure 3: AEX-Notify flow to A opt-in to AEX notifications, B encounter an AEX, C resume the enclave and deliver the notification, and D invoke the EDECCSSA instruction to return to the previous SSA frame without exiting the enclave.

feature, which implies that the (possibly remote) verifier can check that the enclave has been built on a system that supports AEX-Notify, and that AEX-Notify has been enabled.

AEX-Notify only requires changes to one existing Intel SGX instruction, ERESUME, and the addition of one new instruction, EDECCSSA. Moreover, these changes can be applied via a microcode update patch [11]. Hence, the AEX-Notify design can be backported to many existing processors that already support Intel SGX, thus achieving objective *G4(b)*.

Operation Overview. Our proposed AEX-Notify ISA does not require any changes to untrusted software that enters the enclave either synchronously with EENTER or asynchronously with ERESUME (cf. *G4(d)*). Only the trusted runtime within the enclave must be changed to react to AEX notifications, as depicted in Figure 3 and described in detail below:

1. The enclave thread sets the AEXNOTIFY bit to 1 in one of its SSA frames, e.g., in SSA[0]. Note that, apart from a global enable bit per TCS, in our design each SSA frame has an *individual* enable bit, as this allows a handler, e.g., on SSA[1], to resume as normal without AEX-Notify and, hence, make progress in the presence of interrupts (cf. software handler design of Section 7).
2. The enclave thread encounters an AEX, which saves the thread's processor context to the current SSA frame and then increments TCS.CSSA. Note that the AEX-Notify ISA extension does not modify any aspect of AEX.
3. A thread ERESUMEs into the enclave. The AEX-Notify ISA extension causes ERESUME to adopt the semantics of EENTER when the AEXNOTIFY bit is set to 1 in SSA[TCS.CSSA-1]. Hence, the ERESUME instruction does not restore the previous processor context and does not decrement TCS.CSSA. Instead, the enclave thread resumes at the fixed entry point preconfigured in the TCS, allowing the enclave to handle AEXs in a custom exception handler when detecting that TCS.CSSA=1 on entry.
4. The enclave thread invokes a new instruction introduced by AEX-Notify called EDECCSSA, which decrements TCS.CSSA to allow the enclave to resume execution in the prior context without re-exiting the enclave, unlike the legacy flow depicted in Figure 1 (right), which uses EEXIT followed by ERESUME to resume execution. Note that, unlike the legacy ERESUME instruction, EDECCSSA does not restore SSA contents, and enclave software is responsible to first preserve all required state from SSA[TCS.CSSA-1].

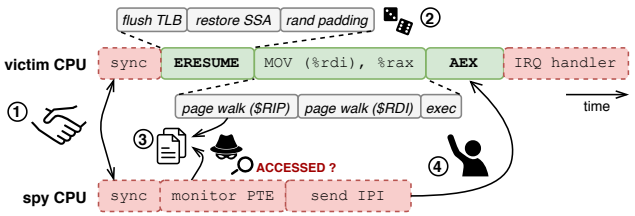

Figure 4: Adversaries can precisely single-step victim enclaves without resorting to timers via a concurrent spy thread that monitors unprotected page-table accesses.

6.1 Considered Design Alternatives

Add a Random Delay to ERESUME. At first sight, it may be tempting to think of a simpler, hardware-only solution that naively attempts to thwart single-stepping by randomizing the execution time of ERESUME. Crucially, however, we do *not* consider mere randomization in itself to considerably raise the bar for SGX-Step adversaries.

First, adversaries may conservatively underestimate the random padding delay and rely on page-table A-bits to filter out the abounding zero-step observations, resulting in a substantially slower, yet accurate single-stepping primitive. Randomization, therefore, may reduce the accuracy of SGX-Step per interrupt, but it does not achieve *G1*. For example, Figure 5 (right) illustrates how adversaries may arm the APIC timer and choose a number n of NOP instructions before resuming the enclave, such that the distribution Y for APIC interrupt arrival has an arbitrarily small overlap m with the the randomized latency distribution X of a modified ERESUME instruction (i.e., most interrupts will zero-step and only very few single-step interrupts arrive just after ERESUME completion).

Second, Figure 4 illustrates that attackers can always resort to a dedicated spy thread that shoots down the victim CPU with near-perfect, instruction-granular IPIs [65] when detecting the page-table walk in untrusted memory *after* the randomized ERESUME instruction has retired and before completion of the enclave instruction.

Move Paging Structures into Enclave Memory. More radical TEE research prototypes [18, 22] have proposed to place enclave page tables completely out of reach of the attacker. However, such an approach clearly violates *G4(b)* by requiring far-going hardware changes. More fundamentally, in the absence of encompassing microarchitectural isolation such as

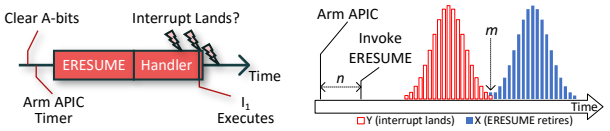

Figure 5: **(Left)** The AEX notification handler prefetches I_1's code and data (if it has a memory operand), thus dramatically reducing I_1's latency. The non-deterministic latencies of the APIC, `ERESUME`, and the handler make it infeasible for the adversary to precisely hit I_1 with an interrupt. **(Right)** A technique that can circumvent the random-delay approach.

full cache partitioning, we argue that merely isolating architectural page-table memory is insufficient to meet *G1*. That is, while this approach prevents a malicious OS from tampering with paging controls (e.g., A-bits), it does not prevent the OS from using cache contention to evict those paging control structures to memory, along with enclave code and data.

Disable A/D-bit Assists while in Enclave Mode. Several works [47,65] have suggested that Intel SGX processors could prevent page-table "accessed" and "dirty" bits from being updated while in enclave mode. However, operating systems require these bits to manage memory, so this approach violates *G4(d)*. Furthermore, this approach does not suffice to meet the crucial *G1* property, as it suffers from the same drawback as the previous approach: there are other demonstrated techniques, such a cache evictions of PTE entries [65], that SGX-Step can use to slow the first enclave instruction.

Postpone Interrupts. Another conceivable approach is to modify the CPU to postpone each interrupt until at least some minimum number of enclave instructions have executed. However, this solution may allow the adversary to detect forward progress, violating *G1*. Moreover, it does not prevent fault-based zero-stepping (also violating *G1*) and its implementation complexity prevents it from meeting *G4(b)*. Naively delaying interrupts in unprivileged user-space enclave programs may, furthermore, interfere with the OS's availability requirements and, thus, hinder *G4(d)*.

7 Enclaved AEX-Notify Software Mitigation

This section describes the software mitigation that uses AEX-Notify to achieve the objectives listed in Section 5. The key idea is to *dynamically*, i.e., after each interrupt, identify and prefetch the code and data memory locations that will be accessed by the enclave application resumption point (EARP). By prefetching these code and data pages in a carefully crafted assembly stub that is *atomically* executed before jumping to the EARP, the mitigation effectively ensures that any subsequent accesses to these locations by the enclave application will hit in the CPU caches. Hence, at least the first EARP instruction will execute fast and any code or data physical page addresses needed by EARP will be served from the processor's trusted TLB, without triggering a fault or assist. Figure 5 (left) highlights how our proposed mitigation effectively closes the root-cause interrupt landing zone exploited by SGX-Step adversaries (cf. top of Figure 2).

Implementation Aspects. We implemented our software mitigation prototype as an extension to the existing two-stage exception handler design (cf. Section 2.1) of the open-source Intel SGX SDK. Particularly, in our design the first-stage exception handler always runs with AEX-Notify disabled (i.e., `SSA[1].AEXNOTIFY=0`), while the second-stage handler is split into a larger, non-atomic part that is run with AEX-Notify disabled and a smaller assembly stub that always runs with AEX-Notify enabled. Note that limiting the use of AEX-Notify in this way to only a small, atomic part ensures that the vast majority of the enclave exception handler's execution can always make progress, as it can be interrupted and resumed as before (i.e., possibly one instruction at a time).

We modified the first stage exception handler to reserve a 120-byte persistent memory area under the red zone of the interrupted enclave application's call stack. This reserved area will be preserved across any later interrupts of the exception handler and can, hence, be used to store selected application registers that are used in our hand-crafted assembly stub, as explained in Section 7.3. We also modified the first stage to copy and clear `SSA[0].AEXNOTIFY`, before making use of the new `EDECCSSA` instruction (cf. Section 2.1) to efficiently switch to the second-stage handler without exiting the enclave. When the second stage C code detects that the originally interrupted enclave application thread was protected by AEX-Notify, it will proceed to decode the EARP instruction's memory operands (Section 7.1) and locate a `RET` instruction on the EARP code page (Section 7.2), before finally handing over to the assembly stub (Section 7.3). Here, we finally re-enable AEX-Notify and, in an atomic manner, prefetch the enclave's working set before jumping to the EARP.

It is worth noting our software mitigation is not in any way restricted or specific to the Intel SGX SDK. In fact, as we make no special assumptions on either the enclave application binary or runtime system, our mitigation may also be integrated into alternative Intel SGX enclave SDKs or library OSs. Such a porting effort would only need to consider the relatively small glue code to extend the exception-handler logic, whereas the crucial constant-time instruction decoding and assembly stub components are largely generic and may be reused across enclave runtimes.

7.1 Constant-Time Instruction Decoding

We first establish a sensible (over)approximation of the EARP instruction's working set, consisting of: *(i)* the current code page; *(ii)* the current stack page; and possibly *(iii)* the next data page that will be accessed (read/write). We, furthermore, extend this base working set with one additional stack page directly below the interrupted stack pointer, as the final stages

of the mitigation restore registers from the reserved area that may lie on this stack page, just before jumping to the EARP.

While code and stack pages can be determined trivially by examining the saved `SSA[0]` frame, any additional global data accesses need to be dynamically determined based on the semantics of the first EARP assembly instruction. Crucially, this decoding process should be performed fast to minimize performance overheads (*G4(a)*), while also not introducing new side-channel leakages that cannot be learned from executing the enclave application without our mitigation (*G2*). Existing commodity disassemblers, such as Capstone and Intel Xed, cannot meet the mitigation objectives because they are not designed for side-channel resilience. They may, for instance, be vulnerable to a cache-based timing channel as they typically rely on one or more memory accesses to an instruction lookup table that maps an opcode to a semantic decoding of each instruction. Such a side channel would leak fine-grained, individual instruction details, which AEX-Notify aims to hide in the first place (*G1*/*G2*). Note that this constant-time requirement for the mitigation code is especially stringent considering that, at this stage, the exception handler is itself not protected by AEX-Notify and may, hence, be single-stepped with a framework like SGX-Step to amplify any subtle side-channel leakages.

To overcome this challenge, we implemented a constant-time decoder (CTD) that extracts the potential memory locations of the next instruction while adhering to constant-time programming guidelines [14]. We implemented our design in around 1.4 K lines of C code, using the `CMOV` family instructions for conditional operations and `AVX2` instructions to accelerate the table lookup process. To balance implementation complexity, performance *G4(a)*, and correctness concerns, we set as an explicit design goal that the CTD should *never* report false-positive memory accesses that are not performed by the application, whereas we allow occasional false negatives for unsupported instructions. We evaluate instruction coverage in Section 8.3, concluding that CTD supports the vast majority of instructions (over 98 %) in real-world enclave binaries. For any remaining, unsupported instructions, CTD will always report that they do not access any memory, thus safeguarding program correctness (*G3*). Even for these unsupported instructions, however, our mitigation will still prefetch the default working set, consisting of the current code and stack pages. Hence, attackers may interrupt or fault on unsupported instructions *only* if they feature non-stack data operands.

The CTD first examines the copied SSA frame of the previously interrupted enclave thread and loads 16 bytes, i.e., the size of the longest x64 instruction, aligned to a power of 2, from the code page at the interrupted enclave application's instruction pointer. Furthermore, as an interesting edge case, our CTD implementation makes sure to never fetch bytes across a page boundary. We use a constant-time instruction sequence that fetches at most 16 bytes from the enclave application's instruction pointer to the end of the code page, padding with up to 15 dummy bytes as needed. This subtlety is necessary to ensure that the CTD never leaks any of the lower 12 page-index bits of the enclave application's instruction pointer, thus preserving the bounded leakage requirement *G2*. Particularly, if the CTD would always unconditionally load 16 bytes across page boundaries, adversaries may trivially distinguish intra-page conditional control flow by observing a page fault on the neighboring code page when one of the paths executes an instruction that falls entirely within the last 15 bytes of the current enclave code page.

Next, based on the loaded buffer, the CTD decodes the memory operand of the instruction and decides whether it is going to access memory, the type of the access (read or write), and the corresponding target address. If the instruction buffer was previously padded with dummy bytes and its decoded length exceeds the current code page boundary, the CTD will simply report that the instruction does not access any memory. The key idea of our CTD is to rapidly decode the necessary memory operands without introducing any opcode-based memory accesses. To calculate the exact address of the target memory operands, the CTD first constructs an algebraic representation of the memory operand, and then it uses a `CMOV`-based constant-time access sequence to evaluate it over the SSA frame register contents, e.g., accessing `RAX` to compute the effective address of a memory operand like `[RAX+0X38]`.

7.2 Verifying Page-Table Permissions

It is essential for the mitigation to be effective that the prefetching performed by the handler is adequate, such that the first EARP instruction will not anymore fault or perform a page-table walk. As outlined in Section 5, privileged adversaries can exercise their control over x86 page-table attributes to distinguish read, write, or execute accesses to the same 4 KiB enclave page. Hence, a naive mitigation stub that would merely *read* the EARP's working set may still violate *G1* when clearing the D-bit in the associated enclave PTE and inducing another assisted page-table walk for EARP write accesses, or by inducing zero-step exceptions when mapping non-executable or read-only pages.

Execute Accesses. To prevent enclaves from being zero-stepped by an execution permission fault, our AEX notification handler finds a `RET` (i.e., `0xC3`) byte within the EARP code page. The address of this `C3` byte is subsequently passed to the assembly stub (Section 7.3), which will atomically `CALL` this `RET` to verify that the code page is indeed executable, i.e., that its execute disable (XD) bit has not been set.

To speed up this process, our implementation uses a thread-local *C3-cache*, a software table that maps a partial address to an offset within the corresponding page that may contain a `C3` byte. A "cache hit" can be confirmed by checking whether the offset indeed refers to a `C3` byte within the code page. In case of a "cache miss", the code page must be scanned sequentially to find a `C3` byte. In the rare event that a `C3` byte

is not found, then the code page cannot be verified in this manner. Note that, if C3 bytes are distributed uniformly at a frequency of 1/256, the probability of not finding a C3 byte on a given 4 KiB page is less than one in a million.

Importantly, our C3-cache does not introduce any *new* side-channel leakage that would violate *G2*, as the trace of executed pages and the position of C3 bytes in code pages are assumed known to the adversary (cf. Section 5)

Write Accesses. Any writes about to be performed by the first EARP instruction need to be similarly anticipated in the mitigation. Not only to prevent zero-stepping through read-only page faults, but also to rule out single-stepping through an assisted page-table walk that sets the PTE "dirty" (D) bit. That is, upon the first read access to a page, the processor populates a TLB entry and sets the A-bit in the associated paging-structure entries. Any subsequent read accesses to that page will now be served from the TLB and will not anymore update the A-bit. However, we experimentally confirmed that the *first* write to the same page will initiate another expensive assisted page-table walk that sets the corresponding D-bit once (as tracked in the corresponding TLB entry).

When the CTD reports that the EARP writes to memory, our atomic mitigation stub takes care to first read one byte from the reported address and then write the same byte back to the same location. Note that we restrict the mitigation to only write one byte to avoid further complicating the CTD with decoding operand lengths. Furthermore, to ensure functional correctness (*G3*) in case of multithreaded enclave applications, CTD never reports write accesses for rare x86 instructions with a LOCK prefix or implicit locking behavior (e.g., XCHG). As per our explicit assumption in Section 5, we assume accesses to data shared among threads are properly synchronized (e.g., any required software locks are held).

7.3 Atomic Prefetching

The final and most critical part of our software mitigation executes entirely as carefully hand-crafted assembly stub. The key challenge addressed by this stub is how to securely prefetch the working set and jump to the EARP in an *atomic* fashion, i.e., without being interrupted in between. This is a crucial requirement to achieve *G1*, as any AEX in between will flush the TLB and again substantially slow down the first EARP instruction (cf. Section 4). The assembly stub totals about 180 lines of code and is broken down into two successive phases. The first mitigation phase still runs without AEX-Notify enabled and can, hence, be freely interrupted with SGX-Step and the normal ERESUME flow. However, the second "atomic" mitigation phase runs with AEX-Notify re-enabled, and any interrupt arriving in this second part will, hence, re-trigger the exception handler flow from the start.

Setup/Rollback Phase. The stub first needs to check whether the last AEX occurred during the enclave application or during the mitigation's atomic prefetching phase. In case of the latter, we explicitly *roll back* the interrupted instruction pointer and parameter registers, so as to restart the subsequent atomic prefetching phase from the beginning. Note that all this needs to be programmed with careful, constant-time constructs, however, since the adversary should not be able to distinguish whether the interrupt landed during or after the mitigation (*G1*), as further discussed in Section 8.1.

Our implementation first compares the saved SSA[0].RIP value against the known, contiguous range of the atomic mitigation assembly code (taking care to also think about the edge case where the interrupt may have landed just after the RET byte on the EARP's code page was called). Next, we use a CMOV instruction sequence to save or restore selected parameter registers from or to the static reserved area under the interrupted enclave application's stack. Hence, after interrupting the atomic mitigation phase, the CPU register and reserved-area memory state will be identical to when the enclave application was originally interrupted.

Atomic Prefetching Phase. This compact, yet critical part of the assembly stub is integrally included in Appendix C for reference. The restartable atomic phase is initiated by first enabling AEX-Notify, ensuring that any future interrupts will be redirected to the start of the exception handler, which will take care of obliviously rolling back any state in case the interrupt arrived before executing the first EARP instruction.

Next, the stub prefetches the working set. In case the CTD reported a write instruction, our implementation first reads and writes back one byte at the provided location. We then enter a tight loop that calls the C3 byte and loads from the beginning of each of the 64 cache lines in each 4 KiB working-set page. Such repetitions in a tight loop may additionally hinder any adversary attempts at concurrently evicting working-set pages. The stub may now restore the small set of registers that were used in the previous steps to their final EARP values that were saved onto SSA[0] during the original AEX.

Finally, before jumping to the EARP instruction at which the original AEX occurred, we insert an additional delay with 50 % probability. This random small delay is not essential for our defense, but we expect that it may make it even harder for a possibly more advanced adversary to reliably time the interrupt directly after the jump to the EARP. The value of the delay is a configurable parameter for the mitigation, but should not be chosen too large, so as to minimize the window for evicting the working set. We set the delay to 20 cycles, as most application instructions will likely have completed within this time given a fully prefetched working set.

8 Evaluation

We first present an overall security argument that our defense meets the mitigation objectives. Next, we empirically evaluate mitigation effectiveness through principled experiments in attacker-favored conditions, guided by our security analysis. We, then, methodologically evaluate constant-timeness,

correctness, and coverage for the crucial CTD component. Finally, we report measured performance overheads.

8.1 Security Analysis

Confidentiality (*G2*). We first explain why our mitigation should not leak secrets that would not otherwise already have been exposed by the application. Generally, we assume proper Intel SGX attestation with the latest recommended microcode [7, 51, 54, 60] or compiler [61] mitigations to rule out transient-execution attacks.

Regarding application memory contents and interrupted SSA register values, we carefully designed CTD to be constant-time (evaluated in §8.3) and made dedicated efforts to only dereference the interrupted instruction pointer at a page-level granularity (§7.1). Furthermore, any application memory accesses performed in the mitigation are legitimate and may also be performed by EARP, as per our explicit assumptions in §5. As a relevant edge case, we took care to not prefetch memory addresses that fall outside the enclave range, as such pointer dereferences are visible to the adversary anyway and may have to be surrounded by additional microarchitectural cleansing and serialization instructions to prevent leakage on certain recent Intel processors [7, 15].

Forward Progress (*G1*). To ensure that adversaries cannot learn whether the interrupt landed during or after the mitigation, we carefully programmed the setup/rollback phase with `CMOV` instructions (§7.3) and validated that it executes in constant time. We, furthermore, made sure that the short, yet critical atomic mitigation stub (§C) is aligned to never cross a page boundary. The only secret-dependent branch in the mitigation stub is to conditionally skip the random padding before jumping to EARP instruction. We explicitly opted to move the random cycle delay *after* the prefetching, as the first iteration of the prefetching loop will be visible to an adversary concurrently monitoring page-table memory [65], and, thus, its timing should not depend on whether the secret cycle delay has been added. While this random branch decision may hypothetically still be later reconstructed via the branch predictor [32, 41], sampling this information after the next interrupt would already be practically useless, as every invocation of the mitigation uses a fresh random bit to decide whether to delay the next EARP instruction. Note that randomness is not generated in the atomic stub itself, as the expensive `RDRAND` instruction may introduce observable delays [64], but instead uses a small randomness buffer populated in the C code of the second-stage exception handler.

Single-Stepping Resilience (*G1*). The key idea behind our mitigation is to minimize the latency of the next EARP instruction by prefetching its working set into the CPU caches and TLB. Hence, the required window (§4) for single-stepping interrupts to reliably arrive is drastically narrowed, and adversaries would have to either *(i)* configure the timer to hit within the restricted EARP time window; *(ii)* exploit any remaining instances of incomplete prefetching by our mitigation prototype; or *(iii)* find ways to undo or prevent the prefetching.

First, regarding accurate timer configuration, we exhaustively experimented (§8.2 and §A) with all possible APIC timer and IPI configurations and further improved over the state-of-the-art SGX-Step setup. Nevertheless, even in this improved setup and while quiescing the system by disabling C-States and SpeedStep technology, we conclude that privileged adversaries remain ultimately restricted by the limited resolution and jitter of the APIC clock. Furthermore, even with a hypothetical cycle-accurate timer, the adversary would have to precisely predict the execution time of the atomic mitigation stub, including the random padding delay, plus the `ERESUME` instruction, which is notoriously variable on modern x86 processors. Crucially, we carefully programmed the mitigation to hide forward progress, as otherwise adversaries may attempt to conservatively underestimate the timer interval and rely on detecting premature interrupt arrivals, similar to bypassing theoretical random `ERESUME` padding (§6.1).

Second, regarding incomplete prefetching, our CTD covers the vast majority of x64 instructions (98 % in real-world Intel SGX binaries; cf. Table 3). For scarce instructions that are not supported by CTD, only any non-stack, global data accesses would not be fully prefetched. Adversaries may, furthermore, still succeed in faulting or interrupting rare x64 instructions that feature multiple memory operands, or in the unlikely event that the instruction itself or its data operands cross page boundaries. We do not expect such uncommon instructions to occur in secret-dependent paths in practice, but, if needed, our CTD implementation could always be further extended.

Finally, to undo prefetching before execution of the next EARP instruction, adversaries may attempt to evict enclave TLB entries concurrently from a sibling logical core [24, 66]. Note that this is only possible when SMT technology is enabled at boot time, which, in response to recent hardware vulnerabilities [7, 54, 60], is also reflected in Intel SGX remote attestation. The most straightforward approach would be to simply reload `CR3` or use the privileged `INVLPG` instruction, which we found, however, to require several hundreds of CPU cycles [1]. Alternatively, adversaries may attempt to evict TLB entries by causing contention on the targeted TLB set. However, this technique can only evict data pages, as the L1 instruction TLB is statically partitioned across logical cores [24]. Furthermore, constructing an eviction set requires several attacker pages sustaining expensive TLB misses [66]. To further frustrate any concurrent TLB eviction attempts, we explicitly designed the mitigation to prefetch working-set pages in a tight loop, such that the TLB would be repopulated in case of any evictions in earlier iterations. In conclusion, we consider the small time window between repeated prefetching and EARP execution to be too narrow to allow reliable concurrent eviction of data pages from the TLB.

Alternatively, instead of targeting the TLB, adversaries may focus on fully disabling [41, 65, 66] or causing contention [27,

44] on the CPU cache. We evaluate both techniques in the next section and conclude that, while they may somewhat increase probabilities for the adversary, they remain largely ineffective at bypassing the mitigation and single-stepping ultimately remains unpredictable and non-deterministic.

8.2 Mitigation Effectiveness Experiments

Experiments were conducted on Intel Ice Lake and Coffee Lake E processor-based server machines supporting Intel SGX2 and AEX-Notify.

APIC Timer Accuracy. The challenge of single-stepping an enclave via privileged timer interrupts is aggravated by the limited resolution of the local APIC, which is not synchronized with the core clock and operates at a much lower frequency [16] (e.g., over 100× on the Intel Ice Lake CPU).

We present detailed microbenchmarks in Appendix A that show that APIC timer interrupt arrival times are normally distributed with a considerable variance of over $\sigma = 70$ CPU cycles for the default one-shot MMIO mode used by SGX-Step. Furthermore, as a contribution of independent interest, we propose an innovative privileged interrupt-gate technique that allows finer-grained APIC timer deadline configuration and reduces variance to about $\sigma = 40$ CPU cycles. Thus, we stress-test our defense in an improved experimental setup that considers adaptive adversaries and further pushes the boundaries of the existing, state-of-the-art SGX-Step setup.

Benchmark Setup. In our experiments, we first use SGX-Step to advance the enclave to the beginning of the atomic mitigation stub, where we force an AEX. Next, we record the timestamp counter, tsc_1, just before `ERESUME`-ing into the benchmark enclave, where we immediately record tsc_2 before the mitigation executes, and finally record tsc_3 at the EARP. Thus, $tsc_2 - tsc_1$ captures the latency of `ERESUME` and $tsc_3 - tsc_2$ captures the latency of the mitigation.

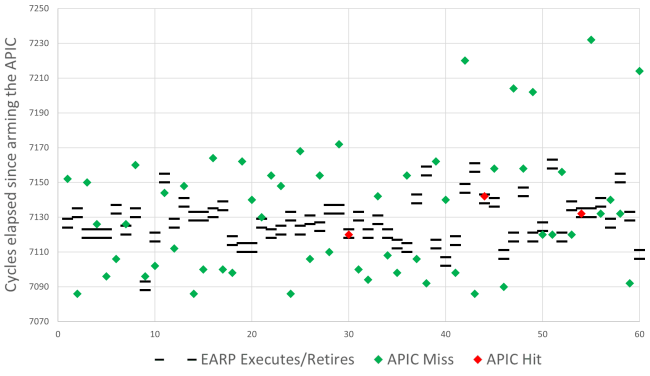

Figure 6: Example experiments resuming to a 5-cycle EARP load instruction with overlayed APIC interrupt arrivals.

As an illustration, Figure 6 depicts the start and end execution times of a 5-cycle EARP load instruction with our mitigation, measured for 60 runs on the Intel Ice Lake processor without attacker interference. We, furthermore, calibrated the APIC to hit after $tsc_3 - tsc_1$ cycles in a separate experiment, and we overlayed the observed actual APIC interrupt arrival times in the figure. Thus, it is visually apparent that successful single-stepping events where the APIC interrupt "hits" within the EARP instruction are infrequent.

Table 1: Single-stepping success rates for different stimuli.

Adversary Action(s)	Single-Step Hit Rate
None	0.042
Clear PTE A-bit	0.107
L1 contention (page/set)	0.118/0.112
L2 contention (page/different set/matching set)	0.114/0.126/0.223
L3 contention (same/separate/all cores)	0.141/0.030/0.104

Cache Contention. Table 1 summarizes the observed single-stepping success rates for 100,000 experiments on the Intel Coffee Lake E processor with several considered stimuli: clearing the PTE A-bit, and L1, L2, and L3 cache contention, affecting either an entire page or a single cache set within the mitigation's working set. We used the open-source Mastik [72] library to create cache contention. While some of the considered stimuli may somewhat increase probabilities for the adversary, e.g., the highest observed single-stepping rate increased from about 4 % without attacker interference to 22 % when applying targeted L2 contention, we conclude that none of the considered stimuli is effective at bypassing the mitigation objective and single-stepping clearly remains non-deterministic.

Cache Disable. We empirically demonstrate that, while enclaves may be drastically slowed down, disabling the cache via the privileged `CR0.CD` interface considerably increases variance, ultimately making single-stepping less predictable.

In our experimental setup on the Intel Ice Lake processor, we pin the victim to a specific core and set the `CR0.CD` bit on that core. With this setting, the latency of all memory loads is increased to over 2,000 cycles. We increased the APIC timer latency to match the execution delay and repeated the single-stepping attack without additional contention interference for 10,000 times (running for about 2 hours). The observed single-stepping success rate drops drastically to 0.08 %. Particularly, among all attempts, 31.98 % of the interrupts land in the mitigation stub and 67.78 % are multi-step interrupts hitting enclave instructions following the first EARP instruction.

The single-stepping attack fails primarily because disabling the CPU cache makes execution latencies highly volatile, due to the inherent timing instability of DRAM memory accesses. Specifically, we measured that the execution time variance to resume the victim with the mitigation significantly increased from $\mu = 8,052; \sigma = 31$ cycles when the cache is enabled to $\mu = 4,117,125; \sigma = 1,063,928$ cycles without the cache.

Combined Success Rates. Importantly, the success rates empirically determined in Table 1 are for stepping over a *single*

instruction, whereas practical attacks need several *consecutive* successful single-step operations, e.g., to count the number of iterations in a secret-dependent `strlen` loop [62, 63], or to distinguish slightly unbalanced branches [2, 3, 45]. Crucially, our defense was explicitly and carefully designed to provide practically no indication of forward progress (*G1*). Thus, when the attacker has no way of knowing how many instructions passed since the previous interrupt, she can only count the total number of interrupts. If one of these interrupts is not perfectly timed to single-step the next EARP instruction, the resulting interrupt count will not anymore correspond to the actual number of executed instructions in the targeted application. Thus, the probability of perfectly stepping through an enclaved instruction stream can be modeled as p^{steps}, where p is the success rate of successfully interrupting a single instruction and $steps$ the number of instructions that need to be successively interrupted for a successful attack.

To illustrate how our defense's strength may add up in practical scenarios, consider, for example, a successful single-step rate of 22 % for an individual instruction, which diminishes considerably when needing 10 successive single-step interrupts: $0.22^{10} = 0.000000266$. Thus, the amount of attack repetitions needed to successfully single-step 10 successive instructions follows the geometric distribution, with mean amount of tries until first success provided as $1/(0.22^{10}) = 3,759,398$. That is, in this example, over 3.7 million of attack iterations would be needed on average in order to observe one run that successfully single-steps all 10 instructions. Crucially, note that the adversary has no way of knowing which of these runs succeeded to single-step all required 10 instructions, and, hence, which of the millions of observations would actually contain the relevant side-channel data and which merely contain irrelevant noise. In contrast to conventional microarchitectural leakages, observations from multiple runs cannot be straightforwardly accumulated or averaged out, as the success rates of individual single-stepping attack iterations are independent of any previous attempts.

In conclusion, we accomplished our objective of thwarting the attacker advantage from deterministic single-stepping, and the adversary would be no better of than falling back to coarse-grained 4KiB page-level or noisy microarchitectural channels. The execution of the latter, furthermore, would typically assume that the victim enclave can be ran several times on the same secret, which is not always possible, e.g., for key generation [45, 49].

8.3 Constant Time Decoder Evaluation

Constant-Timeness. Although we followed best practices while designing and implementing the decoder, constant-time programming can be notoriously challenging and error-prone [34]. We, hence, evaluate the constant-time correctness of our decoder using both the static-analysis tool Pitchfork [21] and the dynamic-analysis tool Dudect [52].

Pitchfork is a static verification tool working on LLVM IR level to verify the constant-time property of control flow and memory access for the target function. Since Pitchfork only reasons about secret input from memory, we slightly change our function API to store all the register values into memory and mark them as secrets. Besides, Pitchfork does not support verification of SIMD instructions, so we manually add special handlers for the SIMD instructions we used in the decoder and verify they are explicitly listed as data operand-independent timing instructions [13]. After making these two changes, Pitchfork runs gracefully and verifies the constant-time correctness of our decoder in LLVM IR. Specifically, we launch Pitchfork with different optimization levels.[1] Pitchfork analyzes all the basic blocks, verifying that the decoder has only one branch and no secret-dependent accesses.

Dudect is a dynamic statistical testing tool to uncover constant-time violations between different execution times. Dudect continuously executes the target program with randomized inputs, until it statistically detects a timing violation between two different executions. We kept the Dudect tool running on the CTD for 24 hours with random 1-byte, 2-byte, and 3-byte opcodes as inputs. Dudect failed to find any violations after executing the CTD over 3.7 million times.

Decoding Correctness. To ensure the correctness of our decoding result, we developed a differential test framework to compare our constant time decoder and Intel Xed v2022.10.11. Because of the composition of x64 instructions, the prefix, opcode, modrm and sib bytes will affect its memory operand and access type, whereas other fields will only affect the exact memory address of the operand, but won't change the way to calculate the memory operand. Therefore, we systematically enumerated all the possible values for each field and feed the generated input to both CTD and Xed.

We validate decoding correctness under different conditions: *(i)* for illegal instruction encodings, we ignore our decoder's output since these instructions may cause undefined behaviors and they should never occur in well-behaved applications; *(ii)* for instructions that our decoder supports, we validate the correctness of the CTD result for both access types and memory operand calculations; *(iii)* for instructions that our decoder does not support, we ensure the CTD indeed outputs no memory access; *(iv)* for the select number of legal instruction encodings that are not supported by Intel SGX [16], we ignore our decoder's output as well. We refer to Appendix B for more details about CTD instruction coverage.

We launched this extensive differential testing approach and enumerated 161.43 million possibilities, passing all the testing criteria listed above. Importantly, this validates that the CTD should not report false-positive memory accesses, which may introduce serious memory-safety misbehavior.

Instruction Coverage on Applications. Instead of evaluating instruction coverage on Intel's opcode table, we argue

[1] We compiled CTD with LLVM 11 under optimization levels `-O1`, `-O2`, `-O3`, `-Oz`, and we hook the return value of the `PSHUFB` intrinsic as secret.

Table 2: CTD instruction coverage on popular SGX runtimes.

Intel SGX Runtime	Number of Binaries	Total Instructions (% - Total Coverage)	Covered Instructions w/o CTD(%)	Covered Instructions w/ CTD(%)
SGX SDK	18	1.37M (98.6%)	0.84M (61.2%)	0.51M (37.4%)
Gramine	53	2.03M (97.5%)	1.44M (71.1%)	0.54M (26.5%)
Occlum	35	1.35M (98.1%)	0.85M (63.0%)	0.47M (35.1%)
Total	106	4.75M (98.0%)	3.13M (65.9%)	1.52M (32.0%)

that the importance of each instruction is uneven in real-world applications. Some instructions like `MOV` are commonly used across all the programs and some SIMD or cryptographic instructions are rarely used only in certain libraries. Therefore, we collect representative real-world binaries from the Intel SGX SDK v2.18.1, and the Gramine v1.3.1 and Occlum 0.29.4 library OSs and statically decompile them to check if instructions are covered by our mitigation. The results are shown in Table 2, where we distinguish coverage without the CTD (i.e., instructions without data memory operands or that only access the stack) versus instructions with explicit data operands that cannot be prefetched in the mitigation without the information from the CTD. Summarized, our mitigation has 98.0 % instruction coverage, of which 32.0 % are protected by the CTD, on the 106 tested binaries.

8.4 Performance Overhead

We first measure instruction decoding time when running the CTD once (i.e., cold cache), then we run it for 1 million times and calculate the average time for each run (i.e., hot cache). We repeat each experiment six times and report the geometric mean. On the Intel Ice Lake processor, our decoder uses 729.15 cycles to finish one round with a cold cache, whereas for a hot cache, the time cost drops to 369.00 cycles.[2]

Overall performance overhead of the entire defense ultimately depends on the number of interrupts, which is already a particularly expensive operation in Intel SGX: we measured that our mitigation slows enclave resumption from 6,500 to 10,300 cycles (i.e., 58 %). Importantly, there is no further performance penalty after resumption. Hence, if a benign OS scheduler interrupts an enclave thread every 1 million cycles, overall performance overhead is below 0.4 %. Note that actual interrupt rates would highly depend on the deployment environment and the overall load of the system.

9 Mitigating Other Attacks

The AEX-Notify ISA extension we propose in this work, provides interrupt-awareness to Intel SGX enclaves, which can be an essential building block for implementing mitigations for a variety of side-channel attacks. Beyond the Intel SGX architecture, we envision that our design for interrupt-aware enclaves may also be useful for other TEEs that have been subject to similar attacks [8, 37, 42, 53, 70]. Interrupt awareness has been assumed [47, 59] or emulated [5, 9, 38, 55] in software in the majority of existing proposals for mitigating interrupt-driven attacks. Thus, we anticipate that AEX-Notify may be leveraged to enable a range of side-channel defenses.

Controlled Channels. The Heisenberg defense [59] proposes to prefetch enclave's pages when an enclave is resumed by forcing it to execute trusted code that does this prefetching right before resuming the enclave's execution. This code effectively populates the TLB, obfuscating the accesses to the respective pages by the enclave logic. AEX-Notify makes it easy to realize this defense in its handler. Compared to our defense, however, such an AEX-Notify extension would require developer or compiler assistance to identify the enclave's secret-dependent working set that needs to be prefetched.

Autarky [47] suggests to invoke a trusted page-fault handler in the enclave, which checks whether the faulting page was actually authorized to be evicted. AEX-Notify can be used to implement this mechanism in the handler. It may differentiate between the AEX due to page faults and the AEX due to other interrupts by inspecting the base address of the faulting page in the SSA structure [16]. It is worth noting that this implementation may work in bare-metal execution, but not in a virtual machine as the address of the faulting page is hidden in the latter case. Furthermore, as page tables still reside in untrusted memory, this defense would not protect against stealthy attacks that do not rely on page faults [65].

T-SGX [55] strives to prohibit enclave page faults by executing enclave code in TSX transactions. This approach is highly prone to false positives. Instead, AEX-Notify can reliably identify the page fault by its base address in SSA.

L1/L2 Cache Side Channels Varys [46] defends against cache-timing (L1/L2) and side-channel attacks. It detects AEXs by watching for changes in the SSA structure, and then flushes the caches. However, this method of detecting AEX is costly due to polling, and it might miss the interrupt event. Our AEX-Notify handler could be adopted to implement this mitigation, or replace it with populating the cache with the sensitive working set similarly to the mitigation for single-stepping attacks discussed in this paper.

10 Conclusion

The continued success of confidential computing will benefit from solutions to problems that are *unique* to TEEs. The work presented in this paper is a novel mitigation for one such problem: fine-grained execution control by a malicious OS or VMM. We designed the mitigation to be practical enough to be enabled by default with negligible performance overhead. Finally, we hope that our analysis of SGX-Step will help to influence the design and implementation of new TEEs.

[2] For reference, general-purpose disassemblers like Capstone or Intel Xed require around 1 million cycles to disassemble one instruction.

Acknowledgments

We thank our shepherd and the anonymous reviewers for their constructive comments and suggestions that helped improve the paper. This research is partially funded by the Research Fund KU Leuven and the Flemish Research Programme Cybersecurity. Jo Van Bulck is supported by a grant of the Research Foundation – Flanders (FWO).

References

[1] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In *ASPLOS*, 2019.

[2] Alejandro Cabrera Aldaya and Billy Bob Brumley. When one vulnerable primitive turns viral: Novel single-trace attacks on ECDSA and RSA. *CHES*, 2020.

[3] Alejandro Cabrera Aldaya and Billy Bob Brumley. Online template attacks: Revisited. *CHES*, 2021.

[4] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *S&P*, 2019.

[5] Fritz Alder, N Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. S-faas: Trustworthy and accountable function-as-a-service using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop*, 2019.

[6] Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. Faulty point unit: ABI poisoning attacks on Intel SGX. In *ACSAC*, 2020.

[7] Pietro Borrello, Andreas Kogler, Martin Schwarzl, Moritz Lipp, Daniel Gruss, and Michael Schwarz. ÆPIC Leak: Architecturally leaking uninitialized data from the microarchitecture. In *USENIX Security*, 2022.

[8] Matteo Busi, Job Noorman, Jo Van Bulck, Letterio Galletta, Pierpaolo Degano, Jan Tobias Mühlberg, and Frank Piessens. Provably secure isolation for interruptible enclaved execution on small microprocessors. In *CSF*, 2020.

[9] Sanchuan Chen, Xiaokuan Zhang, Michael K Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *AsiaCCS*, 2017.

[10] Chitchanok Chuengsatiansup, Daniel Genkin, Yuval Yarom, and Zhiyuan Zhang. Side-channeling the kalyna key expansion. In *CT-RSA*, 2022.

[11] Intel Corporation. Microcode Update Guidance, 2020.

[12] Intel Corporation. Asynchronous Enclave Exit Notify and the EDECCSSA User Leaf Function. White Paper, 2022.

[13] Intel Corporation. Data operand independent timing instructions, 2022.

[14] Intel Corporation. Guidelines for mitigating timing side channels against cryptographic implementations, 2022.

[15] Intel Corporation. Processor MMIO stale data vulnerabilities, 2022.

[16] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined volumes*, 2023.

[17] Victor Costan and Srinivas Devadas. Intel SGX explained. *IACR ePrint Archive*, 2016.

[18] Victor Costan, Ilia A. Lebedev, and Srinivas Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, 2016.

[19] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. Smashex: Smashing SGX enclaves using exceptions. In *CCS*, 2021.

[20] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: efficiently recovering long-term secrets of SGX EPID via cache attacks. *CHES*, 2018.

[21] Craig Disselkoen, Sunjay Cauligi, Dean Tullsen, and Deian Stefan. Finding and eliminating timing side-channels in crypto code with pitchfork. In *TECHCON*, 2020.

[22] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-x: A flexible architecture for hardware-managed isolated execution. In *MICRO*, 2014.

[23] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-LAPD: Thwarting controlled side channel attacks via enclave verifiable page faults. In *RAID*, 2017.

[24] Ben Gras, Kaveh Razavi, Herbert Bos, Cristiano Giuffrida, et al. Translation leak-aside buffer: Defeating cache side-channel protections with tlb attacks. In *USENIX Security*, 2018.

[25] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+ flush: a fast and stealthy cache attack. In *DIMVA*, 2016.

[26] 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.

[27] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *USENIX ATC*, 2017.

[28] Wenjian He, Wei Zhang, Sanjeev Das, and Yang Liu. Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution. In *ICCD*, 2018.

[29] Shohreh Hosseinzadeh, Hans Liljestrand, Ville Leppänen, and Andrew Paverd. Mitigating branch-shadowing attacks on Intel SGX using control flow randomization. In *SysTEX*, 2018.

[30] Senyang Huang, Rui Qi Sim, Chitchanok Chuengsatiansup, Qian Guo, and Thomas Johansson. Cache-timing attack against HQC. *IACR ePrint Archive*, 2023.

[31] Wei Huang, Shengjie Xu, Yueqiang Cheng, and David Lie. Aion attacks: Manipulating software timers in trusted execution environment. In *DIMVA*, 2021.

[32] 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. *CHES*, 2019.

[33] Intel Corporation. *Intel(R) Software Guard Extensions Developer Guide*, 2.18 edition, 2022.

[34] Jan Jancar, Marcel Fourné, Daniel De Almeida Braga, Mohamed Sabt, Peter Schwabe, Gilles Barthe, Pierre-Alain Fouque, and Yasemin Acar. "they're not that hard to mitigate": What cryptographic library developers think about timing attacks. In *S&P*, 2022.

[35] Jianyu Jiang, Claudio Soriente, and Ghassan Karame. On the challenges of detecting side-channel attacks in SGX. In *RAID*, 2022.

[36] Deokjin Kim, Daehee Jang, Minjoon Park, Yunjong Jeong, Jonghwan Kim, Seokjin Choi, and Brent Byunghoon Kang. Sgx-lego: Fine-grained SGX controlled-channel attack and its countermeasure. *Computers & Security*, 2019.

[37] Zili Kou, Wenjian He, Sharad Sinha, and Wei Zhang. Load-step: A precise trustzone execution control framework for exploring new side-channel attacks like flush+ evict. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[38] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. Mole: Mitigation of side-channel attacks against SGX via dynamic data location escape. In *ACSAC*, 2022.

[39] David Lantz, Felipe Boeira, and Mikael Asplund. Towards self-monitoring enclaves: Side-channel detection using performance counters. In *NordSec*, 2023.

[40] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. Byunghoon Kang. Hacking in Darkness: Return-Oriented Programming Against Secure Enclaves. In *USENIX Security*, 2017.

[41] 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*, 2017.

[42] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. Cipherleaks: Breaking constant-time cryptography on amd sev via the ciphertext side channel. In *USENIX Security*, 2021.

[43] Moritz Lipp, Andreas Kogler, David Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. Platypus: Software-based power side-channel attacks on x86. In *S&P*, 2021.

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

[45] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copycat: Controlled instruction-level attacks on enclaves. In *USENIX Security*, 2020.

[46] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. Varys: Protecting SGX enclaves from practical side-channel attacks. In *USENIX ATC*, 2018.

[47] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *EuroSys*, 2020.

[48] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *CT-RSA*, 2006.

[49] Ivan Puddu, Moritz Schneider, Miro Haller, and Srdjan Capkun. Frontal attack: Leaking control-flow in SGX via the CPU frontend. In *USENIX Security*, 2021.

[50] Ivan Puddu, Moritz Schneider, Daniele Lain, Stefano Boschetto, and Srdjan Čapkun. On (the lack of) code confidentiality in trusted execution environments. *arXiv*, 2022.

[51] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. CrossTalk: Speculative data leaks across cores are real. In *S&P*.

[52] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In *DATE*, 2017.

[53] Keegan Ryan. Hardware-backed heist: Extracting ecdsa keys from qualcomm's trustzone. In *CCS*, 2019.

[54] 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.

[55] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *NDSS*, 2017.

[56] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing page faults from telling your secrets. In *AsiaCCS*, 2016.

[57] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util::lookup: Exploiting key decoding in cryptographic libraries. In *CCS*, 2021.

[58] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. Microscope: Enabling microarchitectural replay attacks. In *ISCA*, 2019.

[59] Raoul Strackx and Frank Piessens. The heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. arXiv, 2017.

[60] 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*, 2018.

[61] 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.

[62] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *CCS*, 2019.

[63] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A practical attack framework for precise enclave execution control. In *SysTEX*, 2017.

[64] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Nemesis: Studying microarchitectural timing leaks in rudimentary CPU interrupt logic. In *CCS*, 2018.

[65] 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*, 2017.

[66] 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 *CCS*, 2017.

[67] Nico Weichbrodt, Pierre-Louis Aublin, and Rüdiger Kapitza. sgx-perf: A performance analysis tool for Intel SGX enclaves. In *Middleware*, 2018.

[68] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. Asyncshock: Exploiting synchronisation bugs in Intel SGX enclaves. In *ESORICS*, 2016.

[69] Samuel Weiser, Raphael Spreitzer, and Lukas Bodner. Single trace attack against RSA key generation in Intel SGX SSL. In *AsiaCCS*, 2018.

[70] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt of them all: Inference attacks against secure virtual enclaves. In *AsiaCCS*, 2019.

[71] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *S&P*, 2015.

[72] Yuval Yarom. Mastik: A micro-architectural side-channel toolkit, 2016.

A APIC Precision Microbenchmarks

The local APIC [16] timer can be configured in one-shot or periodic mode to send an interrupt when an MMIO counter register reaches zero, or in TSC-deadline mode when the processor's timestamp counter exceeds a value specified in the dedicated `IA32_TSC_DEADLINE_MSR` model-specific register. Depending on the processor model, the APIC timer operates at the frequency of either the processor's bus clock or its core crystal clock. Neither time source is synchronized with the core clock, which typically operates at a much higher frequency (e.g., more than 100× higher).

All experiments below were performed on an Intel Ice Lake processor-based server platform, with a base frequency of 2.4 GHz and an APIC core crystal clock resolution of 1 tick per 320 core clock cycles (as reported by `CPUID`).

One-Shot Mode. SGX-Step by default operates the APIC timer in one-shot mode with division 2 [63]. SGX-Step requires its user to manually determine the timer duration

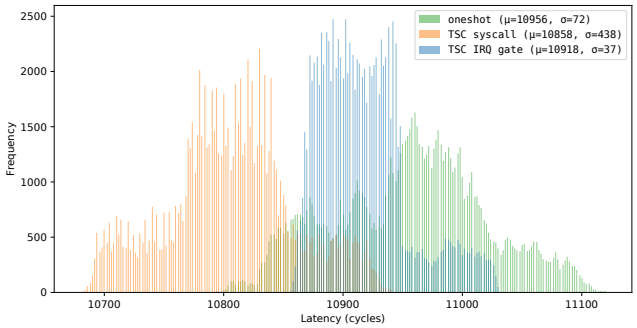

Figure 7: Interrupt arrival timing distributions in elapsed CPU cycles for 100,000 iterations when configuring the APIC timer in MMIO one-shot mode (green, right), TSC-deadline mode via the Linux `msr` driver (orange, left), and TSC-deadline mode via a custom interrupt gate (blue, middle).

(which will vary by platform) for a one-shot interrupt to reliably land on the first enclave instruction, i.e., the first instruction that executes after entering the enclave via `ERESUME`.

In our first microbenchmark experiment, we programmed the APIC timer in one-shot mode to fire an interrupt in 30 ticks (9,600 cycles) – roughly the latency of the `ERESUME` instruction. The benchmark program executes a lengthy instruction slide of register `ADD` instructions immediately following the instruction that arms the APIC timer. We used SGX-Step's page-table manipulation functionality to map the APIC's MMIO interface into user space, so as to avoid any overheads from kernel context switches. We, furthermore, read the counter register in the interrupt handler to derive the number of `ADD` instructions executed before interrupt arrival. Our experiment reads the processor's time-stamp counter using `RDTSC` before arming the APIC and again at the start of a custom handler registered in the processor's interrupt-descriptor table. Sampled over 100,000 repetitions, Figure 7 shows that the measured one-shot interrupt intervals were normally distributed with $\mu = 10,957; \sigma = 73$ cycles (or $\mu = 9,684; \sigma = 49$ `ADD` instructions).

TSC-Deadline Mode. Anticipating more advanced, adaptive adversaries, we hypothesized that operating the APIC timer in TSC-deadline mode may yield more accurate results, as `IA32_TSC_DEADLINE_MSR` can be programmed at the much finer-grained granularity of CPU core clock cycles. In this respect, the SGX-Step authors [63] explicitly mention that they trade timer interval predictability for a lower frequency by operating the APIC timer in one-shot mode via direct, user-space MMIO. That is, the possibly more precise TSC-deadline mode requires privileged `WRMSR` instructions that necessitate user-kernel context switches within the timer interval. Earlier, coarser-grained and more noisy Intel SGX interruption attacks [27, 41, 44] indeed configured the APIC timer in TSC-deadline mode from within the OS kernel, before making an expensive context switch back to user space and resuming the victim enclave.

We first designed an experiment to quantify the accuracy of such a naive approach by measuring the elapsed cycles between the start of our custom interrupt handler and an `RDTSC` snapshot before making a system call to the Linux `msr` driver to program `IA32_TSC_DEADLINE_MSR` with an offset of 9,600 cycles. Over 100,000 repetitions, Figure 7 shows that the measured TSC-deadline interrupts arrived following a remarkably wider distribution with $\mu = 10,859; \sigma = 438$ cycles (or $\mu = 3,070; \sigma = 568$ `ADD` instructions).

Finally, as a contribution of independent interest, we devised an improved setup to considerably reduce the noise from additional kernel context switches. For this, we registered a custom ring-0 interrupt gate in the processor's interrupt-descriptor table. This allows to essentially bypass the OS kernel altogether by directly invoking a minimal assembly handler that programs a specified value in `IA32_TSC_DEADLINE_MSR` using the privileged `WRMSR` instruction via a software interrupt (i.e., by including the `INT` x86 instruction before `ERESUME`). Using this innovative technique, we measured an improved, narrower TSC-deadline interrupt arrival distribution of $\mu = 10,918; \sigma = 38$ cycles (or $\mu = 8,734; \sigma = 67$ `ADD` instructions).

We conclude that our novel interrupt-gate TSC-deadline timer configuration technique yields a considerable improvement in terms of standard deviation of elapsed cycles for both the existing one-shot technique that SGX-Step currently uses, as well as naive kernel-level configuration approaches (cf. as is also visually evident from Figure 7). However, these results also clearly indicate that, even in TSC-deadline mode, privileged adversaries remain limited by the APIC internal clock, which is significantly lower than the core frequency and thus induces inevitable jitter on timer interrupt arrivals.

Inter-Processor Interrupts. In a final set of experiments, we wanted to assess the accuracy of inter-processor interrupts (IPIs), which are similarly sent and received via the local APIC bus and can be triggered by merely writing to memory-mapped APIC registers [16]. Specifically, prior work [65] has demonstrated a reliable technique to reliably interrupt a victim enclave at a near-perfect, instruction-level granularity by issuing IPIs from a kernel-level spy thread that concurrently monitors a victim enclave's page-table accesses using a combination of A/D PTE bits and a high-resolution Flush+Flush [25] side channel.

To assess this advanced type of cross-core adversaries, we extended SGX-Step with support for user-space IPIs via memory-mapped APIC registers. We, furthermore, designed an experiment to measure the elapsed cycles between triggering an IPI from a spy CPU and execution of our custom interrupt handler on the victim CPU. In our setup, we first synchronize spy and victim threads, before triggering the IPI in the spy thread and executing a lengthy `NOP` instruction slide in the victim thread. Over 100,000 samples, we found IPI latency to be distributed with $\mu = 935; \sigma = 27$ cy-

cles (or $\mu = 523; \sigma = 120$ `NOP` instructions) when the spy and victim threads reside on different physical cores. Alternatively, when the spy and victim reside on the same physical CPU with SMT technology, we measured an increased IPI latency of $\mu = 1,772; \sigma = 174$ cycles (or $\mu = 778; \sigma = 506$ `NOP` instructions).

In summary, these results show that cross-core privileged adversaries who may attempt to interrupt victim enclaves through IPIs remain similarly limited by jitter and delays from the limited APIC clock frequency.

B Coverage of the Constant-Time Decoder

Table 3 details the instruction coverage of our CTD prototype. We divide the table by the number of bytes for an instruction's opcode, which is at most 3 bytes for state-of-art x64 instructions. Besides the table, the CTD currently does not support *rip* relative addressing. Note that for instructions not clearly stated as supported, CTD does not support them.

Table 3: Our CTD is designed to support a subset of commonly used instructions for x64.

Opcode	Type	Supported Instructions	Unsupported Instructions
1 byte	Normal	All except right	x87 instructions `VEX/EVEX` instructions
	Extension	All	None
2 byte	Normal	All	None

	Extension	All except right	`PREFETCHIT/XSAVE/XRSTOR`
			`XSAVEOPT/XRSTORS/XSAVEC`
			`XSAVES/PTWRITE/CLRSSBSY`

3 byte	N/A	None	All (94 instructions in total)

C AEX Notification Handler Assembly Stub

```
# \arg %rsi code_tickle_address
#            L-> bit 0: whether to write to data address
#            L-> bit 4: whether to add cycle delay
# \arg %rdx data_tickle_address
# \arg %rbp stack1_tickle_address
# \arg %rbx stack2_tickle_address
# \arg %rdi cr3_code_byte_address
.ct_enable_aexnotify:
    mov   RSVD_AEXNOTIFY_OFFSET(%rsp), %rax
    movb  $1, (%rax)      # Enable AEX-Notify
__ct_mitigation_begin:
    lfence                # Ensure earlier faults taken
.ct_check_write:
    movl  $63, %ecx
    shlx  %rcx, %rsi, %rcx
    jrcxz .ct_clear_low_bits_of_rdx
    lea   -1(%rsi), %rsi # Clear bit 0
    movb  (%rdx), %al
    movb  %al, (%rdx)
.ct_clear_low_bits_of_rdx:
    movl  $12, %ecx
    shrx  %rcx, %rdx, %rdx
    shlx  %rcx, %rdx, %rdx
    mov   $0x1000, %ecx
.ct_warm_caches_and_tlbs: # loops 64 times
    lea   -0x40(%ecx), %ecx
    call  *%rdi
    mov   (%rsi, %rcx), %eax
    mov   (%rbp, %rcx), %eax
    mov   (%rbx, %rcx), %eax
    mov   (%rdx, %rcx), %eax
    jrcxz .ct_restore_state
    jmp   .ct_warm_caches_and_tlbs
.ct_restore_state:
    movzx %sil, %ecx
    mov   RSVD_REDZONE_WORD_OFFSET(%rsp), %rdi
    mov   %rdi, -SE_WORDSIZE(%rsp)
    mov   RSVD_RDI_OFFSET(%rsp), %rdi
    mov   RSVD_RSI_OFFSET(%rsp), %rsi
    mov   RSVD_RBP_OFFSET(%rsp), %rbp
    mov   RSVD_RBX_OFFSET(%rsp), %rbx
    mov   RSVD_RDX_OFFSET(%rsp), %rdx
    mov   RSVD_RAX_OFFSET(%rsp), %rax
    jrcxz  .ct_restore_rcx
    .rept 20
        lea (%rsp), %rsp
    .endr
.ct_restore_rcx:
    mov   RSVD_RCX_OFFSET(%rsp), %rcx
    jmp   *RSVD_RIP_OFFSET(%rsp)
```