TLBlur: Compiler-Assisted Automated Hardening against Controlled Channels on Off-the-Shelf Intel SGX Platforms

Daan Vanoverloop[1], Andrés Sánchez*[2,4], Flavio Toffalini[2,3], Frank Piessens[1], Mathias Payer[2], and Jo Van Bulck[1]

[1]*DistriNet, KU Leuven,* [2]*EPFL,* [3]*RUB,* [4]*Amazon*

Abstract

Intel SGX's vision of secure enclaved execution has been plagued by a continuous line of side channels. Among these, the ability to track enclave page accesses emerged as a particularly versatile and indispensable attack primitive. Despite nearly a decade since the original *controlled-channel* attack, existing mitigations remain focused on detection rather than prevention or depend on impractical developer annotations and hypothetical hardware extensions. This paper introduces TLBlur, a novel approach that leverages the recent AEX-Notify hardware extension in modern Intel SGX processors to essentially limit the bandwidth of controlled-channel attacks to the anonymity set of recently used pages.

Our defense leverages the fact that page translations served from the processor's Translation Lookaside Buffer (TLB), which is forcibly flushed during enclave interruptions, remain oblivious to adversaries. We introduce practical compile-time instrumentation to seamlessly log page accesses within the protected enclave application. Additionally, we utilize AEX-Notify to implement a custom enclave interrupt handler that hides the N most recently accessed application pages by transparently prefetching them into the hardware TLB. Our evaluation on real-world libraries such as libjpeg, yescrypt, wolfSSL, and OpenSSL yields acceptable performance overheads, improving over prior work with several orders of magnitude.

1 Introduction

Intel Software Guard Extensions (SGX) [19, 46], included in selected Intel processors from 2015 onwards, became a popular commercial Trusted Execution Environment (TEE) in both academia and industry. SGX safeguards application compartments, called *enclaves*, from unauthorized access, including by privileged operating system software, making it a key solution for confidential computing in the public cloud, now widely offered by major cloud providers [57].

While enclave memory is protected against direct accesses, SGX's strong root adversary model has led to numerous side-channel attacks. The literature generally distinguishes two main classes of software-based side-channel attacks on TEE platforms [24, 49, 51, 53]. The first type of *microarchitectural* contention attacks [41, 47, 48, 73] exploits shared internal CPU state between enclaves and attackers. These attacks detect subtle timing differences, making them inherently non-deterministic and susceptible to measurement noise. Importantly, microarchitectural contention attacks are not exclusive to TEEs and can be principally mitigated by partitioning or flushing shared microarchitectural resources [20, 22, 25].

The second attack surface, unique to the privileged TEE adversary model, comprises *controlled-channel* attacks [85]. Unlike microarchitectural contention attacks, these rely solely on architecturally defined CPU interfaces, such as page tables and interrupts. By strategically revoking and restoring access rights to specific enclave code and data pages, privileged adversaries can deterministically trace enclave memory accesses at noiseless, 4 KiB spatial resolution. This page-fault channel has been shown to be particularly expressive, enabling the reconstruction of confidential text, JPEG images, and full cryptographic keys [61, 78, 85]. Single-stepping frameworks like SGX-Step [72] can further improve the temporal resolution of these attacks and enhance their ability to reconstruct cryptographic secrets [3, 4, 49]. Controlled channels have also served as a powerful and versatile building block for other SGX attacks, including memory-safety exploitation [40, 70, 77], triggering transient-execution CPU vulnerabilities [11, 68, 69, 75], or reducing microarchitectural noise [44, 63].

Controlled-channel attacks essentially expose a fundamental trade-off, where the TEE hardware safeguards the confidentiality and integrity of enclave memory, but the operating system remains in charge of resource management. Therefore, these attacks have been particularly evasive to mitigate in practice, despite numerous academic proposals (surveyed in §11). Particularly, existing defenses are not *(i) principled,* focussing exclusively on the spatial [53, 61, 86] or temporal [16] dimensions, or resorting to probabilistic detec-

*All the contributions of this author were made prior to joining Amazon.

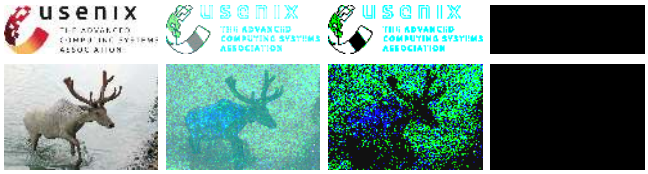


Figure 1: From left to right: example images, recovered via page faults, insufficient leakage-reduction with AEX-Notify single-stepping defense, and TLBlur's leakage elimination.

tion [13,52,60] rather than leakage prevention; *(ii) automated,* relying on manual and application-specific developer annotations [53,61,64,80,86]; and *(iii) practical,* exhibiting prohibitively high performance overheads [61,64,80] or relying on hypothetical hardware extensions [1,20,53,86] that are not supported on off-the-shelf Intel SGX platforms. In this respect, an important advance has recently been made with Intel's release of a new Instruction Set Architecture (ISA) extension for SGX, known as AEX-Notify [16,18]. This extension, already included in recent Intel processors, enables the implementation of trusted handlers that are invoked when an enclave resumes from an exception. AEX-Notify was originally designed to mitigate only single-stepping attacks [72] by using a small software handler that minimally determines the working set of the next instruction and prefetches just these pages into the processor's internal Translation Lookaside Buffer (TLB). This accelerates the next instruction, effectively preventing single-stepping and reducing the temporal resolution for controlled-channel attacks. Crucially, however, the existing AEX-Notify defense lacks any principled reasoning on tackling the spatial bandwidth of controlled-channel attacks, explicitly placing this more fundamental challenge out of scope [16]. As a case in point, Fig. 1 reproduces the seminal controlled-channel attack on libjpeg [85], showing that although the current AEX-Notify mitigation may somewhat reduce leakage in certain cases, it is still clearly insufficient to prevent effective reconstruction.

In this paper, we introduce TLBlur, the first comprehensive mitigation for off-the-shelf Intel SGX platforms that fully automatically addresses both the temporal and spatial bandwidth of controlled-channel attacks. TLBlur offers *principled* leakage prevention by leveraging the fact that enclave page translations served from the processor's internal TLB do not require a page-table walk. Hence, any pages that are transparently prefetched in an enclave interrupt handler will remain oblivious to controlled-channel adversaries until they are evicted from the TLB by its internal page-replacement policy or on the next interrupt. TLBlur uses this observation to effectively reduce the bandwidth of controlled-channel attacks to the anonymity set of recently used pages. Furthermore, TLBlur provides an *automated,* application-agnostic approach that eliminates the need for manual developer annotations, which have been criticized as impractical and error-prone [36]. In this respect, the key challenge we address in this work is how to automatically establish the subset of enclave pages to be prefetched for arbitrary enclave applications. To this end, we develop a compiler plugin and binary rewriting algorithm that automatically instrument the protected enclave application to seamlessly log its own page accesses. Upon resumption after a possibly malicious interrupt, we consult the log to transparently restore the TLB state and effectively hide the N most recently accessed enclave application pages. TLBlur, thus, allows to automatically learn from history during enclave execution and exploit the spatial and temporal locality inherent to real-world programs to considerably reduce—or "blur"—the page-access trace exposed to controlled-channel adversaries. As shown in Fig. 1 for the real-world libjpeg application, this reduction can be sufficient to effectively eliminate all leakage in practice (cf. §9.1). Finally, TLBlur is *practical* in the sense that it can be readily deployed on existing, off-the-shelf Intel SGX processors and yields acceptable performance that improves by an order of magnitude over previous controlled-channel defenses for security-critical enclave applications.

Contributions. To summarize, our main contributions are:

- We propose the first fully automated mitigation to address both the temporal and spatial bandwidth of controlled channels on off-the-shelf Intel SGX platforms.
- We implement an LLVM compiler instrumentation plugin, paired with a BOLT binary rewriting pass, to fully automatically track enclave application page accesses.
- We implement an AEX-Notify software handler that transparently prefetches recently accessed pages into the hardware TLB upon resuming from an enclave interrupt.
- We describe a leakage model and analyze the reduction of temporal and spatial bandwidth.
- We provide practical tools to assess leakage models by profiling enclave memory access patterns.
- We evaluate the effectiveness and runtime overhead of our defense on real Intel SGX hardware and real-world libjpeg, yescrypt, wolfSSL, and OpenSSL libraries.

2 Background

Address Translation. Modern processors translate virtual addresses into physical addresses before interacting with the memory controller. This is managed by the operating system using a hierarchical in-memory data structure called the page table. In the x86 ISA, each Page Table Entry (PTE) typically maps a contiguous 4 KiB physical memory area, but PTEs can optionally also accommodate larger areas of 2 MiB, 4 MiB, or even 1 GiB. Each PTE includes flags to indicate whether the page is present (P), writable (RW), executable (XD), accessed (A), or dirty (D). The A bit is set by the processor the first time a page is accessed, whereas the D bit is set when a page is first modified. These flags and permissions aid in effective

memory management and security enforcement.

TLBs are central to the address translation process, acting as caches for recently used virtual-to-physical address mappings. They enable the processor to quickly retrieve physical addresses without the need for a slower page table lookup. In new Intel x86 processors, TLBs are usually organized into two levels: smaller L1 TLBs for code and data, separately, and a larger unified L2 TLB for both. The TLB levels are non-inclusive and are organized as multi-way, set-associative caches with a (pseudo) least-recently used replacement policy [26, 67]. In processors with Simultaneous Multithreading (SMT), TLBs are shared across logical cores. Due to their performance role, TLB sizes have steadily increased in modern CPUs. For instance, the 4th-gen Intel Xeon CPU can store over 2,048 translations for 4 KiB pages (cf. Appendix B).

Intel SGX. Intel SGX [19] is an x86 extension enabling *enclaves* isolated even to privileged operating systems or hypervisors. SGX is currently a key server-side solution for confidential cloud computing, integrated into recent Intel Xeon processors and offered by leading cloud providers [57]. SGX is also a foundation for recent security technologies, *e.g.,* Intel's Trust Domain Extensions (TDX) [14].

SGX enclaves occupy a contiguous user-space memory region in the virtual address space of an untrusted host process. Enclave code and data pages are isolated in a dedicated Processor-Reserved Memory (PRM) area, using 4 KiB pages (larger pages are not supported for SGX enclaves). The untrusted operating system continues to be in charge of virtual-to-physical address translation and memory management. To prevent active page remapping attacks [19,50], SGX hardware uses an Enclave Page Cache Map (EPCM) to verify physical addresses during the untrusted page-table walk. SGX also provides instructions for securely moving enclave pages in and out of PRM. Crucially, while the processor's internal TLB serves as a trusted cache to speed up repeated page accesses, SGX's architectural security argument requires the TLB to be flushed on every enclave entry or exit [19].

The untrusted application can use the `EENTER` instruction to transfer control to a fixed entry point in the enclave, which can later return control through the `EEXIT` instruction. The Intel SGX SDK abstracts these instructions into secure functions, called `ecall`s, with optional `ocall`s for callbacks to the host. Enclaves can also exit asynchronously due to interrupts or exceptions, triggering an Asynchronous Enclave Exit (AEX) hardware event. AEX securely stores the CPU context in a Save State Area (SSA) within enclave memory before handing control to the untrusted operating system. The `ERESUME` instruction can then be used to restore the CPU context and transparently continue enclave execution, without the enclave ever being aware that it was interrupted.

AEX-Notify. AEX-Notify [16,18] is a recent ISA extension for SGX that allows an enclave to be *interrupt-aware*. This new hardware feature is available on off-the-shelf Xeon CPUs from Sapphire Rapids onwards and (via platform updates) on all previous generation Xeon CPUs that support Intel SGX [8]. AEX-Notify changes the behavior of the `ERESUME` instruction to adopt the semantics of `EENTER` when a configuration flag is set. With the help of AEX-Notify, developers can register trusted exception handlers to be called whenever the untrusted runtime wants to resume from an AEX. Additionally, AEX-Notify includes a new `EDECCSSA` instruction that can be used by the trusted exception handler to seamlessly switch to the originally interrupted context without exiting the enclave.

```
1 fun modpow(m, d, n):
2   rv = 1; nb = nbits(d)
3   for i from nb to 0:
4     rv = square(rv,n)
5     if (d & (0x1 << i)):
6       rv = mult(rv, m, n)
7   return rv;
```

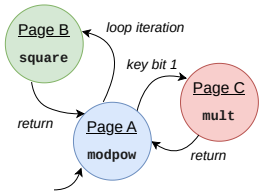


Figure 2: Cross-page conditional control flow in the square-and-multiply algorithm for modular exponentiation.

3 Motivation: Controlled-Channel Attacks

Running Example. Consider the program in Fig. 2, which includes simplified pseudocode for the square-and-multiply modular exponentiation algorithm that is a popular target for controlled-channel attacks [49, 61, 74, 76]. The algorithm computes $r = m^d$ by iterating over the binary representation of d, squaring r each time, and multiplying by m when the current bit of d is set. Crucially, such secret-dependent control flow may reveal bits of the private exponent d through page accesses. The state machine in Fig. 2 illustrates the sequence of code page accesses, assuming the `modpow`, `square`, and `mult` functions reside on separate 4 KiB code pages.

Spatial Bandwidth. Nearly a decade ago, Xu et al. [85] first showed that privileged adversaries can unmap or revoke access rights to a specific enclave code or data page and receive deterministic notifications via a page-fault signal when that page is next accessed. However, as access rights must be restored to allow progress after the initial access, subsequent accesses to the target page will hit in the TLB and remain hidden from page-table adversaries. Xu et al. recognized that strategically revoking access to *another* page referenced some time after the initial target page triggers a new page-fault AEX and full TLB flush, allowing permissions to be revoked again. This way, the *sequence* of relatively coarse-grained 4 KiB page faults can uniquely identify specific points in the victim's execution. For example, the state machine in Fig. 2 shows how to deterministically recover the full private

exponent by alternatingly revoking and restoring access to pages A, B, C and observing different page-access sequences for iterations with a one $\langle A,B,A,C,A\rangle$ vs. zero $\langle A,B,A\rangle$ bit. This technique has proven highly effective, often enabling the extraction of complete cryptographic or application secrets in a single, noise-free run [3, 4, 49, 61, 70, 72, 74, 76, 78, 85]. Furthermore, page faults are a key exploitation primitive in numerous SGX attacks [11, 40, 44, 63, 68–70, 75, 77] and have been generalized to related TEE technologies that expose untrusted page tables, like AMD SEV [29, 50, 79, 82].

Temporal Bandwidth. An improved line of high-resolution controlled-channel attacks [3, 4, 49, 72, 74] leverage timer interrupts to "single-step" the victim enclave one instruction at a time. By observing PTE A/D bits, single-stepping adversaries precisely count the number of enclave instructions executed between different page accesses. This improved temporal resolution has been leveraged to overcome spatial limitations of the page-fault channel, effectively refining a coarse-grained page-fault sequence $\langle A,B,A\rangle$ into an instruction-granular page-access trace $\langle A,B,\ldots,B,A\rangle$. In this case, even if two different functions of varying lengths reside on the same page B, they may still be distinguished by merely counting instructions. The AEX-Notify hardware feature was designed to support a software mitigation that thwarts this temporal advantage for single-stepping attacks [16]. To this end, a trusted exception handler transparently decodes the next enclave application instruction and atomically prefetches all code and data pages about to be accessed into the TLB. Particularly, the existing mitigation decodes the SSA frame of the interrupted enclave application and prefetches: *(i)* unconditionally, the code page of the next instruction; *(ii)* conditionally, the data page accessed by that instruction (if any, determined via a constant-time disassembler); *(iii)* unconditionally, two pages surrounding the current stack pointer. This ensures that executing the next application instruction after `ERESUME` will be fast, minimizing the attacker's window to hit a timer interrupt for successful single-stepping. This mitigation is now integrated by default into the Intel SGX SDK. Importantly, while effectively countering the temporal advantage of single-stepping attacks, the existing AEX-Notify handler does *not* principally address the spatial bandwidth of controlled-channel attacks. Indeed, the more ambitious objective of addressing information leakage from common page-fault sequences, like those exploited in Figs. 1 and 2, was left explicitly out of scope [16].

4 Problem Statement

4.1 Threat Model and Out-of-Scope Attacks

We assume Intel's standard SGX threat model [19, 34], where the adversary controls the operating system or hypervisor. Attackers have access to high-resolution timer interrupts [72] and page-table memory [74, 85] to provoke page faults and distinguish read/write/execute accesses on 4 KiB enclave pages.

Our current mitigation prototype only supports single-threaded enclave applications, the default for Intel SGX and prior attack targets. If needed, multithreading support could be implemented straightforwardly (cf. § 10). Furthermore, in line with secure SGX enclave deployment practices, we assume SMT (Intel HyperThreading) is disabled at the system level, as officially recommended by Intel in response to recent processor vulnerabilities [9, 58, 68, 75]. This assumption excludes uncommon and less reliable attacks [26, 76] that may evict cached page translations from the shared TLB through contention with a sibling logical core, *i.e.,* without interrupting the enclave. Intel's SGX remote attestation procedure [35] now includes boot-time SMT status, which cannot be re-enabled without a reboot, enabling remote parties to validate SMT status securely. Alternatively, existing software techniques [12, 27, 52] could ensure both sibling SMT cores enter and exit the enclave in lockstep. Finally, this work aims to address *architectural* controlled-channel leakage at a 4 KiB page-level granularity. Hence, in line with prior SGX controlled-channel defenses [1, 53, 61, 64, 86, 87], we deem any finer-grained *microarchitectural* side channels, including cache timing [48], explicitly out of scope. Microarchitectural contention attacks pose an orthogonal challenge, not specific to enclave settings, which can be mitigated by using existing techniques [20, 22, 25] in tandem with TLBlur.

4.2 Leakage Model and Security Argument

Page-Access Traces. Similar to existing formalizations [61, 65], we model controlled channels as an observation function that produces a finite ordered sequence of page accesses:

$$\mathcal{A}(E,I) \mapsto \langle P^{r|w|x}, Q^{r|w|x}, \ldots, R^{r|w|x}\rangle \tag{1}$$

In this notation, P, Q, and R are successive page numbers, with $P^{r|w|x}$ indicating a read (r), write (w), or execute (x) access on P. The overall length of the trace reflects the temporal granularity of the page-access observations, *i.e.,* page-fault sequences [85] or instruction-granular single-stepping [49, 72]. Intuitively, a program E leaks under this model when there exist two different secret inputs I_1 and I_2 that an adversary can distinguish solely by looking at their respective page-access traces $\mathcal{A}(E,I_1)$ and $\mathcal{A}(E,I_2)$. This is clearly the case for the running example in Fig. 2, which produces $\langle A^x,B^x,A^x,C^x,A^x\rangle$ and $\langle A^x,B^x,A^x\rangle$ for a 1 vs. 0 bit, respectively.

TLB Leakage Reduction. Crucially, prior formalizations [61, 65] do not consider the trusted TLB cache in their leakage models. We observe that enclave page translations hitting the TLB bypass the page-table walk, making them *oblivious* to controlled-channel attacks. Hence, we model the TLB as a stateful filter with capacity n that can reduce the observed

page-access traces: $\forall n \geq 0 : \mathcal{A}_{TLB=n}(E,I) \subseteq \mathcal{A}(E,I)$. This filter's exact operation depends on the accumulated state $\leq n$ and the internal replacement policy. In this respect, interrupts can be modeled as forcibly resetting the accumulated state, *i.e.,* flushing the TLB, at a chosen point in the instruction-granular page-access trace. Thus, a single-stepping adversary that flushes the TLB after every instruction essentially forces its internal capacity to $n = 0$, resulting in maximal page-access trace leakage: $\forall E,I : \mathcal{A}_{TLB=0}(E,I) = \mathcal{A}(E,I)$.

We make the key observation that, by registering a custom exception handler with AEX-Notify, a non-empty initial state S with $0 < |S| \leq n$ can be loaded into the TLB before resuming the enclave application after an interrupt. In this respect, by prefetching the working set of the immediately next enclave application instruction, the AEX-Notify single-stepping mitigation [16] purposefully hides all subsequent accesses within that working set. Hence, with the existing defense in place, a page fault is observable only when accessing a new code or data page outside the prefetched set, effectively shortening Eq. (1) from an instruction-granular, single-stepping trace to a more concise page-fault sequence. Notably, this page-fault sequence also excludes the two current top-of-stack pages, which are unconditionally prefetched to simplify the constant-time disassembler (cf. §3). As an unintended side-effect of unconditionally prefetching these stack pages, we found that the existing mitigation may in some cases marginally reduce, but certainly not eliminate, the spatial bandwidth of the page-fault channel, *e.g.,* somewhat coarser libjpeg recovery for the example images in Fig. 1.

We conclude that the existing AEX-Notify single-stepping defense, while effectively limiting the temporal resolution, does *not* principally address the spatial aspect of controlled-channel attacks. Indeed, prefetching the working set of the next instruction prevents single-stepping but does not anticipate later code or data page accesses, which remain visible to a page-fault adversary, as exploited in Figs. 1 and 2.

Goal of our Defense. From the above, we conclude that the worst-case leakage $\mathcal{A}_{TLB=0}$ clearly happens when the initial TLB state after an interrupt is left empty. Additionally, *any* non-empty state $S \neq \emptyset$ prefetched in the AEX-Notify software handler will effectively reduce the leakage. In this regard, the easiest case naturally arises when the handler can simply prefetch *all* (secret-dependent) enclave application pages and they can be accommodated within the TLB. This is indeed the implicit assumption underlying prior pinning-based solutions relying on manual annotations [53, 64, 86].

Thus, a key challenge that remains unanswered by prior research is *how to autonomously determine the set of enclave application pages to be prefetched on the next interrupt.* Our key contribution is to devise a novel Page Access Map (PAM) data structure and automated compile-time instrumentation approach to efficiently and transparently record the application's past accesses to code and data pages during its execution. This provides our custom AEX-Notify software interrupt handler with the required information to seamlessly re-fill the hardware TLB with a Prefetch Working Set (PWS) consisting of the N most recently accessed pages on every interrupt. Therefore, under the common assumption that the enclave program exhibits spatial or temporal locality, TLBlur automatically learns and seamlessly prefetches its working set, effectively concealing any secret-dependent page-access patterns therein. Applied to the running example in Fig. 2, TLBlur with a PWS of size $N \geq 3$ reduces the leakage from a lengthy sequence like $\langle A^x, B^x, A^x, C^x, A^x, B^x, A^x, B^x, \dots \rangle$ to the *finite* sequence $\langle A^x, B^x, C^x \rangle$ for a secret key of arbitrary length containing 1 and 0 bits.

As the PWS size parameter N increases, TLBlur effectively extends the application's working set further into the past, thereby further "blurring" the adversary's observations. The upper bound for this blurring effect is naturally dictated by the associativity and size n of the hardware TLB. Intuitively, when $N = n$ and the PAM perfectly mimics the hardware TLB's design, it would appear to the adversary as if no interrupt ever occurred. In this case, the same remaining leakage trace could theoretically also be obtained by continuously monitoring page-table memory [74] without ever triggering interrupts in the first place. However, achieving such a "perfect" PAM would require reverse-engineering undocumented and brittle hardware replacement policies [67], leading to substantial slowdowns due to the complexity of instrumentation and making the solution inherently microarchitecture-specific. Importantly, we do not need to perfectly replicate the unpredictable behavior of hardware TLBs; any non-empty state $PWS \neq \emptyset$ will already contribute to reducing leakage in practice. Thus, as a key contribution, we devise a *generic*, hardware-agnostic PAM data structure that enables a straightforward and deterministic prefetching policy, effectively blurring page-access patterns while maintaining performance and portability.

4.3 Mitigation Objectives

O1: Principled Leakage Reduction. The mitigation must act as a sliding-window filter of configurable size N over the page-access trace $\mathcal{A}(E,I)$ that widens the observational granularity of the attacker. In other words, only the first access to a page that falls out of the current window is allowed to be leaked, and any subsequent page accesses that fall within this window must remain oblivious, even in the presence of advanced interrupt-driven attackers.

O2: Automation. Applying the mitigation must be possible via an automated, application-agnostic compilation pipeline without developer annotations, which have been criticized even by security-aware cryptographic developers [36] and do not generally scale to general-purpose applications [85].

O3: Practicality. The mitigation must be deployable on existing Intel SGX processors without theoretical hardware

changes. Furthermore, it should preserve the effectiveness of the existing mitigation [16] against single-stepping attacks.

Non-Goals. TLBlur represents a pragmatic trade-off, where we dynamically learn the working set during actual execution instead of relying on static and error-prone developer annotations. Therefore, removing *all* leakage from page-table accesses is explicitly out-of-scope. Indeed, in the provided definition, at least the initial access to every page is still leaked, and even in cases where the enclave remains uninterrupted, limited information could still be leaked through page table entries due to the constrained size of the hardware TLB. We discuss further limitations and edge cases, including the effect of the hardware TLB's associativity, in §10.

5 TLBlur Overview

Figure 3 overviews the key components of our defense, which operates through the following high-level steps.

1. Recording Application Page Accesses. Our approach relies on a software-based PAM data structure that records recent application code and data page accesses. We design a fully automated compiler pipeline, described in §6, which inserts instrumentation code to transparently update the PAM at runtime. Our instrumentation passes do not interfere with attestation, as SGX attests the final instrumented binary. Furthermore, we ensure that our compiler pipeline behaves deterministically, allowing reproducible builds.

2. Asynchronous Enclave Exit. During enclaved execution, interrupts or exceptions cause an AEX to the untrusted operating system. Such AEXs can be triggered by a privileged adversary through timer or inter-processor interrupts or manipulation of PTEs. Importantly, as part of SGX's architectural specification, the processor must flush the TLB on every AEX [7, 34]. However, privileged adversaries can still determine which pages have been accessed during enclaved execution by examining the page-fault base address or PTE A/D bits. TLBlur obscures this view to the anonymity set of recently accessed code and data pages.

3. Restoring TLB State. The AEX-Notify hardware extension ensures that a pre-registered trusted exception handler is executed when resuming the enclave. Our mitigation introduces a carefully crafted, constant-time AEX handler, described in §7.1, to seamlessly restore hardware TLB entries before resuming the shielded enclave application. This prefetcher software component transparently parses the PAM to establish the N most recent application page accesses and prefetches those pages into the hardware TLB with the appropriate (maximal) read-write-execute permissions.

4. Leakage Reduction. Consecutive accesses to recently accessed pages hit in the processor's internal hardware TLB. Such TLB hits effectively bypass the untrusted page table, making them *oblivious* to controlled-channel adversaries. In the absence of attacker-induced TLB flushes through interrupts, this scenario naturally holds true. Moreover, in the event of an interrupt, TLBlur's prefetcher component ensures the reloaded state of the flushed hardware TLB. Specifically, accesses to the N most recently accessed pages remain unobservable by adversaries monitoring PTEs. To assess the leakage reduction observable in instrumented enclave applications, we develop a practical profiler tool in §7.2.

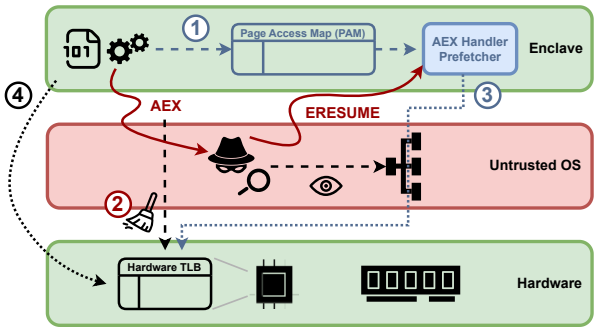


Figure 3: TLBlur maintains a software-based PAM data structure ① during runtime of the protected enclave application. Upon interrupt, the hardware TLB is flushed, ② and the enclave will be resumed at the AEX-Notify exception handler, which transparently parses PAM and prefetches ③ the N most recent pages into the hardware TLB, such that any future accesses ④ to these pages remain oblivious.

6 Static Enclave Instrumentation

The key challenge faced by the general mitigation approach outlined in the previous section is how to adequately instrument an enclave application to efficiently record its own page accesses. A naive approach would be to update the software-level PAM data structure (§6.1) before *every* single instruction, so as to accurately trace the working set of all code and data pages accessed. However, this would incur prohibitive performance overhead. Thus, to reduce the number of instrumentation points, while obtaining a comprehensive representation of the PAM, we design the instrumentation around the observation that TLBlur needs to record memory accesses only at a 4 KiB page-level granularity. This observation poses an additional challenge. The final page layout is only known after linking, whereas classic instrumentation techniques operate at compilation time, before the precise binary layout is established. Figure 4 illustrates how we overcome this challenge through an inventive fusion of compile-time instrumentation (§6.2) and binary-level rewriting (§6.3) without the need for code annotations or application-dependent settings.

We divide the instrumentation task into different sub-cases for distinct code and data accesses, as outlined below:

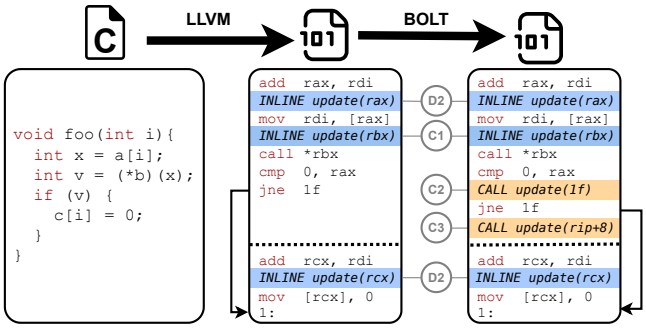


Figure 4: Compilation pipeline with an LLVM backend pass instrumenting global data (D2; blue) and indirect control flow accesses (C1; blue). A post-link binary rewriting BOLT pass instruments direct control flow (C2–C3; orange).

D1. Local Variables. Local variables are allocated on the runtime call stack, which can be easily located by inspecting the stack pointer in the stored SSA frame. Recalling §3, the existing AEX-Notify single-stepping mitigation already protects stack pages by prefetching a number of pages surrounding the interrupted stack pointer [16]. Thus, as an important optimization, we refrain from instrumenting common instructions that only access stack pages.

D2. Global Variables. We conservatively instrument all other data accesses, *i.e.,* allocated in the global data section or on the heap, to record their target address in the PAM.

C1. Indirect Control Flow. For indirect control flow instructions, it is undecidable to statically determine whether they will cross a page boundary as their targets depend on runtime information, *e.g.,* as is the case for `jmp` or `call` with a memory or register operand. Since such indirect control-flow transfers are relatively rare and already come with a performance impact, we choose to conservatively instrument all indirect control-flow transfers at compile time to record their target address in the PAM. This is similar to common indirect control-flow instrumentations by existing memory-safety [66] or speculative-execution [32] mitigations.

C2. Direct Control Flow. For direct `jmp` or `call` instructions with an absolute target, it is in principle possible to statically determine whether they cross a page boundary and instrument only those instructions that explicitly transfer control flow from one page to another. As the page layout is unknown during compilation and available only after linking, we use a fixed-point binary rewriting algorithm to repeatedly extract page layout information and instrument direct control flows that cross a page boundary.

C3. Fallthrough. Finally, compilers generally do *not* align basic blocks to exist entirely within a single page. Thus, normal execution may cross code page boundaries even without any explicit control-flow transfer instructions. Our binary rewriting algorithm covers these subtle cases.

6.1 Page Access Map Data Structure

We represent the PAM data structure as a linear array of 64-bit integers indexed by page numbers, where the values indicate how recently each corresponding page was accessed. Specifically, when recording a page access in the PAM, a global 64-bit counter is incremented, and its value is copied into the corresponding array slot for that page. Consequently, the N most recently accessed pages can be identified by locating the N highest numbers in the PAM array. While updating the PAM incurs minimal runtime overhead, determining the most recent pages to prefetch is somewhat more costly, as it requires sorting the array. Nevertheless, this trade-off proves advantageous since interrupts typically occur relatively infrequently under normal circumstances, compared to the number of runtime instrumentation calls.

To efficiently update PAM, we draw inspiration from existing sanitizer tools that have undergone extensive engineering optimizations to minimize runtime overhead [59]. The resulting hand-crafted assembly routine to update PAM is given in Appendix A. This function takes the address of a memory location and updates PAM accordingly with the page number of this address. We subtract the enclave base from the given address and shift the resulting address to compute the page number. Note that to obtain the page number, the address must be shifted to the right by 12 bits, but since PAM consists of an array of 64-bit integers, we must also shift the address back left by 3 places. We assume a well-behaved, memory-safe program where all accessed addresses fall within the memory address range of the enclave (as required for secure enclave execution [5, 15, 70]). The PAM array needs to be sufficiently large to cover the enclave address range. To avoid any out-of-bounds accesses, we mask the page number with the size of PAM subtracted by one (*e.g.,* `0xffffff` for a PAM of size `0x1000000`). To update the PAM we load the global counter, increment it, and store it back to the array.

6.2 Compile-Time Instrumentation

At compilation time, we instrument the code's non-stack memory accesses ($D2$) and indirect control-flow events ($C1$). We implement our instrumentation pass in LLVM's x86 backend. Our pass iterates over all instructions and identifies non-stack memory accesses, indirect calls, and indirect jumps by inspecting the instruction's addressing mode. The addressing mode refers to the set of operands that describe a memory location and describe symbols referencing code and heap memory locations. We store the accessed address in a new register using a `lea` instruction with the previously identified addressing mode as operands.

Once we determine the accessed address, we inline the code from Appendix A responsible for updating the PAM data structure. This approach is crucial for allowing the compiler's register allocator to optimize register usage, which is

Algorithm 1: Fixed-point binary instrumentation.

```
1 repeat
2 |   code_layout ← calc_code_layout()
3 |   insert_direct_cf_tracing()                ▷ Handle C2
4 |   code_layout_upd ← calc_code_layout()
5 until code_layout == code_layout_upd
6 insert_code_page_crossing_tracing()           ▷ Handle C3
7 fix_direct_call_tracing()
```

typically more efficient than executing a `call` instruction. Notably, physical registers implicitly used by our instrumentation code, such as the `rflags` register, must be saved to maintain program semantics. However, saving `rflags` by pushing it onto the stack introduces a notable performance overhead due to CPU pipeline serialization. Therefore, we employ a liveness analysis to identify the in-use physical registers and avoid saving registers that are not live.

6.3 Binary Instrumentation

TLBlur adopts binary rewriting techniques to infer the code page boundaries from the linked enclave binary and instrument only those direct control-flow transitions ($C2$) or fallthrough code locations ($C3$) that traverse page boundaries. To reduce binary rewriting, we opt for function calls to the PAM update procedure in Appendix A instead of inlining. Injecting function calls require a lower number of opcodes (*i.e.,* two `push` instructions and a `call`), thus introducing a less disruptive modification to the original code. More importantly, as evident from our experimental evaluation in §9, these code page transitions are relatively infrequent.

Injecting binary code and controlling the page layout is non-trivial. Essentially, each injected tracing call modifies the enclave page layout, potentially moving the target of other direct control-flow transfers to a new code page. To address this circularity, we design an inventive fixed-point algorithm, described in Algorithm 1. The algorithm first instruments all direct control flows that cross a page boundary (line 3). Then, we recompute the page layout (line 4). These operations are repeated until the code layout does not change (line 5). Both `calc_code_layout()` and `insert_direct_cf_tracing()` conservatively reserve space for cross-page fallthrough instrumentation, *i.e.,* they assume a page size slightly smaller than 4 KiB. Our approach ensures it is safe to insert page crossing instrumentation without altering the fixed-point enclave page layout.

After reaching the fixed point, the enclave binary holds three properties: *(i)* all cross-page direct control flow transfers are instrumented; *(ii)* some direct control flow transfers may have superfluous instrumentation; and *(iii)* the enclave binary layout is stable. Next, we instrument fallthrough locations by adding the instrumentation at the end of every code page (line 6). As the final step, we fix all the needless instrumentation for the direct control flow events that do not need it due to the final layout of the binary (line 7).

We provide an opt-in mechanism to enable instrumentation of direct control flow by only instrumenting code in a section called `.tlblur.text`. Enclave application code is transparently moved to this section, while other code, such as the SDK's trusted runtime, remains in the original sections.

6.4 Optimizations

Several parameters configure the implementation and enable optimizations that can reduce runtime overhead.

First, instrumentation can be inserted either *before* each control-flow transfer instruction, or it can be inserted *after*, *i.e.,* at the target locations. Placing the instrumentation at the target is beneficial as the PAM only needs to be updated when the branch is taken. Additionally, the `rflags` register is unlikely to be live at this point, so it does not need to be saved. In contrast, when the instrumentation is inserted right before a conditional jump, `rflags` is typically in use and must be saved beforehand, which is a costly operation.

Second, we create another optimization that groups pages into logical sets that are tracked in the PAM as a single unit. When the AEX handler is executed, we make sure that all pages within such set are prefetched together. By grouping a fixed number of subsequent pages, TLBlur effectively simulates larger logical pages and reduces the number of logical page boundaries that need to be instrumented, which considerably lowers the amount of instrumentation code executed. The number of pages grouped together is configured statically using a compiler option, which is applied to all code pages. This optimization is particularly valuable in scenarios where a performance-critical code segment is placed across two subsequent pages, such that the instrumentation at the boundary between these pages would be executed frequently. By grouping contiguous pages, and treating them as a single large page, we can overcome such performance bottlenecks.

7 Dynamic Runtime Components

7.1 AEX-Notify Page Prefetcher

Constant-Time PAM Processing. After an AEX, the enclave executes our custom exception handler, registered via AEX-Notify, to transparently repopulate the hardware TLB before resuming normal execution. In contrast to the shielded enclave application, the majority of the exception handler itself runs without the protection of AEX-Notify and may, hence, fall victim to precise single-stepping attacks [49, 72] that try to learn the contents of the PAM via side-channel analysis. Leaking individual PAM counter values would reveal the amount of times a page has been accessed and, thus, increase the adversary's observational power. Therefore, we carefully implemented a constant-time sorting procedure using side-channel-resistant `cmov` x86 instructions to sort the

PAM array and determine the N most recently accessed pages to be added to the PWS. Next, to prevent attackers from inferring the freshness of PAM entries based on the order in which pages are prefetched, we additionally constant-time sort the resulting PWS by page number.

Note that, in addition to recently accessed pages, we make sure to also include the data page for the PAM array itself, as well as the code page containing the `tlblur_pam_update` function, into the PWS. Lastly, any PAM modifications during sorting and prefetching are avoided by making sure that the exception handler itself is not instrumented.

Atomic Prefetching. We atomically prefetch the selected pages through a practical extension to the existing AEX-Notify single-stepping mitigation included in recent SDK versions [16]. The prefetching stage consists of a hand-crafted assembly stub that first re-enables AEX-Notify, before accessing all PWS pages and jumping to the enclave application resumption point. In case of any interrupts or page faults during prefetching, the prefetcher is restarted as part of the AEX-Notify flow, thus ensuring atomicity.

Similar to the existing single-stepping mitigation, we make sure to prefetch pages with the maximum allowed permissions. For example, if a page is writable, we prefetch the page by reading a byte from that page and writing it back. Similarly, we handle executable pages by locating and calling an x86 `ret` byte (`0xc3`) on that page. This approach maximizes the protection against page-table adversaries, as only reading a page that is also writable or executable would still allow to observe page faults via the *RW* or *XD* PTE attributes.

7.2 Offline Profiler Tool

To evaluate the completeness of the instrumentation and the effectiveness of the "blurring" effect in practice, we develop an automated page profiler tool for offline evaluation of instrumented SGX enclave binaries. Our tool constructs page-access traces for different TLBlur configurations by simulating a set-associative TLB that filters the maximal, instruction-granular page-access trace obtained by single-stepping the enclave in debug mode without AEX-Notify enabled.

We start by initializing the simulated TLB to an empty state. Next, before single-stepping to the next enclave instruction, we clear all PTE accessed bits. After every single-step interrupt, we check whether any of the newly accessed enclave pages $\mathcal{P}$ is not currently in the simulated TLB, in which case we model an AEX event. These AEXs determine the temporal resolution (*i.e.,* relative length) of the resulting trace $\mathcal{T}$. To faithfully determine the blurred spatial observations corresponding to every AEX, we *(i)* log both $\mathcal{P}$ and the current content of the simulated TLB in $\mathcal{T}[aex\_nb]$; *(ii)* flush the simulated TLB; *(iii)* determine the PWS containing all pages to be prefetched; and *(iv)* refill the simulated TLB with PWS.

Step *(iii)* above allows to conveniently implement different mitigation configurations, whereas step *(iv)* enables reasoning about the effect of the simulated hardware TLB size, associativity, and replacement policy. Particularly, the limited temporal reduction achieved by the current AEX-Notify single-stepping mitigation can be modeled by setting PWS to the top-of-stack and next code and data pages (cf. §3). To evaluate different instantiations of TLBlur, we parse the actual PAM data structure in enclave memory to accurately prepare PWS with a configurable size N.

Our profiler tool supports exporting the generated page-access traces to the widely used Value-Change Dump (VCD) format, commonly employed in hardware development. This enables the utilization of standard VCD visualization tools and graphical user interfaces, facilitating the investigation of resulting leakage traces by human analysts.

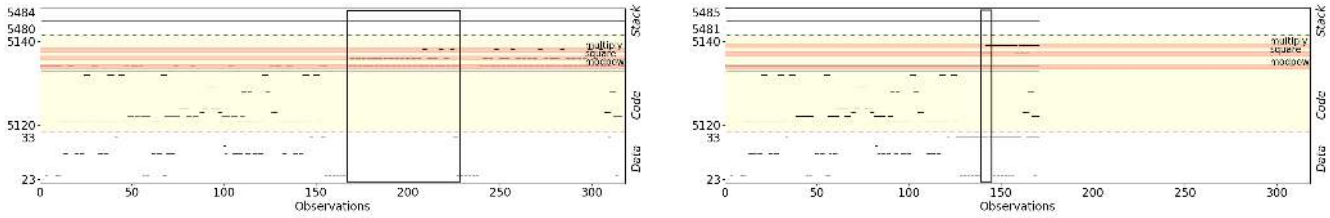
8 Implementation Aspects

Instrumentation. We implemented our compile-time instrumentation as an extension of the LLVM compiler [39], version 18.1.2. Instead of using instrumentation at the source level or in the intermediate representation, we implemented our instrumentation pass in the compiler backend (*i.e.,* the target code generator). This significantly reduces the risk of missing instrumentation on instructions that are inserted when lowering to target instructions, or created by transformation on intermediate code [17, 84]. Our instrumentation pass contains $\sim$ 500 lines of code. Instructions that use thread-local storage or instructions introduced by SIMD extensions are currently not supported by our prototype implementation.

We use BOLT [55], a post-link binary optimization tool, for binary rewriting. BOLT disassembles a compiled and linked binary, reconstructs the control-flow graph and reassembles the binary after applying numerous optimizations. Instead of optimizing the binary layout, we make use of this tool to apply an instrumentation pass that instruments control flow near page boundaries. This pass contains $\sim$ 400 lines of code.

Prefetcher. Our prefetcher is integrated into the standard SGX SDK version 2.23. It consists of 264 lines of C code for the constant-time PAM sorting and 22 lines of assembly code to extend the existing atomic prefetching phase with three loops to set read, write, and execute permissions.

Prefetching with maximum permissions requires determining the permissions of each page. We accomplish this using a custom linker script that inserts symbols to statically mark the boundaries of executable and read-only sections. All pages that need to be prefetched are considered readable. Executable section boundaries are statically configured via the linker script, whereas all other pages outside these sections are considered writable.



(a) AEX-Notify single-stepping mitigation; Lempel-Ziv: 56,220.

(b) TLBlur with PWS of size 3; Lempel-Ziv: 16,866.

Figure 5: Page-access traces collected by our profiler for an example enclave executing the dummy `modpow` function (highlighted). Compared to the existing AEX-Notify single-stepping mitigation (left), TLBlur further reduces temporal resolution (x-axis range) and blurs residual secret-dependent spatial patterns (visible as horizontal lines on the `square` and `multiply` pages).

Profiler. We implemented our profiler tool in $\sim$ 1000 lines of Rust code. The profiler sets the x86 trap flag to execute the enclave in single-step mode, which is available for non-production enclaves signed in debug mode. We observe page accesses at each step by using the SGX-Step framework [72], which enables programs to read and write PTEs of the enclave. We developed practical Rust bindings for SGX-Step and precisely determine the pages accessed by each instruction by clearing the accessed and dirty bits at each step. Additionally, we read the state of the PAM by using the `EDBGRD` instruction.

9 Experimental Evaluation

All experiments were conducted on an Intel Xeon Silver 4410Y processor with support for SGX2 and AEX-Notify. This machine has a unified L2 TLB that can hold 2,048 entries for 4 KiB code and data page translations (cf. Table 2 from Appendix B). To assess the runtime overhead of the instrumentation, we use the processor's internal time-stamp counter register, which reports the number of CPU cycles. Each benchmark is executed in a loop to determine stable averages of the execution times. The number of iterations for each benchmark is chosen to balance a reasonable experimentation time and stable results with low standard deviation. We compare the impact of data-access instrumentation and control-flow instrumentation, as well as the different optimizations, by compiling our code with different parameters to separate enclave binaries.

9.1 Security Effectiveness Validation

To assess leakage when using TLBlur, we investigate the page-access trace of the victim enclave by using our profiler (§7.2), where we assume a TLB with 128 sets and 8 ways per set.

Case Study: Square-and-Multiply. For the running example (§4.2), we validated that with a small PWS of size $N \geq 3$, secret-dependent control-flow patterns to the `modpow`, `square`, and `multiply` pages are indeed reduced to only the first access. This is visualized in Fig. 5, where we also include Lempel-Ziv complexity numbers [42] to quantify the total amount of extractable information (not necessarily secret-dependent) from a single enclave execution.

Case Study: libjpeg. We assess the security effectiveness of TLBlur using a real-world libjpeg benchmark application, which has been a prominent target in controlled-channel attacks [28, 85] and defenses [53, 60, 86, 87]. In this scenario, the victim enclave includes the unmodified libjpeg v9e code and offers an `ecall` to securely decode secret input images in enclave memory. The adversary observes the page access of the victim enclave and uses this information to reconstruct the processed input image. For our experiments, we reimplemented the attack described by Xu et al. [85], which is also included in our open-source artifact (§13) to encourage further research on controlled channels.

The existing AEX-Notify single-stepping mitigation [16] may partially hinder image reconstruction by unconditionally prefetching stack pages. However, it does not fully eliminate leakage, as page-fault adversaries can still observe fault sequences on code pages and heap memory accesses. To demonstrate the attack's remaining potential, Fig. 1 and Appendix C provide example images alongside their reconstructions.

Using the profiler, we find that a PWS of size 30 is sufficient to obscure secret-dependent patterns in practice. With TLBlur, no data leakage remains except for the total number of 8×8 JPEG blocks in the input image. Notably, this information is not even sufficient to reconstruct the image dimensions, meaning the black rectangles in Fig. 1 overestimate the actual remaining leakage. For example, a 1600×1000 image, consisting of 25,000 8×8 blocks, would have 24 possible dimensions. We provide the full page-access traces for libjpeg, along with further details, in Appendix D.

Takeaway. Our experiments clearly demonstrate how TLBlur automatically "blurs" the information leakage observable to controlled-channel adversaries. The libjpeg example underscores the need for automated compiler solutions, given

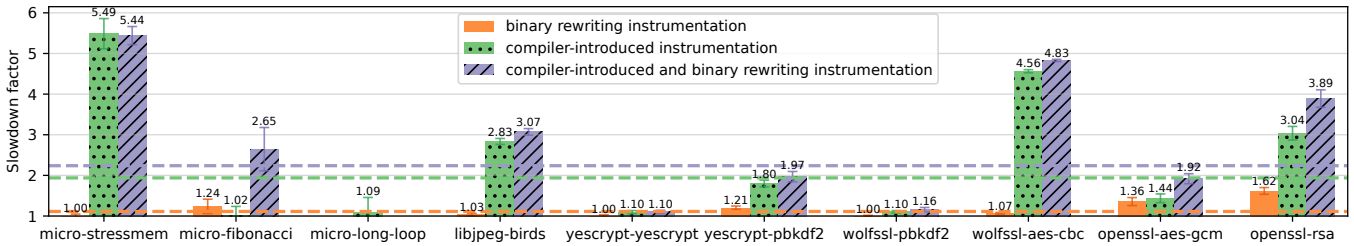


Figure 6: Average execution slowdown across benchmark applications. The dashed line corresponds to the geometric mean.

the infeasibility of manually crafting constant-time code for general-purpose applications. Even in the security-sensitive domain, *e.g.,* cryptography, writing constant-time code remains notoriously challenging and error-prone [36].

9.2 Performance Evaluation

We first assess the runtime overhead imposed by TLBlur's instrumentation via meaningful cryptographic algorithms, the libjpeg enclave from Xu et al. [85], and specialized microbenchmarks (§ 9.2.1). Next, we quantify the additional overhead of sorting the PAM and prefetching pages when resuming from interrupts (§9.2.2).

9.2.1 Instrumentation Overhead

Figure 6 overviews the average execution time slowdown of TLBlur, including the optimizations discussed in §6.4, on various benchmarks. We compare the performance overhead obtained from instrumenting the enclave with only compile-time instrumentation, binary instrumentation, and both. For completeness, Table 3 in Appendix E additionally includes detailed breakdowns of the different optimizations in the binary instrumentation. Overall, the slowdown factor of the optimized instrumentation ranges between $\times 1$ and $\times 5$, depending on the application. In the following, we discuss the performance observed in different benchmark programs.

Data-Access Benchmark. The `micro-stressmem` benchmark measures the overhead on bulky data page accesses by writing to a large memory buffer in a loop. The data-access dominates the performance overhead, while code page instrumentation has a negligible impact.

Control-Flow Benchmark. The `micro-long-loop` benchmark measures the overhead of control-flow instructions. Specifically, the program contains a tight loop that jumps across a page boundary on each iteration to represent a worst-case scenario. Without optimizations, the slowdown factor of the control-flow instrumentation is around $\times 54$. We inspect the reason for this behavior and conclude it is caused by the high-performance impact of the `pushf` and `popf` instructions when invoking the PAM update routine (§ 6.1). A practical way to reduce the impact of page-cross transitions is to set TLBlur to consider larger logical code pages of 16 KiB (§ 6.4). Indeed, this parameter removes most of the performance overhead, as loops are contained within a 16 KiB logical page, thus eliminating the need for instrumenting their jumps.

Libjpeg. Xu et al. [85] demonstrated the impact of page-table attacks on various real-world applications, including libjpeg. We evaluate the performance overhead of our defense on this application. With compile-time instrumentation alone, we observe a slowdown factor of $\times 2.83$. Likewise, when only applying binary instrumentation of code accesses, we find a slowdown of $\times 3.54$ without optimizations. When combining both approaches, the slowdown increases to $\times 7.56$.

By analyzing the execution traces and the instrumented binary, we observe that a performance-critical loop is repeatedly executed when decompressing an image. After instrumentation, such a loop is placed across a page boundary, thus inflating overhead due to code page-cross instrumentation. Similar to the `micro-long-loop` benchmark, using larger logical pages of 16 KiB eases the binary-instrumentation slowdown factor to a more reasonable $\times 1.22$. Another way to boost performance is moving the instrumentation to the targets of jumps. Specifically, the performance-critical loop in libjpeg contains a sequence of conditional jumps. By moving the binary instrumentation to the targets for these jumps, we reduce the number of invocations of the PAM update procedure (§6.1). This also eliminates the need to store the `rflags` register, thus further contributing to lowering the runtime overhead. After applying all of the above optimizations, the binary instrumentation overhead is reduced to $\times 1.03$, and the total slowdown for full mitigation, including compile-time and binary rewriting, decreases to $\times 3.07$.

yescrypt. yescrypt is a library based on Scrypt that implements a key-derivation function for password hashing. We observe a slowdown factor of $\times 1.10$ for the yescrypt function and $\times 1.97$ for the PBKDF2 function within yescrypt.

wolfSSL. We use wolfSSL to evaluate the PBKDF2 function and observe similar results when compared to the yescrypt implementation of PBKDF2. However, for wolfSSL, this overhead can primarily be attributed to the control-flow instrumentation, whereas the overhead in the yescrypt implementation mainly originated from memory-access instrumentation. We notice that wolfSSL executes fewer instruc-

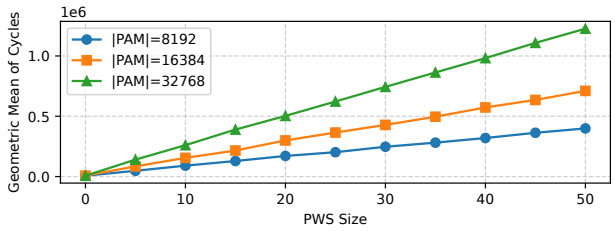


Figure 7: Mean number of cycles required to execute the TLBlur AEX handler, calculated over 100,000 iterations, for varying PWS and PAM size.

tions that access memory compared to yescrypt when using similar parameters. This shows that the exact overhead of our instrumentation can depend on implementation details. Furthermore, we evaluate the AES-CBC implementation of wolfSSL. Here, we find a high overhead caused by memory access instrumentation, resulting in a slowdown factor of $\times 4.83$. This is because wolfSSL does not support AES-NI for hardware-accelerated AES on Intel SGX, which causes it to fall back to an (insecure) software-based implementation. Such implementation uses a significant amount of memory accesses that are all instrumented individually.

OpenSSL. The OpenSSL library implements AES-GCM using AES-NI instructions to accelerate encryption and decryption, demonstrating how different implementation choices of the same algorithm affect performance. In the case of OpenSSL, the amount of instrumented memory accesses is significantly lower when compared to a software-based implementation, resulting in a slowdown factor of $\times 1.92$. Note that parts of the OpenSSL code that execute AES-NI instructions are handwritten in assembly. Since our data access instrumentation is only applied to code lowered by the LLVM backend, any data accesses within this assembly code will not be instrumented. Lastly, we benchmarked OpenSSL's RSA implementation, incurring a slowdown factor of $\times 3.89$.

9.2.2 Interrupt-Resumption Overhead

We conducted a microbenchmark to measure the overhead of sorting the PAM and prefetching N pages in the enclave exception handler, which is executed after every interrupt. Using only the existing AEX-Notify single-stepping mitigation (without enabling TLBlur), we measured an execution time of 13,000 cycles for the AEX handler, which aligns with performance numbers reported in previous work [16]. However, enabling TLBlur significantly increases this overhead, mainly due to the constant-time sorting of the PAM (which need not be executed atomically). For a large application like libjpeg, with 8,192 PAM entries and a PWS size of 30, the overhead rose to approximately 250,000 cycles. Different applications may require different PAM and PWS sizes.

Figure 7 shows that TLBlur's interrupt-resumption overhead scales linearly with both the PWS and PAM sizes. Since the number of PAM entries corresponds to the number of enclave pages, the overhead ultimately depends on the enclave's size and the configured PWS size. Assuming a benign OS interrupts the enclave once every 1 million cycles, an overhead of 250,000 cycles per interrupt would translate to a 25% performance overhead. However, actual interrupt rates vary based on the deployment environment, and the current implementation is not yet optimized. Particularly, we expect that developing faster constant-time sorting algorithms with vector extensions may further reduce the performance overhead.

9.2.3 Takeaway

Based on our review of the literature (§ 11), the work by Shinde et al. [61] aligns most closely to our objectives and approach. Both defenses are compiler-based and explicitly designed to address page-table attacks. From a design perspective, the main difference is that the work of Shinde et al. needs extensive manual code annotations, while TLBlur is application agnostic. To the best of our knowledge, there is no public implementation of Shinde et al.'s defense. Therefore, we interpolate the overhead from their publication's results. In terms of performance, Shinde et al. find an average slowdown of $\times 700$ [61, Table 2]. From our evaluation, when optimizations are in place, we measured an average slowdown of $\times 2.24$ from instrumentation. Assuming an additional prefetching overhead around 25%, the overall average slowdown would be $2.24 \cdot 1.25 = \times 2.80$. Furthermore, in contrast to the work of Shinde et al., we did not have to terminate any benchmarks early due to timeouts. This result positions TLBlur as a more practical approach for off-the-shelf Intel SGX platforms.

10 Discussion and Future Work

Limitations and Edge Cases. Our PAM implementation does not consider TLB set-associativity. As such, re-filling could lead to high pressure on certain sets of a set-associative TLB. Therefore, when the size of the PWS matches the maximum size of the TLB, which is computed as the number of sets multiplied by the number of ways, re-filling the TLB is likely to lead to evictions of previously prefetched entries. To mitigate this, we conservatively choose the size of PWS, and provide tools to evaluate the effectiveness of the defense. In the worst case, all accessed pages map to the same TLB set, reducing the effective value for the PWS size to the number ways in a set, *e.g.,* 8 (cf. Table 2 from Appendix B). Even if this unlikely case happens, there would still be a significant reduction in spatial granularity from 4 KiB to 32 KiB. For memory-intensive applications with a working set exceeding the PWS size, some data may still leak. However, we note

that this would still only be a subset of their leakage under the current AEX-Notify single-stepping mitigation, as illustrated in Figs. 9c and 9d of Appendix D.

Our current mitigation prototype only supports single-threaded enclave applications, the default for Intel SGX and prior attack targets. If needed, multithreading support could be implemented by using thread-local instead of global storage, such that each thread maintains its own PAM and fills its TLB individually. Shared variables would be logged in the PAM of each thread that accesses them.

Microarchitectural State Restore. We envision that TLBlur's protective design philosophy for the TLB could be extended to other microarchitectural components. Specifically, we propose the generic concept of replicating hidden microarchitectural state in software and seamlessly restoring microarchitectural buffers on (asynchronous) context switches between security domains. With a finer-grained software data structure, such transparent state restore defenses could also be applied to other microarchitectural components, *e.g.,* CPU caches. Furthermore, novel hardware-software co-design mechanisms could ease or entirely alleviate microarchitectural state tracking in software, *e.g.,* through custom update instructions or by transparently writing microarchitectural buffers to protected memory regions on context switch.

Alternative TEE Architectures. While TLBlur was instantiated on off-the-shelf Intel SGX, we believe its design is generalizable to other TEE technologies. The primary architectural requirement for TLBlur is *interrupt-awareness*, enabling TLB pre-filling through a custom exception handler.

Some academic TEEs [6,20] are inherently interrupt-aware, as they restore the interrupted enclave state from software. Confidential VM architectures, on the other hand, are typically not interrupt-aware. We consider protecting legacy VMs orthogonal to hardening security-sensitive enclave programs. However, recompiling at least user-space applications could be feasible in dedicated, security-sensitive deployments or custom VM runtimes like Enarx or Mushroom. Notably, AMD SEV exposes untrusted page tables, which have been exploited in controlled-channel attacks [29, 50, 79, 82] and related frameworks like SEV-Step [83]. Implementing TLBlur on AMD SEV would require a firmware update to make SEV VMs interrupt aware, akin to Intel SGX's AEX-Notify, along with recompiling the user-space application. Interestingly, Intel's TDX confidential VM architecture does not directly expose extended page tables, but includes an untrusted `TDH.MEM.RANGE.BLOCK` VMM call that has been exploited to successfully mount controlled-channel attacks [2]. This has led to explicit calls for a similar "TDExit-Notify" feature [62], which would similarly enable exploring TLBlur on TDX.

SGX prevents reading encrypted memory, making ciphertext side channels [43] ineffective for observing PAM updates. In contrast, other TEEs like AMD SEV may be vulnerable to such attacks and would require orthogonal architectural [19, 21] or compiler [81] mitigations.

Enclave Security Considerations. SGX enclaves are security containers that host dedicated portions of code. Only the cost of calling an enclave and returning is 8,000 cycles, which is an order of magnitude above regular system calls [54]. This indicates developers using SGX already accept to sacrifice performance for security. As a case in point, due to continuous mitigation addressing microarchitectural attack vectors, SGX enclaves have already experienced significant performance slowdowns, *e.g.,* Intel's official compiler-based mitigation for LVI attacks introduces slowdown factors of $\times 2$ to $\times 9$ [38,69]. In comparison, the overhead introduced by TLBlur is much less impactful than similar defenses for controlled-channel attacks [61]. Even more important, TLBlur is more practical than fully homomorphic encryption approaches, which are the only alternative to TEEs for untrusted cloud environments [45].

Further Performance Optimizations. Certain code patterns, such as performance-critical loops crossing page boundaries, can cause significant slowdowns. We discussed optimizations that group code pages to reduce the number of instrumentations in §6.4. A more principled approach may use a performance profile to optimally place basic blocks on code pages. Other potential optimizations involve copying memory in loops, where we could move the instrumentation outside the loop and trace all pages in one step. However, identifying bulky data transfers is challenging and requires dedicated software-analysis efforts.

11 Related Work

Table 1 comprehensively compares TLBlur to prior defenses, where we computed the minimum and maximum overheads for TLBlur based on the instrumentation slowdown factors from the real-world applications in Fig. 6, with an additional 25% interrupt-resumption overhead applied.

Interrupt Detection. Initial approaches [13, 52, 60] detect suspicious AEX rates that may be caused by controlled-channel attacks. As these solutions were devised before the AEX-Notify hardware extension was available, they use other mechanisms to detect AEXs, such as Intel Transactional Synchronization Extensions (TSX) [13, 60], which yields considerable performance impacts and is deprecated on newer processors, or by imperfectly monitoring SSA memory writes [52]. More fundamentally, such heuristic approaches are inherently fragile, suffering from both false positives and false negatives [30, 37], and may ultimately be circumvented by more stealthy page-table-based attacks [74, 76].

Table 1: Overview of prior controlled-channel defenses.

	Defense	Scope	Proactive	Compat	Auto	Resol	Overhead % min	Overhead % max
Detection	T-SGX [60]	◐	○	◐	●	–	10	90
	Deja-Vu [13]	●	○	◐	●	–	1	4
	Varys [52]	●	○	●	●	64 B	5	40
Random	InvisiPage [1]	◐	◐	○	●	4 KiB	16	1,800
	Klotski [87]	◐	●	●	●	4 KiB	25	1,022
	Dr.SGX [10]	●	◐	●	●	64 B	500	1,100
	Obelix [80]	●	●	●	●	64 B	16,200	10,290,900
Pinning	Heisenberg [64]	◐	●	◐	○	4 KiB	300	3,000
	Autarky [53]	◐	◐	○	○	4 KiB	18	25
	SG^{XL} [86]	◐	◐	○	○	2 MiB	–	–
	AEX-Notify [16]	○	◐	●	●	–	–	–
Compile	Pigeonhole [61]	◐	◐	●	○	4 KiB	29	400,000
	TLBlur (this paper)	◐	●	●	●	4 KiB	28	504

Scope: covers a subset (○), only (◐), or a superset (●) of controlled channels; *Proactive:* fully (●) or partially (◐) prevents leakage of code and data accesses; *Compat:* off-the-shelf Intel SGX (●), non-standard TSX (◐), hypothetical hardware (○); *Auto:* manual developer intervention (○); *Resol:* spatial resolution at which accesses are hidden, if applicable (page = 4 KiB; cacheline = 64 B); *Overhead:* as reported in the respective papers.

Randomization. An orthogonal line of work developed randomization-based defenses, such as customized oblivious RAM solutions, to probabilistically hide any secret-dependent enclave memory accesses at the granularity of pages [1, 87] or cache lines [10, 80]. While such obfuscations can be generic and largely invisible to the enclave application, they commonly incur prohibitive performance overheads, reaching up to $\times 10^5$. Additionally, certain solutions [10] solely target data accesses without obscuring prevalent conditional code accesses. More complete solutions [1], require invasive hardware changes, *e.g.,* enclave-private page tables [20, 23], that are unavailable on off-the-shelf Intel SGX platforms. Furthermore, recent research [56] has demonstrated that subtle leakages may persist even when implementing demand-paging with an oblivious page access module.

Page Pinning. Defenses conceptually closest to our work aim to "pin" certain enclave pages to make them oblivious to the adversary. Similar to TLBlur and AEX-Notify [16], Heisenberg [64] prefetches sensitive pages into the TLB when resuming from an interrupt. However, Heisenberg requires hypothetical hardware extensions or using Intel TSX, which is scarcely available and induces exuberant performance overheads up to $\times 30$. Alternatively, Autarky [53] proposed theoretical SGX hardware extensions to hide enclave page-fault addresses, forcing the untrusted operating system to cooperate with the enclave to evict or restore pages. However, Autarky implies extensive alterations in both hardware, operating system components, and enclave software. Furthermore, since page tables still reside in untrusted memory, Autarky does not safeguard against stealthy attacks taregtting the cache status of PTEs [74]. Lastly, SG^{XL} [86] proposed to change Intel SGX's EPCM hardware to securely support large 2 MiB pages. While this would in itself not eliminate the page-fault channel, it would considerably limit the spatial resolution with negligible overheads. However, SG^{XL} requires intrusive hardware changes unsupported on current off-the-shelf SGX platforms, and its temporal protection remains unclear.

More fundamentally, from a usability perspective, none of these pinning solutions have addressed the key question of *how to determine the subset of enclave pages that need to be pinned.* Prior solutions simply assume that the enclave is small enough to be entirely pinned [64], or by relying on impractical and error-prone developer annotations [53, 86].

Compile-Time Rewriting. Intel's official SGX developer's guide advices to "aligning specific code and data blocks to exist entirely within a single page" [31]. However, they do not specify a concrete procedure and do not consider larger programs with secret-dependent patterns spanning beyond a single page. Shinde et al. [61] devised a software-based compiler scheme that achieves page-fault-oblivious execution by iteratively copying secret-dependent code and data to a single staging area. However, their approach introduces prohibitive slowdowns, exeeding $\times 4000$ for real-world cryptographic libraries. Additionally, it imposes severe restrictions on input programs, requiring developers to *(i)* use a restrictive subset of C; *(ii)* ensure a manually balanced execution tree devoid of secret-dependent loops; *(iii)* manually annotate secrets; and *(iv)* engage in extensive manual optimizations. Finally, focussing solely on page-fault obliviousness, the impact of more powerful interrupt-driven page-table-based attacks [49, 72, 74] on this prior defense remains unclear.

Temporal Resolution. While the previous solutions mainly targeted the spatial dimension of controlled-channel attacks, the original AEX-Notify hardware-software co-design [16] was explicitly designed to limit the *temporal* resolution of single-stepping attacks [72]. This paper is the first to show that the hardware extension provided by AEX-Notify can tackle the *spatial* bandwidth of controlled-channel attacks.

12 Conclusion

TLBlur is a practical and effective software defense that counteracts controlled-channel attacks on Intel SGX platforms using the AEX-Notify feature. TLBlur mitigates such attacks by dynamically tracking application page accesses through transparent compiler instrumentation and seamlessly prefetching them into the hardware TLB upon interrupt resumption. Our results show an acceptable performance overhead ($\times 2.80$ on average), making TLBlur a viable option for securing enclaves against controlled-channel attacks on existing hardware.

13 Ethics Considerations and Open Science

Ethics Considerations. This is core defensive research, aiming to harden applications for a widely deployed real-world security technology (*i.e.,* Intel SGX). As part of this research, we also developed improved attack techniques and tooling (cf. page-access profiler in §7.2). We firmly believe that the benefits of better understanding and being able to explore and quantify remaining side-channel leakage in SGX applications outweighs any harms of making these tools openly available. This is also recognized by the community at large, *e.g.,* the open-source SGX-Step [72] framework on which we based our tooling was recognized with a long-term ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award [71]. All of the case-study applications we used to evaluate our defense were already attacked in prior works [61,74,76,85], thus we found no new, unknown vulnerabilities that would need responsible disclosure.

Open Science. All of our code, including the compiler, runtime, profiler, and case studies, is available on Zenodo [1] and GitHub [2]. We hope that a high-quality open-source release of our mitigation prototype can help foster further research or even influence real-world applications in the industry.

Acknowledgements. This research is partially funded by the Research Fund KU Leuven, the Research Foundation – Flanders (FWO) via grant #1261222N, by the Cybersecurity Research Program Flanders, SNSF PCEGP2_186974, ERC #850868 CodeSan, and by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

References

[1] Shaizeen Aga and Satish Narayanasamy. Invisipage: oblivious demand paging for secure enclaves. In *ISCA*, 2019.

[2] Erdem Aktas, Cfir Cohen, Josh Eads, James Forshaw, and Felix Wilhelm. Intel trust domain extensions (TDX) security review, 2023.

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

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

[5] Fritz Alder, Lesly-Ann Daniel, David Oswald, Frank Piessens, and Jo Van Bulck. Pandora: Principled symbolic validation of Intel SGX enclave runtimes. In *S&P*, 2024.

[6] Fritz Alder, Jo Van Bulck, Frank Piessens, and Jan Tobias Mühlberg. Aion: Enabling open systems through strong availability guarantees for enclaves. In *CCS*, 2021.

[7] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP*, 2013.

[8] Scott Bair. Intel Labs contributes key technologies to new Intel Core Ultra and Intel Xeon Scalable Processors. https://community.intel.com/t5/Blogs/Tech-Innovation/Client/Intel-Labs-Contributes-Key-Technologies-to-New-Intel-Core-Ultra/post/1553289, 2023.

[1] https://doi.org/10.5281/zenodo.14650953

[2] https://github.com/TLBlur-SGX

[9] 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.

[10] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. Dr. SGX: automated and adjustable side-channel protection for SGX using data location randomization. In *ACSAC*, 2019.

[11] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre attacks: Stealing Intel secrets from SGX enclaves via speculative execution. In *Euro S&P*, 2019.

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

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

[14] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *ACM Computing Surveys*, 2023.

[15] Tobias Cloosters, Michael Rodler, and Lucas Davi. TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *USENIX Security 20*, 2020.

[16] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting precise single-stepping attacks through interrupt awareness for intel SGX enclaves. In *USENIX Security*, 2023.

[17] Bart Coppens, Ingrid Verbauwhede, Koen De Bosschere, and Bjorn De Sutter. Practical mitigations for timing-based side-channel attacks on modern x86 processors. In *S&P*, 2009.

[18] Intel Corporation. Asynchronous enclave exit notify and the EDECCSSA user leaf function, 2022.

[19] V. Costan and S. Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016(086), 2016.

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

[21] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In *S&P*, 2025.

[22] Ghada Dessouky, Tommaso Frassetto, and Ahmad-Reza Sadeghi. HybCache: Hybrid side-channel-resilient caches for trusted execution environments. In *USENIX Security*, 2020.

[23] 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.

[24] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. Security vulnerabilities of SGX and countermeasures: A survey. *ACM Computing Surveys (CSUR)*, 54(6), 2021.

[25] Qian Ge, Yuval Yarom, Tom Chothia, and Gernot Heiser. Time protection: The missing os abstraction. In *EuroSys*, 2019.

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

[27] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *USENIX Security*, 2017.

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

[29] Felicitas Hetzelt and Robert Buhren. Security analysis of encrypted virtual machines. *ACM SIGPLAN Notices*, 52(7), 2017.

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

[31] Intel. *Intel Software Guard Extensions Developer Guide: Protection from Side-Channel Attacks*, 2017.

[32] Intel. Retpoline: A branch target injection mitigation. White Paper, 2018. Reference no. 337131-001.

[33] Intel. *Intel 64 and IA-32 Architectures Optimization Reference Manual*, 2020. Order no. 248966-040.

[34] Intel. *Intel 64 and IA-32 Architectures Software Developer's Manual – Combined volumes*, 2020. Reference no. 325462-062US.

[35] Intel. Intel SGX attestation technical details. https://www.intel.com/content/www/us/en/security-center/technical-details/sgx-attestation-technical-details.html, 2024. Online. Last accessed April, 28 2024.

[36] 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.

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

[38] Michael Larabel. The brutal performance impact from mitigating the LVI vulnerability, 2020.

[39] Chris Lattner and Vikram S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO*, 2004.

[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] Abraham Lempel and Jacob Ziv. On the complexity of finite sequences. *IEEE Transactions on information theory*, 22(1), 1976.

[43] 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.

[44] 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.

[45] Paulo Martins, Leonel Sousa, and Artur Mariano. A survey on fully homomorphic encryption: An engineering perspective. *CSUR*, 2017.

[46] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

[47] Daniel Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In *Cryptographers' Track at the RSA Conference*, 2018.

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

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

[50] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's virtual machine encryption. In *EuroSec*, 2018.

[51] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. A survey of published attacks on Intel SGX. *arXiv preprint arXiv:2006.13598*, 2020.

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

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

[54] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for SGX enclaves. In *EuroSys*, 2017.

[55] Maksim Panchenko, Rafael Auler, Bill Nell, and Guilherme Ottoni. BOLT: A practical binary optimizer for data centers and beyond. *CoRR*, abs/1807.06735, 2018.

[56] Nirjhar Roy, Nikhil Bansal, Gourav Takhar, Nikhil Mittal, and Pramod Subramanyan. When oblivious is not: Attacks against OPAM. In *WOOT*, 2020.

[57] Mark Russinovich. Confidential computing: Elevating cloud security and privacy. *Communications of the ACM*, 67(1), 2023.

[58] 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.

[59] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

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

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

[62] Pradyumna Shome. Closing the Intel TDX page fault side channel, or, the case for TDExit-Notify. https://collective.flashbots.net/t/closing-the-intel-tdx-page-fault-side-channel-or-the-case-for-tdexit-notify/3775/1, 2024.

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

[64] Raoul Strackx and Frank Piessens. The Heisenberg defense: Proactively defending SGX enclaves against page-table-based side-channel attacks. *CoRR*, abs/1712.08519, 2017.

[65] Pramod Subramanyan, Rohit Sinha, Ilia Lebedev, Srinivas Devadas, and Sanjit A Seshia. A formal foundation for secure remote execution of enclaves. In *CCS*, 2017.

[66] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal war in memory. In *S&P*, 2013.

[67] Andrei Tatar, Daniël Trujillo, Cristiano Giuffrida, and Herbert Bos. TLB;DR: Enhancing TLB-based attacks with TLB desynchronized reverse engineering. In *USENIX Security 22*, 2022.

[68] 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.

[69] 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.

[70] 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.

[71] Jo Van Bulck and Frank Piessens. SGX-Step: An open-source framework for precise dissection and practical exploitation of Intel SGX enclaves. In *ACSAC 2023 Cybersecurity Artifacts Competition and Impact Award Finalist Short Paper*, 2023.

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

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

[74] 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.

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

[76] 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.

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

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

[79] 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.

[80] J. Wichelmann, A. Rabich, A. Pätschke, and T. Eisenbarth. Obelix: Mitigating side-channels through dynamic obfuscation. In *S&P*, 2024.

[81] Jan Wichelmann, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. Cipherfix: Mitigating ciphertext side-channel attacks in software. In *USENIX Security*, 2023.

[82] L. Wilke, J. Wichelmann, M. Morbitzer, and T. Eisenbarth. SEVurity: no security without integrity – breaking integrity-free memory encryption with minimal assumptions. In *S&P*, 2020.

[83] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step: A single-stepping framework for AMD-SEV. *TCHES*, 2023.

[84] Hans Winderix, Jan Tobias Mühlberg, and Frank Piessens. Compiler-assisted hardening of embedded software against interrupt latency side-channel attacks. In *EuroS&P*, 2021.

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

[86] Sujay Yadalam, Vinod Ganapathy, and Arkaprava Basu. SGXL: Security and performance for enclaves using large pages. *TACO*, 2020.

[87] Pan Zhang, Chengyu Song, Heng Yin, Deqing Zou, Elaine Shi, and Hai Jin. Klotski: Efficient obfuscated execution against controlled-channel attacks. In *ASPLOS*, 2020.

A Page Access Map Update Assembly Code

```
    # TLBlur page-access map update,
    # called by instrumentation
    #   args: %rdi = accessed address
    #         %rax = scratch
    __tlblur_pam_size = 0x1000000
    __tlblur_pam_mask = 0x0fffff8
    .global tlblur_pam_update
tlblur_pam_update:
    # Compute index in PAM
    # %rdi  = (((%rdi - <enclave_base>) >> 12) *8)
    # %rdi &= (<pam_size> - 1)
    lea     __ImageBase(%rip), %rax
    sub     %rax, %rdi
    shr     $12, %rdi
    shl     $3, %rdi
    and     $__tlblur_pam_mask, %rdi

    # Add PAM base address
    leaq    __tlblur_pam(%rip), %rax
    add     %rax, %rdi

    # Increment global counter in %rax
    movq    __tlblur_counter(%rip), %rax
    incq    __tlblur_counter(%rip)

    # Update the PAM
    movq    %rax, (%rdi)
    ret

    .data
    .align 0x1000   # 4KiB
    .global __tlblur_pam
__tlblur_pam:
    .zero   __tlblur_pam_size

    .global __tlblur_counter
__tlblur_counter:
    .quad   1
```

B Hardware TLB Details

Table 2 provides the full TLB details of the Intel Xeon Silver 4410Y evaluation platform, as reported by `cpuid` and documented in the Intel optimization manual [33]. It contains separate L1 TLBs for code and data (separated into loads and stores), plus a unified shared L2 TLB for code and data.

Table 2: TLB details of the Intel Xeon Silver 4410Y.

	Page Size Support						
Type	**4KB**	**2MB**	**4MB**	**1GB**	**Ways**	**Sets**	**Tot**
L1 iTLB	✓	✗	✗	✗	8	32	256
L1 iTLB	✗	✓	✓	✗	8	4	32
L1 dTLB (store)	✓	✓	✓	✓	16	1	16
L1 dTLB (load)	✓	✗	✗	✗	4	16	64
L1 dTLB (load)	✗	✓	✓	✗	4	8	32
L1 dTLB (load)	✗	✗	✗	✓	8	1	8
L2 sTLB	✓	✓	✓	✗	8	128	1024
L2 sTLB	✓	✗	✗	✗	8	128	1024

C Libjpeg Examples

See Fig. 8 for additional examples of images reconstructed by our reimplementation of the original controlled-channel

Table 3: Slowdown factor for benchmarks with different instrumentation parameters. The left side of the table shows results for binary instrumentation of direct control flow under various optimizations. The "data + Indirect Control-Flow (ICF)" column lists the results when applying only compile-time instrumentation. The right side displays the full mitigation, combining compile-time instrumentation and binary rewriting. The last row reports the geometric mean slowdown across all benchmarks.

Benchmark	direct control-flow				data+ICF	control-flow and data			
	unoptimized	16KiB pages	jump targets	both		unoptimized	16KiB pages	jump targets	both
micro-stressmem	1.00	1.00	1.01	1.00	5.49	5.55	5.46	5.42	5.44
micro-fibonacci	9.69	9.82	1.24	1.24	1.02	9.71	9.81	1.39	2.65
micro-long-loop	54.35	0.90	12.09	0.80	1.09	54.31	2.33	12.09	0.79
libjpeg-birds	3.54	1.22	1.09	1.03	2.83	7.56	3.93	3.31	3.07
yescrypt-yescrypt	1.15	1.17	1.00	1.00	1.10	1.28	1.25	1.14	1.10
yescrypt-pbkdf2	1.43	1.41	1.20	1.21	1.80	2.53	2.12	2.04	1.97
wolfssl-pbkdf2	1.69	1.21	1.22	1.00	1.10	2.18	1.44	1.52	1.16
wolfssl-aes-cbc	1.35	1.04	1.11	1.07	4.56	5.30	5.21	4.87	4.83
openssl-aes-gcm	6.58	4.99	1.43	1.36	1.44	8.48	5.22	2.07	1.92
openssl-rsa	10.47	6.94	1.54	1.62	3.04	16.17	9.83	4.05	3.89
geometric mean	3.70	1.94	1.50	1.11	1.94	6.39	3.72	2.91	2.24

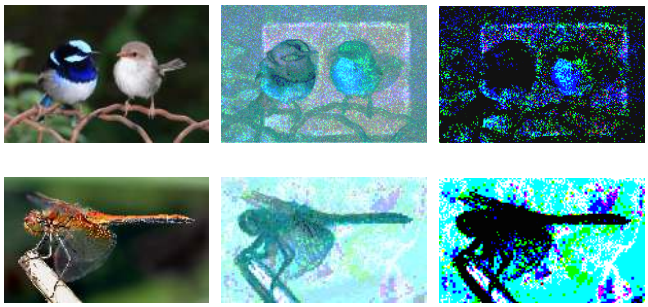


Figure 8: Example images (left) recovered via libjpeg page faults (middle). TLBlur fully eliminates leakage vs. insufficient protection from existing AEX-Notify mitigation (right).

attack on libjpeg by Xu et al. [85]. Although the existing AEX-Notify single-stepping mitigation [16] may somewhat hinder image reconstruction, by unconditionally prefetching stack pages, it is clearly insufficient to fully prevent leakage.

D Detailed Security Effectiveness Evaluation

Figures 9 and 10 show the traces of libjpeg obtained with different TLBlur prefetch sizes. The original page-access trace in Figure 9a clearly shows a repeating page-access pattern that can be used to identify the start of the decompression of each 8×8 block of the input image, as well as the start of each row of blocks. This information, combined with the number of accesses to stack and heap, provides significant information to deduce the input image. We zoom into a single 8×8 block decompression step for each figure to better illustrate this.

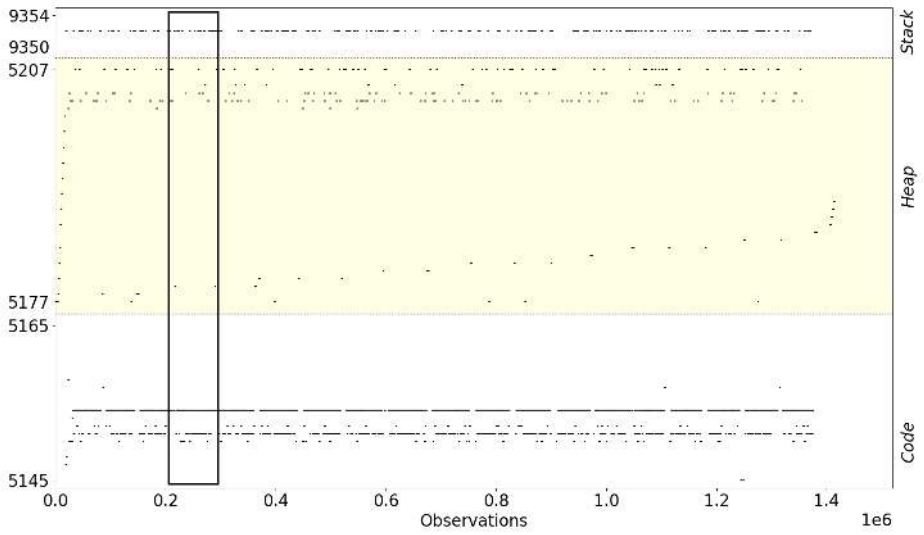
When using the AEX-Notify single-stepping defense [16], the temporal resolution decreases significantly, as shown in Fig. 9b. However, the same page-access patterns that can be used to reconstruct images remain observable.

Already after applying TLBlur with a PAM of limited size 20 (Figure 9d), we observe that individual calls to the 8×8 block decompression function become indistinguishable, although an adversary is still able to determine the start of each row. When increasing the size of the PAM further, any remaining control-flow patterns during decompression become obscured. A similar effect occurs on the data pages, making it infeasible for adversaries to reconstruct images through page table leakage using state-of-the-art techniques.

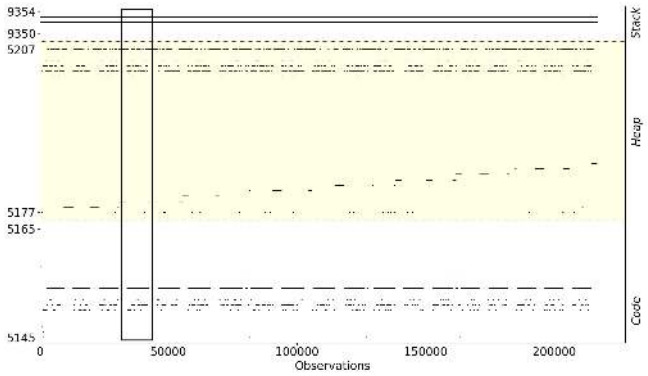
The bottom heap pages of the traces in Fig. 9 are data pages accessed by `memcpy` between decompression operations. These accesses still leak with higher PAM sizes, as we have chosen to not instrument standard library functions to reduce the performance overhead. For this application, such standard library functions are not used in a secret-dependent way, as copying segments of an image leaks no information about the image besides its size. However, we note that it is possible to instrument these functions for applications that would otherwise leak secret information.

E Detailed Performance Benchmarks

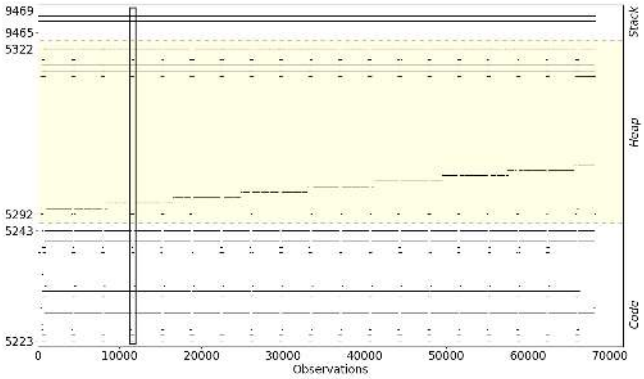
Table 3 provides detailed performance results of the benchmarks in §9.2.1.



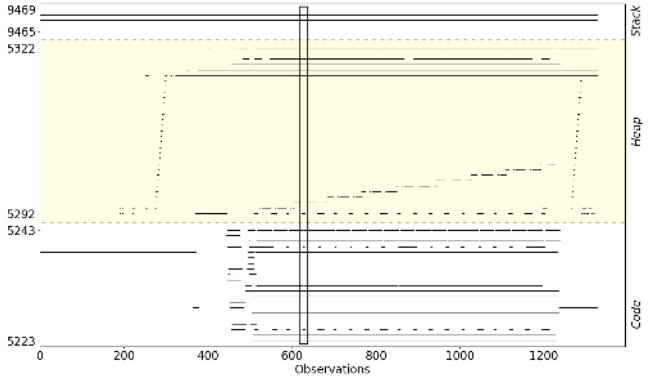
(a) Maximal page-access trace for single-stepping attacker; Lempel-Ziv: 10,313,182.



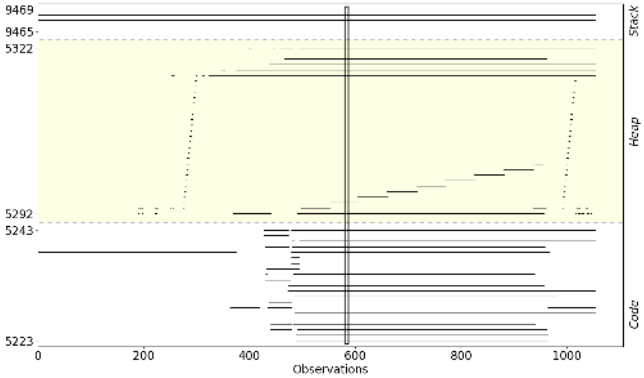
(b) AEX-Notify single-stepping mitigation; Lempel-Ziv: 6,150,926.



(c) TLBlur with PWS of size 10; Lempel-Ziv: 3,536,653.

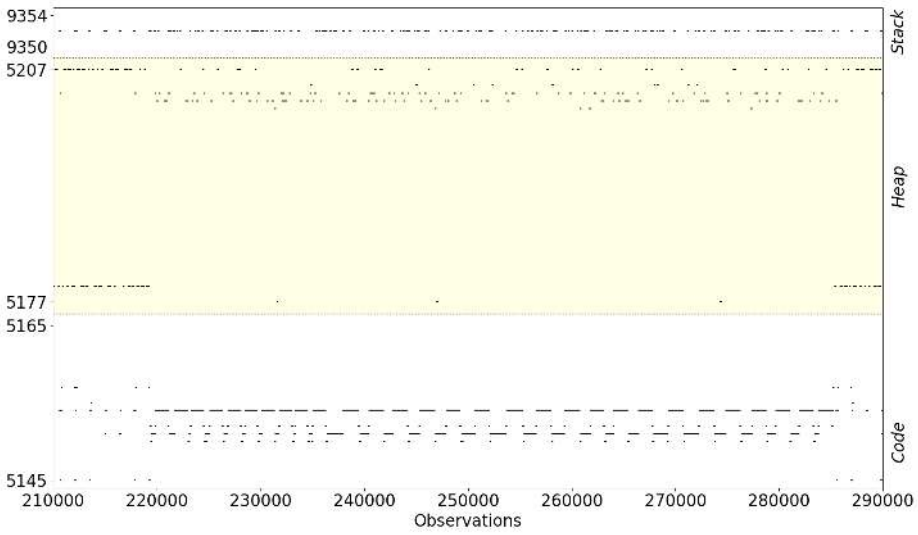


(d) TLBlur with PWS of size 20; Lempel-Ziv: 472,775.

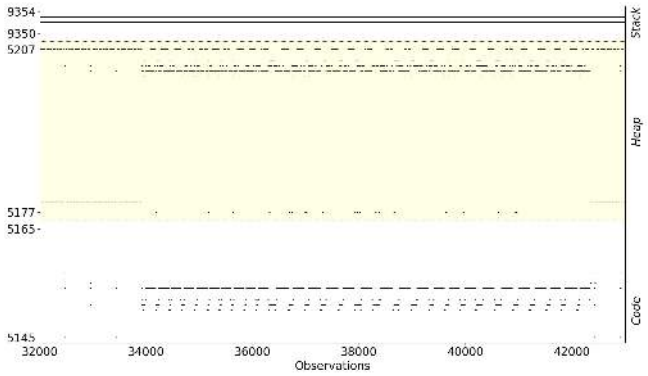


(e) TLBlur with PWS of size 30; Lempel-Ziv: 414,905.

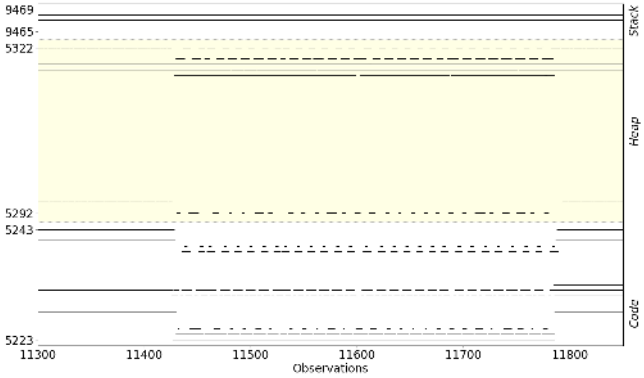
Figure 9: Full page-access trace from our profiler during image decompression with libjpeg (x-axis: attacker observation points; y-axis: page numbers). The existing AEX-Notify single-stepping mitigation merely reduces temporal resolution (x-axis range) and minimally obscures spatial accesses to stack pages. As the PWS parameter increases, TLBlur principally reduces both temporal and spatial dimensions: less page faults on the x-axis and "blurred" access patterns. The highlighted rectangles show a single, secret-dependent libjpeg iteration, with a zoomed trace in Fig. 10. We include Lempel-Ziv complexity numbers [42] to quantify the amount of information that can be extracted from these traces.



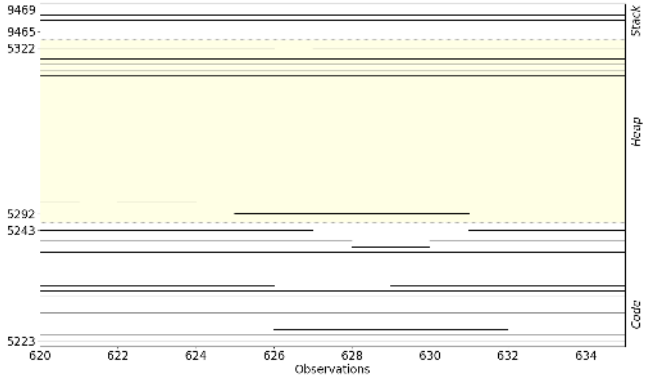
(a) Maximal page-access trace for single-stepping attacker; Lempel-Ziv: 2,416,952.



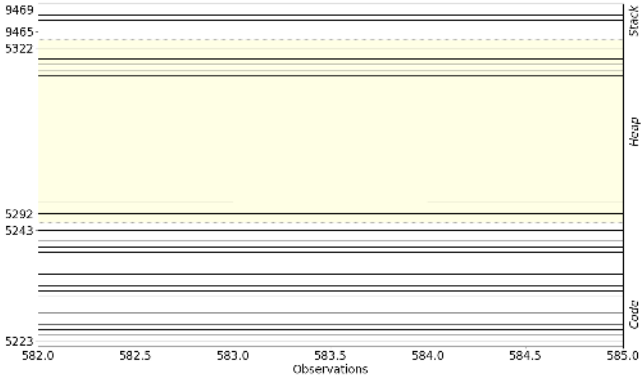
(b) AEX-Notify single-stepping mitigation; Lempel-Ziv: 1,369,130.



(c) TLBlur with PWS of size 10; Lempel-Ziv: 308,035.



(d) TLBlur with PWS of size 20; Lempel-Ziv: 48,054.



(e) TLBlur with PWS of size 30; Lempel-Ziv: 19,212.

Figure 10: Detailed page-access trace for a single, secret-dependent libjpeg iteration, zoomed-in from Fig. 9 (x-axis: attacker observation points; y-axis: page numbers). The existing AEX-Notify single-stepping mitigation merely reduces temporal resolution (x-axis range) and minimally obscures spatial accesses to stack pages. As the PWS parameter increases, TLBlur principally reduces both temporal and spatial dimensions: less page faults on the x-axis and "blurred" access patterns, effectively hiding the secret-dependent page-access patterns successfully exploited in previous works. We include Lempel-Ziv complexity numbers [42] to quantify the amount of information that can be extracted from these traces.

F Artifact Appendix

F.1 Abstract

We provide the automated mitigation pipeline, including LLVM and BOLT instrumentation passes, a modified Intel SGX SDK that includes our custom prefetcher, as well as the benchmark applications to evaluate performance overhead. To assess leakage in practice, we furthermore provide an independent reproduction of the controlled-channel attack on libjpeg and a profiler tool to seamlessly extract page-access traces from victim enclaves.

F.2 Description & Requirements

F.2.1 Security, privacy, and ethical concerns

This artifact includes attacks on real-world Intel SGX processors, which require the installation of the privileged SGX-Step kernel module. We recommend that evaluators run all code on an isolated test machine.

The attack and mitigation code in this artifact is intended solely for reproducing our results. Any use of these results should be conducted responsibly. The mitigation is a research prototype, and we do not recommend its deployment in production environments as-is.

F.2.2 How to access

The artifact files are available on Zenodo (https://doi.org/10.5281/zenodo.15194120) and GitHub (https://github.com/TLBlur-SGX/tlblur/tree/usenix-artifact).

F.2.3 Hardware dependencies

Building and running TLBlur requires hardware with support for Intel SGX2 and AEX-Notify.

F.2.4 Software dependencies

- A Linux distribution supported by the Intel SGX SDK. We recommend using Ubuntu 22.04.
- C/C++ compiler (`gcc` or `clang`) and linker (`lld` strongly recommended)
- Build tools: `make`, `cmake`, `meson` and `ninja`.
- Stable Rust toolchain (1.76 or later)

F.2.5 Benchmarks

No external benchmarks were used for our evaluation.

F.3 Set-up

F.3.1 Installation

Download artifact. Clone the repository with all submodules.

```
$ git clone --recurse-submodules \
    --branch usenix-artifact \
    https://github.com/TLBlur-SGX/tlblur.git
```

Build. Run `build.sh` to build and install LLVM with TLBlur instrumentation passes, the Intel SGX SDK with TLBlur prefetcher, SGX-Step, benchmark enclaves and tools.

Load SGX-Step. Load the SGX-Step kernel module by running `make clean load` in `sgx-step/kernel`.

F.3.2 Basic Test

1. Run the test program with an enclave binary, e.g. `./install/bin/test-rsa ./install/lib/s-encl-rsa-instrumented-relocs-bolt-opt.so`. Output should contain the following, indicating that the attack succeeded:

```
[../enclaves/rsa/main.c] secure enclave
    encrypted '1234' to '21921'; decrypted '
    1234'
[../enclaves/rsa/main.c] --> RECONSTRUCTED KEY
     '20771' (0x5123)
```

F.4 Evaluation workflow

F.4.1 Major Claims

(C1): A practical profiler (Section 7.2) to extract enclave page-access traces (Figs. 5, 9 and 10).
(C2): A reproduction of the seminal controlled-channel attack by Xu et al. on libjpeg, and demonstrate that it still leaks with AEX-Notify single-stepping defense (Figs. 1 and 8).
(C3): An automated compiler pipeline (Section 6) that reduces information leakage observable to controlled-channel adversaries by "blurring" page-access traces (Section 9.1).
(C4): An average slowdown of $\times 2.24$ from instrumentation across a variety of benchmark programs (Sections 9.2.1 and 9.2.3, Fig. 6, and Table 3).
(C5): An interrupt-resumption overhead that scales linearly with the size of the PWS and the size of the enclave (Section 9.2.2, Fig. 7).

F.4.2 Experiments

(E1): *[Profiler] [10 human-minutes + 10 compute-minutes]: extract a page-access trace from libjpeg.*

Preparation: Run `cd sgx-step/app/libjpeg`.
Execution: Run the profiler as follows:

```
$ sudo ../profiler/target/release/sgx_tracer \
    --so ./profiler-libjpeg.so \
    -e ./Enclave/encl.so \
    --output trace.vcd \
    --args img/birds.jpg 10000000 10000000
```

Collecting the full trace can take up to several hours. Interrupt the program after a few minutes.
Results: The partial trace in the `trace.vcd` file can be opened in a VCD viewer like GTKWave. This demonstrates the practicality of using the profiler to extract page-access traces and, thus, validates C1.

(E2): *[Attack on libjpeg] [10 human-minutes + 3 compute-hours]: reconstruct libjpeg image with page-fault attack.*
Preparation: Run `cd sgx-step/app/libjpeg/attack`.
Execution: Run the attack as follows:

```
$ cargo run --release -- -o reconstruct.bmp \
    -i ../img/Wapiti_from_Wagon_Trails.jpg \
    --color enclave \
    -e ../Enclave/encl.so
```

The attack can take several hours to complete, but progress is indicated by the progress bar.
Results: The resulting `reconstruct.bmp` file should contain a reconstruction of the original image, as in Fig. 1, validating C2.

(E3): *[Prefetching simulation] [10 human-minutes + 30 compute-minutes]: use the instrumented binary to evaluate the effectiveness of TLBlur.*
Preparation:

```
$ export TLBLUR_LIB="$PWD/install/lib"
$ cd sgx-step/app/libjpeg
```

Execution: Run the profiler as follows:

```
$ sudo \
  ../profiler/target/release/sgx_tlblur_sim \
  --so $TLBLUR_LIB/libprof-libjpeg.so \
  -e $TLBLUR_LIB/s-encl-libjpeg-instrumented-
      relocs-bolt.so \
  --output trace_30.vcd \
  --args img/birds.jpg 10000000 10000000 \
  --pws-size 30 --irq-pat page-fault \
  --hw-tlb set-associative \
  --ways 8 --sets 1024
```

This will run the profiler with the instrumented libjpeg binary and a PWS size of 30, simulating the effect of TLBlur with a set-associative hardware TLB with 8 ways and 1024 sets. Collecting the full trace can take up to several hours. Interrupt the program after around 30 minutes have passed to collect sufficiently long traces.
Results: The partial trace in the `trace_30.vcd` file can be opened in a VCD viewer like GTKWave. Compared to the trace from E1, page-access patterns are "blurred", as in Fig. 10, validating C3. Note that the binary instrumentation pass moves code to higher addresses, hence the code pages start at page 5120 instead.

(E4): *[Instrumentation benchmarks] [10 human-minutes + 2 compute-hours]: run performance benchmarks to measure instrumentation overhead.*
Preparation: Ensure that all benchmark enclave binaries are installed in `install/lib`. Each benchmark program and combination of optimizations shown in Table 3 has a corresponding `.so` file in this directory. Refer to the table in the README for the naming scheme of enclave files. Change to the `evaluation` directory.
Execution: To run the benchmarks, execute the `run_benchmarks.sh` script. This may take several hours to complete.
Results: The results can be found in the `out` directory. Run `./plot.py` to reproduce Fig. 6. Set `FULL_PLOT = True` in `plot.py` to reproduce Table 3, validating C4.

(E5): *[Interrupt-resumption benchmark] [10 human-minutes + 5 compute-minutes]: benchmark the interrupt-resumption overhead.*
Preparation: Run the following to prepare the microbenchmark:

```
$ cd prefetch-benchmark
$ source /opt/intel/sgxsdk/environment
$ make clean all
```

Configure the enclave size in `Enclave/encl.config.xml`.
Execution: The `run.sh` script executes the microbenchmark for varying PWS sizes, and a fixed PAM size based on the configured enclave size.
Results: The results can be found in the `out` directory. Repeat the experiment with different enclave sizes to obtain results for varying PAM sizes. Run `./plot.py` to reproduce Fig. 7, showing linearly scaling overhead in both PWS and PAM size and, thus, validating C5.

F.5 Notes on Reusability

F.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2025/.