SECURING INTEL SGX AGAINST SIDE-CHANNEL ATTACKS VIA LOAD-TIME SYNTHESIS

A Dissertation
Presented to
The Academic Faculty

By

Ming-Wei Shih

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Georgia Institute of Technology

Georgia Institute of Technology

December 2019

SECURING INTEL SGX AGAINST SIDE-CHANNEL ATTACKS VIA LOAD-TIME SYNTHESIS

Approved by:

Dr. Taesoo Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Wenke Lee
School of Computer Science
Georgia Institute of Technology

Dr. Marcus Peinado
Cloud and Infrastructure Security Group
Microsoft Research

Dr. Michael Steiner
Security and Privacy Research Group
Intel Labs

Dr. Brendan Saltaformaggio
School of Computer Science
Georgia Institute of Technology

Date Approved: Octobor 23, 2019

To my dear wife, Tiffany, and my family.

ACKNOWLEDGEMENTS

Foremost, I would like to express my sincere gratitude to my advisor Dr. Taesoo Kim for all the support and guidance he provided to me throughout my Ph.D. study. He has taught me how to conduct research in system security since my first year—when I literally know nothing about how to do so. His solid technical background and logical, critical thinking have always inspired me.

I would also like to thank Dr. Marcus Peinado for always providing constructive comments when we collaborated on T-SGX and SGX-Armor. I have learned a lot from working with him. Besides, I would like to thank the rest of my thesis committee: Dr. Wenke Lee, Dr. Michael Steiner, and Dr. Brendan Saltaformaggio for their insightful comments that helped me improve this thesis.

I would like to express my special thanks to Dr. Sangho Lee for the fruitful discussions and the help when we collaborated on T-SGX and PRIDWEN. Additionally, I would like to thank my colleagues at SSLab: Sanidhya, Meng, Insu, Ren, Wen, Jinho, Soyeon, Fan, Seulbae, Hanqing, and former members, Steffen and Mohan for all the discussions and the good time spent together.

Last but not least, I would like to thank my family: my wife, Tiffany, my parents, and my brothers for supporting me spiritually throughout writing this thesis and my life in general.

Finally, I thank my God for letting me through all the difficulties. I have experienced Your guidance day by day. You are the one who brought me here and let me finish my degree. I will keep on trusting You for my future.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

In response to the growing need for securing user data in the cloud, recent Intel processors have supported a new feature, Intel Software Guard Extensions (SGX). SGX allows a program to execute in isolation from the rest of the underlying system. Thus, even after compromising the system, neither cloud providers nor attackers can gain access to data that the program processes. Unfortunately, recent studies have shown that such isolation is bypassable via side-channel attacks (SCAs). In particular, SCAs against SGX are more critical under the extreme assumption (*i.e.*, attackers compromise the system), allowing attackers to infer fine-grained information from an SGX-protected program.

Toward practical defenses against SCAs on SGX, the first part of the thesis presents two mitigation techniques, SGX-Armor and T-SGX. SGX-Armor is a general-purpose defense based on Address Space Layout Randomization (ASLR) that obfuscates the memory layout of the program, preventing attackers from interpreting side-channel information. Unlike traditional ASLR implementations, SGX-Armor incorporates a provably secure algorithm that shuffles memory layout without revealing the information of the layout through any of the known side channels. T-SGX is a novel defense against controlled-channel attacks that exploit page faults as a side channel. By using Intel Transactional Synchronization Extensions (TSX) as a primitive that suppresses page faults, T-SGX automatically transfers a program into a protected one at compile time.

The second part of the thesis presents PRIDWEN, a framework that addresses the challenges of composing multiple mitigation techniques such as SGX-Armor and T-SGX, thereby providing a broader scope of protection against SCAs on SGX. Using load-time synthesis, PRIDWEN adaptively enforces mitigation schemes to a program in distinct cloud environments. The prototype of PRIDWEN has supported four mitigation schemes that secure SGX programs again various SCAs while minimizing the incurred runtime overhead according to the configuration of the environment.

CHAPTER 1
INTRODUCTION

Over the past decade, an emerging computing paradigm that has rapidly evolved and gained significant popularity is cloud computing. Cloud computing refers to a collection of computing resources that provides on-demand services to remote users. Well-known examples of such services include Amazon AWS, Microsoft Azure, IBM Cloud, and Google Cloud Platform. An important benefit of these services is eliminating the need for users or businesses to purchase and to maintain hardware by themselves. However, this convenience comes with security concerns. One of the concerns is that once deploying data to a cloud platform, a user loses complete control over the data. More specifically, the user cannot prevent a cloud provider— who owns the platform and therefore has free access to the data— from inspecting or misusing the data. This concern has become a significant obstacle that hinders cloud adoption.

Solutions to the concern require protecting the data against cloud providers throughout its lifecycle, including data in transit, at rest, and in use. However, existing practices focus mostly on securing data in transit (*e.g.*, cryptographic protocols) and at rest (*e.g.*, disk encryption) while failing to protect data in use. To fill this gap, research and industry communities propose confidential computing, a new type of cloud computing. Confidential computing ensures that the data in a cloud platform remains encrypted during the data-in-use stage. The decrypted data is only available when the data enters a trusted execution environment (TEE) that provides isolation against the rest of the platform. By utilizing data encryption and TEEs, confidential computing allows users to process their data in the cloud without exposing the content of the data to cloud providers.

More recently, cloud providers have started supporting confidential computing (*e.g.*, Azure Confidential Computing and IBM Cloud Data Shield). The critical enabling imple-

mentation of TEE is Intel Software Guard Extensions (SGX) [1]. SGX provides a set of instructions that allows a program to execute within an isolated region of memory, an *enclave*. By incorporating a memory encryption engine (MEE) [2], SGX ensures that the content of the enclave stays encrypted, except when the content enters a central processing unit (CPU). The SGX-capable CPU enforces a strict access control that prevents all the code —even privileged one in an operating system (OS) or a hypervisor— outside the enclave from accessing its content. Consequently, users can deploy their programs to remote enclaves and assure that these programs securely process their data.

Although SGX enforces enclave-based isolation, recent studies have shown that this isolation is bypassable through side-channel attacks (SCAs). SCAs allow attackers to infer the information of programs from the side effects of their executions. These side effects usually stem from the implementation of underlying OSes or hardware. In-enclave programs, without exception, cause similar side effects as normal programs do. Known, exploitable side effects include the usage of page tables [3, 4], translation lookaside buffers (TLBs) [5], CPU caches [6, 7, 8], branch prediction units [9, 10], and CPU pipelines (*i.e.*, speculative execution [11]). In typical settings, measuring the usage of these resources is mostly limited; *i.e.*, the attacker needs to have root privileges or to deal with a significant level of noise from the OS. However, as SGX assumes powerful attackers (*e.g.*, cloud providers) who can eliminate these limitations, SCAs against enclaves become a more realistic, critical threat.

1.1 Research Objectives

This thesis work aims to devise practical defenses against SCAs on SGX. As the root cause of SCAs stems from the implementation of hardware, a fundamental solution is analyzing and modifying Intel CPUs or the SGX itself. Unfortunately, the implementation details of Intel hardware are proprietary and therefore are mostly inaccessible to research communities. Another type of solution is adopting cryptographic-based techniques such as oblivious RAM (ORAM) [12, 13]. However, these techniques incur significant runtime

overhead (*e.g.*, causing the execution tens or hundreds times slower) and usually require manual efforts to apply (*e.g.*, rewriting source code), both of which hamper their adoption in practice. In summary, we demand a practical, general solution that 1) requires no hardware modifications, 2) incurs reasonable runtime overhead to enclaves, and 3) minimizes manual efforts for its adoption. We present the objectives of the thesis toward this goal in the rest of the thesis.

1.1.1 Retrofitting Existing Side-Channel Defenses

An existing defense technique that meets our requirements and has the potential to mitigate SCAs is address space layout randomization (ASLR). ASLR is a software-based technique that obfuscates the memory layout of a program binary. Initially, ASLR aims to prevent the exploitation of memory corruption vulnerabilities in the program (*i.e.*, hindering an attacker from predicting target memory addresses). Since the goal of SCAs is inferring information of the program through side-channel leakage (in terms of memory accesses), ASLR ideally causes similar effects that prevent side-channel attackers from interpreting the leakage. With the potential, researchers have proposed ASLR-based defenses against SCAs on SGX [14, 15, 16]. Unfortunately, these defenses fail to protect the implementation of ASLR itself, allowing attackers to derandomize the layouts even before the program starts to execute. The thesis attempts to address this problem by investigating the challenges of enabling ASLR on SGX against SCAs. Moreover, the thesis seeks solutions that address these challenges.

1.1.2 Designing New Side-Channel Defenses

Based on the unusual assumption of SGX (*i.e.*, attackers with root privileges), researchers have demonstrated new types of SCAs. A representative example of such SCAs is controlled-channel attacks [3] that use page faults as a side channel. To exploit page faults, an attacker needs the abilities to manipulate the page table and to observe the occurrence of exceptions (*e.g.*, through the exception handler), both of which require root privileges. Having these

abilities, the attacker unmaps the memory pages of an enclave, causing all the corresponding accesses to trigger page faults. Consequently, the occurrence of these page faults allows the attacker to learn precise, page-granular memory accesses of the enclave. Despite being a standard mechanism that supports virtual memory in modern OSes, page faults impose a unique threat to SGX. As a result, the thesis intends to defeat page-fault-based SCAs by proposing new defenses.

1.1.3 Developing a General Side-Channel Defense Framework

The results of the thesis include two SCA defenses: an improved ASLR-based defense, SGX-Armor, and a new defense, T-SGX [17], which targets at controlled-channel attacks. An observation from these defenses is that using a single defense is insufficient to defeat all the knows SCAs on SGX. Regarding the ASLR-based defense, even with our improved design, the defense still subject to runtime attacks. Regarding T-SGX, the scheme focuses specifically on the controlled-channel attacks and therefore fails to prevent other types of SCAs (*e.g.*, cache attacks). In addition to these two defenses, we note that all the other proposed defenses [18, 19, 20, 21, 22] against SCAs on SGX suffer from similar limitations. A naïve approach to address such limitations is combining distinct defenses that collectively provide a broader scope of protection. However, this approach may not work in practice because some of the defenses are 1) *non-deployable* in target environments lacking the support of dependent hardware features, 2) *redundant* if target environments are immune to some types of SCAs, and 3) *incompatible* with each other by design or implementation. Statically identifying all of such conditions and preparing potential workarounds are challenging, primarily when a program targets multiple cloud platforms that abstract or manipulate their configurations. The thesis aims to propose a general SCA defense framework that addresses these challenges.

1.2 Thesis Contributions

SGX-Armor. The first contribution of the thesis work toward fulfilling the objective §1.1.1 is an improved ASLR-based defense, SGX-Armor. In this work, we thoroughly analyze enabling ASLR on SGX-like environments that are subject to SCAs. Our results include two new SCAs against SGX-Shield [14], the only public implementation of ASLR for SGX. One of the attacks completely infers the ASLR code layout as the implementation positions the code in memory. The other attack targets the runtime of the program, allowing the attacker to derandomize the majority of the memory layout. In response to these attacks, we present SGX-Armor, which incorporates an alternative algorithm for producing randomized memory layouts, which is immune to all known types of SCAs on SGX. Besides, our results indicate that using ASLR alone is insufficient to defeat SCAs that target its runtime phase. On the other hand, ASLR may still be effective against broader types of SCAs when combing our layout generation algorithm with a defense that limits side-channel leakage during the runtime of the program.

T-SGX. The second contribution of the thesis work regarding the objective §1.1.2 is a new defense, T-SGX, which target at controlled-channel attacks. T-SGX presents a novel usage of a commodity component of the Intel CPU, Transactional Synchronization Extensions (TSX), which implements hardware transactional memory. More specifically, TSX allows programs to execute concurrently inside hardware-assisted transactions that automatically detect conflicts among these executions. A conflict, including not only access to shared memory but also an exception such as a page fault, results in aborting the ongoing transaction. An attractive property of TSX is that the abort also suppresses the notification of exception to the underlying OS, which implies that the attacker cannot know whether a page fault has occurred during the transaction. T-SGX, by utilizing this property of TSX, carefully isolates the effect of attempts to tap running enclaves, thereby entirely eradicating the controlled-channel attacks. To be practical, we implement T-SGX as a compiler-based

scheme that transforms a program into a secure one without requiring manual source code modification or annotation.

PRIDWEN. The last contribution of the thesis work with regard to the objective §1.1.3 is a general SCA defense framework, PRIDWEN, which addresses the challenges of composing multiple defenses, thereby providing general protection against all the known SCAs on SGX. PRIDWEN is a framework that dynamically and selectively applies multiple defenses when loading a program according to the configurations of a cloud environment. PRIDWEN allows a developer to deploy a program in the form of WebAssembly (Wasm) to distinct environments along with a pre-installed, in-enclave loader. Upon receiving a Wasm binary, the loader probes the current hardware configuration and synthesizes a program (*i.e.*, a native binary) with an optimal set of defenses applied. Through the selection of defenses, PRIDWEN strives to maintain a similar level of protection across environments while minimizing the incurred runtime overhead. A PRIDWEN prototype supports software-only or hardware-assisted defenses that protect arbitrary in-enclave programs from various SCAs.

1.3 Outline of the Thesis

The outline of the thesis is as follows. Chapter 2 provides the necessary background information for this thesis. Chapter 3 presents SGX-Armor along with two new SCAs against typical ASLR implementations under the assumptions of SGX. Chapter 4 details the design of T-SGX, including the novel use of TSX and a compiler-based scheme that automatically transforms an SGX program into a TSX-enforced one. Chapter 5 introduces PRIDWEN, including design, implementation, the support of multiple mitigation schemes, and evaluation. Chapter 6 concludes the thesis and discusses future work.

CHAPTER 2

BACKGROUND

2.1 Intel Software Guard Extensions (SGX)

To ensure the security (i.e., integrity and confidentiality) of applications running in untrusted environments (e.g., the cloud), Intel has introduced a set of ISA extensions, Intel Software Guard Extensions (SGX) [1], to its recent CPUs. SGX consists of a set of instructions that supports the instantiation of isolated containers, enclaves, that are protected from all other software on the computer. SGX maps the code and data inside the enclave to a dedicated memory region, the Enclave Page Cache (EPC), within the main memory. To secure the enclave memory during the enclave runtime, SGX enforces a strict memory access policy that prevents accesses from not only user-level software (e.g., applications or other enclaves) but also privileged ones (e.g., an operating system or a hypervisor). To mitigate physical attacks such as memory bus spoofing or cold boot attacks [23], SGX incorporates a memory encryption engine, the Memory Encryption Engine (MEE), that keeps EPC pages encrypted while residing in the main memory. The MEE maintains a Merkle tree for the entire EPC region to detect any modification of code and data inside [2].

Enclave initialization. The operating system creates and initializes the enclave. It allocates EPC memory for the enclave and copies the target code and data into it. Relying on untrusted code is acceptable because SGX allows users to verify that the enclave was created correctly. The CPU maintains an enclave digest that captures exactly how the enclave was constructed. SGX supports remote attestation [24] such that users can verify that an enclave with the correct digest was created. However, putting the operating system in charge of enclave construction also means that the memory layout of the enclave is known to the attacker who controls the operating system. Similar to previous work [14], our ASLR model (§3.2.2)

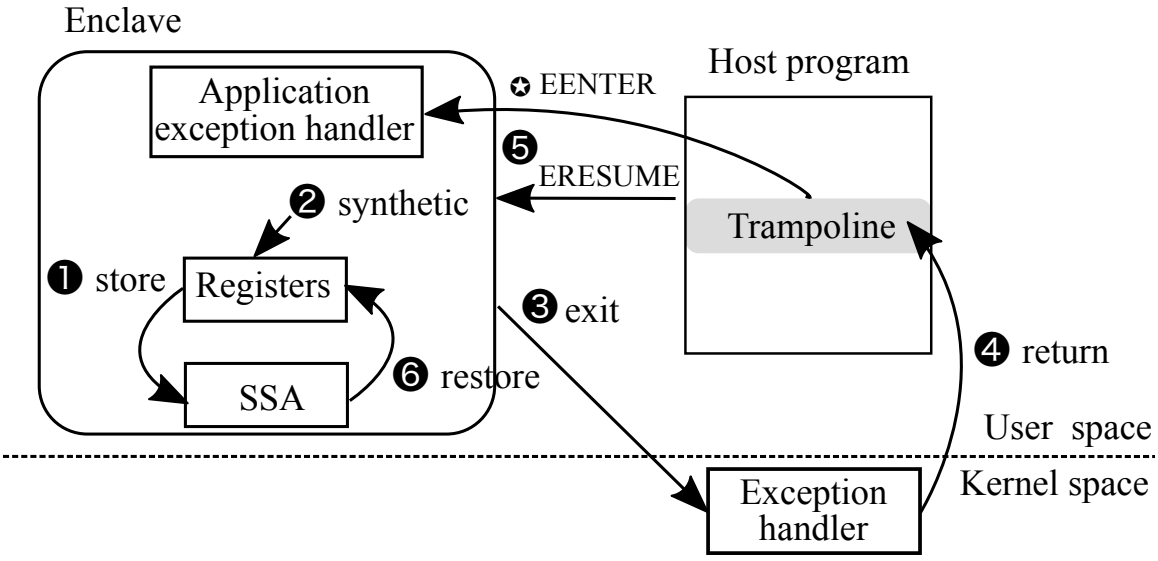

Figure 2.1: Steps of an SGX asynchronous enclave exit (AEX). ❶The CPU stores register values and the exit reason into the state save area (SSA) inside the enclave. ❷The CPU loads synthetic data into registers. ❸The enclave exits directly to the kernel space exception handler. ❹The exception handler handles the interrupt and returns to the trampoline. ❺The trampoline resumes the enclave. ❻The CPU restores the stored register values and resumes enclave execution. Besides, the trampoline can call an application exception handler inside the enclave to handle exceptions the OS cannot process.

tackles this fundamental problem.

2.2 Exceptions in SGX

An exception is an erroneous condition that occurs during a CPU execution, such as divide by zero (`#DE`), invalid opcode (`#UD`), general protection fault (`#GP`), and page fault (`#PF`). Conventionally, exceptions are delivered to system software for further investigation using default handlers or user-defined handlers. Such handlers can retrieve the exception code and the entire process context (e.g., register values). However, conventional exception handling does not work for SGX because it does not trust system software such that revealing the exception code and the process context to the systems software should be prohibited. To minimize information leakage during exception handling, SGX defines two mechanisms, Asynchronous Enclave Exit (AEX) and a custom exception handler [25, 26].

Asynchronous enclave exit (AEX). During the AEX, the original exception code and

register context are stored into the State Save Area (SSA) inside the enclave and then overwritten by synthetic data. Figure 2.1 depicts how the AEX is conducted. The CPU first stores the enclave's register context and exit reasons (e.g., exception code) in SSA and loads synthetic values into the registers. In the case of page faults, the CPU provides the OS only with the base address of the faulting page and not with the exact address. It then transfers control to the regular OS kernel exception handler. Eventually, the exception handler will return control to a user-mode trampoline function outside the enclave, which can call the `ERESUME` instruction. `ERESUME` will restore the enclave's saved register context and resume enclave execution.

Custom exception handlers. SGX allows developers to define custom exception handlers to process exceptions inside an enclave. These custom exception handlers can retrieve the SSA to check the stored exception code and registers (i.e., `GPRSGX.EXITINFO.VECTOR` and `GPRSGX.<registers>`), and to change them (e.g., `GPRSGX.RIP`) to resume execution.

Limitations. Although the AEX hides the register context and the exact address of an exception, information about the exception still leaks to the OS. For example, after each page fault, the OS learns which page the enclave attempted to access. This information is the basis for the controlled-channel attack [3]. Whenever an exception is generated during an enclave execution, the CPU exits from the enclave asynchronously.

2.3 Side-Channel Attacks (SCAs)

Unlike attacks that exploit the vulnerabilities inside a program (e.g., code-reuse attacks) and disclose its sensitive information or hijack its control flow, SCAs infer information about the program by observing physical effects caused by its operation. One classic example is timing-based attacks [27] that extract information from a program by measuring its execution time. More recently, the focus has shifted from timing attacks to microarchitectural side-channel attacks that exploit microarchitectural components (e.g., caches [28, 29] and branch predictors [30]) and allow attackers to infer finer-grained information. However, given the

high complexity of modern computer systems, side-channel attacks generally suffer from a significant amount of noise in practice.

SCAs against SGX. Enclaves share most microarchitectural components with the rest of the underlying system and therefore are inherently vulnerable to SCAs. Given the assumption that an attacker controls the operating system, several side-channel attacks have been demonstrated. Revealing page access patterns [3, 4, 31, 32, 33] by monitoring page faults or access and dirty bits is one of the most unique and well-known SCAs against SGX. Traditional cache SCAs work against SGX with different configurations [6, 8, 34, 35, 5]. Interrupt execution time can be used to infer which instructions have been executed within an enclave [36, 37]. Branch prediction behaviors [9, 10] are exploited to infer the results of branch instructions in an enclave. Speculative SCAs [38, 39] are feasible to Intel SGX. Foreshadow [11], or L1 Terminal Fault (L1TF), can leak enclave memory through the L1 cache. Recently, Microarchitectural Data Sampling (MDS) SCAs that leverage several internal buffers of Intel CPUs to leak sensitive information have been discovered [40, 41, 42]. In the adversary model of this thesis work, we assume that an attacker can launch the major types of well-known attacks against an enclave.

2.4 Address Space Layout Randomization (ASLR)

Among the numerous defenses against code-reuse attacks such as return-oriented programming (ROP) [43], the most widely adopted one in the real world is address space layout randomization (ASLR). ASLR aims to block the attacks' requirement of knowing the exact addresses of target code snippets (gadgets) by randomizing the memory layout of the program such that guessing the address of a gadget is computationally difficult. For example, initial ASLR techniques [44] that are widely deployed in mass-market operating systems randomize the base address of executables. More advanced (fine-grained) ASLR techniques randomize the process layout at a much finer granularity. This includes rearranging the addresses of functions [45, 46, 47, 48, 49], basic blocks [50, 51], or even individual

instructions [52].

Attacks on ASLR. One apparent threat to ASLR is memory disclosure [53]. These attacks require the existence of a *memory disclosure vulnerability* in the victim program; i.e., a vulnerability that allows the attacker to read arbitrary addresses in the victim's address space. This is a strong assumption. In particular, a memory disclosure vulnerability in an enclave would be equivalent to a complete breakdown in enclave confidentiality, thus negating one of the main benefits of running under SGX. Our attacks do not rely on this assumption.

Several authors use side channels to attack ASLR [54, 30, 55, 56, 57]. These attacks have been demonstrated only against traditional *non-fine-grained* variants of ASLR, in which the attacker has to find only a single pointer (i.e., code or data pointers) that can be used to infer the base address of the program. This limitation probably arises from the noisy nature of side channels in a traditional (non-SGX) setting. In contrast, we consider the impact of stronger SGX side-channel attacks against *fine*-grained ASLR in the work of SGX-Armor.

2.5 Intel Transactional Synchronization Extension (TSX)

In this section, we explain Intel Transactional Synchronization Extensions (TSX), which is Intel's implementation of hardware transactional memory (HTM) [58]. HTM was originally proposed to reduce the overhead of acquiring locks for mutual exclusion and to simplify concurrent programming. With HTM, a thread can transactionally execute in a critical section without any explicit software-based lock such as a spinlock or mutex. If a transaction completes without conflict, all of its read and write attempts are committed to memory. Otherwise, all of intermediate read and write attempts are rolled back (never exposed to the real memory) and a fallback (or abort) handler that was registered at the beginning of the transaction is invoked. The fallback handler decides whether to retry the transaction. Intel TSX supports two different interfaces, namely, hardware lock elision (HLE) and restricted transactional memory (RTM). Regarding the work of T-SGX, we focus only on RTM.

Intel TSX provides four instructions: `XBEGIN`, `XEND`, `XABORT`, and `XTEST`. A thread can

```
1  unsigned status;
2
3  // begin a transaction
4  if ((status = _xbegin()) == _XBEGIN_STARTED) {
5    // execute a transaction
6    [code]
7    // atomic commit
8    _xend();
9  } else {
10   // abort
11 }
```

Figure 2.2: A basic example of Intel TSX. `_xbegin()` initiates a transaction region to execute `[code]` and `_xend()` closes the region. An exception at `[code]` makes the control flow go to the `else` block.

initiate a transactional execution using `XBEGIN` and terminate it using `XEND`. It can use `XABORT` to terminate a transaction and `XTEST` to test whether it is currently executing in a transaction.

Figure 2.2 shows a code snippet that uses TSX. It first executes `_xbegin()` (*i.e.*, `XBEGIN`) to begin a transaction. If it succeeds, `_xbegin()` returns `_XBEGIN_STARTED` and continues to execute the code inside the `if` block (line 6). If there is no conflict, the program will eventually execute `_xend()` (*i.e.*, `XEND`) to atomically commit all the intermediate results. However, if there is a conflict or an exception, the transaction is rolled back and the program executes the `else` block (line 10) to handle the error.

Technical details. Understanding the technical details of the TSX implementation [59] is important because such details can explain why TSX exhibits the fallback behavior upon a transaction. During a transaction, HTM needs a buffer to store intermediate data read or written, so it can commit them to memory at the end of the successful transaction. Instead of introducing a separate buffer, TSX uses the L1 cache as a buffer. This choice was made not only to avoid extra storage requirements but also to simplify the implementation of TSX; it piggybacks on the existing cache coherence protocol to detect memory read or write conflicts without introducing complex new logic. The cache coherence protocol maintains data consistency between the caches of different cores such that TSX can detect data conflicts at the granularity of cache lines and roll back a transaction when a conflict occurs.

Transaction Abort. When two transactions conflict with each other (*e.g.*, their read and write sets overlap), one will be canceled, and thus aborted (see §2.5). TSX also aborts

a transaction when encountering an exception because a ring transition is not possible while executing a transaction. In the case of *synchronous exceptions* that occur during the execution of a specific instruction (*e.g.*, page fault, general protection fault, divide-by-zero), the exception is *not delivered* to the OS because TSX intentionally suppresses it ([26, §15.3.8.2]). On the other hand, an asynchronous exception (*e.g.*, timer interrupt and I/O interrupt) will be delivered right after a transaction is aborted and rolled back, because suppressing such interrupts would make user-space processes non-preemptable, which would interfere with OS scheduling.

Handling SGX exceptions with TSX. Using TSX inside SGX enclaves makes it possible to route exceptions such as page faults to TSX abort code inside the enclave and not to the ring-0 exception handler of the untrusted OS. This deprives attackers of all information about page faults and allows the enclave to identify potential attacks. We describe a design based on this observation in §4.4.

2.6 WebAssembly (Wasm)

The World Wide Web Consortium (W3C) proposes Wasm [60] as a platform-independent compilation target for various high-level languages (e.g., `C/C++` and `Rust`). A Wasm binary has language-like syntax and structure that are suitable for compilation and instrumentation. The basic executable unit of code in Wasm is a module that consists of multiple sections, where each section contains specific definitions of the module such as global variables, functions, and a sequence of instructions of each function. The execution model of a Wasm instruction is based on a stack machine; *i.e.*, each instruction manipulates values on an implicit operand stack, popping argument values and pushing result values. Besides, the model supports only the structured control flow such as if-else and loops without goto statements, enabling single-pass fast compilation. Lastly, to provide memory safety, Wasm prohibits any direct memory access and function call. Instead, they have to be indirectly accessed or invoked via indices. Because the Wasm binary is well-structured and

friendly for efficient Just-In-Time (JIT) compilation, PRIDWEN adopts Wasm as light-weight Intermediate Representation (IR) for supporting load-time synthesis.

CHAPTER 3
SECURING ASLR ON SGX AGAINST MULTIFACETED SIDE-CHANNEL ATTACKS

3.1 Introduction

Trusted execution environments (TEEs) such as ARM Trustzone [61] and Intel SGX [62] have been attracting significant interest in the research community and are seeing adoption in industry [63, 64]. They allow small security-sensitive programs to run in isolation from the rest of the software on the platform, which is typically large, complex, and potentially compromised. TEEs protect the confidentiality and integrity of code running inside them even if the rest of the platform (including the operating system) is compromised.

As compelling as these systems are, they do not offer protection from vulnerabilities within the trusted software itself. For example, Lee et at. [65] demonstrate that a single memory corruption vulnerability in an SGX enclave program can result in a complete breakdown of confidentiality and integrity even if the vulnerable binary is not known to the attacker. This situation makes generic defenses that can protect entire classes of vulnerabilities from being exploited highly desirable. Techniques such as control-flow integrity (CFI) [66], address space layout randomization (ASLR) [44], data execution prevention (DEP) [67], and stack canaries [68] are widely deployed in mass-market commercial systems and have kept countless bugs from becoming exploitable vulnerabilities.

Despite the undeniable benefits of these generic defenses, their deployment in TEEs is, at best, incomplete. The reason lies in the additional challenges posed by the TEE environment. Some of the defenses such as CFI or stack canaries are compiler-based and can be effortlessly deployed into TEEs. However, other defenses require system support that is not readily available in existing TEEs. In particular, ASLR, which is the focus of this thesis

work, typically requires assistance by the operating system in producing the randomized memory layout. However, in the TEE threat model, the operating system is the attacker. SGX-Shield [14] solves this problem by placing a loader into the enclave that produces a randomized layout of the application binary.

A more fundamental obstacle to deploying ASLR on TEEs is the existence of various side channels that may leak the randomized layout to the attacker. For SGX, highly accurate versions of cache- [6, 7, 8] and page-table-based channels [3, 4] have been demonstrated. These channels reveal the locations of memory accesses such as those used to read instructions as they are executed. In addition, the branch-prediction-based channel [9, 10] leaks whether conditional branches were taken. Various other cache-based side channels have also been demonstrated for Trustzone [69, 70]. Using processor hardware that only the operating system can access (e.g., hardware performance counters, page tables), these attacks obtain perfect or near-perfect signals, allowing the attacker to observe where code is originally placed in memory and where code is executed once the application starts running.

This thesis claims that when multiple side-channels (e.g., cache, page-table, and branch prediction) are combined in a *multifaceted side-channel attack*, implementing a perfectly secure ASLR scheme is difficult in practice, especially under the strong threat model of Intel SGX. In particular, we identify two attack vectors: First, the attacker can try to observe how the randomized binary is laid out in memory. Second, the attacker can observe the execution of the application binary. We present a simple attack on the initial layout generation phase of SGX-Shield that completely derandomizes the layout. The attack applies not only to SGX-Shield, but to a broad class of ASLR schemes that partition a binary into *code units* and use an in-enclave loader to place the code units into random locations. Our attack combines the page-table- and the cache-based side channels, which are complementary to each other, to track the actions of the loader as it copies the code units to their destinations. In light of a number of results that demonstrate the high-resolution and accuracy of these channels in the SGX model [3, 6, 7, 8, 4, 71], we follow the approach of prior work (e.g.,

Stacco [72] and DATA [73]) and use an abstract side-channel model, rather than building a physical side-channel implementation (which has been done many times).

One of the key challenges for the in-enclave loader is to produce the layout without leaking critical information through side channels. We solve this challenge by designing an efficient layout algorithm that is *provably* secure against side-channel leakage. The algorithm makes use of results from the theory of switching networks [74], which guarantee that an array of n elements will be almost randomly shuffled after applying $O(n \log n)$ appropriately positioned switches. A switch receives as input two fixed array positions that are public and known to the attacker and a secret bit. Depending on the secret bit, the switch either swaps the contents of the two array positions or leaves them unchanged. We treat the binary as an array of code units and use a switching network together with a carefully designed non-leaking switch to obtain a randomized layout. Our construction also handles relocations (adjusting relative and absolute addresses in the binary) without leaking information.

This construction solves the problem of creating the initial layout. However, information about the memory layout can still leak as the application executes. We present a runtime attack that leverages this information to identify the precise locations of the code units executed by the program. The attack builds a dynamic control-flow graph (DCFG) from the execution trace the attacker observes through multiple side channels. It then matches the observed DCFG against the known DCFG of the non-randomized binary. We evaluate the attack on SQLite [75], Lighttpd [76], and Libjpeg [77], three widely deployed open source libraries. Our results show that the attack recovers on average 94.9% of the fine-grained randomized layouts produced by SGX-Shield.

While several side-channel attacks against ASLR are known [54, 30, 55, 56, 57], these attacks target traditional (non-fine-grained) ASLR. In contrast, this thesis presents the first attack against a state-of-the-art *fine-grained* ASLR system. Attacking fine-grained ASLR poses significant, additional challenges as the amount of entropy in a fine-grained ASLR

layout is many times higher than in the traditional case. Traditional ASLR can typically be defeated by disclosing a single address in the randomized binary. This level of information disclosure reveals only a negligible fraction of a fine-grained ASLR layout. We are able to solve the additional challenges by working in the stronger TEE attack model, in which the attacker controls the operating system. In contrast, existing attacks on ASLR assume unprivileged attackers.

Finally, we implement a system, SGX-Armor. Its main component is a secure in-enclave loader that uses our layout algorithm to prevent the attack against initial layout generation. We evaluate the security and performance of SGX-Armor and discuss several other countermeasures that could potentially thwart the runtime attack.

In summary, this thesis work makes the following contributions:

- We construct a multifaceted side-channel attack against the layout generation of ASLR schemes in a general form, and demonstrate it against SGX-Shield.
- We present a similar attack that infers the randomized layout from the execution of the application (as opposed to layout generation).
- We design a new countermeasure for layout generation that is provably resistant to side channels.

The conclusion from these results is that SGX-Shield, the only existing ASLR system for SGX, does not provide robust ASLR under the SGX attack model. This is not a trivial shortcoming of SGX-Shield, but appears to apply to a broad class of existing fine-grained ASLR systems if one were to transplant them directly into the SGX attack model. We demonstrate that an ASLR system can robustly generate a randomized layout in the presence of powerful side-channel attackers. However, the problem of keeping the application from leaking this layout as it executes is still open.

3.2 Threat Model

This section outlines our threat model, which includes: a system model for running an enclave; an ASLR model that defines the class of ASLR systems we consider; and an adversary model that allows an attacker to use the three powerful side-channels to attack ASLR on SGX.

3.2.1 System Model

An untrusted, adversarial operating system runs on a fully functional SGX-enabled processor. The processor suffers from the standard known side channels in x86/64 processors, including cache, branch-prediction, and timing channels. However, the processor does not have further uncontrolled side-channel leaks. In particular, we assume that the timing, memory access pattern, and possible branch-prediction artifacts of simple integer instructions such as `add` and memory access operations such as `mov` do not depend on the values being added or stored. While integer division as well as floating point division and square root do not have this property on at least some x86 processor models [78], our security arguments will not depend on these instructions.

A program runs inside a correctly configured enclave that guarantees the integrity of the program and prevents external accesses from the operating system. However, the operating system knows the initial layout of the enclave. Moreover, the enclave is subject to side-channel attacks. The enclave does not adopt any existing defense mechanism against side-channel attacks. We discuss such defenses in §3.7. The enclave program has access to a random number generator whose outputs are not visible outside the enclave (i.e., the `rdrand` instruction).

Table 3.1: The summary of published fine-grained ASLR systems. Level represents the granularity of the randomization. `A`: base address. `F`: function. `P`: page. `B`: basic block. `I`: instruction.

Project	Description	Level	Load-time support
PaX ASLR [44]	Base address randomization	`A`	✓
Bhatkar et al. [45]	Source code transformation	`F`	
ASLP [46]	Binary re-writing	`F`	
Giuffrida et al. [47]	Link-time transformation	`F`	
Readactor [48]	Compile-time transformation	`F`	
Marlin [49]	Binary re-writing	`F`	✓
pagerando [79]	Compile- and link-time transformation	`P`	✓
Oxymoron [80]	Binary re-writing	`P`	✓
Binary Stirring [50]	Binary re-writing	`B`	✓
XIFER [51]	Binary re-writing	`B`	
kR^X [81]	Compile-time transformation	`B`	
SGX-Shield [14]	Compile-time transformation	`B`	✓
ILR [52]	Binary re-writing & VM support	`I`	
ORP [82]	Binary re-writing & In-place randomization	`I`	
Homescu et al. [83]	Compile-time NOP insertion	`I`	

3.2.2 ASLR Model

The program running inside the enclave is protected by ASLR. More specifically, we assume fine-grained ASLR that is immune to the known attacks [54, 30, 55, 56, 57] against traditional, non-fine-grained variants. Given the diversity of the published ASLR systems as shown in Table 3.1, we define a simple ASLR model to make clear what types of ASLR systems our results apply to. Our model is based on SGX-Shield, the only published, fine-grained ASLR system for SGX. However, it also aims to include the hypothetical systems that would result from a simple port to the SGX environment of a large portion of previous work.

Our model ASLR system partitions the program into *code units* and places them independently at random addresses. A code unit can be a function, a page, a basic block, or a similar unit. To address the problem that the operating system knows the initial layout of the enclave, the model defers the process of generating the randomized layout to the enclave

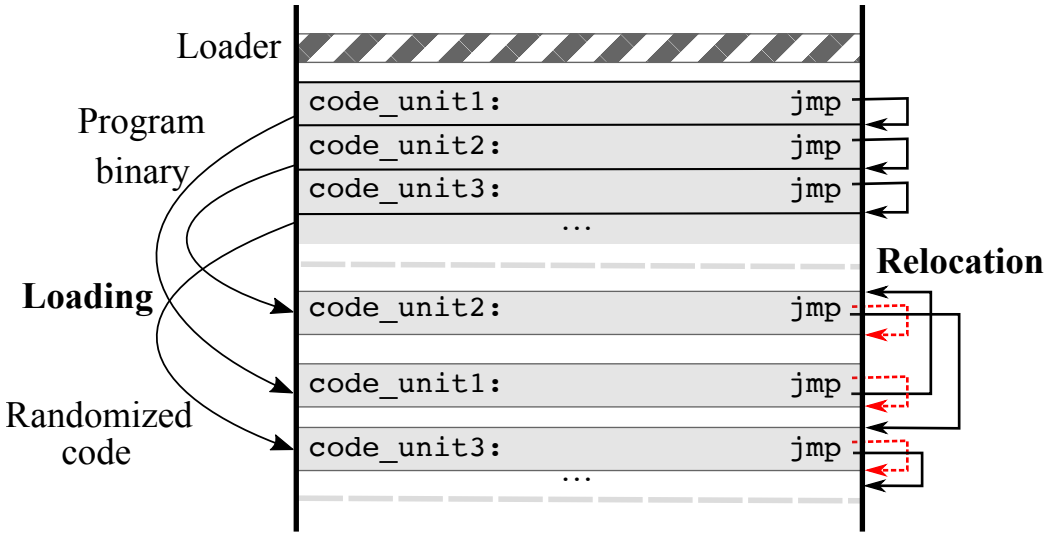

Figure 3.1: An example enclave under our ASLR model. Red arrows indicate that the destinations of `jmp` instructions require updates.

runtime. When the enclave first starts running, a small piece of code inside the enclave, the *loader*, copies each code unit a randomized address. The original, non-randomized program binary can either be placed inside the enclave at enclave creation time or be read by the loader from a location outside the enclave.

The loader and the program binary are public and known to the adversary. As a result, the random bits used by the loader to generate the randomized layout and the layout itself are the only secret information used in our model. The binary contains metadata that the loader can use to identify code units and to perform relocations. The loader consists of two phases. The loading phase copies each of the program's code units to a random address. The relocation phase adjusts the relocatable addresses within all the code units. After finishing the two phases, the loader transfers control to the program. As in SGX-Shield, the loader has no provision for hiding or obfuscating its memory accesses.

In this model, every program execution has a different randomized layout. This should prevent even sophisticated attacks [84, 65] in which the attacker does not know the target binary. In these cases, an attacker has to find gadgets by trial-and-error, which entail a large number of program crashes and restarts. The attack breaks down if the location of the

gadgets changes after each restart.

3.2.3 Adversary Model

Similar to the original threat model of SGX, we assume that the attacker controls the operating system. The enclave program may suffer from one or more vulnerabilities (e.g., memory corruption). The attacker's ultimate goal is to exploit one such vulnerability by means of control-flow hijacking. Because the binary of the program is public, the attacker can freely analyze it and find vulnerabilities. He also sees the static code layout of the program. Based on this knowledge, he can, in the absence of ASLR, locate code gadgets in the program and launch code-reuse attacks such as ROP. However, the enclave implements fine-grained ASLR based on our model, which effectively mitigates such attacks.

As fine-grained ASLR prevents the attacker from knowing the addresses of the code gadgets, his goal becomes derandomizing the code layout. To this end, the attacker learns about the code layout by launching side-channel attacks against the enclave that do not require causing crashes. More precisely, our attacker can launch the following three state-of-the-art side-channel attacks.

① **Page-table-based attacks.** This type of attack manipulates the page-table entries (e.g., present [3] or A/D bits [4]) that maintain the mapping between the virtual address of the enclave and the corresponding physical address. By observing either page faults or A/D bit changes in page-table entries, the attacker can infer the exact *page-granular* memory accesses of the enclave.

② **Cache-based attacks.** Similar to traditional cache-based attacks, this type of attack utilizes the prime+probe technique [29], which infers the *cache-line-granular* memory accesses of the enclave. Several authors [85, 6, 7, 8] have demonstrated that the strong adversary model (i.e., the attacker controls the operating system) allows the attack to infer the cache-level accesses with much less noise than the traditional settings do.

③ **Branch-prediction-based attacks.** This type of attack infers information about a branch

instruction by exploiting branch-prediction units. Previous work [9, 10] has shown that an attacker who knows the address of a branch instruction inside an enclave and its target can determine if the enclave program has taken the branch or not.

Channel resolution. A series of results use the local APIC timer in attacks on SGX that achieve high temporal resolution [8, 9, 7, 71]). SGX-Step [71], the latest of these results, demonstrates true single-stepping through enclave code. In particular, SGX-Step programs the local APIC timer, and thus interrupting the enclave execution after precisely one instruction has been executed. This has been demonstrated for enclave executions involving hundreds of thousands of fast instructions without a single instance of more than one instruction being executed between interrupts. In light of this result, we assume that the attacker can single-step through the enclave. That is, the attacker can interrupt enclave execution after every single executed instruction.

The spatial resolution of each side channel is given by the underlying hardware. The resolution of the page-table side channel is limited to the size of a 4 kB page, which provides address bits only 12 to 63 of each memory access. The resolution of the cache-based side channel on the L1 cache is limited to a 64-byte cache line, which allows the attacker to distinguish addresses that differ only in bits 6 to 11. The branch-prediction channel can provide detailed control-flow information if the attacker knows the address of the targeted branch instruction. The latter may not be the case under ASLR. Therefore, our multifaceted side-channel attacks use only the first two channels, which allows the attacker to observe address bits 6 to 63 of each memory access.

The addresses of code units that are smaller than 64 bytes may differ only in bits 0 to 5. For example, SGX-Shield supports a mode with 32-byte code units. Even a perfect L1 cache attack will not be able to identify the position of the code unit within the cache line. In the case of 32-byte SGX-Shield, this results in 1 bit of uncertainty for the attacker. The code unit could be at offset 0 or 32 within the identified 64-byte address range (a cache line), since SGX-Shield places code units at 32-byte-aligned addresses. The typical instruction

size combined with the need for an instruction at the end of the code unit to transfer control to the next code unit makes it difficult to envision code units that are consistently smaller than 16 bytes. Thus, the uncertainty resulting from the limited resolution of the cache side channel seems to be bounded by 2 bits.

An attacker can try to resolve this uncertainty by guessing and rerunning the victim until he succeeds. For 32-byte code units, the attacker will need an average of two guesses to find the precise address of a code unit. If the attacker requires three gadgets that are the same as the ones in the RIPE benchmark [86], he will need a maximum of $2^3 = 8$ guesses to make his ROP attack succeed.

Since the focus of this thesis work is the viability of ASLR under SGX, rather than demonstrating the effectiveness of physical side channels (which has been done previously [6, 7, 8, 3, 4]), we follow the approach of prior work [72, 73] and construct our attacks assuming that the underlying side channels exist. That is, rather than implementing three separate known physical side-channel attacks, we extract the information that could be obtained from each of the side channels from the trace of an enclave execution (see §3.5.2).

3.3 New Attacks against ASLR

This section describes two new multifaceted side-channel attacks against ASLR. One attack targets the layout generation phases of the loader. Another attack recovers the randomized layout through collecting the runtime execution trace of the target program.

3.3.1 Attacks on Randomized Layout Generation

Regardless of the details of the ASLR scheme, the loader has to move all code units from their locations in the original binary to their randomized addresses. Furthermore, the loader has to update all relocatable addresses to produce functional code. Importantly, the memory accesses involved in these operations are subject to side-channel leakage. Thus, by observing the loader's memory accesses through the multifaceted side channels—the

combination of cache- and page-table-based side-channels—the attacker can try to learn the entire randomized layout even before the program starts. We show two separate attacks against the two phases of the loader.

Attack on loading. Assume the program consists of n code units. The goal of the attack is to learn the randomized address of each code unit, which we denote as $addr_{rand}[i]$, where $i = 1, 2, ..., n$. Similarly, we use $addr_{init}[i]$ to represent the address of each code unit prior to loading. During loading, the loader copies each code unit i from $addr_{init}[i]$ to $addr_{rand}[i]$. Because the static layout of the program (i.e., the layout in the binary) and the base address of the program before loading are public, the attacker knows $addr_{init}[i]$. Based on this knowledge, he explicitly constructs a table T with n entries that are indexed by $addr_{init}[i]$.

The goal of the attacker is to add the correct $addr_{rand}[i]$ to each table entry i. The attacker uses the cache- and the page-table-based side channel to collect a trace of all the memory accesses made during the loading and tries to identify the pattern that associates a write to $addr_{rand}[i]$ with a read from $addr_{init}[i]$ for each i. The details of this step depend on the implementation of the loader. However, since the loader's code is public and the loader does not try to obfuscate its memory accesses, it should be easy for the attacker to identify the accesses to $addr_{rand}[i]$.

Attack on relocation. After the loading phase, the loader has to adjust the relative and absolute addresses based on the randomized layout. The metadata part of the binary contains a relocation table $Relo\_T$. An entry in the relocation table $Relo\_T$ includes a pointer to a code address (e.g., the address of an instruction's operand within a code unit) of the program that contains a relocatable address (e.g., the value of the operand) requiring an update and the necessary information for the update.

A typical loader begins at the start of $Relo\_T$ and sequentially reads its entries. For each entry, the loader retrieves the code address, which contains a relocatable address, and computes the value for updating the relocatable address. Then, the loader writes the new value to the code address. An attacker observes the loader's memory accesses during

this phase (via the cache- and page-table-based side channels) and identifies the relevant reads from $Relo\_T$ and writes to the corresponding code addresses. As a consequence, the attacker learns the final positions of each code address listed in $Relo\_T$ in the randomized layout. This reveals the randomized addresses of all code units that contain relocatable addresses. As in the previous attack, the attacker should be able to find the relevant memory operations in the trace because both the program binary and the loader are public and the loader does not attempt to obfuscate its memory accesses.

As mentioned earlier, the address resolution of our multifaceted side channel is up to 64 bytes, and the attacks in this subsection will not be able to observe the exact addresses if the code units are smaller. This does not affect the operation of the two attacks.

3.3.2 Attack on ASLR at Runtime

In addition to the attacks on layout generation, we present an attack that works even if the loader leaks no information about the randomized layout. The core idea of the attack is that, because every executed instruction involves an instruction fetch (memory access), an attacker listening on the cache- and page-table-based side channels will see the randomized addresses of the code units as the enclave program executes them. This trace of executed code units contains information about the program's runtime control flow. For example, short patterns of code units that are repeated several times are indicative of loops, and the number of code units in the loop, the number of repetitions in the trace (i.e., loop iterations) and their position in the trace may uniquely identify one particular loop in the binary, allowing the attacker to associate the observed randomized addresses with concrete code units in the binary.

Algorithm 1 Runtime attack

Step 1: Side-channel attack

1: Given an enclave that runs a program P with an unknown layout and an unknown input:

2: $side\_channel\_trace \leftarrow$ collect the runtime trace of P

Step 2: Off-line analysis

3: $DCFG_t \leftarrow build\_DCFG(side\_channel\_trace)$

4: Given the same program binary P with a known memory layout and a set of inputs Inp:

5: **for all** $i \in Inp$ **do**

6: $\quad trace_i \leftarrow$ collect the runtime trace of P given i

7: $\quad DCFG_i \leftarrow build\_DCFG(trace_i)$

8: $\quad I_i \leftarrow match\_DCFGs(DCFG_t, DCFG_i)$

9: $I_{final} \leftarrow I_i$ with the maximum number of matched vertices, $\forall i \in Inp$

The attack is described in detail in Algorithm 1. The first step (line 2) is to perform a side-channel attack while the victim is running. As stated earlier, the attacker can obtain a single-instruction-granular trace of the instructions executed by the victim by using the technique of SGX-Step [71] to single-step through the victim. After each instruction, he uses the multifaceted side channels to record the address of the last instruction with the given granularity (64 bytes in our case). The attacker gathers all these observations into a trace of the victim's execution. All remaining steps of the attack can be performed off-line using the trace.

In order to extract control-flow information from the trace, the attacker converts it into a dynamic control-flow graph (DCFG) [87] (line 3). The set of vertices of the DCFG is the set of unique addresses from the trace. There is a directed edge from vertex v to vertex w if and only if there is a transition from address v to address w.

The attacker's goal is to establish a mapping between the vertices of $DCFG_t$ (i.e., addresses of unknown code units) and the known code units in the binary P. This randomized

mapping is produced by the ASLR algorithm. The attack builds the mapping in two steps. First, the attacker generates several DCFGs for which he knows the mapping by running program binary P himself with a known memory layout and known inputs, gathering an execution trace (line 6) and converting the trace into a DCFG (line 7). Second, the attacker tries to find a graph isomorphism between $DCFG_t$ and at least one of the DCFGs he generated in line 6 and for which he knows the mapping between code units and layout addresses. By composing these two mappings, the attacker reconstructs the randomized ASLR layout. The rest of this section provides more detail on DCFG construction ($build\_DCFG$) and matching ($match\_DCFGs$).

DCFG construction. The construction of the basic DCFG from a trace is evident from the definition. The procedure processes the trace in linear order. Every time the procedure encounters a new address, it adds the address to the set of vertices of the DCFG. Every time the procedure observes an address v followed directly by a different address w in the trace, it adds a directed edge (v, w) to the edge set of the DCFG if that edge does not already exist.

Our DCFG construction procedure, $build\_DCFG$, augments this basic construction with additional information from the trace and encodes the information as vertex and edge labels. The information includes: 1. Given the observation that the basic DCFGs tend to have many *linear chains* (linear sequences of nodes with in- and out-degree one), $build\_DCFG$ replaces every linear chain by a single directed edge and labels that contain the length of the chain; 2. $build\_DCFG$ computes the order in which the addresses first appear in the trace. This is effectively the order in which the basic DCFG construction procedure adds vertices to the vertex set. The procedure adds the rank of each address in this order as a label to the corresponding DCFG vertex. The procedure also adds a similar label for each edge in the DCFG; 3. $build\_DCFG$ computes the relative frequency of each address in the trace and adds this number as a label to the corresponding DCFG vertex. The procedure also adds a similar edge label for transition frequency.

DCFG matching. Abstractly, our problem of matching two DCFGs is closely related to

the graph isomorphism problem [88] for which no polynomial-time algorithm is known. However, our concrete problem differs from standard graph isomorphism in several critical ways. Importantly, we have additional data for the vertices and edges in our DCFGs that simplify the problem significantly. Such data include the relative frequency with which individual vertices and edges appear in the trace and their order of first appearance in the trace. In addition, our graph isomorphism problem is approximate in two ways. The two graphs may not be completely isomorphic since different inputs may have sent the two executions along somewhat different paths. Furthermore, the attacker does not require a complete isomorphism. A correct matching between a reasonably large number of vertices may provide him with plenty of gadgets. These properties also distinguish our problem from previous work that performs the CFG matching in the context of malware detection [89, 90, 91, 92]. These papers try to find tiny signatures inside a large CFG and rely critically on information (instructions in CFG nodes) that is not available to our attacker.

Given the idiosyncratic nature of our problem, we design our own DCFG matching algorithm. We model our problem as a constrained optimization problem and use a simple heuristic to find an approximate solution. While more powerful and elegant algorithms may well exist, this simple scheme is sufficient for our attack.

Given two directed graphs $G = (V, E)$ and $\bar{G} = (\bar{V}, \bar{E})$, we are looking for a set $I = \{(e_1, \bar{e}_1), (e_2, \bar{e}_2), \ldots\}$ of edge pairings such that $e_i \in E$, $\bar{e}_i \in \bar{E}$ (for $i = 1, 2, \ldots$) that is of maximal size and satisfies the following condition:

Condition 1 (Graph structure preservation): If $(e_i, \bar{e}_i), (e_j, \bar{e}_j) \in I$ and e_i and e_j ($\bar{e}_i$ and $\bar{e}_j$) are neighboring edges (i.e., edges that have a vertex in common) then $\bar{e}_i$ and $\bar{e}_j$ (e_i and e_j) must have the corresponding vertex in common.

The condition guarantees that the edge mapping I respects the graph structure of G and $\bar{G}$ and that it is a bijection. In particular, no edge appears more than once in I.

Problems of this type can be approximated by various standard optimization algorithms such as simulated annealing [93]. Our algorithm uses the following greedy heuristic. We

define a confidence function $c : E \times \bar{E} \rightarrow IR$ that assigns a confidence value to each possible pairing of edges from G and $\bar{G}$. The confidence value is computed from the information contained in the edge and vertex labels. More similar values between the labels of e and $\bar{e}$ result in a higher confidence score $c(e, \bar{e})$.

Algorithm 2 $match\_DCFGs$

1: **for all** $e \in E$ **do**
2: $Cand(e) = \{\}$
3: $I \leftarrow \{\}$
4: **for all** $(e, \bar{e}) \in E \times \bar{E}$ **do**
5: **if** $c(e, \bar{e}) > c_{limit}$ **then**
6: add $\bar{e}$ to $Cand(e)$
7: **repeat**
8: $I \leftarrow I \cup \{(e, \bar{e}) | e \in E$ **and** $Cand(e) = \{\bar{e}\}\}$
9: **while** I violates *Condition 1* **do**
10: Remove $(e, \bar{e})$ from I, where $(e, \bar{e})$ is the conflicting pairing with the lowest confidence value
11: Remove all elements from the candidate lists that are inconsistent with I
12: **until** no new edge pairings

Algorithm 2 summarizes the scheme. The algorithm maintains a candidate list $Cand(e)$ for each edge $e \in E$, which is initialized to the set of edges $\bar{e} \in \bar{E}$ for which the confidence score $c(e, \bar{e})$ exceeds the threshold c_{limit}. The algorithm also maintains the partial edge mapping I which is initialized to the empty set. The main loop adds edge pairings to I for all edges $e \in E$ that have a unique partner in $\bar{E}$ (i.e., $Cand(e) = \{\bar{e}\}$). This indiscriminate addition of edge pairings to I may result in a violation of Condition 1. The next step is to remove pairings that cause inconsistencies from I until I satisfies Condition 1 again (line 10). After that, the algorithm prunes the candidate lists to remove all candidates that are

inconsistent with the current edge pairings in I (line 11). This process is repeated until no new edge pairings can be added to I. Each edge pairing in I matches not only the two pairs of vertices that make up the two edges, but also all vertices in the linear chains represented by the edges.

The intuition behind the algorithm is that the additional data contained in the vertex and edge labels allows us to establish an initial, small set of edge pairings that are typically correct. Once the algorithm has a correct edge pairing, the search space for the neighboring edges is vastly reduced from almost |E| to the in- or out-degrees of the two vertices that make up the paired edge. This reduction is captured in the pruning of the candidate lists. Effectively, the algorithm starts with a small number of seed pairings and grows I outwards from there by repeatedly pairing edges that are neighbors of edges that are already paired. This simple greedy heuristic requires most of the pairing decisions to be correct in order to work well. This has been the case in our experience. In situations where this is not the case, a variant of the algorithm that allows for backtracking may be required.

3.4 Mitigation against Attacks on Layout Generation

In this section, we introduce our defense system, dubbed SGX-Armor, that embodies a new mitigation against the attacks on layout generation. We discuss the possible mitigation and their limitations against our runtime attack in §3.7.

3.4.1 Secure Layout Generation

This section describes our algorithm for producing a randomized layout for the enclave program in the presence of a strong adversary who can observe all memory accesses as well as the outcomes of all conditional branches. We show that the algorithm leaks no information about the randomized layout to that adversary. In addition to producing a random permutation of the code units, our algorithm has to adjust all absolute and relative addresses in the program (e.g., the operands of conditional branch instructions) based on the

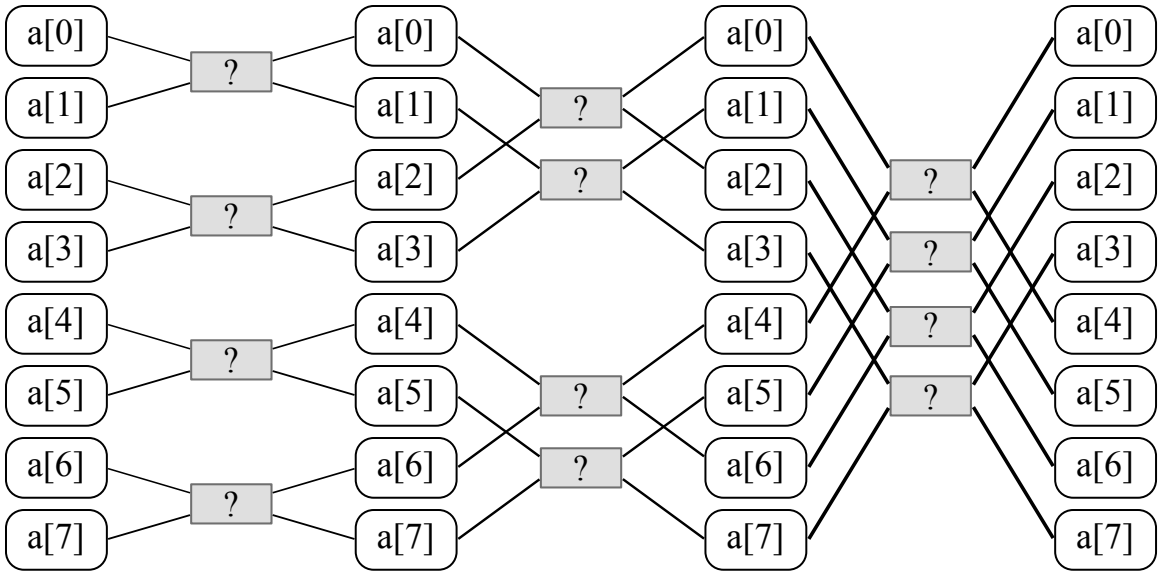

Figure 3.2: An example of a switching network with input size 8. The number of rounds is 3. The number of switches per round is 4.

Algorithm 3 OSWAP (a_1, a_2, b)	*Variable*	$b = 0$	$b = 1$
0 : **Load** X, Y **from** a_1, a_2			
$1 : B64 = 0 - b$	$B64$	$\{0\}^{64}$	$\{1\}^{64}$
$2 : notB64 = \neg B64$	$notB64$	$\{1\}^{64}$	$\{0\}^{64}$
$3 : 0orX = X \wedge B64$	$0orX$	$\{0\}^{64}$	X
$4 : Yor0 = Y \wedge notB64$	$Yor0$	Y	$\{0\}^{64}$
$5 : Y' = 0orX \vee Yor0$	Y'	Y	X
$6 : Xor0 = X \wedge notB64$	$Xor0$	X	$\{0\}^{64}$
$7 : 0orY = Y \wedge B64$	$0orY$	$\{0\}^{64}$	Y
$8 : X' = Xor0 \vee 0orY$	X'	X	Y
9 : **Store** X', Y' **to** a_1, a_2			

chosen permutation without leaking information about the latter. To ensure these security requirements, our algorithm assumes fixed-size code units.

Oblivious swap (OSWAP). Our algorithm relies on a primitive (OSWAP) that works as follows. Given two code unit addresses a_1, a_2 that are known to the attacker and a random bit b that is *not* known to the attacker, oswap(a_1, a_2, b) will swap code units at a_1 and a_2 if $b = 1$. If $b = 0$, it will leave the code units unchanged. In either case, oswap(a_1, a_2, b) will produce identical sequences of attacker-observable events (i.e., branches, memory accesses).

Algorithm 3 shows our implementation of oswap(a_1, a_2, b) for a fixed data length given

by the register size (8 bytes on x64). Longer data lengths can be accommodated by repeating the operation on subsequent 8-byte segments. Line 0 loads the content of addresses a_1 and a_2 to variables X and Y. Line 1 extends the one bit of b to all 64 bits of the variable $B64$, and line 2 stores the bitwise complement of $B64$ in another variable $notB64$. Line 3 computes the bitwise AND of X and $B64$. Line 4 does the analogous operation for Y and $notB64$. Next, line 5 computes the first output Y' by combining the results of lines 3 and 4 using bitwise OR. Effectively, the first five lines implement the following if-statement: IF $b == 0$ THEN $Y' = Y$ ELSE $Y' = X$. Lines 6 through 8 perform the analogous operations for the second output X'. Finally, line 9 stores the outputs X' and Y' to the addresses a_1 and a_2. We implemented an optimized version of this pseudocode using 15 lines of assembly code and three extra 64-bit CPU registers.

The sequence of memory accesses (lines 0 and 9) is clearly independent of b. Furthermore, in contrast to the *oblivious compare-and-swap* function from [94] which uses the `cmov` instruction and thus might leak secret information through microarchitectural channels, OSWAP uses only bitwise logic instructions and an integer subtraction and contains no conditionals at all. This makes OSWAP immune to a much broader class of side channels than those listed in §3.2.3, including channels with sub-cache-line granularity such as MemJam [35]. As long as bitwise `not`, `and`, `or` and `xor` as well integer subtraction do not leak the values of their operands, OSWAP is safe from all side channels.

Random permutation. If a single OSWAP operation does not leak information then any fixed sequence of OSWAP operations will be similarly side-channel resistant. Thus, we can use sequences of OSWAP operations to generate a random permutation of the code units in a side-channel-resistant way.

A series of results shows that sequences of OSWAP operations can rapidly produce an almost random permutation [95, 96, 97, 98, 74]. The high-level idea is to use an OSWAP-like operation as a switch and construct a switching network (see Figure 3.2 for an example). A recent work [74] shows that various types of switching networks with $O(n \log n)$ switches

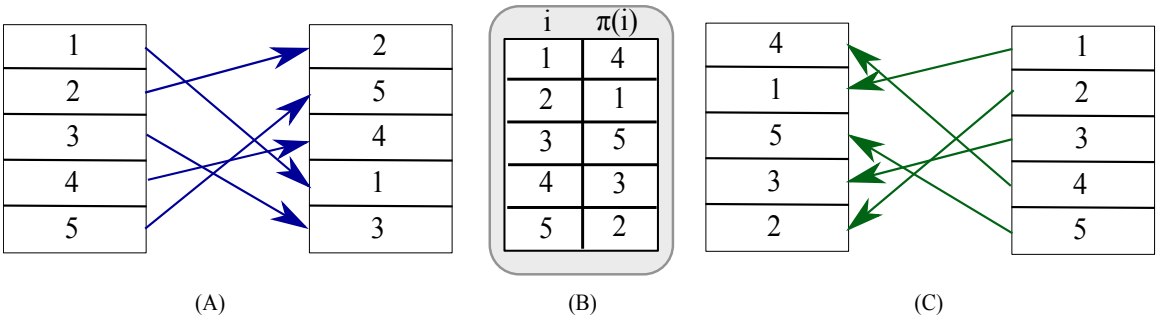

Figure 3.3: A: The switching network (blue arrows) permutes the code units (labeled $1, 2, 3, 4, 5$ on the left). B: The table π maps initial positions to final positions. C: Running the switching network backwards (green arrows). The output array equals to the table π.

produce near uniformly distributed permutations over n input elements. In other words, after $O(n \log n)$ OSWAP operations on appropriately chosen code unit addresses, the program (a sequence of n code units) will be close to a random permutation. This should be sufficient for ASLR. Note that permuting n code units in a larger region can be achieved by treating extra space as dummy code units.

Relocations. In general, the program's instructions will contain relative and absolute addresses that have to be adjusted based on the permutation. The challenge is to do so without leaking information about the permutation to the adversary. If the adjustments are performed on the permuted program layout, the code that adjusts the addresses has to be very carefully crafted to avoid leaking information about the permuted layout. Adjustments require reading and writing operands in the program, and these memory accesses can be observed by the adversary. For this reason, we adjust addresses on the original program layout *before* permuting the code units. This requires knowledge of the permutation in order to compute the correct relative and absolute addresses. An address a of a code unit can be calculated as

$$a = base + s \times i, \tag{3.1}$$

where $base$ is the base address of the code region, s is the size of a code unit, and i is the index of the s-byte slot in which the code unit resides. According to Equation 3.1, knowing an index i is sufficient for computing the address of an code unit.

We construct a table π that stores the position of each code unit will have under the final permutation. Figure 3.3 shows an example. Figure 3.3a shows how the network (represented by the blue arrows) permutes the code units. Figure 3.3b shows the corresponding table π. $\pi(i)$ is the position of the i-th code unit after the permutation (represented by the target of the corresponding blue arrow). Unfortunately, simply running our network will produce the right-hand side of Figure 3.3a and not table π. The key insight is that we can compute table π by running our switching network *backwards* as shown in Figure 3.3c. The network has the same arrows. However, their direction is reversed (from right to left). The input (right-hand side) is the sequence of array positions (1, 2, ...). Input i is moved to position $\pi(i)$ in the output array (left-hand side). The output array is identical to the table π.

To run the network backwards, we execute the switches of the original network, but in *reverse* order (i.e., from right to left in Figure 3.2). Each switch is fed the same random bit as in the original network. This requires remembering the entire random bit sequence and being able to access it in reverse order. The implementation could store the entire sequence in memory. However, more space-efficient options exist. This computation of π does not leak secret information by the same argument used for the original switching network. We are using the same network, except that we are executing it in reverse order.

Given table π and Equation 3.1, the absolute and relative addresses can be easily computed. We describe an example for a relative address as follows. Assume code units c_i and c_j originally reside in the i-th and the j-th slots. The code unit c_i contains a relocatable address dst at offset f. The target of the relocatable address is within code unit c_j at offset g. With the help of π, we can compute the relative address under the permuted layout for dst as

$$dst = (base + \pi(j) \times s + g) - (base + \pi(i) \times s + f). \tag{3.2}$$

Equation 3.2 shows that, once having the table π, the loader can perform the relocation process before permuting the code units. More importantly, the order of the process remains the same (i.e., follow the order on the relocation table). The attacker who knows the

relocation table (because the binary is public) cannot learn extra information about the randomized layout. Critically, secret information from table π is only used in the arithmetic operations of Equation 3.2. The accesses to table π are fixed and determined completely by the order of the relocation table entries, which is known to the attacker. Each calculation involves two lookups in table π ($\pi(i)$ and $\pi(j)$). Both i and j depend only on the binary, but not on information read from π. The secret values read from π are used only in a subtraction, an addition, a multiplication, and a store to memory (as the value being stored, not as the address). Our hardware model assumes that these operations do not leak information to the attacker. The argument for absolute addresses is similar.

3.5 Implementation

In this section, we describe our attack framework, and show how we used it to carry out the attacks on SGX-Shield. Finally, we describe the implementation of SGX-Armor in detail.

3.5.1 Automated Attack Analysis Framework

The input to the framework is a trace of the memory accesses of a program of the kind that fine-grained cache- and page-table-based attacks against SGX can recover. The purpose of the framework is to perform various common processing steps on the trace. As mentioned earlier, we did not implement the physical side channels. Instead, we collected the traces of the programs' memory operations using GDB [99], a widely used debugger. We implemented the framework in python with 1,750 lines of code. It consists of the following components.

① **Side-channel modeler.** The modeler applies models of side-channels to the collected trace. To model a cache-based side channel, we extract a cache-granular trace such that the address of each memory access in the trace has only bits 6-11. Moreover, we remove consecutive accesses with the same values for these bits. Similarly, to model a page-table-based side channel, we remove bits 0-11 from all addresses.

② **Layout generation attack analyzer.** The analyzer implements the attacks on loading and relocation, as outlined in §3.3.1. For the attack on loading, the analyzer requires additional information, including the original position of each code unit in the binary, the base address of the binary within the enclave, and the memory range to which the loader copies the code units. For the attack on relocation, the analyzer also requires the information of the relocation table.

③ **Runtime attack analyzer.** The analyzer realizes the attack described in §3.3.2. For the DCFG construction, the analyzer takes a side-channel trace generated from the side-channel modeler as an input and builds a DCFG based on the trace. For DCFG matching, the analyzer takes two DCFGs as inputs and performs the matching algorithm.

3.5.2 Attacks against SGX-Shield

Overview. The implementation of SGX-Shield consists of two components: a customized LLVM compiler and an in-enclave loader. The compiler partitions the executable into code units of constant size (32 or 64 bytes) such that the in-enclave loader can easily position each code unit at a different random memory location. This transformation places a jump instruction at the end of each code unit to transfer control to the next code unit. The compiler also adds symbol and relocation information that allow the loader to find the code units and to update absolute and relative addresses. At enclave creation time, the loader and the application binary are placed inside the enclave. The enclave also contains 32 MB of memory space for the randomized code layout. When the enclave starts running, the loader generates a randomized layout for the application binary by placing each code unit independently into a random (32- or 64-byte aligned) location within the 32 MB reserved space (the loading phase). Finally, it fixes up all relative and absolute addresses and transfers control to the application (the relocation phase).

Implementation of loading in SGX-Shield. Figure 3.4 shows the relevant code snippet of the implementation of loading. For each code unit in the table `symtab`, the function picks

```
1  static void load(void) {
2    for (unsigned i = 1; i < n_symtab; ++i, ++_n_symtab) {
3      Elf64_Addr symoff = pshdr[symtab[i].st_shndx].sh_offset
4                          + symtab[i].st_value;
5      ...
6      symtab[i].st_value = (Elf64_Addr)reserve(
7        pshdr[symtab[i].st_shndx].sh_flags, symtab[i].st_size,
8        pshdr[symtab[i].st_shndx].sh_addralign);
9      ...
10 ★    cpy((char *)symtab[i].st_value,
11 ★        GET_OBJ(char, symoff), symtab[i].st_size);
12   }
13 }
```

Figure 3.4: The code snippet of loading in SGX-Shield.

a randomized address from the reserved space (line 6-8). Next, the function copies the code unit from its original location (with the offset `symoff` to the base address of the binary within the enclave) to the randomized one (line 10-11).

Attack on loading. The memory accesses involved in `cpy()` provide the necessary pattern that the attack needs to infer the randomized address of a code unit. To implement the attack, we collect a runtime log of the loading phase and extract a trace of memory accesses with both page- and cache-granular information (i.e., bits 6-63 of each address are known) using the side-channel modeler. To identify the randomized addresses of the code units in the trace, the information we provide to the layout generation attack analyzer includes the base address of the binary inside the enclave, the offset of each code unit to the base address, which allows the analyzer to know each of `symoff`, the base address of the reserved space, and also its size (32 MB).

Attack on loading. During the loading phase, the loader iterates through all the code units. For each code unit, the loader picks a randomized address from the reserved space and copies the code unit from its original location to the randomized one. The memory accesses involved in the copy operation provide the necessary pattern that the attack needs to infer the randomized address of a code unit. To implement the attack, we collect a runtime log of the loading phase and extract a trace of memory accesses with both page- and cache-granular information using the side-channel modeler. To identify the randomized addresses of the code units in the trace, the information we provide to the layout generation attack analyzer

```
1  static void relocate(void) {
2    for (unsigned k = 0; k < n_rel; ++k)
3      for (unsigned i = 0; i < n_reltab[k]; ++i) {
4        unsigned int ofs = REL_DST_OFS(reltab[k][i].r_offset);
5        unsigned int dst_sym = REL_DST_NDX(reltab[k][i].r_offset);
6        addr_t dst = (addr_t)symtab[dst_sym].st_value + (addr_t)ofs;
7        ...
8        if (type == R_X86_64_64) {
9  ★      *(addr_t *)dst =
10 ★        symtab[src_sym].st_value + reltab[k][i].r_addend;
11       } else if (type == R_X86_64_32) {
12 ★      *(uint32_t*)dst =
13 ★      (uint32_t)(symtab[src_sym].st_value+reltab[k][i].r_addend);
14       } else if (type == R_X86_64_32S) {
15 ★      *(int32_t*)dst = (int32_t)(symtab[src_sym].st_value
16 ★                        + reltab[k][i].r_addend);
17       } else if (type == R_X86_64_PC32||type == R_X86_64_PLT32) {
18 ★      *(uint32_t*)dst = (uint32_t)(symtab[src_sym].st_value
19 ★                          - dst + reltab[k][i].r_addend);
20       } else if (type == R_X86_64_GOTPCREL) {
21 ★      *(uint32_t*)dst =
22 ★        (uint32_t)((uint64_t)&(symtab[src_sym].st_value)
23 ★        - dst + reltab[k][i].r_addend);
24       }
25     }
26 }
```

Figure 3.5: The code snippet of relocation in SGX-Shield.

includes the base address of the binary inside the enclave, the offset of each code unit to the base address, the base address of the reserved space, and also its size.

Implementation of relocation in SGX-Shield. Figure 3.5 shows the relevant code snippet of the implementation of relocation. The function iterates through the relocation table `reltab`. After getting information from the table `reltab`, the function calculates the correct value and updates the relocatable address to which `dst` points (line 9-10, 12-13, 15-16, 18-19, and 21-23).

Attack on relocation. The parts of the implementation that are vulnerable to the attack include the memory accesses to `reltab` and `dst`. We use the same granularity of side-channel traces as we used in the loading attack. We provide the base address and the size of both `reltab` and the reserved space to the analyzer.

Attack on relocation. During the relocation phase, the loader iterates through the relocation table. For each of the entry on the table, the loader obtains the target address points to the relocatable address, calculates the correct value, and writes the value to the target address.

The parts of the implementation that are vulnerable to the attack include the memory accesses to the relocation table and the target address. We use the same granularity of side-channel traces as we used in the loading attack. We provide the base address and the size of both the relocation table and the reserved space to the analyzer.

Runtime derandomization attack. To conduct the attack, we obtain the same granularity of runtime traces from a program with randomized layout and analyze the traces using the runtime attack analyzer. Because the attack does not depend on the implementation of SGX-Shield, we do not need to provide additional information to the analyzer.

3.5.3 SGX-Armor Prototype

```
1  oswapq:
2    movq (%rdi), %rax
3    movq (%rsi), %rcx
4    neg  %rdx
5    movq %rax, %r9
6    andq %rdx, %r9
7    not  %rdx
8    andq %rcx, %rdx
9    orq  %rdx, %r9
10   xorq %rcx, %rax
11   xorq %r9, %rax
12   movq %r9, %rcx
13   movq %rax, (%rdi)
14   movq %rcx, (%rsi)
15   ret
```

Figure 3.6: The implementation of OSWAP for two 64-bit inputs.

We implemented the prototype of SGX-Armor on top of SGX-Shield by replacing the layout and relocation code of its loader with the proposed, secure alternatives: OSWAP and *random permutation*. Our prototype used 32-byte code units.

OSWAP. We implemented OSWAP in 40 lines of assembly code, which follows the standard Unix 64-bit calling convention. The implementation includes a function that supports a single swap of two 64-bit operands as shown inFigure 3.6. The function uses registers `rdi`, `rsi`, `rdx` to hold the parameters a_1, a_2, b, respectively. In addition, the function uses three extra registers (`rax`, `rcx`, and `r9`) for storing the intermediate values of the algorithm. To support 32-byte code units, we provide a function with a loop that executes a series of the

64-bit swap.

Random permutation. The implementation of the algorithm consists of two phases: initialization and permutation. The initialization phase, which can be done offline, generates a random switching network. As previous work [74] has proved that almost every switching network generates an almost random permutation, a random switching network is sufficient for ASLR. Alternately, we could have used one of several explicit constructions [74, 100] with slightly different bounds. We chose to use a random switching network for ease of implementation. The permutation phase first generates a random bit (using `rdrand`) for each of switch and stores it in an array. Next, it applies the switching network to the code units and finally outputs the randomly permuted layout.

To secure the process of the relocation, we update the relocatable addresses before permuting the code units, as outlined in §3.4.1. To run the backward switching network, we execute the switches in reverse order using the random bits stored in the array. The permutation scheme is implemented by adding 863 lines of code to SGX-Shield's loader.

Safeguard code pages. To guarantee that the code pages are both non-writable and non-readable to the application after the layout generation, we use the same mechanism, software data execution protection (DEP), as SGX-Shield [14] does. This mechanism is not needed for SGX version 2 which supports modification of page permissions [101] by the enclave and which appears to have just become commercially available [102].

3.6 Evaluation

In this section, we present our evaluation results with the goal of answering the following questions.

- Is SGX-Shield vulnerable to the layout generation attacks?
- How effective is the runtime attack against SGX-Shield?
- What is the performance impact of SGX-Armor?

Experimental setup. The experiment was carried out on a machine with a 4-core Intel

i7-6700K CPU (Skylake) operating at 4.00 GHz with 64 GB of RAM. The machine ran Linux kernel 4.8. Each core had private 32 KB L1 and 256 KB L2 caches, and all cores shared an 8 MB L3 cache. All SGX applications were compiled with clang (based on customized LLVM 4.0.0 for supporting ASLR) and executed on top of the Intel Linux SGX SDK 2.1.102.

Target applications. In our experiments, we selected three applications and ported them to the SGX environment. The applications included an image compressor, a database, and a web server that were built on top of real-world, open-source projects. The image compressor is based on Libjpeg [77], a library for processing JPEG images, that had been chosen as the target of side-channel attacks in previous work [3, 17, 8]. For the database and the web server, we used SQLite3 [75], a popular SQL database engine, and, Lighttpd [76], a widely used web server. Both the database and the web server are commonly used in web applications and therefore would benefit from software defense mechanisms such as ASLR.

3.6.1 Attacking Layout Generation

This section examines how vulnerable SGX-Shield is to the layout generation attack we describe in §3.5.2. We ran the attack against the loader of SGX-Shield while it loaded each of our target applications. To estimate the impact of the code unit size, we ran our experiments for both 64- and 32-byte code units. For each of the settings, we ran each of the applications three times and report the averages over the three runs.

We present the results in terms of the fraction of code units from the binary for which the attack obtains the correct randomized address. We denote this fraction as the layout derandomization rate (DRR). The resolution of the cache side channel is inherently limited to 64-byte cache lines. Thus, we count a code unit as correctly identified if the attack finds the aligned 64-byte address range in which it resides. This makes no difference for 64-byte code units since there is only one way to place a 64-byte code unit into an aligned 64-byte address range. However, for 32-byte code units there are two possible aligned placements

Table 3.2: The results of initial layout attacks on target applications.

Application	Attacks on loading				Attacks on relocation			
	64-byte unit		32-byte unit		64-byte unit		32-byte unit	
	#units	DRR	#units	DRR	#units	DRR	#units	DRR
Libjpeg	4,756	100%	10,733	100%	4,606	96.8%	10,497	97.8%
Lighttpd	7,350	100%	16,105	100%	6,982	95.0%	15,455	96.0%
SQLite	14,851	100%	32,491	100%	14,097	94.9%	31,283	96.3%
Average		100%		100%		95.6%		96.7%

Table 3.3: The entropy averaged on the three target applications under different ASLR schemes. base: baseline entropy. loading, relocation, runtime: entropy left after each of the corresponding attack.

	base	loading	relocation	runtime
SGX-Shield 64-byte	19.00	0.00	0.84	0.96
SGX-Shield 32-byte	20.00	1.00	1.63	6.23
SGX-Armor 32-byte	20.00	20.00	20.00	6.23

(at offset 0 and at offset 32) within an aligned 64-byte address range, and the attacker will have to resolve the remaining bit of uncertainty as outlined in §3.2.3.

Results. Table 3.2 shows the results. For 64-byte code units, the attack on loading identifies all the code units of each target application. The attack on relocation identifies more than 95% of the code units. This number is below 100% because not all code units contain relocatable addresses. However, the attack identified 100% of the code units that contain relocatable addresses. For 32-byte code units, Table 3.2 shows that the attacks are as effective as in the 64-byte case. However, as explained above, the attacker is left with one bit of uncertainty, as the attack cannot determine if a code unit is placed at offset 0 or 32 within the 64-byte address range.

Entropy analysis. Table 3.3 displays the entropy averaged over all code units. The baseline entropy is determined by the size of reserved space (32 MB) and the size of code units. The results show that for 64-byte code units, the loading attack completely removes the entropy (i.e., becomes 0) because the attack identified all the code units. Using smaller code units

Table 3.4: The results of the runtime attack on target applications.

		SGX-Shield 64-byte unit			SGX-Shield 32-byte unit		
Application	**Test case**	**#units**	DRR	FDR	**#units**	DRR	FDR
Libjpeg	100x100 image	1,010	96.1%	0.5%	2,187	89.4%	2.2%
	200x200 image	996	97.4%	0.0%	2,159	77.1%	2.1%
	300x300 image	992	94.5%	0.5%	2,164	79.1%	2.8%
Lighttpd	HTML	1,946	94.6%	0.3%	4,391	62.7%	5.7%
	IMG	1,964	94.1%	0.8%	4,339	68.8%	5.0%
	CSS	1,958	94.0%	1.7%	4,394	57.0%	6.1%
	Javascript	1,952	97.1%	0.6%	4,357	70.6%	4.0%
SQLite	create	3,711	98.5%	0.1%	8,190	83.2%	5.8%
	insert	3,322	91.0%	1.5%	7,279	75.3%	6.7%
	update	3,532	93.6%	1.7%	7,966	76.5%	7.1%
	select	3,071	96.0%	0.9%	6,696	78.1%	7.3%
	delete	3,468	92.6%	1.2%	7,645	79.2%	5.7%
Average			94.9%	0.8%		76.3%	5.0%

(32-bytes) only provides 1 bit of randomness after the attack. For the relocation attack, although the number of code units that have relocatable addresses is smaller than the total number of code units, the results show a similar entropy reduction. The reason lies in the branch instruction that SGX-Shield inserts at the end of almost all the code units, which is relocatable. Finally, the results show that, by using SGX-Armor, the entropy remains the same as the baseline value because SGX-Armor is immune to the layout generation attack.

3.6.2 Attacking Runtime Randomization

The goal of this section is to evaluate how effective the runtime attack is against the fine-grained ASLR layout produced by SGX-Shield. We applied the runtime attack from §3.5.2 against all three target applications. For the image compressor, we used a 100x100 image as an input seed to get a baseline DCFG and a 100x100, a 200x200, and a 300x300 image to get target DCFGs. For the database application, we collected baseline DCFGs by separately executing create, insert, update, select, and delete operations. We collected target DCFGs by executing different sequences of these operations with distinct input parameters (i.e.,

operating on different databases). For the web server, we constructed DCFGs based on the executions of serving an HTML page, an image, a CSS file, and a JavaScript file. For the target DCFGs, we requested the files with the same types but distinct content. We executed each application with both 64- and 32-byte code units using SGX-Shield. We present the results in terms of layout derandomization rates (DRR) and false detection rates (FDR), which represent the fractions of correctly and falsely matched vertices out of all vertices of a target DCFG, respectively. For each of the settings, we ran each of the applications three times and report the averages over the three runs. We built the ground truth using debugging information. Similar to the layout generation attack, we also show the results in terms of entropy.

Results. Table 3.4 presents the results. For the case of SGX-Shield with 64-byte code units, the attack successfully identified nearly all the code units through DCFG matching (DRR=94.9%). This indicates that the algorithm effectively captures common behavior across different executions. The case of SGX-Shield with 32-byte code units introduced noise to the attack, which not only reduced DRR (to 76.3%) but also increased FDR. The likely explanation lies in the fact that some of the code units share their 64-byte address range with a second code unit. As result, the attacker is only able to observe the union of the transitions into and out of the two unknown code units. However, DRR is still quite high, which indicates that this situation is rare. The reason lies in the size of the reserved space is much larger than the size of total code units.

Entropy analysis. Table 3.3 displays the entropy averaged over all the code units observed during the runtime of the three applications. The results show that the case of SGX-Shield with 64-byte code units has almost no entropy after the attack. For SGX-Shield with 32-byte code units, the attack reduces the entropy from 20 to about 6 bits. The still considerable entropy of 6 bits is almost entirely the result of the fact that the attack failed to derandomize about 25% of the code units. For each of the 76.3% (DRR) of the units that were successfully derandomized, the residual entropy is only 1 bit (due to the 64-bit limit on address resolution).

Table 3.5: The load-time performance of SGX-Armor.

	Layout generation time		**Memory requirement**	
Application	**baseline**	SGX-Armor	**baseline**	SGX-Armor
Libjpeg	1.91 ms	3.34 ms	343 KB	609 KB
Lighttpd	3.31 ms	5.34 ms	515 KB	698 KB
SQLite	6.44 ms	11.36 ms	1,039 KB	1,911 KB

Table 3.6: The results of running nbench SGX-Armor.

Application	**baseline**	**SGX-Shield**	**SGX-Armor**
Num sort	951 μs	1.43%	1.34% ($-$0.09%)
String sort	5,932 μs	35.12%	35.33% ($+$0.21%)
Fp emu.	9,061 μs	6.78%	6.26% ($-$0.52%)
Assignment	36,205 μs	0.61%	0.89% ($+$0.28%)
Idea	291 μs	0.09%	0.03% ($-$0.06%)
Huffman	334 μs	24.73%	24.80% ($+$0.07%)
Neural net	21,596 μs	26.69%	26.95% ($+$0.26%)
Lu decomp.	795 μs	3.22%	3.23% ($+$0.01%)
Average		12.33%	12.35% ($+$0.02%)

Thus, the *average* entropy of 6 bits may not offer much protection, as an attacker may find plenty of gadgets in the 76.3% of the code units. For SGX-Armor with 32-byte code units, the entropy reduction is the same as the case of SGX-Shield with 32-byte code units. This indicates that SGX-Armor produces similar layout as SGX-Shield does and therefore is also vulnerable to the attack.

3.6.3 Overhead of SGX-Armor

The goal of this section is to demonstrate that SGX-Armor is practical. We evaluate the effect of SGX-Armor on both program load time and runtime. SGX-Armor adopts a code unit size of 32 bytes. For the load-time evaluation, we measure the time SGX-Armor takes to generate the layout. We use the time of loading the binary without permutation as the baseline. We also show the amount of memory SGX-Armor requires. We use the size of the code section in the binary as the baseline. For the runtime evaluation, we use the same

set of benchmark programs [103] used to evaluate SGX-Shield [14] and measure the time SGX-Armor takes to finish a program run. We obtained the baseline by directly porting the benchmark programs to SGX and measuring the execution time of each program (i.e., without instrumentation and ASLR). For comparison, we also reproduced the results of SGX-Shield with 32-byte code units. We show the overheads of both SGX-Shield and SGX-Armor as a percentage of the baseline. We collected the results of both load-time and runtime evaluations from ten independent runs and reported the averages over these results.

Results. Table 3.5 displays the results of the load-time evaluation for our three target applications. The impact of SGX-Armor on program startup time is less than 12 ms in all cases and is thus negligible. The memory requirements of SGX-Armor are noticeably larger than the baseline numbers. However, they are still small when compared to the total size of enclave memory ($\sim$100 MB). Importantly, all extra memory used by SGX-Armor is released by the time the application starts running. Thus, SGX-Armor has no impact on the memory available to running applications.

Table 3.6 shows the runtime performance. The runtime overhead of SGX-Armor over the baseline is only 12.35% averaged over the benchmark programs. The overhead comes mostly from the need to execute additional jump instructions at the end of each code unit. Compared to the results of SGX-Shield which executes the same number of extra jump instructions, SGX-Armor has negligible overhead (0.02% difference). This shows that SGX-Armor improves the security of layout generation over SGX-Shield without imposing additional runtime overheads.

3.7 Discussion: Mitigation against Runtime Attacks

We have shown in §3.6.1 that attackers who monitor the page-table- and cache-based side channels can completely derandomize the fine-grained ASLR layout generated by SGX-Shield. As discussed in §3.3.1, this is not just an implementation deficiencyagainst of SGX-Shield, but the result of the memory accesses of typical ASLR systems, which can be

observed through side channels. As we show in §3.4.1, layout generation can be protected against all the known side channels.

§3.6.2 showed that, by simply observing the combination of page- and cache-line-granular addresses of the executed instructions, the adversary can reconstruct DCFG and use it to identify the locations of a large part of the code units. On the other hand, the results also indicated that using code units that are smaller than the side-channel resolution can affect the accuracy of the reconstructed DCFG. Based on such observation, one simple mitigation can be to pack the code units densely (e.g., force them to share cache lines) and thereby obfuscating the reconstructed DCFG.

Unfortunately, it does not appear that this mitigation can provide general protection against attacks based on runtime leakage. In our model, the attacker knows the code units into which the ASLR system decomposes the binary; he just does not know their positions in memory. However, using page-table- and cache-based side channels, he can observe the addresses at which the enclave program executes instructions. Starting his observations when the enclave program begins to execute, he will immediately identify the location of the code unit that contains the program's entry point. If that code unit contains no unpredictable control-flow instructions, the adversary can follow program execution to the next code unit and so forth. In the extreme case (straight-line programs without control flow that the adversary cannot predict), the adversary can, thus, trivially derandomize the entire layout. The same is true if the attacker can use a complete and noise-free branch-prediction side channel. However, in general, this simple attack will break down because the attacker does not know which control-flow path the application is taking.

Yet, this mitigation relies on a property of the program and not of the ASLR system. Programs with simple DCFGs or whose DCFGs depend on data that the adversary can predict appear to be inherently vulnerable to our runtime attack. Below, we discuss potential defenses that lie outside our attacks and ASLR model.

Side-channel defenses. Several authors describe systems to mitigate certain SGX side

channels [17, 20, 15]. Without the strong SGX-specific side channels, the attacker would face a situation much closer to that faced by the traditional ASLR attackers. At present, it is still unclear if this approach can provide a general protection, as the published side-channel defenses, suffer from various limitations [20], have high overhead [15], or have been followed up be new side-channel attacks that overcome the defenses [17].

Program transformations. The defender could try to apply program transformations beyond ASLR. For example, Pappas et al. [82] randomize the code itself, rather than its location in memory. One could envision additional transformations to obfuscate or randomize the DCFG. The challenge with such approaches is that the component performing the transformations would have to execute inside the enclave subject to side-channel observations by the adversary. It appears challenging to design side-channel-resistant versions of these fairly complex components. Our side-channel-resistant layout generator from §3.4 has to solve a much simpler problem. In the presence of side-channel leakage, the transformations are subject to being observed by the adversary.

Rerandomization. Another approach could be to frequently rerandomize the memory layout [104, 105, 106, 107], either partially or completely. For example, one could interleave OSWAP operations with program execution. It is still unclear, however, how frequently the rerandomization would have to occur and what their security benefit and performance impact would be.

3.8 Related Work

Attacks on ASLR. Strackx et al. [108] showed an attack against ASLR by exploiting memory disclosure vulnerabilities in the program. Snow et al. [53] presented an attack against fine-grained ASLR. The attack relies on a serious memory-disclosure vulnerability that gives the attacker unrestricted read access to the victim's address space. Our attacks do not rely on such an assumption. Several authors attack ASLR using side channels [54, 30, 55, 56, 57]. These attacks target non-fine-grained ASLR in a traditional attack model. In contrast,

this paper targets fine-grained ASLR in the SGX attack model that the attacker controls the operating system. Recent work [30] speculates about the possibility of attacking fine-grained ASLR with the branch-prediction side channel, but does not describe or demonstrate a concrete attack.

Side-channel attacks against SGX. In addition to the attacks mentioned in §3.2.3, we discuss the known multifaceted side-channel attacks against SGX. Schwarz et al. [109] proposed an attack that combines the DRAM-based and the L3 cache side channels against an enclave without ASLR. Recent speculative execution attacks [39, 38, 11] combine the L1 cache side channels to infer the speculatively fetched memory content of an enclave. In contrast, our attacks combine the page-table-based and the L1 cache side channels against an enclave with fine-grained ASLR.

Oblivious swap operation. Ohrymenko et al. [94] described a primitive similar to OSWAP that depends on `cmov` instruction while OSWAP does not. Further, their target applications (machine learning algorithm) and the higher-level techniques employed are quite different from ours.

Attacks against SGX-Shield. Recent work [110] demonstrated code-reuse attacks against an enclave under the protection of SGX-Shield. The attacks exploit a limitation of SGX-Shield that does not randomize the code from Intel SDK. In contrast, our attacks do not require such a limitation in an ASLR implementation.

Switching networks applications. Kwon et al. [111] used a switching network to protected a messaging system against traffic-analysis attacks. Some authors [97, 98] used switching networks for encrypting small-sized data. Our work utilizes switching networks to securely produce randomized layout.

CFG matching. Several authors [89, 90, 92, 91] proposed to detect malware by searching for small signatures (CFGs with only a few vertices) in the CFGs of large binaries. In contrast to this line of work, our problem is to find an isomorphism between two large DCFGs. Furthermore, the matching in all these results uses knowledge of the concrete

instructions in each basic block. This information is not available to our side-channel attacker who can only observe the *addresses* of executed instructions and only at limited granularity. Finally, while these results match CFGs that were obtained though static analysis of the binary, our attacker has to resort to dynamic analysis to obtain the victim's DCFG. Consequently, he has to cope with the incompleteness and input-dependence of this DCFG. All this makes our problem quite different from previous work and requires a different solution. Stacco [72] uses DCFGs to decide if slightly different inputs fed to the same binary result in different control flow. This problem is quite different from ours. Our attacker has to find an isomorphism between two DCFGs that were scrambled as a result of ASLR. In contrast, Stacco has to decide if two DCFGs that should be obviously identical are different.

CHAPTER 4
ERADICATING CONTROLLED-CHANNEL ATTACKS AGAINST ENCLAVE PROGRAMS

4.1 Introduction

Hardware-based Trusted Execution Environments (TEEs) have become one of the most promising solutions against various security threats, including malware, remote exploits, kernel exploits, hardware Trojans, and even malicious cloud operators [112]. ARM's TrustZone [61] and Samsung's KNOX [113] are now widely deployed on mobile phones and tablets. To secure traditional computing devices, such as laptops, desktops, and servers, the Trusted Platform Module (TPM) [114], Intel's Trusted Execution Technology (TXT) [115], and Software Guard Extensions (SGX) [116] have been developed and are being adopted into mainstream products.

Among these hardware-based TEEs, Intel SGX is getting considerable attention because it can be the basis for practical solutions in an important security domain: the trustworthy public cloud, which provides strong guarantees of both confidentiality and integrity, which are known to be the biggest obstacle to wider cloud adoption [112, 117]. Homomorphic encryption [118] has been proposed as a software-only solution to this problem, but, so far, it is too slow for practical uses. More critically, sensitive operations are often executed on potentially malicious clients [119, 120, 121], which significantly weakens the overall, end-to-end security of the system. In contrast, hardware-based Intel SGX provides strong security guarantees for running enclaves in combination with Intel's efforts on formal verification of the hardware specification and implementation of cryptographic operations [122]. The resulting security guarantees enable a variety of new applications, including data analytics [123], MapReduce [124], machine learning [94], Tor [125], Network

Function Virtualization (NFV) [126], and library OSs [127, 128, 129].

However, researchers have recently demonstrated two critical side-channel attacks against SGX programs, namely, page fault and cache-based side channels [3, 33, 130]. The page fault attack, also known as the controlled-channel attack, is particularly dangerous because it gives the malicious OS complete control over the execution of SGX programs. In contrast, cache-based side-channel attacks have to passively, thus non-interactively, monitor the execution from the outside. Specifically, to launch a controlled-channel attack, the malicious OS can stop an enclave program, unmap the target memory pages, and simply resume its execution. By using the leaked addresses, researchers [3] could reconstruct input text and image files from running enclave programs [127]. Similarly, the pigeonhole [33] attack could extract bits of encryption keys from cryptographic routines in OpenSSL and libgcrypt.

Broadly, two types of countermeasures have been proposed, namely, obfuscating memory accesses [131, 13, 33] and isolating page faults [33, 132], but both are limited in terms of performance or compatibility. First, memory access obfuscation suffers from huge performance degradation: up to 4000$\times$ overhead without significant developer effort [33]. Second, more efficient schemes, such as self-paging [132] and contractual execution [33], require new page fault delivery mechanisms that do not exist in mainstream processors and are unlikely to be included in them in the foreseeable future. For example, Intel considers side-channel attacks as out of scope for SGX [122] and is unlikely to disrupt core processor components to accommodate such proposals.

In this thesis work, we propose a new, practical enclave design, called T-SGX, that can protect any enclave program against controlled-channel attacks using only existing commodity hardware. At a high level, T-SGX transforms an enclave program such that any exception or interrupt that occurs during the execution is redirected to one specific page (see §4.4.2). We provide strong security guarantees against controlled-channel attacks under a conservative threat model (see §4.6).

T-SGX realizes this mechanism with a commodity hardware feature, called Intel Transactional Synchronization Extensions (TSX), that was introduced with the Haswell processor. The key enabling property of TSX is the way it aborts an ongoing transaction when encountering an erroneous situation, such as a page fault or interrupt. In particular, when a page fault occurs, TSX immediately invokes a user-space fallback handler *without notifying the underlying OS*. The fallback handler recognizes whether the very recent attempt to execute a code page or access a data page has triggered a page fault. If it did, T-SGX carefully terminates the program. Further, TSX ensures that such traps and exceptions are *never exposed to system software including the OS and hypervisor*, implying that the controlled-channel attack relying on page fault monitoring is no longer possible with T-SGX because *even the OS cannot know whether a page fault has occurred.*

However, obtaining a working, efficient TSX-secured enclave binary requires careful program analysis. First, TSX is very sensitive to cache usage; it treats cache conflicts and evictions as errors [26, §15.3.8.2]. Thus, we have to carefully compose transactional code regions based on their memory access patterns. Second, TSX treats any interrupts and exceptions as errors (*e.g.*, timer and I/O interrupts), so we cannot run a code region for a long time even if it makes no memory accesses. Third, setting up a TSX transaction is very expensive (around 200 cycles on our test machine with an Intel Core i7-6700K 4 GHz CPU). This creates a naïve solution, wrapping individual instructions with TSX, impractical. Finally, we need to carefully arrange transactional code regions in memory to hide transitions between them from attackers (see §4.4.2).

T-SGX is based on a modified LLVM compiler satisfying the following three important design requirements. First, T-SGX automatically transforms a normal enclave program into a secured version, all of whose code and data pages are wrapped with TSX. Second, T-SGX isolates the specific page for the fallback handler and other transaction control code, called *springboard*, from the original program's code and data pages to ensure that exceptions including page faults and timer interrupts can only be triggered on the

springboard. The OS can identify whether an exception has occurred at the springboard, but this does not reveal any meaningful information. Third, T-SGX ensures that there are no unexpected transaction aborts due to benign errors (*e.g.*, transaction buffer overflow and timer interrupts), by carefully splitting a target enclave program into a number of *small execution blocks* satisfying the TSX cache constraints. A conservative splitting strategy (*e.g.*, secure individual basic blocks) significantly slows down T-SGX (§4.7). We develop compiler-level optimization techniques, such as loop optimizations and cache usage profiling, to increase the size of execution blocks as much as possible for performance (§4.5).

Our evaluation results show the effectiveness of T-SGX in terms of security, compatibility, and performance. First, we applied T-SGX to the three programs attacked by the controlled-channel attack (JPEG, Hunspell, and FreeType) and confirmed that an attacker was no longer able to obtain meaningful information and there was no compatibility issue. We also checked the overall overhead of T-SGX with the programs. On average, the execution time increased by 40% and the memory consumption increased by 30%. Second, we applied T-SGX to a popular benchmark suite, nbench, and confirmed that the performance overhead of T-SGX was 50% on average.

In summary, this thesis work makes the following contributions:

- **New security mechanism.** We develop a new security mechanism, T-SGX, to protect enclave programs from a serious threat: the controlled-channel attack. At compilation time, it transforms an enclave program into a secure version without requiring annotations or other manual developer efforts, and, most important, it does not require hardware modifications.
- **Novel usage of TSX.** To the best of our knowledge, T-SGX is the first attempt to use TSX to detect suspicious exceptions. Mimosa [133] was the first application of TSX to establish a confidential memory region, but it focuses on detecting read-write or write-write conflicts, which is the original use case of TSX. In contrast, we use TSX to isolate exceptions such as page faults and redirect them to a user space handler

under our control.

- **Springboard and program analysis.** It was believed that TSX should be applied only to a small portion of a program due to its sensitivity to cache usage and interrupts. Our springboard design and program analysis make a breakthrough: we can run any program in transactions without compatibility problems.
- **Evaluation and analysis.** To evaluate its security, we applied T-SGX to previous controlled-channel attack targets (*i.e.*, libjpeg, Hunspell, and FreeType) and confirmed a 40% performance overhead on average. To understand the performance in detail, we also ported and ran a benchmark suite, called nbench: we observed a 50% performance degradation on average. T-SGX is also easy to use; it was able to transform all our macro- and micro-benchmarks with no source code modification.

4.2 Controlled-channel Attack Revisited

In this section, we briefly explain the controlled-channel attack [3] (called pigeonhole attack in [33]) that allows a malicious OS to infer sensitive computation and data inside a TEE such as Haven [127] and InkTag [134]. We limit the discussion to attacks against SGX, which is the focus of this thesis work and which provides stronger security guarantees than a trusted hypervisor (*e.g.*, InkTag).

4.2.1 Threat Model

We explain the threat model of the controlled-channel attack. Note that our system, T-SGX, assumes the same threat model.

First, the attack assumes that an OS can manage (*e.g.*, map and unmap) enclave memory pages although it cannot see their contents. Whenever an enclave program attempts to access an unmapped page, the OS will receive a page fault to handle it; then the handler either remaps the page and resumes the program or generates an access violation error. However, this attack does not assume that the OS knows the exact offset of a page fault because TEEs

can hide this information from the OS.

Second, the attack assumes that an attacker knows the detailed behavior of a target enclave program, especially its memory access patterns according to inputs. The attacker has already analyzed a target enclave program's source code and/or binary in detail to obtain the information. Also, this attack ignores programs with obfuscated memory access patterns (*e.g.*, Oblivious RAM (ORAM) [13, 135]) because they do not have visible behavior characteristics.

Third, the attack assumes that an attacker cannot arbitrarily run a target enclave program. Due to remote attestation, a user will know how many times his/her enclave program is executed in the public cloud such that it is difficult to run the target enclave program many times without the user's approval.

Fourth, the attack relies only on a noise-free side channel: page fault information. Other noisy side channels, including cache and memory bus, are out of the scope for this thesis work.

4.2.2 Controlled-channel Attack

The controlled-channel attack uses page faults as a controllable side channel. Since a malicious OS can manipulate the page table of an enclave program, it can know which memory pages the enclave program wants to access by setting a *reserved* bit in page table entries and monitoring page faults.

In contrast to a normal execution environment, the malicious OS cannot see the exact faulting address but only the page frame number because SGX masks the exact address, as explained in §2.2. To overcome this limitation, the controlled-channel attack analyzes *sequences of page faults* rather than individual page faults.

The final step of the controlled-channel attack is correlating the page fault sequences with the results of offline, in-depth analysis of a target enclave program. This allows the attacker to infer the input to the enclave program if the memory access pattern of the program

varies sufficiently with the input.

Effectiveness. The original controlled-channel attack was demonstrated against three popular libraries: FreeType, Hunspell, and libjpeg. The evaluation results show that the attack can accurately infer the input text and images to the libraries [3]. Shinde *et al*. [33] use a similar attack to extract bits of cryptographic keys from the OpenSSL and libgcrypt libraries.

4.2.3 Known Countermeasures

A few countermeasures against controlled-channel attacks have been discussed, but most of them are neither practical nor secure. Intel has revised its SGX specification to support an option for recording page faults and general protection faults in the SSA [116]. However, this countermeasure is incomplete because a malicious OS can cause the SSA to be overwritten (details in §4.2.4). Second, Intel has suggested static and dynamic analysis to eliminate all feasible input-dependent code and data flows [122]. But, this requires significant developer effort and incurs non-negligible performance overhead. Third, Shinde *et al*. [33] have proposed deterministic multiplexing, a software-only solution against the controlled-channel attack. However, its performance overhead is tremendous without developer-assisted optimizations. Finally, Shinde *et al*. also have proposed a new execution model (contractual execution) that makes a contract between the enclave program and the OS to ensure that a specified number of memory pages reside in the enclave. Their proposal, however, requires modifications to core processors components. Such changes appear difficult and unrealistic.

4.2.4 Overwriting Exit Reason

As mentioned in §2.2, the SSA stores the exit reason for each AEX. However, we found that a malicious OS can easily overwrite the exit reason by sending an arbitrary interrupt to an enclave program because the SSA stores only the last exit reason[1]. This makes an enclave

[1]There is an SSA stack for handling nested exceptions. However, this overwriting attack is not about nested exceptions because it sends a new interrupt right after handling the previous interrupt.

program unaware of page faults, even if it uses an option SECS.MISCSELECT.EXINFO=1 to record page faults and general protection faults.

We experimentally confirmed that a malicious OS can overwrite the exit reason of a page fault by using a fake general protection fault. When a page fault is generated, a corresponding address is stored in SSA.MISC.EXINFO.MADDR and PFEC is stored in SSA.MISC.EXINFO.ERRCD for later use [116]. However, a general protection fault could overwrite these fields: it stores 0 in SSA.MISC.EXINFO.MADDR and GPEC in SSA.MISC.EXINFO.ERRCD. We have found that a malformed Advanced Programmable Interrupt Controller (APIC) interrupt generates a general protection fault. The OS can program the APIC to generate such interrupts and thus general protection faults during enclave execution. Therefore, if a malicious OS generates a malformed APIC interrupt for an enclave program right after handling a page fault, the fields in the SSA are overwritten such that an enclave program cannot know whether or not a page fault has occurred. Further, the OS can generate another normal interrupt (*e.g.*, a timer interrupt) later to even clear up the GPEC flag. Thus, we conclude that relying on the exit reason cannot protect an enclave program from the controlled-channel attack.

4.3 System Model

In this section we explain our ideal system model. An ideal enclave (*uncontrollable enclave*) protects any enclave program from the security threats explained in §4.2. The basic requirement of the uncontrollable enclave is to enable an enclave program to know every interrupt and page table manipulation, and stop its normal execution when it detects that the OS has unmapped any of its sensitive memory pages.

To achieve these goals, the uncontrollable enclave allows an enclave program to have two kinds of memory pages: *secured pages* and *controller pages*, as shown in Figure 4.1. First, the secured pages are *unobservable* pages containing all code and data of an enclave program. The OS cannot interrupt the enclave program when it executes or accesses the secured pages, and it cannot monitor page faults generated due to the execution or access of

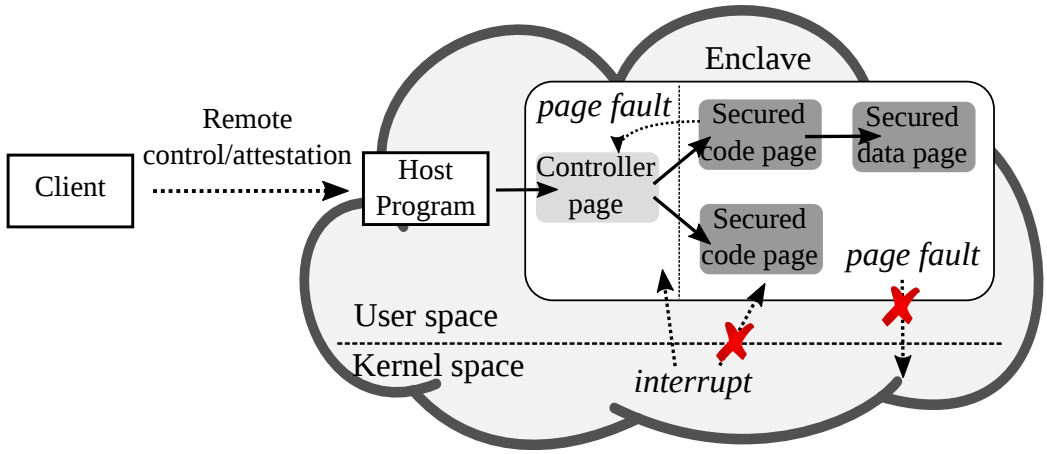

Figure 4.1: The uncontrollable enclave model. It consists of secured and controller pages. The secured page is not interrupted by the OS and its page fault is delivered to the controller page instead of to the OS. The controller page manages control and data flows between secured pages and handles page faults generated by accessing secured pages.

these pages. Since secured pages are uninterruptible, the uncontrollable enclave needs to ensure that execution with secured pages is short (*e.g.*, up to the interval of a timer interrupt) to prevent a malicious enclave program from fully occupying a CPU core. Also, when the uncontrollable enclave detects that any of the secured pages are unmapped, it treats the OS as malicious.

Second, the controller pages relay the control and data flow between the secured pages, check whether access to the secured pages is hindered by the OS (*i.e.*, unmapped), and interact with the OS for scheduling and system calls. The OS can interrupt their execution and monitor page faults generated by accessing them. However, revealing their behavior does not leak much information because they are just trampoline pages and the actual execution of an enclave program is performed inside the secured pages.

The uncontrollable enclave ensures that no page fault sequence (*i.e.*, inter-page accesses) is revealed to an OS. First, when the uncontrollable enclave identifies that a secured page is unmapped, it stops its execution. This could reveal up to a single page fault to the OS. Second, the enclave program can let its remote client know whether or not it has successfully terminated by sending an acknowledgment message. A lack of acknowledgment also means that there was a problem. Third, the uncontrollable enclave prevents the OS from running the

enclave program arbitrarily. To achieve this, the enclave program checks whether its client allows the OS to run itself during a remote attestation process. The remote client would completely disallow any further execution if the program did not send acknowledgment messages before. Note that it is natural to assume that an enclave program runs in the cloud and its remote client controls its execution.

Based on these requirements, we implement a prototype scheme, T-SGX. T-SGX does not ensure perfect information leakage prevention, but we believe it is sufficient to make the known controlled-channel attacks impractical (see §4.6 for details).

4.4 Design

In this section, we describe in detail the design of T-SGX, which is a practical realization of the uncontrollable enclave model (§4.3). In particular, we explain how to realize the model's various components using Intel TSX.

4.4.1 Overview of the TSX-based Design

This section describes a working instantiation of our architecture that relies only on a widely deployed standard processor feature (TSX). This approach yields a practical and effective side-channel mitigation that can be used today.

Intuitively, the main value of TSX as a side-channel mitigation lies in its ability to suppress page faults and other synchronous exceptions. A page fault that occurs during a TSX transaction will not be delivered to the untrusted ring 0 page fault handler. Instead, the processor will abort the transaction and transfer control to the transaction's abort code. Thus, our strategy will be to run enclave code inside transactions and to place a trusted exception handler in the TSX abort code path.

Figure 4.2 shows a simple example of an enclave program and its TSX-based transformation. The code between `_xbegin` and `_xend` is executed as a transaction. The `else` branch contains the abort code path. TSX guarantees that any page fault that occurs while executing

```
// original code
void foo(char *msg, size_t len) {
  const char *secret = "key";
  ...
}

// protected code
void ecall_foo(char *msg, size_t len) {
  if ((status = _xbegin()) == _XBEGIN_STARTED) {
    foo(msg, len);
    _xend();
  } else {
    // abort: e.g., page fault detected
    abort_handler();
  }
}
```

Figure 4.2: A straw man example that wraps the entire enclave code in a TSX transaction to prevent controlled-channel attacks.

`foo(msg,len)` is suppressed and control is transferred directly to the `abort_handler` in the `else` branch.

A simple design idea could be to wrap the entire enclave program in a single TSX transaction. However, for typical programs, such transactions will never complete because (a) TSX will abort a transaction if its write or read set is too large to fit into the L1 or L3 cache, respectively, and (b) long-running transactions are highly likely to be aborted by interrupts. Thus, we have to partition the program into small execution blocks and wrap each execution block in a transaction.

This requires the ability to perform detailed static analysis as well as a number of program transformations. For this reason, we integrate T-SGX into the compiler. As the source code is compiled into an enclave binary, T-SGX computes an appropriate partitioning into execution blocks and makes sure each execution block is protected by TSX by conservatively placing `XBEGIN` and `XEND` instructions (see §4.4.3 for details).

4.4.2 The Springboard

Using many small transactions entails a new problem. As page faults are not suppressed across transactions, an attacker may still see all page faults he/she is interested in, unless transactions are carefully arranged in memory. Figure 4.3 shows an example in which the

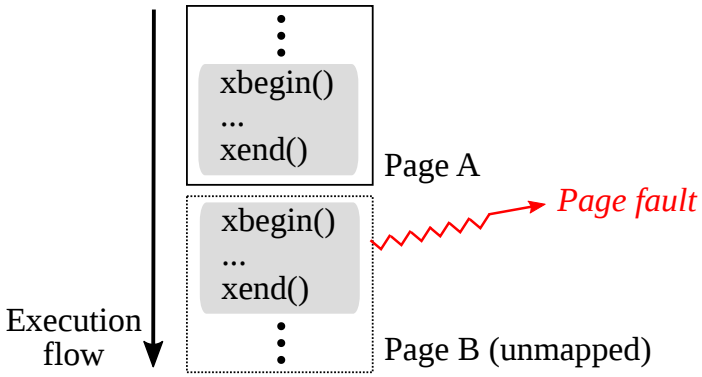

Figure 4.3: Careless usage of TSX revealing a page fault. An attacker can monitor the page fault at Page B because a transition between Page A and Page B is not in a transaction.

transition between two transactions is leaked because they are at a page boundary. An attempt to execute the first instruction on page B causes an observable page fault outside a transaction.

T-SGX solves this problem by placing all code that executes outside transactions on a single page. We call this page the *springboard*. Figure 4.4 displays the code that performs the transitions between consecutive transactions. An important property of this code is that *it does not access memory on any other page (e.g., stack, heap)*. Upon receiving an enclave function call from the host, the entry function begins by jumping to the `springboard.begin` block, which starts a transaction with an `XBEGIN` instruction followed by a jump to the start of first the block (`call` in this example). At the end of each block, the T-SGX compiler inserts two instructions that load the address of the next block into a register and jump to the `springboard.next` block. Code on the springboard then ends the current transaction (`XEND`), begins the next transaction (`XBEGIN`), and jumps to the start of the next block, as indicated by the register value provided by the previous block. Right before the end of execution, `springboard.end` ends the last transaction (`XEND`).

T-SGX also places the transaction abort code on the springboard page. Like the code that transitions between transactions, the abort code also executes outside a transaction. It is thus subject to page faults, and we ensure that it does not access memory outside the springboard. With this code layout, the only enclave page for which access could possibly

```
springboard:
springboard.next:
  xend
springboard.begin:
  xbegin springboard.abort
  jmpq   *%r15

springboard.end:
  xend
  jmpq   *%r15

springboard.abort:
  # (abort handler code)
  ...
  # resume execution
  jmp    springboard.begin
```

```
# entry point to the function wrapper
entry_point:
  leaq  EB.start(%rip), %r15
  jmp   springboard.begin
EB.start:
  # (load parameters)
  call   _function
  # (save return value)
  leaq  EB.end(%rip), %r15
  jmp   springboard.end
EB.end:
  ...
  enclu[EEXIT]

# transformed function
_function:
  subq $40, %rsp
  ...
  leaq EB.1(%rip), %r15
  jmp  springboard.next
EB.1:
  ...
  leaq EB.2(%rip), %r15
  jmp  springboard.next
EB.2:
  ...
```

Figure 4.4: Transaction transition code on the springboard and at the end of each execution block (denoted EB).

result in a page fault is the springboard. This could happen (a) at the transaction transition (`springboard.next`, `springboard.begin`), and at `springboard.end` in Figure 4.4 and (b) in the transaction abort code (`springboard.abort`).

Example. Figure 4.5 shows how a host program and an OS interact with an enclave program secured by T-SGX.

1. The host program uses the SGX `EENTER` instruction to call a function inside the enclave.

2. `EENTER` transfers control to the enclave's *springboard*. The springboard starts the first transaction and jumps to the first execution block. As execution blocks complete and jump back to the springboard, the springboard completes and initiates transactions and jumps to subsequent execution blocks. While these execution blocks may be distributed over many memory pages, only the springboard contains code that is not wrapped in a transaction.

3. If an exception occurs inside an execution block, the processor transfers control directly to the abort handler whose address is specified at `XBEGIN`.
4. The abort handler determines whether it has to restart the transaction or terminate the enclave program. The operating system will only see exceptions on the springboard page.

4.4.3 Execution Blocks

This section explains how the T-SGX compiler partitions a program into execution blocks that can be executed as transactions. We begin with a simple partitioning scheme that yields correct and functional programs. After that, we introduce various optimization techniques that drastically reduce the overhead of the simple scheme.

T-SGX computes the control flow graph of the program and tests for each basic block if it satisfies the transaction limits imposed by TSX and an execution time bound we establish. In particular, T-SGX makes a conservative estimate of the write and read sets of the basic blocks with respect to a cache model, as explained in §4.4.4. We approximate execution time by counting the number of instructions in the basic block. Most basic blocks satisfy the two constraints. The remaining basic blocks are split by T-SGX into smaller blocks until all split blocks satisfy the transaction constraints. The resulting set of blocks is the partitioning into execution blocks under the basic scheme.

4.4.4 Transaction Constraints

TSX imposes strict bounds on the read and write sets of each transaction. The write set must fit into the L1 data cache. That is, the L1 data cache must be able to hold all memory writes of a transaction.

For example, on Skylake processors, the L1 data cache has a size of 32 kB. It is 8-way set associative with 64-byte cache lines. We can visualize this cache as eight copies (ways) of a 4 kB page partitioned into 64 slots of size 64 bytes (see Figure 4.6). The 64-byte cache

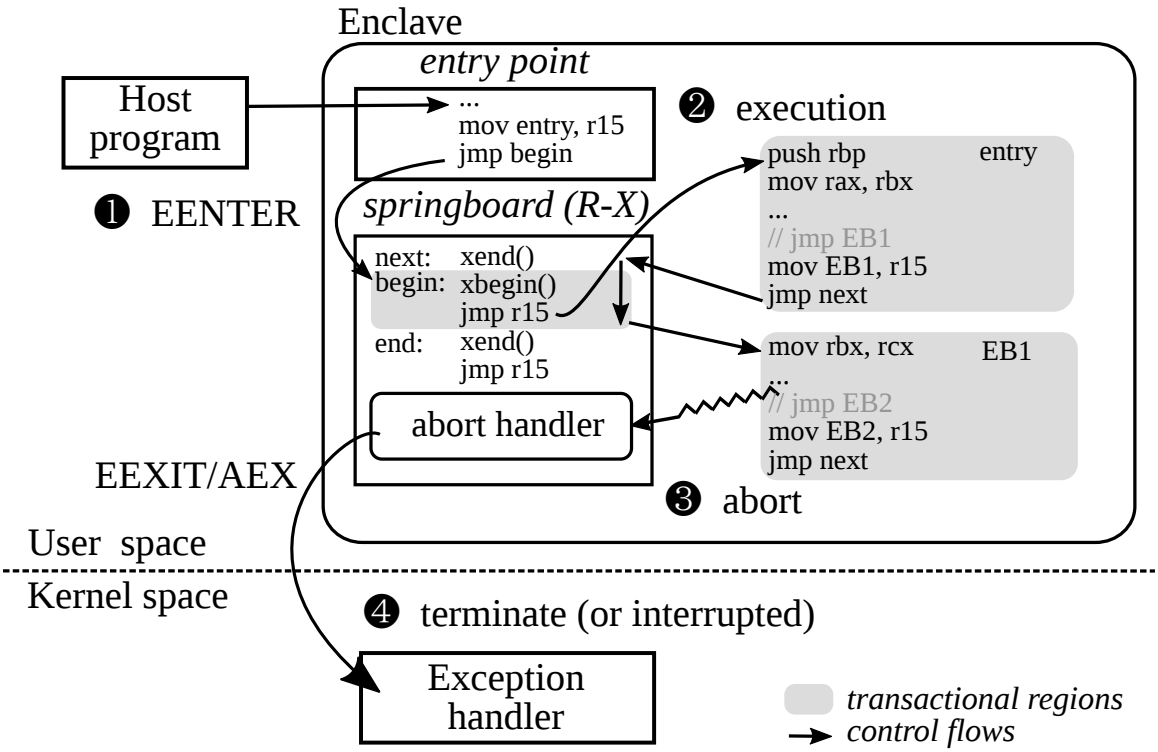

Figure 4.5: Overall procedure of T-SGX: ❶ A host program calls an enclave program. ❷ Enclave execution is managed by the springboard that jumps into execution blocks scattered across multiple pages. The execution blocks jump back to the springboard when they are successfully executed. ❸ When an exception occurs in a execution block, control goes directly to the abort handler on the springboard. ❹ The enclave program either terminates or is interrupted. The OS can only identify the page containing the springboard.

line size is the granularity at which cache space is assigned. Multiple write operations within a single 64-byte aligned 64-byte address range occupy a single cache line. This 64-byte line is mapped to the slot in the L1 cache at the same page offset. A memory access within the 64-byte line causes it to be loaded into one of the eight ways at the corresponding slot in the L1 cache. This will cause the previous content of the of the way to be evicted from the L1 cache.

As the write set of a transaction must be kept in the L1 cache until the end of the transaction, a transaction will fail if its write set includes more than eight cache lines that map to a single slot. The T-SGX compiler uses this cache condition to determine if the write set of an execution block is too large. If it is, it will split the execution block into smaller units (as explained above).

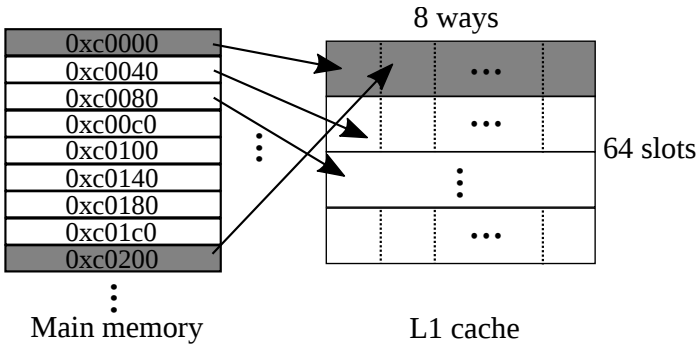

Figure 4.6: Mapping of memory addresses to L1 cache slots.

Since the exact addresses of memory write operations may not be known at compile time, we have to use a conservative approximation. That is, given uncertainty about the addresses of memory accesses at run time, we assume the worst possibility. This may cause T-SGX to split the program into unnecessarily small execution blocks. However, it guarantees that the write set will fit into the L1 cache.

More precisely, we distinguish between three types of memory accesses. First, addresses that are known completely at compile time can be mapped directly to a cache slot. Second, memory accesses given by an unknown base address and a known fixed offset (e.g., `rsp+8`, `rsp+16`) are grouped by the base pointer. We compute the maximum number of occupied ways separately for each group (base pointer) and add the maxima over all groups. Third, we model an access whose address is completely unknown as occupying one way of every slot. Finally, we add the largest way-count from each of the three cases to obtain an upper bound on the L1 requirements of the execution block.

We use a similar strategy to analyze the read set of execution blocks, and we count instructions as a proxy for execution time.

4.4.5 Optimization Techniques

An empty transaction with `XBEGIN` and `XEND` costs about 200 cycles. This can have a significant performance impact on the end-to-end application run time (see §4.7). The simple, basic-block-based partitioning uses basic block boundaries as the default place

```
# TSX Basic
EB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq 32(%rsp)
  leaq loop.header(%rip), %r15
  jmpq springboard.next
loop.header:
  ...
  cmpq $100, 32(%rsp)
  leaq loop.body(%rip), %r15
  jbe  springboard.next
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

```
# T-SGX
EB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq 32(%rsp)
loop.header:
  ...
  cmpq $100, 32(%rsp)
  jbe  loop.body
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

Figure 4.7: An example of the loop optimization.

to begin and end transactions. However, it is often possible to place transactions around larger units of program execution, such as loops or functions. This subsection describes optimization techniques that follow this strategy and that can remove most of the overhead of the simple partitioning. Note that these optimizations do not interfere with OS scheduling, since interrupts cause transactions to be aborted (§2.5).

Loops. Simple partitioning places transactions inside the loop body. That is, every iteration of even the simplest loops (*e.g.*, `memcpy()`) is at least one separate transaction. Our first optimization technique is to pull the transaction out of the loop where possible. Rather than creating a transaction for every loop iteration, we create a single transaction for the entire loop execution.

The main difficulty is to determine the write set of a loop. In general, this is not a tractable problem. However, in practice, many loops have simple relationships between the iteration number and the addresses of memory accesses in that iteration. For example, we frequently observe a simple linear relationship. That is, during the k-th iteration, the loop will access address $a + b * k$, where a and b are constants known at compile time. We use data-flow analysis to determine such relationships.

Given the write set of the loop, we perform the tests of §4.4.4 to determine if the optimization can be applied. If the test fails because the number of loop iterations is too large or unknown, we can still apply the optimization by partially unrolling the loop. For example, if for up to 100 iterations the write set of the loop fits into the L1 cache, we place a transaction around every 100 iterations of the loop. This allows us to amortize the transaction cost over possibly many loop iterations (see Figure 4.7).

Functions and if-statements. This optimization attempts to merge all execution blocks within a function into a single execution block covering the entire function. We attempt to compute the write and read sets and instruction count for the entire function. If the function is complicated (*e.g.*, contains loops), this may not succeed, and we do not optimize the function. If we can obtain the read and write sets and the instruction count and if they pass the tests of §4.4.4 then we merge the entire function into a single execution block.

Similarly, if we can determine the read and write sets of if-then-else statements and if they meet the conditions of §4.4.4, we merge the if-then-else statement into a single execution block.

4.4.6 Abort Sequence

If a transaction fails, TSX will transfer control to the abort address specified in the `XBEGIN` instruction. T-SGX places this address on the springboard.

A simple version of the abort code restarts the transaction unconditionally until it succeeds. The appeal of this design lies in its simplicity. The abort code is stateless and only a few instructions long. However, if a transaction has to access a page that has been unmapped, this design will restart the transaction indefinitely, which is not an optimal defense strategy (§4.6).

The alternative is for the abort code to monitor transaction aborts for signs of attacks and to stop program execution if an attack is detected. Lacking hardware support for distinguishing between page faults and regular interrupts as the cause of a transaction abort,

we use the following criterion. *If a transaction aborts more than n times, the abort code will terminate program execution,* where n is a parameter that must be chosen such that the likelihood of seeing n consecutive transaction aborts due to benign causes under normal operation is very low. Based on the analysis of §4.7.2, we set $n = 10$. The controlled-channel attack [3] requires millions of page faults to obtain sensitive information, so that 10 would be a reasonable threshold to defeat it.

Aborting execution when an unmapped page is detected may leak to the attacker that the enclave was trying to access this page. However, as explained in §4.6, this strategy ensures that the attacker does not learn anything else through the page fault channel.

An implementation difficulty arises from the fact that the attack detection code is not stateless, as it has to count the number of times a transaction is aborted. In order to maintain the important springboard property that the only memory accesses of springboard code outside transactions are execute accesses to the springboard page, we store the counter in a CPU register that we reserve in the compiler.

4.4.7 Preventing Reruns

The decision to run the enclave should be made by its owner and not by the attacker. This can be easily enforced by having the enclave code wait for a cryptographically secured authorization before it accesses sensitive data. We make this authorization an optional part of T-SGX.

The attack model of [3] assumes that the victim runs only once. Furthermore, the use model described in [3] (a remote user controlling a SGX-protected Haven instance or VM in the cloud via a remote desktop protocol) effectively includes an authorization to run (the remote user's commands) that is cryptographically secured (through the remote desktop protocol).

4.4.8 External calls

T-SGX places an `XEND` before calls from the enclave into the untrusted part of the address space (an `EEXIT` instruction). Similarly, T-SGX places a `XBEGIN` instruction close to the enclave entry points specified in the SGX Thread Control Structures (TCSs).

4.4.9 Illegal instructions

The T-SGX compiler ensures that no instructions that are illegal under SGX or TSX are generated for an enclave binary. This is unproblematic, as those instructions are not necessary to generate regular application code.

4.5 Implementation

We have built a prototype of T-SGX based on the LLVM compiler. Our prototype produces T-SGX-enabled binaries that can be run in an enclave just like the original binary. Our prototype can handle arbitrary C and C++ code.

The main part of our prototype is integrated into the back-end of LLVM. It starts by performing the analysis described in §4.4 based on the basic blocks produced by LLVM. After that, it modifies the instruction sequence as it is being emitted by LLVM. In particular, it places two instructions (to load the address of the next execution block into the `r15` register and to jump to the springboard) at the end of each execution block. In the case of 64-bit code, we reserve the `r15` register for this purpose (*i.e.*, to communicate the address of the next block to the springboard). Jump and call instructions (including indirect jumps and calls) are also made to jump to the springboard with the destination address loaded into `r15`.

As TSX uses the `rax` register, we reserve a second register to save the value of `rax` at the end of each execution block that writes to `rax` and to restore it at the beginning of each execution block that reads `rax` before writing to it.

Our prototype also includes a plugin to the LLVM front-end that injects a function

wrapper for each exported function into the LLVM intermediate representation. The entire prototype consists of 4,110 lines of C++ code.

4.6 Security Analysis

For T-SGX-based enclave programs, the attacker can only observe page fault locations (faulting addresses) on (a) the springboard page and (b) the unsecured pages containing function wrappers for the external enclave entry points. The latter can be ignored, as they do not access sensitive data.

Furthermore, as shown below, the attacker cannot use page faults to obtain deterministic notification when enclave execution accesses the springboard. These two properties of T-SGX disable the two main uses of the page-fault channel in the attacks of [3] and [33]: (a) leaking page numbers of memory accesses and (b) giving the attacker synchronization points that allow him/her to track the the victim's execution. An example of the latter is the strategic unmapping of code pages in [3] such that, every time the victim calls a function that accesses sensitive data (looking up a word in the Hunspell hash table, rendering a character, decoding an 8×8 pixel block of an image), a page fault interrupts (stops) the victim and invokes the attacker, who can then advance him/her state machine and update the set of unmapped pages. By blocking these two mechanisms, T-SGX effectively protects enclave programs against the known page-fault-channel-based attacks.

For the full-strength T-SGX variant that requires the enclave-owner's consent to run the enclave program (§4.4.7) and that aborts enclave execution as soon as a page fault is detected (§4.4.6), a stronger statement can be made: *The attacker will learn at most one page access by the victim.* Recall that the attacks of [3] required millions of page faults.

Given a reliable attack detection mechanism, the argument is straightforward. The first time one of the victim's transactions aborts, T-SGX will detect an attack and stop the execution. As the attacker will not be able to run the victim again, he/she cannot observe more than the one page access that caused the springboard to abort the execution. Whether

the attack detection described in §4.4.6 is sufficiently robust is arguable.

Attack detection. For enclave execution, it is reasonable to require that the enclave encounters no unexpected page faults. Thus, attack detection reduces to detecting page faults for T-SGX-secured pages. The problem would be trivial if the TSX hardware would distinguish between page faults and interrupts in the `eax` value provided to the abort handler. Lacking such hardware support, our abort handler declares an attack after a small number of consecutive transaction failures. This approximation is motivated by our evaluation (§4.7): Transactions tend to be short (1,000 to 2,000 cycles), and we have never observed more than three consecutive transaction aborts or false positives.

Restarting the transaction several times before aborting enclave execution could, in principle, give the attacker an opportunity to observe that the page has been accessed and to make the page accessible before the springboard terminates enclave execution. However, it appears that the attacker would have to rely mainly on other mechanisms (beyond the page-fault channel) to detect that the page had been accessed. In other words, while we cannot exclude the possibility that the attacker could gain more information, it appears that he/she would have to rely primarily on powerful mechanisms beyond the page-fault channel (e.g., cache side channels), which is not the focus of T-SGX.

Attacks on the Springboard. We noted above that the attacker cannot obtain deterministic notification of springboard accesses through the page-fault channel. More precisely, the attacker cannot force page faults on springboard accesses.

Before execution of sensitive enclave code starts, the springboard must be mapped and accessed since any sensitive code is called from the springboard. The attacker can, of course, unmap the springboard in the page tables at any time. However, accesses to springboard will continue to succeed (without causing page faults) as long as the springboard's mapping is in the TLB. Furthermore, this mapping is unlikely to be evicted quickly from the TLB, as the springboard is accessed very frequently (§4.7).

All reliable methods for removing the springboard's mapping from the TLB (*e.g.*,

flushing the TLB) require the attacker to run code on the enclave's core or send an inter-processor interrupt (IPI) to the core, interrupting the enclave execution eventually. The key observation is that, by construction of T-SGX, the instruction pointer value at which enclave execution will resume is on the springboard. Thus, if the attacker wants enclave execution to proceed, he will have to map the springboard in the page tables before resuming the enclave (`ERESUME`). The instruction at which enclave execution resumes will be on the springboard, and its execution will establish a new TLB entry for the springboard, which sets the attack back to the beginning.

In summary, while the attacker can interrupt enclave execution asynchronously, he cannot use the page-fault channel to obtain deterministic notification of accesses to the springboard.

4.7 Evaluation

We evaluate T-SGX by answering the following questions.

- How general is the T-SGX approach? Can this approach be applied to a wide range of legacy real world applications without manual effort?
- What are the performance characteristics of T-SGX-based programs?
- What is the performance impact of running multiple instances of T-SGX-based applications simultaneously?

Experimental setup. The experiments were conducted on a generic PC with a Supermicro X11SSQ motherboard, an Intel Core i7-6700K 4 GHz (Skylake) CPU, and 64 GB of RAM. The machine ran Windows 10 Pro. We disabled hyperthreading because avoiding cache-timing attacks in the public cloud is recommended.

Target applications. We evaluate T-SGX by using the programs in the nbench benchmark suite and the three applications that were used by Xu *et al*. [3] to demonstrate the controlled-channel attack. Table 4.1 describes each program in detail, including source code size, description, and binary code size before and after applying T-SGX. The applications are

Table 4.1: Benchmark programs (top) and applications (bottom) used to evaluate T-SGX.

Application	LoC	Description	#exec. blocks	Code segment size		Memory Overhead	Avrage increase bytes per block
				Baseline	T-SGX		
numeric sort	211	Numeric heap sort	23	1,014 B	1,276 B	25.8%	11.4 B
string sort	521	String heap sort	46	2,745 B	3,358 B	22.3%	13.3 B
bitfield	225	Bit operations	24	1,182 B	1,472 B	24.5%	12.1 B
fp emulation	1,396	Floating-point emulation	80	5,636 B	6,467 B	14.7%	10.4 B
fourier	235	Signal processing	20	1,163 B	1,386 B	19.2%	11.2 B
assignment	490	Assignment algorithm	92	3,605 B	4,758 B	32.0%	12.5 B
idea	353	Crypto	36	3,101 B	3,553 B	14.6%	12.6 B
huffman	448	Compression	44	2,960 B	3,648 B	19.2%	15.6 B
neural net	746	Back-propagation network simulation	82	4,183 B	4,941 B	18.1%	9.2 B
lu decomposition	441	Linear equations solving algorithm	62	3,307 B	4,136 B	25.1%	13.4 B
AVERAGE						22.0%	
libjpeg (9a)	34,763	JPEG library	4,557	272,881 B	350,274 B	28.4%	17.0 B
Hunspell (1.5.0)	24,794	Spell checking library	8,641	356,298 B	471,617 B	35.0%	13.3 B
FreeType (2.5.3)	135,528	Font rendering library	12,060	615,862 B	796,105 B	29.3%	14.9 B
AVERAGE						28.6%	

fairly diverse, including cryptography, text processing, and image compression. While the nbench applications are generally small, the other three applications are one to two orders of magnitude larger, with FreeType exceeding 100,000 lines of code.

4.7.1 Application Binaries

This section shows various properties of T-SGX binaries. The main effort in obtaining these binaries lies in porting the applications into the SGX environment. Once we had working SGX applications, no further manual effort was required to apply the T-SGX protections.

After manually adapting the source code of each application to run on SGX (resolving header and linker dependencies), we compiled the code with Clang-Cl, a `cl.exe`-compatible driver mode program for Clang (based on LLVM version 3.7.1). We linked the resulting object files into executables with the Microsoft linker (`link.exe`) version 14.00.23506.0.

We built three versions of each application. (a) The baseline version runs in an SGX enclave without any protection. (b) The TSX-basic version is secured with TSX on SGX, yet without any optimization. (c) The T-SGX version is secured with TSX and optimized as described in §4.4.5. These optimization techniques improve performance without affecting security.

Execution Block Counts and Code Size. We first measure basic statistics of each

application, in particular, static information such as the number of execution blocks and the impact on code size. Table 4.1 shows the results. The reported code sizes are the sizes (in bytes) of the code (`.text`) segments of all object files associated with the application. In the case of nbench where several applications share the same source file (and the same object file), we built per-application versions of nbench by commenting out all source code that did not belong in the application.

The code size increase (excluding the springboard page) from baseline to T-SGX varies between 15% and 32%. These overheads will likely result in somewhat increased pressure on the L1 instruction cache. However, there will be no effect on the application's data accesses. Thus, the increase in the overall memory requirements will be significantly lower than 30%, depending on the application and its inputs.

The table also reports the number of execution blocks in T-SGX. Dividing the size increase by the number of execution blocks reveals an average size increase of 9 to 17 bytes per execution block. This is roughly the space needed to store the two additional instructions that jump to the springboard and the occasional instructions to save and restore `rax` (§4.5).

Distribution of Execution Block Sizes. The next measurement studies the size of execution blocks. Figure 4.8 displays the distribution of the number of instructions per execution block for T-SGX and TSX-basic across the 10 nbench applications.

We observe that the optimizations of §4.4.5 have noticeably shifted the distribution for T-SGX toward larger blocks. The small blocks (containing at most 10 instructions) are mostly the result of (a) non-mergeable cases, such as a block immediately before or after a loop, (b) nested loops, and (c) calls to functions that may not satisfy the cache constraint.

We manually inspected two large outlier blocks (up to 120 instructions). Both correspond to functions that were merged into a single execution block by our optimizations.

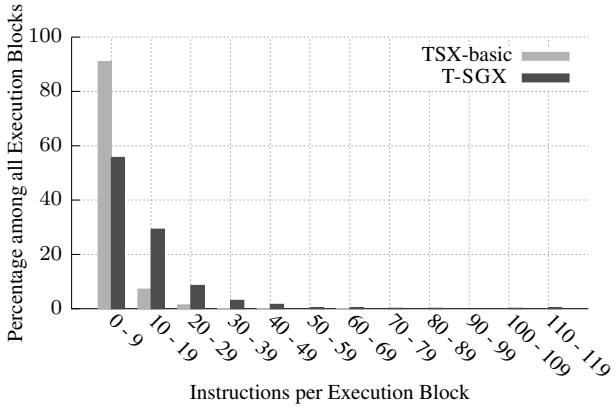

Figure 4.8: Distribution of execution block sizes: The optimizations increase the size of a typical execution block.

4.7.2 Run-time Performance

This section demonstrates the run-time performance of T-SGX. Unless stated otherwise, measurement values are averaged over five runs of nbench and the real applications. For the nbench suite, we ran each program for five second and measured the number of iterations per second.

For jpeglib, we measured how long it takes to decompress a 1220×813 (203,446 bytes) compressed jpeg image. The size of the decoded image is 8,926,740 bytes. The measurement includes the image decompression time but not general startup and initialization.

For Hunspell, we picked the book *Around the World Eighty Days* as the input. The number of words extracted from the book is 63,704. We performed a spell check (using `Hunspell::spell`) on these words with the "en_US" dictionary as a single call into the enclave and measured the total time spent.

We used the same input for FreeType. The number of characters in the book is 375,338. We measured the time required for a single enclave call that renders all these characters (using `FT_Load_Char`).

Run-time Overhead. Table 4.2 displays the run-time of the baseline, TSX-basic, and T-SGX versions of the applications and the associated overheads. We took the numbers for the nbench applications directly from the nbench outputs.

Table 4.2: Run-time overhead of TSX-basic and T-SGX over baseline.

Application	Baseline	TSX-basic	(overhead)	T-SGX	(overhead)
numeric sort	12,682 iter/s	1,149.1 iter/s	(9.1×)	8,390.1 iter/s	(1.5×)
string sort	8,872.3 iter/s	1,991.1 iter/s	(4.1×)	7,218.7 iter/s	(1.2×)
bitfield	516,000,000 iter/s	26,100,000 iter/s	(17.9×)	443,000,000 iter/s	(2.1×)
fp emulation	319.8 iter/s	25.3 iter/s	(11.9×)	146.4 iter/s	(2.2×)
fourier	186,000 iter/s	31,847 iter/s	(5.4×)	98,847 iter/s	(1.9×)
assignment	1,741.9 iter/s	82.6 iter/s	(18.4×)	1,196 iter/s	(1.5×)
idea	3,814.1 iter/s	275.3 iter/s	(13.0×)	3,665.7 iter/s	(1.0×)
huffman	3,264.7 iter/s	162.6 iter/s	(16.6×)	1,641.5 iter/s	(2.0×)
neural net	45.7 iter/s	3.8 iter/s	(11.1×)	27.3 iter/s	(1.7×)
lu decomposition	1,197.6 iter/s	82.4 iter/s	(13.6×)	883.4 iter/s	(1.4×)
GEOMEAN			11.0×		1.5×
libjpeg	6,784.5 kB/s	846.4 kB/s	(8.0×)	4,674.1 kB/s	(1.5×)
Hunspell	176,000 word/s	36,333.3 word/s	(4.9×)	114,000 word/s	(1.6×)
FreeType	37,747.2 char/s	3,047.7 char/s	(12.4×)	28,394.5 char/s	(1.3×)
GEOMEAN			7.8×		1.4×

The overhead of T-SGX ranges from 4% to 118% with a geometric mean of 50%. While this overhead is significant, it does not appear prohibitive. The table also demonstrates the effectiveness of the optimizations of §4.4.5. Without these optimization techniques, the overhead would have been significantly higher (as high as 17.9×). It seems that additional optimization could reduce the overhead even further.

Transaction Properties. Table 4.3 displays the rate at which T-SGX transactions are aborted and the reason, as indicated by the value of the `eax` register at the time of the abort. We observe up to about 500 aborted transactions per second with an `eax` value of 0, indicating an interrupt or exception. This rate follows closely the per-core interrupt arrival rate, which we observed using Windows performance counters.

We observed no transaction abort with `eax` bit 3 set (CAP). This bit is set if an "internal buffer overflowed," which includes the case when a transaction's read or write set does not fit into the corresponding caches. This confirms the conservative nature of our cache model.

We also observed small numbers of aborted transactions with `eax` bits 1 and 2 set (CON). These bits indicate "transaction may succeed on retry" and "another logical processor conflicts with read or write set," respectively.

Table 4.3: Rate and type of transaction aborts for the nbench applications for T-SGX.

	TX	CON	CAP	Abort Rate
numeric sort	481 times/s	20 times/s	0 times/s	0.0020%
string sort	317.3 times/s	5.3 times/s	0 times/s	0.0020%
bitfield	532 times/s	2.3 times/s	0 times/s	0.0120%
fp emulation	314 times/s	8.5 times/s	0 times/s	0.0006%
fourier	221.5 times/s	1.5 times/s	0 times/s	0.0006%
assignment	572.5 times/s	13.5 times/s	0 times/s	0.0020%
idea	707 times/s	9.5 times/s	0 times/s	0.0160%
huffman	530.7 times/s	8 times/s	0 times/s	0.0013%
neural net	485.5 times/s	35.2 times/s	0 times/s	0.0015%
lu decomposition	480 times/s	27.3 times/s	0 times/s	0.0016%

Transaction Duration. Figure 4.9 displays the distribution of transaction duration. We measured the duration of each transaction by instrumenting the springboard code that begins and ends transactions with `rdtsc` instructions. As the rdtsc instruction is illegal under SGX and entering and leaving enclaves add significant noise to the measurement, we performed this experiment by running the applications outside SGX enclaves.

Figure 4.9 shows the distribution of transaction duration for T-SGX and TSX-basic. For TSX-basic, most transactions take less than 1,000 cycles. As a result of the optimizations, a typical transaction for T-SGX takes between 1,000 and 2,000 cycles. Still, the transaction duration is short enough to easily meet the execution time constraint imposed by the interrupt frequency. For example, our 4 GHz processor should be able to complete 2,000-cycle-transactions even for interrupt rates of up to 2 million interrupts per second per core. Such a rate is orders of magnitude higher than the interrupt rates we have observed under normal conditions (thousands of interrupts per second per core). This observation is also consistent with Table 4.3.

Transaction Abort Counts. We study the number of times a transaction aborts before it succeeds. We gather these counts by instrumenting the TSX management code on the springboard.

Table 4.4 displays the distribution of abort counts across the 10 nbench applications. The overwhelming majority of transactions succeeds on the first try. A tiny fraction of

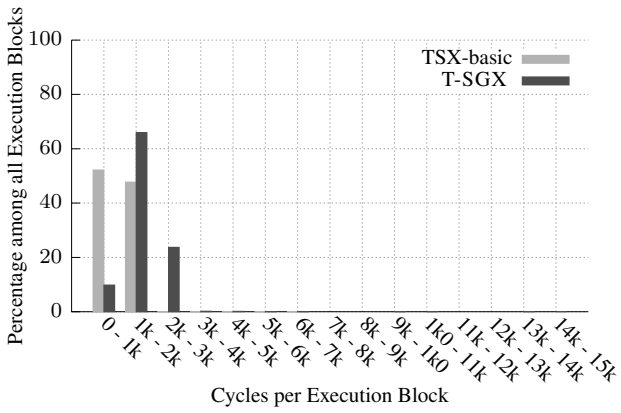

Figure 4.9: Distribution of transaction times: Most transactions take less than 3,000 cycles.

transactions requires up to three retries. After executing many millions of transactions, we observed no transaction requiring more than three retries to complete. This observation can be used as the basis for a mechanism to detect attacks or anomalies.

Table 4.4: Distribution of the number of times a transaction aborts before it succeeds.

Number of aborts	Percentage
0	99.9%
1	$1.7 \cdot 10^{-3}$%
2	$9.8 \cdot 10^{-6}$%
3	$3.7 \cdot 10^{-7}$%
4	0 %

Multiple instances. The next experiment analyzes the performance of multiple T-SGX-protected enclaves running side by side. Our goal is to analyze whether T-SGX scales to multiple protected enclaves.

We measured the running time of baseline and T-SGX for the nbench applications, varying the number of concurrent instances from one to eight. For each measurement, we created n identical enclaves in n separate Windows processes ($n \in \{1, \ldots, 8\}$) running one of the 10 nbench applications for baseline or T-SGX and recorded the timing output of nbench for one of the enclaves. We repeated the measurement for n from 1 to 8, for all 10 nbench applications and for both configurations (baseline and T-SGX).

Figure 4.10 displays the results. The x-axis displays the number of concurrent instances.

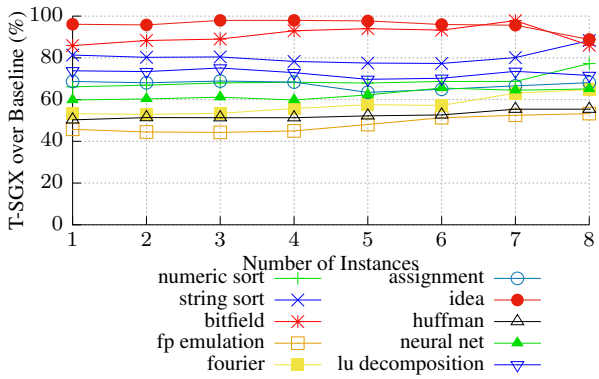

Figure 4.10: Overhead with increasing number of parallel instances. It shows that T-SGX can be scaled for system-wide uses in practice.

Each line corresponds to one nbench application. The y-value is the ratio of the number of iterations per second for T-SGX and for baseline. In other words, it is the inverse of the overhead. All lines are roughly constant, indicating that one can run multiple T-SGX-protected enclaves without affecting the overhead.

4.8 Discussion

In this section we explain limitations of T-SGX and possible approaches to overcome them. Also, we explain other potential attacks against T-SGX and show how we can cope with them.

4.8.1 Limitations

One limitation of T-SGX is that it cannot correctly identify what causes an exception. A transactional execution aborts when an exception has been generated, but it does not let a program know the vector number of the exception (§2.5). T-SGX can distinguish a synchronous exception from an asynchronous exception by repeatedly executing a transactional region, but it cannot know whether the synchronous exception is a page fault, a divide-by-zero error, or something else. This could be a problem because T-SGX may mistake errors in the enclave software for an attack by the OS. To avoid this problem, we plan to develop an application exception handler (§2.2) running inside an enclave that dynamically inspects

the code and execution status to know the exception reason and to fix it to ensure continuous execution.

Second, T-SGX cannot protect libraries without source code because it is a compiler-based approach. This problem could be solved when library developers apply T-SGX to their closed-source libraries. Also, we plan to improve T-SGX to support binary instrumentation.

Third, T-SGX does not support page-level swapping between enclave memory and main memory, as Sanctum [136] does. This limitation would be problematic, especially when T-SGX runs in the public cloud while sharing the limited enclave memory with other processes. One possible solution to this problem is to swap out the whole memory of an enclave program to the main memory. We plan to study the effectiveness of this approach in the future.

Finally, T-SGX cannot support a multithreaded enclave program that wants use TSX for its original purpose: lock elision. However, this does not hurt the program's functionality because lock elision is just an optional feature. Instead, it can use a traditional lock for synchronization between different threads without any problem.

4.8.2 Other Side-channel Attacks

We explain other side channel attacks that could attack T-SGX and discuss possible countermeasures against them.

Cache timing attack. A cache timing attack by a malicious OS is a serious threat because the OS manages the virtual address mapping of every program [130]. To mitigate the threat, an enclave program needs to flush its private cache whenever the OS resumes its execution, but, generally, it cannot obtain such information. Fortunately, with T-SGX, an enclave program can know exactly when it is resumed by the OS such that it only needs to flush its private cache at that point. However, this mitigation is not enough to cope with asynchronous cache timing attacks that use the last-level cache (LLC) [137, 138]. We plan to study how to secure enclave programs from such attacks.

Memory bus snooping attack. A memory bus snooping attack is a hardware-level attack. By monitoring memory bus traffic, a malicious peripheral device can know which memory addresses are currently accessed by a CPU although the memory contents are encrypted by SGX. To prevent such an attack, SGX needs to provide software-level or hardware-level ORAM techniques [135, 139, 124, 13]. Also, we can minimize the number of memory accesses as much as possible by using cache-based [140, 141, 142, 133] or register-based [143, 144] computations.

4.9 Related Work

In this section, we discuss a number of important studies that are related to T-SGX.

Trusted execution environments. Mainstream computing environments are typically very complex. They provide only limited assurance for confidentiality and integrity in light of various attacks such as malware, kernel exploits, and malicious peripherals. Numerous researchers and companies have proposed a variety of TEEs to protect critical data and computations with higher assurance. TEEs typically do not trust the main OS because it could be compromised. Thus, they are implemented in places that even the OS cannot control, such as a trusted hypervisor or hardware. For example, Overshadow [145], NOVA [146], TrustVisor [147], Cloud Terminal [148], InkTag [134], MiniBox [149], and Sego [150] are TEEs based on trusted hypervisors. The basic idea of these systems is to provide isolated memory for each trusted process or module by using nested page tables or the extended page table feature of hardware-based virtualization. Also, all the interactions between a trusted process and the OS (*i.e.*, system calls) have to be managed by the trusted hypervisor.

However, a hypervisor is also software and potentially vulnerable to various attacks [151]. Flicker [152] and TrustVisor [147] use trusted hardware (TPM [114]) and attempt to minimize the complexity of their TEE software. ARM's TrustZone [61], Intel's TXT [115] and SGX [116], and Samsung's KNOX [113] are widely-deployed hardware-based TEEs. Numerous researchers have proposed hardware-based TEE designs, such as TrInc [153],

SICE [154], SecureSwitch [155], OASIS [156], TrustLite [157], and Sanctum [136].

OS attacks against TEEs. Although TEEs are designed to protect user processes from a malicious OS, the latter still has opportunities to attack the processes because they cannot access system resources (*e.g.*, storage, network) without the help of the OS. Iago attacks [158] exploit this limitation. For example, an Iago attack may manipulate the return value (*i.e.*, a virtual address) of the `mmap()` system call to make a target application overwrite a portion of its stack and, thereby, hijack control flow. Since any system call could potentially be exploited for this type of attack, the TEE has to carefully validate the return values of all system calls [134, 159]. The controlled-channel attacks [3, 33] this thesis work focuses on also rely on the fact that the OS manages system memory. Finally, AsyncShock [160] demonstrates that synchronization bugs that are mostly harmless in a traditional environment can allow an adversarial OS to compromise SGX enclaves.

SGX applications. Among the various hardware-based TEEs, Intel SGX recently has been receiving much attention because it is widely deployed (all Intel Skylake CPUs support it) and because it allows developers to use almost the full unprivileged instruction set of the Intel CPU. For example, Haven [127], Graphene-SGX [128, 129], and SCONE [161] are SGX-based platforms to securely run an unmodified application in an untrusted cloud. VC3 [123], M2R [124], and Ohrimenko *et al*. [94] use SGX to perform data analytics, MapReduce computations, and machine learning computations while ensuring confidentiality and integrity. Also, Kim *et al*. [125], S-NFV [126], Pires *et al*. [162], and SecureKeeper [163] show how we can use SGX for securing network services, content-based routing, and distributed computing.

Moat [164] and CONFIDENTIAL [165] design verification methodologies for enclave programs to check whether they are secure. Also, OpenSGX [159] is an emulator for the execution of enclave programs for software development and in-depth debugging and testing. Further, SGX-Shield [14] implements fine-grained address space layout randomization (ASLR) for SGX. Ryoan [166] introduces a distributed two-way sandbox to run untrusted

enclave programs with sensitive user data while preventing possible information leakage.

CHAPTER 5
SECURING SGX PROGRAMS AGAINST SIDE-CHANNEL ATTACKS VIA LOAD-TIME SYNTHESIS

5.1 Introduction

Ensuring *confidential computing* is necessary as many applications tend to be executed in remote and shared environments, e.g., the public cloud. In such environments, users or tenants just specify the performance and characteristics of the hardware they want to lease and the duration, reducing costs and maintenance efforts [167]. However, since the hardware is not only owned and managed by service providers but also shared with other tenants, their sensitive computation and data potentially suffer from information leakage [168].

Hardware-based Trusted Execution Environments (TEEs), such as Intel SGX [62, 122], AMD Secure Encrypted Virtualization (SEV) [169], and ARM TrustZone (TZ) [61], are considered as a promising approach to realize confidential computing. Especially, Intel SGX ensures even underlying system software and hardware cannot compromise the authenticity, confidentiality, and integrity of userspace SGX applications running inside *enclaves*. Leading cloud service providers, such as Microsoft and Google, are developing Software Development Kits (SDKs) and frameworks for confidential computing [170, 171, 172] to support their customers.

Unfortunately, Intel SGX is not a silver bullet for all security problems especially because it does not cover side-channel attacks (SCAs) [173], which are serious threats to shared computing environments that it aims to be deployed. Numerous researchers have shown that SGX is vulnerable to various SCAs including cache-based SCAs [6, 8, 34, 35, 5], page-table-based SCAs [3, 4, 31, 32, 33], and speculative execution SCAs [38, 39], which can infer sensitive control flows or exfiltrate secret data. To avoid such security threats, researchers

Table 5.1: Side-channel attacks against SGX and countermeasures. Hyper-Threading Technology (HT), L1 Terminal Fault (L1TF), and Microarchitectural Data Sampling (MDS).

Attack	Countermeasure
Cache	Cache flushing [18] and cache eviction detection [20]
Page	Page fault detection [17] and huge page [174]
HT	HT disabling [18] and co-location detection [22, 21]
Interrupt	Frequent AEX monitoring [22, 175]
Branch prediction	Branch obfuscation [9]
Speculation	Branch prediction control [176], lfence [177]
L1TF	Cache flushing and HT disabling [18]
MDS	HT disabling [19]

have also proposed software- and/or hardware-based countermeasures or mitigation against individual SCAs, such as cache flushing [18, 19] or eviction detection [20], page fault detection [17], and HT disabling [18, 19] or co-location detection [21, 22] (Table 5.1).

To ensure the security of an SGX program, developers must ensure that it is immune to at least all known, defeated SCAs. One naïve approach to achieve this goal is incorporating all existing countermeasures, but, this does not work in practice because of 1) *deployability*, 2) *overprotection*, and/or 3) *incompatibility* problems.

First, some countermeasures, especially, hardware-assisted ones, highly depend on underlying hardware and system configurations, so deploying them to public cloud environments abstracting configuration details [178] is challenging. For example, Cloak [20] and T-SGX [17] leverage a CPU instruction, TSX [179], to detect cache- and page-table-based SCAs, respectively. However, since only high-end Intel CPUs support TSX (e.g., Xeon and Core i7), they cannot be deployed to a platform with a low-end CPU, such as Pentium Silver. Further, even if a CPU itself supports a required instruction, administrators or system software has a chance to deactivate it via microcode update or control registers. Thus, developers might have to write a *bloated* program considering all feasible combinations of configurations, or compile a program for different configurations whenever they want to execute it in the public cloud, which is time and resource-consuming especially when they want to deploy the program to multiple machines for distributed computing [180, 123].

Second, the overhead of individual countermeasures can be accumulated if developers do not try to eliminate redundant protections during combining them. For example, HT is the fundamental source of cache-based and speculative execution SCAs, and co-location detection [22, 21] can be used to disrupt this channel. However, enabling this technique is a waste of resources if a target platform does not support HT or is patched with a recent microcode update to disable HT within SGX [18, 19]. Unless developers accurately identify the detailed configurations of a target platform in advance, which is difficult to be assumed in the (potentially malicious) public cloud, they cannot identify and cancel out redundant protections.

Third, existing countermeasures are independently developed to prevent each SCA such that they do not consider any potential incompatibilities between each of them. For example, shared-memory-based thread co-location detection [21, 22] cannot directly work with TSX-based mitigation [17, 20] because such conflicting shared-memory accesses can abort TSX without cache-line eviction or page fault. Therefore, developers have to identify conflicting techniques and customize them to protect an SGX program from more than one SCAs (e.g., detecting co-location outside TSX transactions).

In this thesis, we propose PRIDWEN, a framework to *dynamically synthesize SCA-free SGX programs* according to the hardware and system configurations of a target platform. PRIDWEN has a universal loader that securely transforms and loads a given SGX program inside an enclave by using the four components: 1) *configuration prober*, 2) *pass manager*, 3) *program synthesizer*, and 4) *validator*. First, the configuration prober reliably identifies the target platform's hardware and system configurations based on SGX exception handling logic and remote attestation procedure (§5.3.3), which are robust against system software's malicious manipulation. Second, based on the identified configurations, the pass manager determines a subset of instrumentation passes that are necessary to protect an SGX program from all feasible SCAs at the target platform while having low incompatibility and performance problems (§5.3.4). Third, the program synthesizer hardens a given SGX

program with the chosen instrumentation passes. To realize lightweight program synthesis that demands a small amount of memory such that can fit into an enclave, PRIDWEN uses Wasm [60, 181] as its IR. That is, PRIDWEN assumes that SGX programs provided to it are compiled into Wasm. PRIDWEN develops comprehensive instrumentation interfaces to transform such Wasm-based SGX programs both at Wasm IR and native code levels, and eventually produces a final executable (§5.3.5). Lastly, the validator confirms whether the final executable has been correctly produced to avoid any potential incompatibility and security problems (§5.3.6).

As a case study, we write four passes for PRIDWEN to show its flexibility: T-SGX [17] to prevent a page-fault SCA with a hardware support, Varys [22] to mitigate a page-fault attack in a software-only manner, QSpectre [177] to mitigate the Spectre attack, and fine-grained Address Space Layout Randomization (ASLR) [14] as a general-purpose mitigation, and show how PRIDWEN can selectively incorporate them according to hardware configurations.

Our evaluation shows the acceptable performance overhead and faithfulness of PRIDWEN. PRIDWEN synthesized programs within 0.5 s while temporarily spending up to 25 MiB of enclave memory only during a synthesization. Compared to software-only, redundant mitigation, PRIDWEN with hardware-assisted, non-redundant mitigation improved the runtime performance by up to 2×. The runtime overhead of synthesized and instrumented programs was around 1–6× depending on applications and configurations, which is acceptable as a recent study shows that Wasm could have up to 3× slowdown [182] without adopting any defenses. Also, we confirmed that PRIDWEN faithfully compiled and ran all 73 programs from the official Wasm specification test suite [183] without notable problems.

We will make the design and implementation of PRIDWEN publicly available as an open-source project, allowing communities to use, test, and contribute. The openness of PRIDWEN should help not only improving its security, but also constructing a foundation for the SCA-resistant SGX ecosystem.

In summary, this thesis work makes the following contributions:

- **Hardware-aware load-time software hardening.** To the best of our knowledge, PRIDWEN is the first framework that can dynamically synthesize hardened software within an enclave by choosing optimal (robust and efficient) hardware-assisted or software-only security mechanisms according to underlying hardware supports.

- **Comprehensive Wasm instrumentation.** PRIDWEN's Wasm instrumentation is comprehensive: it can instrument Wasm both at IR and native code, unlike related studies that only consider IRs [184, 185]. Native code instrumentation are especially necessary to adopt countermeasure for SCAs dealing with machine code.

- **Reliable hardware probing.** PRIDWEN's exception- and remote-attestation-based hardware probing is robust against malicious system software. What attackers can do is just denying its execution.

- **Instrumentation validation.** PRIDWEN supports validation passes to confirm whether final binaries are generated as expected, enabling secure, functional, and efficient adoption of various countermeasures.

5.2 Threat Model

Our threat model is similar to the threat models of other SGX-related studies, such as [3, 17, 22]. Our Trusted Computing Base (TCB) consists of an SGX enclave provided by an Intel CPU and code running in the enclave including the PRIDWEN loader (i.e., prober, pass manager, synthesizer, and validator) and a target Wasm binary prepared by a developer. We assume that the developer uses remote attestation to confirm the validity of the CPU and the PRIDWEN loader, and establishes a secure channel with the loader to securely transmit his/her binaries. Any threats due to the potential vulnerabilities of the CPU and the code running in an enclave are out of our consideration.

We assume that adversaries have already compromised the underlying systems software to attack the PRIDWEN loader and the target binary it will run. What the adversaries can

leverage to attack them via side channels include 1) the page table of an enclave program to intentionally trigger page faults and monitor access and dirty flags; 2) the cache status of an enclave program to monitor it before, after, and during the execution of an enclave; 3) the assignment of an enclave program to logical cores (i.e., processor affinity) to concurrently monitor its behaviors through a co-located attack thread; 4) the (frequent) interrupts to an enclave program to increase the accuracy of side-channel attacks; and 5) other speculative side channels (i.e., indirect branch predictor, L1TF, and MDS). Also, we assume that the adversaries try to prevent the PRIDWEN hardware prober from identifying correct hardware information to make it use weaker or software-only mitigation.

5.3 PRIDWEN Design

This section outlines the design of PRIDWEN, including goals, an overview, and the detail of each key component.

5.3.1 Goals

❶ **Adaptivity.** A goal of PRIDWEN is to be adaptive to the hardware configuration of the underlying system. Such adaptivity allows PRIDWEN to take advantage of the hardware capabilities fully and therefore to provide a similar level of protection against SCAs in distinct environments while minimizing the runtime overheads.

❷ **Extensibility.** Another goal of PRIDWEN is being extensible enough to support not only existing but new mitigation techniques against SCAs. Also, the PRIDWEN aims to support combinations of multiple mitigation techniques. The extensibility of PRIDWEN should also allow for the smooth integration of unsupported mitigation techniques.

❸ **Transparency.** The last goal of PRIDWEN is to transparently apply various mitigation techniques while doing not affect the functionality of a target program. Except for potential slowdown and additional memory usage, PRIDWEN should maintain the primary behavior of the program.

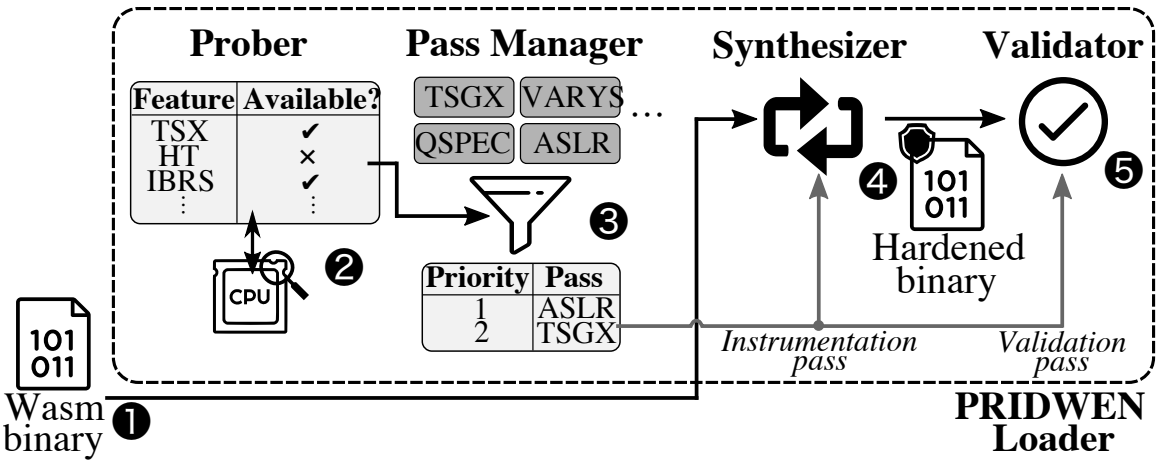

Figure 5.1: Overview of PRIDWEN. ❶ A developer compiles his/her program into a Wasm binary and transmits it to the loader via a secure channel. ❷ PRIDWEN probes the current CPU configurations. In this example, it founds that the CPU enables TSX and IBRS while disabling HT. ❸ PRIDWEN selects and prioritizes mitigation passes. Here, it chooses T-SGX and then ASLR because the CPU enables TSX (leveraged to detect page-fault attacks) and IBRS (mitigating some Spectre variants), and disables HT (no HT-required attacks). ❹ PRIDWEN synthesizes a native binary based on the Wasm binary while hardening it with the chosen passes. ❺ PRIDWEN validates whether the final native binary is correct according to given validation passes, and then executes it.

5.3.2 Overview

Figure 5.1 shows an overview of PRIDWEN. The core of PRIDWEN is an in-enclave loader that implements four key ideas with corresponding components: *user-mode hardware probing* with the prober, *optimal pass selection* with the pass manager, *load-time program synthesis* with the synthesizer, and *post-synthesis validation* with the validator. Given that each countermeasure may depend on specific hardware features, the prober interacts with the CPU and dynamically determines the availability of these features. Based on the probing results, the pass manager determines an optimal set of countermeasures (i.e., instrumentation passes) and the order of their enforcement. Next, the pass manager informs the selection to the synthesizer and the validator. The synthesizer takes a Wasm binary (via a secure network channel) as an input and compiles it into a native one. During the compilation, the synthesizer hardens the binary with the optimal pass set the pass manager provides. The validator takes the synthesized binary as an input. Then, it performs the binary analysis specified in the validation passes. The analysis ensures the correct enforcement of each

countermeasure and prevents the countermeasures from conflicting each other. Once the binary successfully passes the validation, the loader starts executing the binary.

5.3.3 User-mode Hardware Probing

The goal of hardware probing is to allow the PRIDWEN loader to acquire the specific hardware configurations of the underlying system dynamically. Based on the knowledge of the hardware configurations, the PRIDWEN loader determines the optimal set of mitigation schemes to enforce during the later stages. Figuring out the hardware configurations typically requires interactions with the underlying system. For example, executing the `cpuid` instruction that system software can hook and virtualize, and retrieving privileged registers (i.e., Model-Specific Register (MSR) and control registers). Unfortunately, we cannot rely on these approaches because the system software is not trustworthy in our threat model. Instead, we leverage exception handling and remote attestation to probe hardware configurations inside an enclave.

Exception-based probing. Our user-mode hardware probing relies on the custom exception handler for SGX (§2.2) to test whether an exception is generated when we try to use a specific hardware feature. Systems software cannot suppress exceptions generated during an enclave execution. Thus, to proceed with the execution of an enclave, systems software has to call a custom exception handler inside an enclave that inspects exception information to handle it and proceeds the execution by changing `GPRSGX.RIP`.

Intel CPU's instructions show two different behaviors when they are disabled or not supported: 1) generating a `#UD` exception (e.g., TSX) or 2) being treated as a `nop` instruction (e.g., Memory Protection Extensions (MPX) [26, §17.4]). If executing a disabled or unsupported instruction results in a `#UD` exception, we can use a custom exception handler to detect it as shown in Figure 5.2. If it is treated as `nop`, we have to rely on its semantics to generate other exceptions. For example, to test whether MPX is available, we can setup an arbitrary boundary with `bndmk` and execute either `bndcl` or `bndcu` with out-of-bound

```
#define UD 6 /* Invalid opcode exception */

bool tsx_support = false;
check_tsx_support:
_xbegin();
tsx_support = true;
_xend();
skip_tsx_check:
...

exception_handler:
if (SSA.GPRSGX.EXITINFO.VECTOR == UD &&
    SSA.GPRSGX.RIP == check_tsx_support) {
  GPRSGX.RIP = skip_tsx_check;
}
```

Figure 5.2: Exception-based probing code for TSX. If a CPU does not support TSX, there will be a `#UD` exception that needs to be handled by an in-enclave exception handler to proceed execution (i.e., changing `GPRSGX.RIP`).

ranges. If it results in a bound range exceeded (`#BR`) exception, we can confirm that MPX is available. To test whether Execute Disable (XD) is configured, we can try to execute a data page consisting of ud instructions and check whether it results in a `#GP` (XD is enabled) or a `#UD` (XD is disabled).

Remote attestation for hardware configuration. Remote attestation allows PRIDWEN to accurately determine several hardware configurations, i.e., HT and IBRS. If a remote device turns on HT, an attestation verification report can contain `CONFIGURATION_NEEDED` in the `isvEnclaveQuoteStatus` field since the attestation API version 3 [186, 187]. PRIDWEN could leverage this information to selectively adopt mitigation for preventing hyperthread co-locations [22, 21]. Also, if a remote device does not install the microcode update for indirect branch control mechanisms, a remote attestation protocol will fail with `GROUP_OUT_OF_DATE` [188]. If a developer still wants to run his/her code in such a remote device, he/she can adopt software-based approaches [177] against speculative side-channel attacks.

5.3.4 Pass Manager

For extensibility, we implement a pass manager as a part of the PRIDWEN loader allowing for smooth integration of a pass that supports a new mitigation scheme. More specifically, the manager provides a set of high-level APIs that allows the developers of side-channel

Table 5.2: The APIs for the instrumentation (top) and validation (bottom) support. `CCTX`: `CompilerContext`. `MI`: `MachineInstr`. `MCTX`: `MachineContext`. `MB`: `MachineBasicBlock`.

API	Hooking point
`onFunctionStart(CCTX *c)`	Beginning of a function
`onFunctionEnd(CCTX *c)`	End of a function
`onControlStart(CCTX *c)`	Beginning of a control statement
`onControlEnd(CCTX *c)`	End of a control statement
`onInstrStart(CCTX *c)`	Before a IR-level instruction
`onInstrEnd(CCTX *c)`	After a IR-level instrcution
`onMachineInstrStart(CCTX *c, MI *i)`	Before a native instruction.
`onMachineInstrEnd(CCTX *c, MI *i)`	After a native instrcution
`validate(MCTX *c, MB *b)`	Beginning of a basic block

mitigation schemes to implement their instrumentation and validation passes and plug them into the loader. During the load time, the pass manager maintains a list of plugged-in passes, determining the optimal set of passes for the synthesizer and the validator to execute, and resolving the correct execution order of each selected pass.

Pass APIs. Table 5.2 lists the high-level APIs for implementing instrumentation and validation passes. For instrumentation, we expose all the hooks as APIs. To reflect the structure of a Wasm module, we classify the IR-level hooks into the granularity of functions, controls, and instructions. Each hook can obtain the information about the hooking IR instruction and the current states of compilation via the `CompilerContext` (`CCTX`) data structure. For the native level, the hook should consult the information of the native instruction via the `MachineInstr` (`MI`) data structure. The reason for this is that `CompilerContext` does not track such information (i.e., not necessary for the compilation process to proceed). Similar to instrumentation, we expose the basic-block-level procedure as an API for implementing a validation pass. The procedure can obtain the raw bytes of the basic block from the `MachineBasicBlock` (`MB`) data structure. We also allow the procedure to obtain more high-level information of the corresponding basic block via the `MachineContext` (`MCTX`) data structure.

Pass selection and ordering. When plugging into the PRIDWEN loader, each pass is

associated with a configuration file. The file specifies the type of SCA that it intends to mitigate, hardware features or other passes that it depends on, and a list of passes incompatible with it. Optionally, the file allows a pass to specify soft-dependent passes, which indicates that the pass depends on any of its soft-dependent passes only when such passes are available. During the initialization phase, the pass manager adds all the plugged-in passes into a pass queue. For selecting the optimal set of passes, the pass manager consults the prober about the current hardware configuration. Next, the pass manager checks the dependency of each pass in the queue and drops a pass if its dependent hardware feature is not available. Next, the pass manager checks the type of the side channels that each active pass mitigates. If the pass manager identifies more than one passes targeting the same SCA, it assigns a priority value to each of the passes and retains the one with the highest value. For example, the pass manager assigns a high priority value to an active pass that has lower overhead. This simple rule is sufficient for our current design. Optionally, we can adopt sophisticated rules and optimization algorithms when the number of supported passes increases.

To determine the execution order of active passes, the pass manager builds a dependency graph of all the passes given the dependencies specified in configuration files. Next, the pass manager uses the topological ordering of the graph as the execution order. The pass manager may drop passes if their non-soft dependency does not meet or incompatible passes are in the active pass set. Note that we assume the graph contains no circular dependencies. If the graph does, our current design terminates the execution of the loader. Besides, if all passes are independent, the pass manager uses the order of the passes in the queue as the execution order.

5.3.5 Load-time Synthesis

The high-level goal of load-time synthesis is allowing the PRIDWEN loader to generate a hardened binary from a given program and the optimal set of countermeasures (§5.3.4) based

on the current hardware configuration. For this goal, our design adopts a Wasm binary as input to the loader. Moreover, the loader supports the compilation chain, including parsing and compilation, of the Wasm binary. We also extend the compilation chain to support both IR- and native-level instrumentation that is flexible enough to implement various types of SCA mitigation schemes.

Parsing. The parsing performs standard decoding on a Wasm binary and converts it into the IR of a Wasm module. During the decoding, the process also validates the format of the binary with several checks (e.g., type checking of functions) to guarantee the binary follows the specification. Because of validations, any modification to the binary before parsing can easily result in a rejection by the loader. For example, inserting an instruction that causes the inconsistency on the stack machine renders the binary to be invalid.

Compilation. For generating a native binary given an IR of the Wasm module, the synthesizer performs a single-pass compilation over each function in the module (similar to the baseline compilation of SpiderMonkey [60] and V8 [189]). During the compilation of a function, the synthesizer virtually executes each instruction based on the execution model of the Wasm stack machine and generates the corresponding native code. More specifically, the synthesizer maintains the operand stack internally as the virtual execution proceeds (i.e., manipulating the stack based on the operations of each virtually executed instruction). Based on the values on the stack and the type of an executed instruction, the synthesizer generates one or more native instructions. The synthesizer also keeps track of the metadata about each value on the operand stack. Such metadata includes the data type (e.g., `i32`, `i64`, `f32`, or `f64`) and the actual position (e.g., a physical register, a location to the physical stack frame of the function, or a constant value stored in the operand stack). The metadata of the values on the operand stack is necessary for generating correct native code and for allowing the synthesizer to perform type-checking when consuming each value as the operands of the instruction. For determining the position of a value, the synthesizer adopts the linear scan algorithm for register allocation [190]. Optionally, the synthesizer can adopt more

sophisticated algorithms such as graph coloring [191] to improve the efficiency of register allocation.

In addition to the operand stack, the synthesizer maintains a control stack that keeps track of the control flow of the function. Pushing a value to the control stack indicates the function initiates a new control statement (e.g., `block`, `if`, and `loop` instructions). On the other hand, popping a value from the stack implies reaching the end of the current statement (e.g., an `end` instruction). The control stack provides sufficient information, allowing the synthesizer to resolve the target of a branch (e.g., a `br` instruction). The reason for this is that Wasm permits only a structured control flow; i.e., the target of a branch can only be the beginning of or the end of a previously initiated control statement.

After finishing the native code generation of all the functions, the synthesizer performs relocation. The process patches all the unresolved address values in native instructions such as `call` and the instructions for memory accesses. Note that branch instructions besides `call` that are corresponding to IR-level branches do not require relocation because their destinations are resolved during the phase of native code generation.

Instrumentation. To support flexible instrumentation, we extend the design of the compilation to provide hooks at both IR- and native-level. For IR-level hooks, we place them both before and after the position that synthesizer processes an IR instruction. Both the hooks before and after the instruction support code insertion, modification, and deletion. However, only the hooks before the instruction can manipulate the instruction itself. For each hook, we provide sufficient information about the corresponding instruction and the states of the compilation at the given point, such as the operand and control stacks, that enable program analysis and instrumentation. Besides, all the instrumentation made via the hooks also updates the states of the compilation accordingly.

The synthesizer may generate more than one native instructions for a single IR instruction. In such a scenario, the IR-level hooks are not sufficient to support mitigation schemes that require the information about native instructions. For this reason, we provide similar hooks

at the native level (i.e., surrounding the generation of native instructions). To allow for inserting or modifying a native instruction that requires relocation, we provide an option to create instructions with symbols. A symbol refers to a target location that allows the synthesizer to recognize and resolve it during the relocation phase.

Despite being flexibility, all the instrumentations made via hooks should respect the state of compilation. More concretely, an arbitrary manipulation that affects the states such as the operand or control stack may result in a compilation failure or unexpected behavior during the runtime of the instrumented binary. For the case of the compilation failure, an example is inserting an IR instruction that pushes values to the operand stack, causing a type checking failure when the next instruction consumes the values. For the case of the unexpected runtime behavior, one example is inserting a control instruction that results in modifying the target of the following branch instructions. Another example is inserting a native instruction that overwrites a value stored in an in-use register or memory location. The PRIDWEN validator checks whether a compiled binary suffers from the above-mentioned errors (§5.3.6).

System call support. Wasm is designed to run in a sandboxed environment within a web browser. Thus, a Wasm binary by default does not use any system calls. Invoking a system call from a Wasm requires additional runtime support. For example, the binary makes a call to an imported function that is outside of the binary. PRIDWEN provides runtime support that is compatible with an Emscripten-compiled [192] Wasm binary, which consists of wrapper functions for all the system calls. In addition, PRIDWEN implements OCall-based system call interfaces for these wrapper functions, which are similar to those of [14, 17, 193]. Recently, Mozilla starts to standardize WebAssembly System Interface (WASI) to run Wasm outside a web browser [194]. We plan to extend PRIDWEN to make it compatible with WASI.

5.3.6 Post-synthesis Validation

Except for cases that result in compilation failures, the synthesizer does not assume how an instrumentation pass manipulates a binary. On the one hand, the flexibility of instrumentation provides the potential for implementing various types of mitigation schemes. On the other hand, such flexibility indicates that an instrumentation pass can arbitrarily modify the binary. Such modifications can potentially affect the other passes (e.g., creating security holes or invalidating the expected protection) or break the binary itself (e.g., causing unexpected runtime behavior). To avoid such cases, we add the validator to the PRIDWEN loader that enables post-synthesis validation. Post-synthesis validation aims for supporting static analysis over a PRIDWEN-synthesized binary (i.e., the binary generated by the synthesizer) before its execution. Unlike typical binary analysis that assumes a stripped binary, post-synthesis validation enables more sophisticated analysis by taking advantage of the metadata (e.g., the control-flow information) that the synthesizer provides.

Validation passes. An analysis of post-synthesis validation takes the form of a validation pass. Based on the control-flow information, the validator executes the pass at the basic-block level. More specifically, the validator iterates all functions in the binary and, at the beginning of each basic block, invokes a procedure that the pass implements. The procedure performs a series of checks based on the content of the basic block (i.e., raw bytes). For example, the procedure determines whether specific instrumentation is applied based on pattern matching. If any of the checks fail, the procedure rejects the binary. Optionally, the procedure can utilize other metadata such as the original IR instructions that map to the basic block to facilitate the analysis beyond binary scanning.

5.4 Case Study

In this section, we examine four SCA mitigation schemes (ASLR, Varys, T-SGX, and QSpectre) that demonstrate how we support these schemes on top of PRIDWEN. These

Table 5.3: Attack surfaces and software-only or hardware-assisted mitigation schemes PRIDWEN implements. CPUs with recent microcode update do not have some of the attack surfaces.

Attack surface	**Mitigation**	
	SW-only	**HW-assisted**
Cache timing	Interrupt (**Varys**)	Cache flushing (microcode)
Page fault	Interrupt (**Varys**)	**T-SGX**
HT	Co-location (**Varys**)	HT disabling (microcode)
Speculative execution	**QSpectre**	IBRS (microcode)
Static layout	**ASLR**	N/A

schemes cover five different SCA surfaces: cache timing, page fault, HT, speculative execution, and static layout (Table 5.3), in which three of them (cache, HT, and speculative execution) are closed by recent microcode update [18, 19]. That is, some of them are redundant depending on hardware configurations. It is worth noting that the primary goal of our case studies is to show that PRIDWEN is general enough to support various mechanisms. It is not about faithfully implementing and/or improving them.

5.4.1 Fine-grained ASLR

In addition to mitigating memory corruption vulnerabilities, fine-grained ASLR that diversifies the runtime behavior of a binary offers general protection against SCAs [14]. However, enabling fine-grained ASLR in an SGX enclave is not straightforward because the enclave relies on the system software to set up the memory layout during the initialization phase. As a result, the memory layout, even if it is randomized, of the enclave is known by an attacker who controls the system software. Fortunately, PRIDWEN does not suffer from this problem because it dynamically generates final binaries.

Instrumentation. Our instrumentation pass of fine-grained ASLR randomizes the locations of each basic block. Similar to SGX-Shield, our pass inserts a `jmp` instruction at the end of every basic block. More specifically, the pass uses the `onControlStart` and `onControlEnd` APIs (Table 5.2) to identify the structure of a basic block. Next, the pass inserts a `jmp` with

a symbol that points to the successor of the basic block if it does not end with a `jmp`. The pass also updates the targets of all the other branches that point to a basic block accordingly by using `onMachineInstrEnd`. After the code generation phase, the synthesizer shuffles the placement of each basic block if the ASLR pass is enabled. During the relocation, the symbols that the pass generates allow the synthesizer to resolve the target of each branch, pointing to a basic block at a randomized location.

Validation. To validate the instrumentation done by the pass, we implement a validation pass performing the following checks: 1) whether a basic block terminates with a `jmp` and 2) whether each branch points to the correct target (based on the control-flow information).

5.4.2 T-SGX

T-SGX [17] is a compiler-based mitigation scheme targets specifically at page-fault SCAs. The key idea of T-SGX is to execute an enclave inside TSX transactions. As a result, all page faults occur during the execution are suppressed (i.e., not delivered to the system software). However, one challenge of utilizing TSX is that each transaction can execute only a limited number of instructions (bounded by the capacity of L1 cache and the frequency of interrupts). To overcome the challenge, T-SGX splits a program binary into multiple code blocks (consisting of several basic blocks) and places each block into an individual transaction. To determine the splitting points, T-SGX performs static analysis that estimates both cache usage and execution time of each basic block. Based on the estimation, T-SGX composes code blocks concerning the cache and time constraints. Another challenge is that a transaction does not include the instructions for initiating (`xbegin`) and terminating (`xend`) the transaction. To solve this problem, T-SGX adopts a springboard design. Instead of surrounding every code block with `xbegin` and `xend`, T-SGX modifies the target of branch instructions in each code block to a springboard. The springboard includes a piece of code that manages the transitions between consecutive transactions. Note that as transactions also abort upon interrupts, T-SGX is also able to detect interrupt-based SCAs.

Instrumentation. The T-SGX pass implements cache usage and execution time analyzers for native instructions with the `onMachineInstrEnd` API. Based on the analysis results, the pass determines the scope of a code block. Next, the pass replaces branch instructions at the end of the block with the instructions for jumping to the springboard (i.e., a `lea` for saving the address of the next code block and a `jmp` to the springboard). Similar to the pass for fine-grained ASLR, the T-SGX pass identifies basic blocks in the binary by using the `onControlStart` and `onControlEnd` APIs. For the springboard support, the pass places the springboard code before the entry function (e.g., `main`) of the binary by using `onFunctionStart`.

Validation. The validation pass for T-SGX checks 1) the existence of the springboard, 2) the existence of the instructions to jump into the springboard at the end of every code block, and 3) the target of the instructions correctly points to the springboard. Optionally, the pass can re-analyze cache usage and execution time to ensure the correctness of the code splitting.

5.4.3 Varys

Varys is software-based mitigation against high-frequency, interrupt- and HT-based SCAs.

High-frequency AEX detection. Varys consists of two key designs: interrupt detection and interrupt frequency estimation. Similar to exceptions, an interrupt causes an AEX and an update to the SSA. As a result, Varys detects the occurrence of an interrupt by periodically polling the value of SSA. To estimate the interrupt frequency, Varys instruments the binary by inserting a piece of checking code into every basic block. The code maintains a counter that tracks the number of instructions has been executed since the last SSA polling and compares the counter against a configurable threshold. The threshold indicates the frequency of AEX detection; i.e., if the counter exceeds the threshold, the code invokes the SSA polling routine. The routine either resets the counter for two cases. One is if no interrupt has occurred. Another is that the program has made sufficient progress since the last AEX

(i.e., the number of executed instructions is large enough). Otherwise, the routine terminates the execution, which ensures that an attacker cannot interrupt the program too frequent. To prevent potential leakage through caches, Varys also manually evicts cache lines upon detecting an AEX. However, recent microcode update provides the same effect and therefore eliminates such manual evictions.

Limitation. The security of Varys depends on how frequently it checks SSA and the value of the threshold. For example, to ensure a high-security level, the scheme has to insert the checking code as frequently as possible (i.e., at each basic block). Less-frequent insertion of checks otherwise allows an attacker to frequently interrupt the program between two checks such that preventing the counter from reaching the threshold. Consequently, this tradeoff prevents performance optimization (e.g., inserting checks outside of a loop body) without compromising the security (see examples in §5.6).

Co-location test. To prevent HT-based SCAs, Vary incorporates a method that prevents privileged attackers from scheduling an arbitrary thread on the same physical core running a target enclave. The idea is always requesting the OS to schedule an in-enclave thread alone with the original enclave on the same physical cores. To verify the co-location of two threads, Varys establishes a cache-based covert channel between the two threads and determines whether they share L1 or last-level caches. Sharing L1 cache indicates the two threads co-locate. Varys performs the co-location test whenever detecting an occurrence of an interrupt. Since the initial version of SGX does not support trusted time source (e.g., `rdtsc`), Varys uses an alternative approach that spawns an in-enclave thread. The thread keeps incrementing a global counter, representing ticks.

Instrumentation. The instrumentation pass of Varys inserts the checking code at the beginning of every basic block by using `onControlStart` and `onControlEnd`. Unlike the original Varys design that counts the number the instructions at the LLVM IR level, our pass counts the number of native instructions with the help of `onMachineInstrEnd`. For the SSA polling routine, the pass inserts the code before the entry function of the binary via

`onFunctionStart`. For supporting the co-location test, the pass adds a piece of code that does the test to the SSA polling routine (i.e., after detecting an AEX). Unlike the detection of high-frequency AEX, the co-location test is not overlapped with T-SGX. As a result, we add the optional support of the co-location test to the T-SGX. More concretely, we put the same piece of code at the springboard, before initiating a transaction with `xbegin`.

Validation. The validation pass performs the checking of 1) the existence of the checking code, 2) the correctness of the instruction number added to the counter, 3) the existence of the SSA polling code, and 4) whether the target of the `call` in the checking code points to the SSA polling routine.

5.4.4 QSpectre

For mitigating the Spectre attack, one software-based approach is utilizing serializing instructions (e.g, `lfence`) that prevent the CPU from speculatively executing instructions beyond the placement of these instructions. Following this idea, the Microsoft Visual Studio has implemented a compiler-based scheme, QSpectre [177]. During the compilation, the scheme tries to find potentially vulnerable code patterns and inserts the `lfence` instruction to the code.

Instrumentation. Unlike inserting `lfence` based on the pattern matching, which could be bypassed [195], our instrumentation pass for QSpectre adopts a simple, yet effective strategy: inserting `lfence` to all if-else structures. More concretely, inserting `lfence` right after a conditional branch in the code with an if-else structure. For implementing the strategy, the pass uses the `onMachineInstrEnd` API and determines if a conditional branch is in a if-else structure by consulting `CompilerContext`.

Validation. To validate QSpectre, the pass simply checks the existence of the `lfence` instruction in every if-else structure.

5.4.5 Pass Integration

Assuming the PRIDWEN loader supports all the abovementioned passes, we show how the pass manager selects the optimal passes. Moreover, we demonstrate how a pass resolves the potential conflicts with other passes.

Pass selection. Given that T-SGX and Varys have similar goals, we consider the passes of the two schemes targeting the same type of SCAs (e.g., high-frequency, page-fault attacks). To determine which pass to enable, the pass manager consults the prober and checks whether TSX is available in the current hardware environment. If TSX is available, the pass manager puts the T-SGX pass into the active set. Meanwhile, the pass manager drops the Varys pass according to the priority assignment rule (i.e., low runtime overhead). However, if TSX is not available, the pass manager performs oppositely. When detecting both TSX and HT are available, the pass manager additionally enables the co-location test in the T-SGX pass (e.g., inserting the co-location test code in the springboard).

Regarding the QSpectre pass, the pass manager activates it only if the prober reports that the HT is enabled and/or the microcode is outdated (i.e., no IBRS). Both cases indicate that the binary in the current hardware environment may be vulnerable to the Spectre attack. Lastly, the pass manager enables the fine-grained ASLR pass by default because it does not depend on any hardware features. It is worth mentioning that the explained pass selection policy is just an example. PRIDWEN can transparently support any developer-provided policies by design.

Conflicts resolving. Enabling multiple passes at the same time may lead to conflicts among the passes. For example, the ASLR and T-SGX passes can compete with each other to instrument the branch at the end of a basic block. Also, when enabling T-SGX pass alone, figuring out the address of the next code block for the pass is easy because of the fixed the memory layout. However, this is not the case when enabling the T-SGX and ASLR passes together. To resolve such conflicts, a pass (instrumentation and validation) should

implement additional logic that detects the existence of other active passes and handles the potential conflicts with each of the passes. For example, the T-SGX pass consults the pass manager about the existence of the ASLR pass. The ASLR pass inserts `jmp`s at the end of each basic block and associates each `jmp` with a symbol pointing to its successor. With such knowledge, the T-SGX pass replaces each `jmp` with a `lea`, reuses the symbols for the lea, and inserts another `jmp`, pointing to the springboard. This approach requires the execution order of the ASLR pass is ahead of that of the T-SGX pass. For this purpose, the T-SGX pass specifies the ASLR pass as a soft dependency in the configuration file.

5.4.6 Security Analysis

For each of the mitigation schemes, PRIDWEN provides a *similar level* of protection compared to its original design. We have experimentally verified such protection with both static (e.g., validation passes) and dynamic (e.g., runtime testing) approaches. For example, a T-SGX-enforced binary successfully suppresses page faults during the runtime, and a Varys-enforced binary terminates under frequent AEXs. Besides, by combining multiple mitigation schemes, PRIDWEN provides either a *higher level* of protection or better runtime performance compared to enforcing single mitigation. For example, although ASLR diversifies the runtime behavior of a program, the program is still subject to SCAs, which allows an attacker to learn the randomized layout of the program incrementally. Instead, combining ASLR and Varys prevents an attacker from obtaining fine-grained side-channel information of the program runtime. As a result, the combination secures the program against other types of SCAs (e.g., low-frequent SCAs) that are effective when enforcing only one of the mitigations. Another example is supporting the co-location test on top of T-SGX. This combination provides not only a similar level of protection of Varys (against high-frequent SCAs) but also a better runtime performance.

5.5 Implementation

We implement a prototype of PRIDWEN with 25k lines of `C` code on top of the Intel Linux SGX SDK 2.5.102. For native code generation, we implement an `x86` backend as part of the prototype that provides minimal support for the Wasm instructions set. Because the dynamic memory management in SGX [196] has yet to receive public support from CPUs, our prototype reserves 8 MiB of memory with `RWX` permissions for placing the synthesized native code, which is sufficient for all the programs in our evaluations. Wasm by default runs in a sandbox with memory safety, so using `RWX` memory pages is acceptable. In addition, to prevent validation passes from overwriting the synthesized binary, our current implementation makes the pass operate on a copy of the binary.

Runtime support. Our prototype provides an Emscripten-compatible runtime support that allows for running fairly large, complex applications such as Lighttpd as shown in §5.6. The application is directly compiled from unmodified `C` source code to a Wasm binary via the Emscripten compiler.

Pass implementations. Our prototype includes all of the four passes mentioned in §5.4. Each pass takes on average 800 lines of code. Because the control-flow information (including the definition of basic blocks in the binary) is generally required by most of the passes, we implement a control-flow analysis pass to share the information with other passes.

Attestation of synthesized binaries. The standard remote attestation of SGX works for the PRIDWEN loader (which is static), but not for the synthesized binaries that the PRIDWEN loader generates (which are dynamic). However, since the PRIDWEN loader can be attested, we can build a chain of trust from it. Further, the PRIDWEN loader accepts input Wasm binaries and passes only via a secure channel. These attested loader and secure binary deployment are similar to the remote attestation support of SGX-LKL [197].

5.6 Evaluation

In this section, we evaluate PRIDWEN by answering the following questions.

- What are the performance characteristics of the PRIDWEN loader?
- Does the execution of a PRIDWEN-synthesized binary remain faithful to the semantics of the input Wasm program?
- What is the performance of PRIDWEN-synthesized binaries? How much overheads do mitigation schemes incur?

Experimental setup. We ran all the experiments on a machine with a 4-core Intel i7-6700K CPU (Skylake microarchitecture) operating at 4 GHz with 32 KiB L1 and 256 KiB L2 private caches, an 8 MiB L3 shared cache, and 64 GiB of RAM. The machine was running Linux kernel 4.15. The PRIDWEN loader is compiled with gcc 5.4.0 and executed on top of the Intel Linux SGX SDK 2.5.102.

Applications. We base our evaluation on a benchmark suite, PolyBench [198], and three real-world applications or libraries: Lighttpd 1.4.48 [76], libjpeg 9a [77], and SQLite 3.21.0 [75]. The benchmark suite consists of 23 small `C` programs with only numerical computations (i.e., no system calls) that are used to evaluate the runtime performance of just-in-time compiled Wasm binaries against that of native ones [60]. In addition to the micro-benchmark suite, we use Lighttpd, libjpeg, and SQLite as a macro-benchmark suite that represents large, complex applications. We compile the original source code of each micro- or macro-benchmark program into Wasm using Emscripten [192], an LLVM-based compiler. We also directly port all of the programs using SGX SDK that serve as baseline versions.

Methodology. For each run of experiments, we take the application in the form of Wasm as an input to PRIDWEN. To evaluate PRIDWEN-synthesized binaries with distinct sets of defense schemes enforced, we manually configure PRIDWEN before each run. We use **PRIDWEN-base** to represent the configuration of baseline compilation (i.e., without

Table 5.4: Comparison of lines of code and binary size between PRIDWEN and LLVM backends.

	Line of code	**Binary size** (MiB)
PRIDWEN backend	8,166	1.26
LLVM backend	80,449	1026.00

instrumentation) and the name of defense schemes to represent the configuration of enforcing the corresponding schemes. For example, **TSGX** indicates the configuration with T-SGX-enforced and **TSGX+CoTest** indicates the configuration with both T-SGX- and co-location-test-enforced. For the ease of comparing Varys and T-SGX, the rest of the section uses Varys, or **VARYS**, to represent its original design without co-location test. To measure the execution time of each application, we use the `rdtsc` instruction via an OCall inside an enclave. The reported results are averaged over 10 runs.

5.6.1 Performance Characteristics of the PRIDWEN Loader

Complexity. To evaluate the complexity of the PRIDWEN loader, we compare its implementation against the implementation of LLVM 6.0.0 in terms of lines of code and binary size. More specially, we focus on comparing the implementation of `x86` backend between the PRIDWEN loader and LLVM. For lines of code, we calculate the lines of `C/C++` code in both implementations. For the binary size, we compare the size of the entire loader binary against the `llc` binary, which is the backend compiler of LLVM (i.e., generating an `x86` binary from a LLVM bitcode).

Results. Table 5.4 shows the results. Both lines of code and the binary size of LLVM backend are significantly larger than that of the PRIDWEN loader. The results indicate that adopting the LLVM backend in the SGX environment may be unrealistic. More specifically, given the limited size of SGX memory ($\sim$96 MiB), the size of the `llc` binary is too large to fit into an enclave. Further, long lines of code increase the size of the trusted computing base of an SGX program and the level of difficulties for security analysis. On the other hand, the implementation of PRIDWEN loader, which provides a minimum backend support for

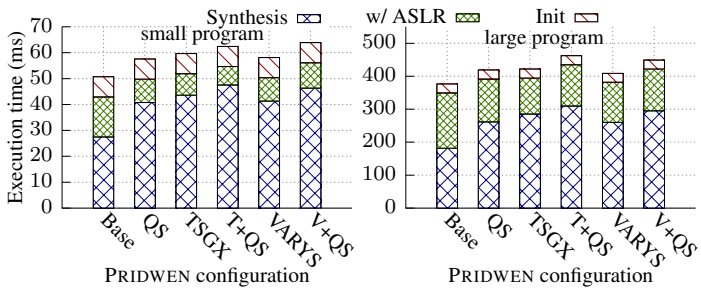

Figure 5.3: The time the PRIDWEN loader takes to synthesize programs.

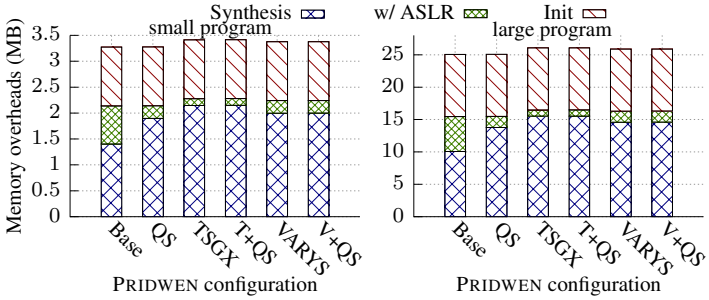

Figure 5.4: The memory overheads of the PRIDWEN loader on program synthesis.

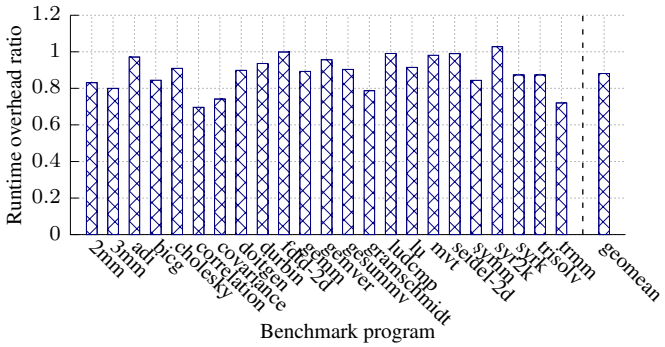

Figure 5.5: The runtime performance of PRIDWEN-synthesized programs compared to the native binary.

compiling Wasm binary, is more compact and suitable for the SGX environment.

Runtime and memory overheads. To show both runtime and memory overheads of the PRIDWEN loader, we measured the execution time that the loader takes to generate native binaries and the additional memory that the loader allocates during the entire process (by hooking `malloc`). To demonstrate the impact on the size of input, we used one small (`2mm`, 52 kB) and one large (`lighttpd`, 462 kB) Wasm binaries as inputs. We also ran experiments with multiple configurations of PRIDWEN that show the impact of enforcing each of the

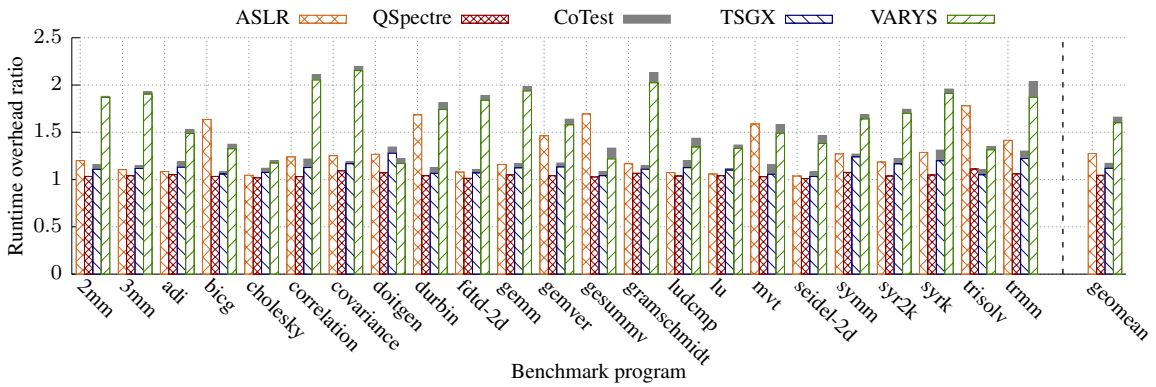

Figure 5.6: The runtime performance of PRIDWEN-synthesized programs secured with mitigation schemes.

defense schemes or combinations of them. Note that as the co-location test depends on either T-SGX or Varys and requires only adding a piece of code to each scheme, we do not include the test in the selected configurations.

Results. Figure 5.3 and Figure 5.4 present the results. We divide each bar into three parts: the execution time of the initialization stage (red), the execution time of the synthesis stage (blue), and the additional execution time to the synthesis stage when the ASLR is enforced together on top of the corresponding configuration (green). The initialization stage includes the time spent on hardware probing, pass manager initialization, and Wasm decoding. The synthesis state represents the time spent on compilation, instrumentation, and validation.

For the runtime overheads, the results show that, given the same input, the execution time of the initialization stage is fixed regardless of the configurations. Moreover, the loader spends more time during the initialization stage for the larger input, which is mostly contributed by the process of parsing the Wasm binary. However, the proportion of the execution time spent in the initialization stage decreases, which indicates that the loader spends more time on the synthesis stage for the large input. We also observe that enabling ASLR for the larger input incurs higher overhead than for the smaller one. The reason lies in that a larger program usually results in many basic blocks. As a result, the loader takes more time to place these blocks. In addition, enforcing more schemes incurs high overheads as expected. However, the execution time is generally reasonable (less than 500 ms even

with a large input). Moreover, each input requires only one-time initialization and synthesis before starting the execution.

For the memory overheads, the results show that the loader requires a fixed amount of memory during the initialization stage for the same input. The majority of the required memory is used to store the IR of the input program during the parsing process, which also explains that the loader requires a significantly large amount of memory for a large input. We also observe a similar memory requirement for the loader with the **PRIDWEN-base** configuration in the synthesis stage. Such a requirement represents the amount of memory that the loader needs for maintaining the metadata during the baseline compilation. Another observation is that enabling ASLR on top of **PRIDWEN-base** incurs the highest overhead. The reason behind this is that the instrumentation passes of each scheme all depend on a pass for obtaining control-flow graph (CFG) information, which contributes to the majority of the memory overhead when enabling ASLR on top the **PRIDWEN-base**. On the other hand, as the ASLR pass shares the CFG information with other passes, enabling ASLR on top of them incurs less overhead. Although the memory overhead of the loader is relatively high for large inputs, such memory is only required before the execution of the synthesized binary and therefore does not affect the runtime of the binary.

5.6.2 Faithfulness of Execution

One important aspect of PRIDWEN is whether a synthesized binary follows the semantics of the corresponding Wasm program regardless of its configuration. To validate whether the synthesized program behaves as expected, we use the official Wasm specification test suite [183], which provides comprehensive test cases for all Wasm instructions. The test suite consists of 73 programs. Each program includes a set of functions and test cases that specify the expected output of a function with corresponding input. We ran the test suite on PRIDWEN with all configurations and reported the results in terms of pass or fail on each program. In addition to the test suite, we also record the intermediate values of

```
# Varys
BB:
  ...
  jmp loop.header
loop.body:
  call varys_check
  ...
  incq %rcx
loop.header:
  call varys_check
  ...
  cmpq $100, %rcx
  jbe  loop.body
loop.end:
  call varys_check
  ...
```

```
# T-SGX
BB:
  ...
  leaq loop.header(%rip), %r15
  jmp  springboard.next
loop.body:
  ...
  incq %rcx
loop.header:
  ...
  cmpq $100, %rcx
  jbe  loop.body
  leaq loop.end(%rip), %r15
  jmp  springboard.next
loop.end:
  ...
```

Figure 5.7: The comparison of Varys and T-SGX on a loop structure.

the benchmark programs (by manually inserting `printf`) for both baseline and PRIDWEN-synthesized version and compare them.

Results. The results from the test suite show that programs with all the configurations successfully pass all the test cases, which indicates that not only the baseline compilation of PRIDWEN (**PRIDWEN-base**) faithfully follows the specification of Wasm but also the enforcement of schemes does not modify the behavior of the program. For the results of comparing intermediate values between baseline and PRIDWEN-synthesized binaries, we did not aware of any difference, which indicates a similar conclusion from the results of the test suite.

5.6.3 Performance of Synthesized Binaries

This section demonstrates the performance of PRIDWEN-synthesized binaries including runtime and memory overheads, and the sizes of the binaries. We compare the results of the baseline compilation against those of native binaries that we port directly to the SGX environment. In addition, we show the impact of defense schemes that PRIDWEN supports by comparing the results of using each configuration against that of the **PRIDWEN-base** configuration.

Runtime performance. Figure 5.5 shows the results of running the Polybench with the

PRIDWEN-base configuration, which are normalized to the execution time of the baseline programs. Our results indicate that PRIDWEN-synthesized binaries have negligible slowdown or are even faster than native binaries, without any mitigation schemes enforced. The execution time of PRIDWEN-synthesized binaries are 0.7–1.0× of that of the native binaries. In contrast, the evaluation of the in-browser compiler reports similar results with the majority of programs within 1–2× slowdown [60].

Figure 5.6 demonstrates the results of running Polybench each of defense schemes enforced. We show the results of enforcing multiple schemes in the following case studies of real-world applications. The bar on the figure represents the relative execution time of the program to the **PRIDWEN-base** configuration. The results indicate that **ASLR** incurs various overheads, while lies in the characteristics of the program: The number of randomized basic blocks being executed, which is not cache-friendly. Similarly, **QSpectre** also incurs various but smaller overheads, which come from the number of `lfence` instructions being executed.

Regarding **TSGX** and **VARYS**, the former incurs less overhead than the latter. This is mainly because Varys has to check the number of AEXs even inside a loop structure (see an example in Figure 5.7). Varys cannot avoid this issue unless it compromises its security guarantees (i.e., less frequent security checks). T-SGX incurs lower overhead than Varys because it supports loop optimization (i.e., puts an entire loop into a single transaction when possible). This comparison also illustrates the performance benefit of using hardware-assisted mitigation schemes over software-based ones. In addition, the evaluation results show that **CoTest** incurs small overheads (less than 10%) to both T-SGX and Varys. Note that **CoTest** requires two more concurrent threads for the software timer and the co-located memory access. PRIDWEN can eliminate these overheads when it detects a target platform disables HT.

Memory overhead. Figure 5.8 shows how much memory each mitigation demands on top of the binaries with **PRIDWEN-base**. On average, the instrumented binaries require up to 1.2× more memory than the baseline binaries. Figure 5.9 shows the sizes of PRIDWEN-

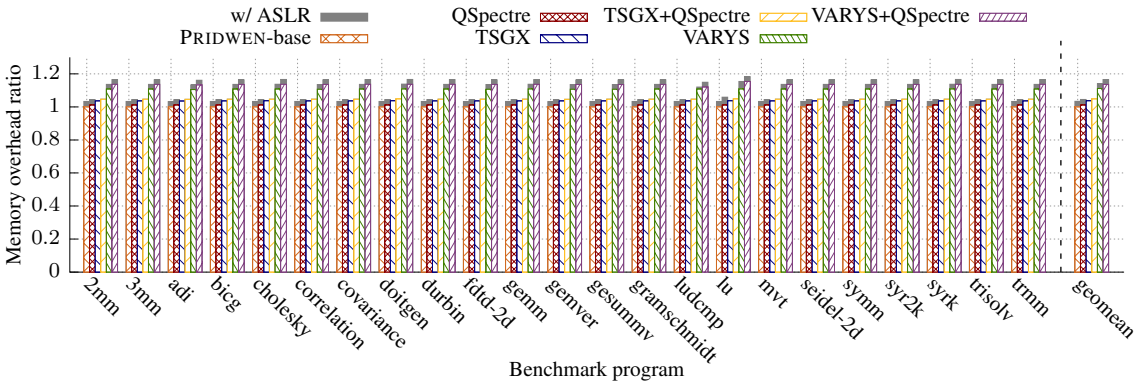

Figure 5.8: The memory overheads of mitigation schemes.

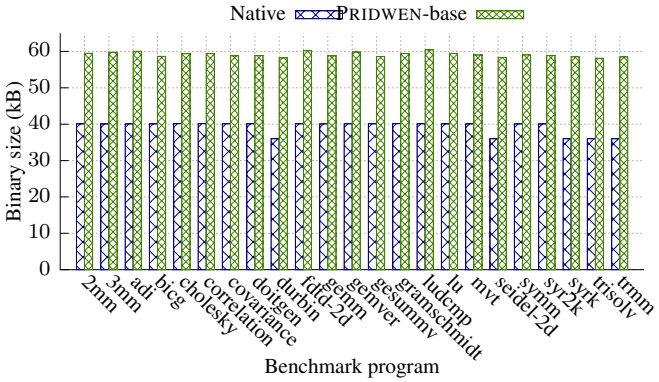

Figure 5.9: The binary size of PRIDWEN-synthesized programs compared to the native binary.

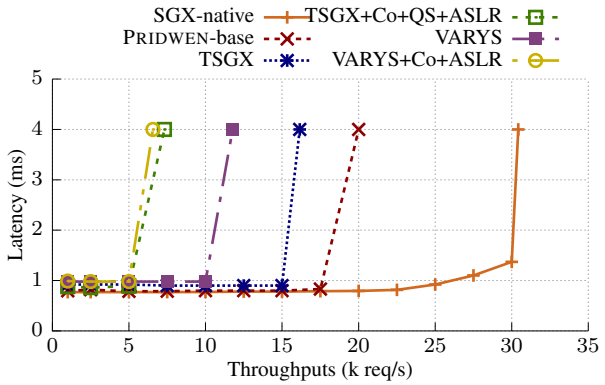

Figure 5.10: The runtime performance of lighttpd with various settings. QS: QSpectre.

synthesized Polybench binaries are on average 1.5× larger than those of native binaries. This is because Emscripten generates redundant code when compiling a program from C source to Wasm. Also, the Wasm compiler of PRIDWEN generates less-optimized code compared to the gcc compiler.

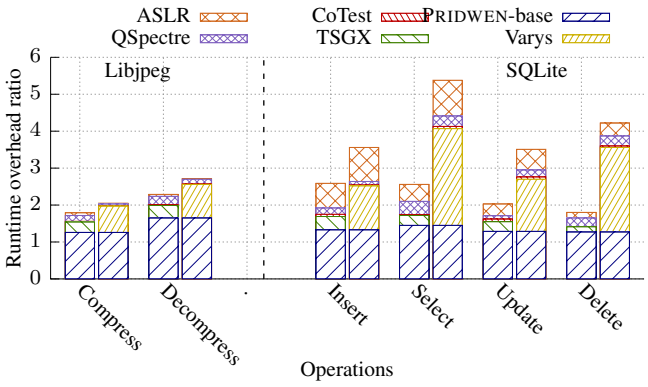

Figure 5.11: The runtime performance of synthesized libjpeg and SQLite.

Case study: Real-world applications. We use three real-world applications as case studies for demonstrating that PRIDWEN provides sufficient support for large, complex programs. In addition to the Lighttpd (a webserver), the other two applications are based on libjpeg and SQLite libraries. More specifically, the libjpeg application supports both compressing and decompressing a jpeg image and the SQLite application supports basic database operations including insert, select, update, and delete. Regarding the methodology, we use the HTTP benchmarking tool, wrk, for evaluating the throughputs of the Lighttpd. For the other two applications, we measure the time of each supported operation and report the average over multiple runs. We apply various configurations to these applications, which mainly focus on the performance impact of applying combinations of multiple mitigation schemes.

Results. Figure 5.10 shows the results of Lighttpd. The slowdown of **PRIDWEN-base** is 1.5× to the native version. Similar to the results of Polybench, **TSGX** incurs fewer overheads compared to **VARYS**. However, **ASLR** incurs significant overhead because the Lighttpd has a large number of small-sized basic blocks. This shortens the gap between **TSGX** and **VARYS** when they are enforced with **ASLR**. When enforcing multiple mitigation schemes, the slowdown of Lighttpd is up to 6× compared to the native binary.

Figure 5.11 presents the results of libjpeg and SQLite in terms of their operations. We use the stacked bars to represent the incurred overheads when applying a mitigation scheme

on top of a configuration. The slowdowns of **PRIDWEN-base** is 1.2–1.7× to the native versions. We note that such slowdowns are a way better than state-of-the-art results, which shows that Wasm can be up to 3.14× slower than native execution [182]. The overheads of an individual mitigation scheme are similar to the results of PolyBench (e.g., **TSGX** incurs fewer overheads than **VARYS** does and **ASLR** incurs various overheads). The slowdown of applying multiple mitigation schemes on top of Varys is up to 5.3× while the case of T-SGX is bounded by 2.7×. Thus, depending on hardware configurations, hardware-assisted mitigation schemes can be around 2× faster than software-only ones.

5.7 Discussion

Caching intermediate binaries. Instead of repeatedly synthesizing (and validating) SGX programs, PRIDWEN might be able to cache intermediate binaries before adopting dynamic passes (e.g., ASLR) along with the input Wasm binaries, static instrumentation passes, and probing results into a sealed storage. Whenever PRIDWEN synthesizes an SGX program again, it can first check whether the current Wasm binary, static instrumentation passes, and probing results are identical to the cached ones. If they are identical, PRIDWEN can just apply dynamic passes to the cached intermediate binary to generate a final binary.

Other hardening techniques. Although PRIDWEN mainly focuses on instrumentation for microarchitectural side-channel attack mitigation, it can be extended to adopt other hardware-assisted or software-only hardening techniques, such as Control-Flow Integrity (CFI) and memory safety. For example, instead of relying on Wasm's index-based CFI, we can write a CFI pass with the Intel Control-flow Enforcement Technology (CET) [199]. Also, we can use Intel MPX or SGXBounds [200] for the spatial memory safety of SGX programs. However, to fully support such hardening techniques, we need to extend Wasm to provide more detailed information about memory object types and allocations [201]. In addition, instead of the standard Wasm, we could extend PRIDWEN to use CT-Wasm [202] for secure information flow and constant-time execution.

5.8 Related Work

In-enclave loader. Developing in-enclave loaders is one of the actively researched topics to enhance the security and deployability of Intel SGX. For security, several loaders leverage randomization and encryption. SGX-Shield [14] is a compiler-based framework to load SGX applications while enforcing fine-grained ASLR. VC3 [123] and SGXElide [203] deploy encrypted SGX code while decrypting it within an enclave. Obfuscuro [204] obfuscates SGX code with Oblivious RAM.

For the deployability, several loaders abstract the interface between code running inside SGX and the outside. Ryoan [166] implements a two-way sandbox to securely execute untrusted code inside an enclave. Haven [127], Graphene-SGX [129], and SGX-LKL [197] run library Operating Systems (OSs) inside an enclave to executed unmodified programs. Similarly, Scone [161] abstracts system call interfaces and Panoply [193] abstracts POSIX interfaces to run unmodified programs with SGX.

Unlike these approaches, PRIDWEN focuses on how to customize SGX applications according to underlying hardware characteristics to improve their security and performance.

SGX and Wasm. To the best of our knowledge, there are a few initial efforts to execute Wasm interpreters inside an enclave. Rust-SGX [205] can be configured to use Wasm as a backend. Se-Lambda [206] executes serverless functions written in Wasm inside an enclave. Both approaches, however, run existing Wasm interpreters without notable improvements, that is, supporting instrumentation and probing hardware configurations, which are the key contributions of PRIDWEN.

Load-time program transformation. The basic concept of load-time program synthesis or transformation is not completely new. Several Java frameworks, such as JOIE [207], JMangler [208], and ASM [209], have been developed to transform Java classes according to user-provided transformers. Unlike them, PRIDWEN is specialized for improving the security of SGX programs while probing and leveraging the underlying hardware features.

Wasm instrumentation. Other studies also instrument Wasm binaries to detect security attacks and enable dynamic analysis. SEISMIC [184] instruments Wasm binaries to inject an inline monitor for detecting . Wasabi [185] is a Dynamic Binary Instrumentation (DBI) tool that statically instruments Wasm binaries to inject hooks and dynamically runs JavaScript-based analysis code on them to find potential performance and security bugs. However, unlike PRIDWEN, they do not consider instrumenting native binaries compiled from Wasm binaries, which is necessary to adopt low-level security mitigation sensitive to machine code.

CHAPTER 6
CONCLUSION AND FUTURE WORK

Side channels, which stem from the implementation of the hardware, is not a unique problem to SGX; *i.e.*, any multi-tenant architecture such as the cloud generally suffers from side-channel problems. However, SGX aims to provide a high-security standard: ensuring the security of an application in the cloud without trusting the provider. Such an unusual setting —attackers with root privileges— enables exceptionally powerful SCAs, making these attacks become a critical threat to SGX. This thesis work attempts to address the threat of SCAs practically. Toward this end, the thesis presents two defenses, including SGX-Armor, an improved ASLR-based defense that provides general protection against SCAs, and T-SGX, a novel, TSX-based defense that defeats controlled-channel attacks. Besides, the thesis presents PRIDWEN, a general framework that addresses the limitations of composing multiple defenses, including SGX-Armor and T-SGX, and thereby provides a broader scope of protection against various types of SCAs. Despite not completely eradicating SCAs on SGX, the proposed defenses have demonstrated the feasibility of practical mitigation that is sufficient to render these SCAs ineffective.

Since this thesis primarily focuses on instrumentation-based defenses against SCAs that automatically transform a program into a protected one, such defenses unavoidably impose non-negligible overheads on the execution. A more fundamental reason for taking this direction is that, unlike typical software vulnerabilities (*e.g.*, memory corruptions), side-channel vulnerabilities are difficult to identify; *i.e.*, even a program may leak sensitive information through side channels, and the presence of such leakage is stealthy to the program. As a result, an alternative direction is to identify side-channel vulnerabilities using static side-channel analysis. Similar to finding software bugs with static analysis, side-channel analysis aims to detect code patterns that potentially cause information leakage

through side channels. Consequently, side-channel analysis assists developers not only to evaluate the implications of SCAs to their programs but to prevent the programs from leaking information through side channels during the development phase.

PRIDWEN provides a broader scope of protection against SCAs by composing multiple defenses. A potential concern with this approach is that when the number of supported defenses raises, PRIDWEN may incur high runtime overhead to the program. Therefore, instead of composing defenses, another potential direction is proposing a new software-hardware co-design for SCA mitigation. The core idea of this direction is proposing new, dedicated hardware instructions for SCA mitigation. Next, similar to T-SGX, the idea is to design a software model for the instructions and to provide the same level of protection as the composition of multiple defenses.

The other potential direction is to design an SCA defense framework for general TEEs. Since SGX is merely one implementation of TEEs, the SCAs on SGX should be mostly applicable to other implementations of TEEs, such as ARM TrustZone and AMD SEV. The current trend in the industry community is also to create a TEE abstraction that is agnostic to the implementations [170, 171]. Therefore, having an SCA defense framework for general TEEs may be necessary in the near future.

REFERENCES

[1] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.

[2] S. Gueron, "Memory Encryption for General-Purpose Processors," *IEEE Security & Privacy*, vol. 14, no. 6, pp. 54–62, 2016.

[3] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[4] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[5] W. Wang, G. Chen, X. Pan, Y. Zhang, X. Wang, V. Bindschaedler, H. Tang, and C. A. Gunter, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[6] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, Canada, Aug. 2017.

[7] A. Moghimi, G. Irazoqui, and T. Eisenbarth, "Cachezoom: How sgx amplifies the power of cache attacks," in *International Conference on Cryptographic Hardware and Embedded Systems*, 2017.

[8] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jul. 2017.

[9] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[10] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," in *Proceedings of the 23st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Williamsburg, VA, Mar. 2018.

[11] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[12] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: An extremely simple oblivious RAM protocol," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[13] A. Rane, C. Lin, and M. Tiwari, "Raccoon: Closing digital side-channels through obfuscated execution," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[14] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[15] F. Brasser, S. Capkun, A. Dmitrienko, T. Frassetto, K. Kostiainen, U. Müller, and A.-R. Sadeghi, "Dr. sgx: Hardening sgx enclaves against cache attacks with data location randomization," *arXiv*, 2017.

[16] S. Hosseinzadeh, H. Liljestrand, V. Leppänen, and A. Paverd, "Mitigating branch-shadowing attacks on intel SGX using control flow randomization," *CoRR*, vol. abs/1808.06478, 2018. arXiv: 1808.06478.

[17] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, "T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[18] Intel, *Q3 2018 Speculative Execution Side Channel Update*, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018.

[19] ——, *Intel Side Channel Vulnerability MDS*, https://www.intel.com/content/www/us/en/architecture-and-technology/mds.html, 2019.

[20] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa, "Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[21] G. Chen, W. Wang, T. Chen, S. Chen, Y. Zhang, X. Wang, T.-H. Lai, and D. Lin, "Racing in hyperspace: Closing hyper-threading side channels on SGX with contrived data races," in *Proceedings of the 39th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[22] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer, "Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks," in *Proceedings of the 2018 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jul. 2018.

[23] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: Cold-boot attacks on encryption keys," *Communications of the ACM*, vol. 52, no. 5, pp. 91–98, 2009.

[24] Intel, *Code Sample: Intel Software Guard Extensions Remote Attestation End-to-End Example*, `https://software.intel.com/en-us/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example`, 2018.

[25] ——, *Exception Handling in Intel Software Guard Extensions (Intel SGX) Applications*, 2019.

[26] ——, *Intel 64 and ia-32 architectures software developer's manual combined volumes: 1, 2a, 2b, 2c, 2d, 3a, 3b, 3c and 3d*, Sep. 2016.

[27] D. Brumley and D. Boneh, "Remote timing attacks are practical," *Computer Networks*, 2005.

[28] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games–bringing access-based cache attacks on aes to practice," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2011.

[29] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of aes," in *Cryptographers' Track at the RSA Conference*, 2006.

[30] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazalch, "Jump over ASLR: Attacking branch predictors to bypass ASLR," in *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Taipei, Taiwan, Oct. 2016.

[31] S. Weiser, R. Spreitzer, and L. Bodner, "Single Trace Attack Against RSA Key Generation in Intel SGX SSL," in *Proceedings of the 13th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Seoul, South Korea, Jun. 2018.

[32] J. Gyselinck, J. Van Bulck, F. Piessens, and R. Strackx, "Off-limits: Abusing legacy x86 memory segmentation to spy on enclaved execution," in *International Symposium on Engineering Secure Software and Systems*, Springer, 2018, pp. 44–60.

[33] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets," in *Proceedings of the 11th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Xi'an, China, May 2016.

[34] F. Dall, G. D. Micheli, T. Eisenbarth, D. Genkin, N. Heninger, A. Moghimi, and Y. Yarom, "CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks," in *Proceedings of the Conference on Cryptographic Hardware and Embedded Systems (CHES)*, 2018.

[35] A. Moghimi, T. Eisenbarth, and B. Sunar, "Memjam: A false dependency attack against constant-time crypto implementations in sgx," in *Cryptographers' Track at the RSA Conference*, Springer, 2018.

[36] J. Van Bulck, F. Piessens, and R. Strackx, "Nemesis: Studying microarchitectural timing leaks in rudimentary cpu interrupt logic," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, Canada, Oct. 2018.

[37] W. He, W. Zhang, S. Das, and Y. Liu, "Sgxlinger: A new side-channel attack vector based on interrupt latency against enclave execution," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, IEEE, 2018, pp. 108–114.

[38] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *arXiv preprint arXiv:1802.09085*, 2018.

[39] E. M. Koruyeh, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh, "Spectre returns! speculation attacks using the return stack buffer," in *Proceedings of the 12th USENIX Workshop on Offensive Technologies (WOOT)*, Baltimore, MD, Aug. 2018.

[40] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida, "RIDL: Rogue in-flight data load," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[41] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," *arXiv:1905.05726*, 2019.

[42] M. Minkin, D. Moghimi, M. Lipp, M. Schwarz, J. Van Bulck, D. Genkin, D. Gruss, B. Sunar, F. Piessens, and Y. Yarom, "Fallout: Reading kernel writes from user space," 2019.

[43] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Oct. 2007.

[44] PaX Team, *PaX address space layout randomization (ASLR)*, `https://pax.grsecurity.net/docs/aslr.txt`, 2000.

[45] S. Bhatkar, R. Sekar, and D. DuVarney, "Efficient techniques for comprehensive protection from memory error exploits," in *Usenix Security Symposium*, 2005.

[46] C. Kil, J. Jun, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (ASLP)," in *Annual Computer Security Applications Conference*, 2006.

[47] C. Guiffrida, A. Kuijsten, and A. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[48] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A.-R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[49] A. Gupta, S. Kerr, M. S. Kirkpatrick, and E. Bertino, "Marlin: A fine grained randomization approach to defend against rop attacks," in *International Conference on Network and System Security*, 2013.

[50] R. Wartell, V. Mohan, K. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS)*, Raleigh, NC, Oct. 2012.

[51] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[52] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. Davidson, "Ilr: Where'd my gadgets go?" In *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[53] K. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A. Sadeghi, "Just in time code reuse: On the effectiveness of fine-grained address space layout randomization," in *IEEE Symposium on Security and Privacy*, 2013.

[54] Y. Jang, S. Lee, and T. Kim, "Breaking Kernel Address Space Layout Randomization with Intel TSX," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[55] D. Gruss, C. Maurice, A. Fogh, M. Lipp, and S. Mangard, "Prefetch side-channel attacks: Bypassing SMAP and kernel ASLR," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, Oct. 2016.

[56] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space aslr," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2013.

[57] B. Gras and K. Razavi, "Aslr on the line: Practical cache attacks on the mmu," Feb. 2017.

[58] M. Herlihy and J. Moss, "Transactional memory: Architectural support for lock-free data structures," in *Proceedings of the 20th ACM/IEEE International Symposium on Computer Architecture (ISCA)*.

[59] V. Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Proceedings of the 30th IEEE International Conference on Data Engineering (ICDE)*, Chicago, IL, Mar. 2014.

[60] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the Web up to Speed with WebAssembly," in *Proceedings of the 2017 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Barcelona, Spain, Jun. 2017.

[61] ARM, *Building a secure system using TrustZone technology*, PRD29-GENC-009492C, Dec. 2008.

[62] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Tel-Aviv, Israel, 2013, pp. 1–8.

[63] Fortanix, *Runtime encryption*, https://www.fortanix.com/, 2018.

[64] Russinovich Mark, *Introducing azure confidential computing*, https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/, Sep. 2017.

[65] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in Darkness: Return-oriented Programming against Secure Enclaves," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, Canada, Aug. 2017.

[66] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, Alexandria, VA, Nov. 2005.

[67] S. Andersen and V. Abella, "Changes to functionality in windows xp service pack 2, part 3: Memory protection technologies," Aug. 2004.

[68] C. Cowan, C. Pu, D. Maier, J. Walpole, and P. Bakke, "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks.," in *Proceedings of the 7th USENIX Security Symposium (Security)*, San Antonio, TX, Jan. 1998.

[69] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard, "Armageddon: Cache attacks on mobile devices.," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[70] N. Zhang, K. Sun, D. Shands, W. Lou, and Y. T. Hou, "Truspy: Cache side-channel information leakage from the secure world on arm devices.," 2016.

[71] J. Van Bulck, F. Piessens, and R. Strackx, "SGX-Step: A practical attack framework for precise enclave execution control," in *2nd Workshop on System Software for Trusted Execution (SysTEX 2017)*, 2017.

[72] Y. Xiao, M. Li, S. Chen, and Y. Zhang, "Stacco: Differentially analyzing side-channel traces for detecting ssl/tls vulnerabilities in secure enclaves," in *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[73] W. Samuel, Z. Andreas, S. Raphael, M. Katja, M. Stefan, and S. Georg, "Data - differential address trace analysis: Finding address-based side-channels in binaries," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[74] A. Czumaj, "Random permutations using switching networks," in *ACM Symposium on Theory of Computing (STOC)*, 2015.

[75] *SQLite*, https://www.sqlite.org/index.html, 2000.

[76] *Lighttpd*, https://www.lighttpd.net/, 2003.

[77] *libjpeg*, https://libjpeg.sourceforge.net/, 1991.

[78] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[79] S. Crane, A. Homescu, and P. Larsen, "Code randomization: Haven't we solved this problem yet?" In *Cybersecurity Development (SecDev), IEEE*, 2016.

[80] M. Backes, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing.," in *Proceedings of the 23rd USENIX Security Symposium (Security)*, San Diego, CA, Aug. 2014.

[81] M. Pomonis, T. Petsios, A. D. Keromytis, M. Polychronakis, and V. P. Kemerlis, "Krx̂: Comprehensive kernel protection against just-in-time code reuse," in *Proceedings of the Twelfth European Conference on Computer Systems*, 2017.

[82] V. Pappas, M. Polychronakis, and A. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[83] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2013.

[84] A. Bittau, A. Belay, A. Mashtizadeh, D. Mazières, and D. Boneh, "Hacking blind," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2014.

[85] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on intel sgx," in *Proceedings of the 10th European Workshop on Systems Security*, 2017.

[86] J. Wilander, N. Nikiforakis, Y. Younan, M. Kamkar, and W. Joosen, "Ripe: Runtime intrusion prevention evaluator," in *Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[87] Intel, *Dynamic control-flow graph generation with pinplay*, https://software.intel.com/en-us/articles/pintool-dcfg, 2015.

[88] V. Lyzinski, D. E. Fishkind, M. Fiori, J. T. Vogelstein, C. E. Priebe, and G. Sapiro, "Graph matching: Relax at your own risk," *IEEE transactions on pattern analysis and machine intelligence*, 2016.

[89] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Polymorphic worm detection using structural information of executables," in *International Workshop on Recent Advances in Intrusion Detection*, 2005.

[90] D. Bruschi, L. Martignoni, and M. Monga, "Detecting self-mutating malware using control-flow graph matching," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2006.

[91] S. Alam, I. Traore, and I. Sogukpinar, "Annotated control flow graph for metamorphic malware detection," *The Computer Journal*, 2015.

[92] S. Anju, P Harmya, N. Jagadeesh, and R Darsana, "Malware detection using assembly code and control flow graph optimization," in *Proceedings of the 1st Amrita ACM-W Celebration on Women in Computing in India*, 2010.

[93] E. W. Weisstein, "Simulated annealing," 2000.

[94] O. Ohrimenko, F. Schuster, C. Fournet, A. Mehta, S. Nowozin, K. Vaswani, and M. Costa, "Oblivious multi-party machine learning on trusted processors.," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[95] B. Morris, "The mixing time of the thorp shuffle," *SIAM Journal on Computing*, 2008.

[96] ——, "Improved mixing time bounds for the thorp shuffle," *Combinatorics, Probability and Computing*, 2013.

[97] V. T. Hoang, B. Morris, and P. Rogaway, "An enciphering scheme based on a card shuffle," in *Advances in Cryptology–CRYPTO 2012*, 2012.

[98] T. Ristenpart and S. Yilek, "The mix-and-cut shuffle: Small-domain encryption secure against n queries," in *Advances in Cryptology–CRYPTO 2013*, 2013.

[99] *GDB*, `https://www.gnu.org/software/gdb/`, 1986.

[100] J. Håstad, "The square lattice shuffle," *Random Structures and Algorithms*, 2006.

[101] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel software guard extensions (Intel SGX) support for dynamic memory

management inside an enclave," in *Proceedings of the 5th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[102] Greg W., *Sgx2 support for cpus that already support sgx1*, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/757950, 2018.

[103] *nbench*, https://www.math.utah.edu/~mayer/linux/bmark.html, 1996.

[104] D. Bigelow, T. Hobson, R. Rudd, W. Streilein, and H. Okravi, "Timely rerandomization for mitigating memory disclosures," in *ACM Conference on Computer and Communications Security (CCS)*, 2015.

[105] K. Lu, S. Nürnberger, M. Backes, and W. Lee, "How to make ASLR win the clone wars: Runtime re-randomization," in *Network and Distributed System Security Symposium*, 2016.

[106] D. Williams-King, G. Gobieski, K. Williams-King, J. Blake, X. Yuan, P. Colp, M. Zheng, V. Kemerlis, J. Yang, and W. Aiello, "Shuffler: Fast and deployable continuous code re-randomization," in *USENIX Symposium on Operating Design and Implementation*, 2016.

[107] Y. Chen, Z. Wang, D. Whalley, and L. Lu, "Remix: On-demand live randomization," in *Proceedings of the Sixth ACM Conference on Data and Application Security and Privacy*, 2016.

[108] R. Strackx, Y. Younan, P. Philippaerts, F. Piessens, S. Lachmund, and T. Walter, "Breaking the memory secrecy assumption," in *Proceedings of the Second European Workshop on System Security*, 2009.

[109] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, 2017.

[110] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against intel sgx," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[111] A. Kwon, H. Corrigan-Gibbs, S. Devadas, and B. Ford, "Atom: Horizontally scaling strong anonymity," in *ACM Symposium on Operating Systems Principles (SOSP)*, 2016.

[112] I. Ion, N. Sachdeva, P. Kumaraguru, and S. Čapkun, "Home is safer than the cloud!: Privacy concerns for consumer cloud storage," in *Proceedings of the Seventh Symposium on Usable Privacy and Security (SOUPS)*, Pittsburgh, Pennsylvania, 2011.

[113] Samsung, *White paper: An overview of Samsung knox*, Enterprise Mobility Solutions, 2013.

[114] Trusted Computing Group, *Trusted platform module (TPM) summary*, `http://www.trustedcomputinggroup.org/trusted-platform-module-tpm-summary/`.

[115] J. Greene, "Intel trusted execution technology," *Intel Technology White Paper*, 2012.

[116] Intel, *Intel software guard extensions programming reference (rev2)*, 329298-002US, Oct. 2014.

[117] H. Takabi, J. B. Joshi, and G.-J. Ahn, "Security and privacy challenges in cloud computing environments," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 24–31, 2010.

[118] C. Gentry, "Fully homomorphic encryption using ideal lattices," in *Proceedings of the 41st Annual ACM Symposium on Theory of Computing (STOC)*, 2009.

[119] J. Li, M. Krohn, D. Mazières, and D. Shasha, "Secure untrusted data repository (SUNDR)," in *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, CA, Dec. 2004.

[120] A. J. Feldman, W. P. Zeller, M. J. Freedman, and E. W. Felten, "SPORC: Group collaboration using untrusted cloud resources," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[121] P. Mahajan, S. Setty, S. Lee, A. Clement, L. Alvisi, M. Dahlin, and M. Walfish, "Depot: Cloud storage with minimal trust," in *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, Canada, Oct. 2010.

[122] Intel, *SGX Tutorial, ISCA 2015*, `http://sgxisca.weebly.com/`, Portland, OR, Jun. 2015.

[123] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "VC3: Trustworthy data analytics in the cloud using SGX," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[124] T. T. A. Dinh, P. Saxena, E.-C. Cang, B. C. Ooi, and C. Zhang, "M2R: Enabling stronger privacy in MapReduce computation," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[125] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han, "A first step towards leveraging commodity trusted execution environments for network applications," in *Proceedings of*

the 14th ACM Workshop on Hot Topics in Networks (HotNets), Philadelphia, PA, Nov. 2015.

[126] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska, "S-NFV: Securing NFV states by using SGX," in *Proceedings of the 1st ACM International Workshop on Security in SDN and NFV*, New Orleans, LA, Mar. 2016.

[127] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Broomfield, Colorado, Oct. 2014.

[128] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and security isolation of library OSes for multi-process applications," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[129] C.-C. Tsai and D. Porter, *Graphene / Graphene-SGX Library OS - a library OS for Linux multi-process applications, with Intel SGX support*, `https://github.com/oscarlab/graphene`.

[130] V. Costan and S. Devadas, *Intel SGX explained*, Cryptology ePrint Archive, Report 2016/086, `http://eprint.iacr.org/`, 2016.

[131] B. Coppens, I. Verbauwhede, K. D. Bosschere, and B. D. Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[132] S. M. Hand, "Self-paging in the Nemesis operating system," in *Proceedings of the 3rd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, New Orleans, LA, Feb. 1999.

[133] L. Guan, J. Lin, B. Luo, J. Jing, and J. Wang, "Protecting private keys against memory disclosure attacks using hardware transactional memory," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[134] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel, "InkTag: Secure applications on an untrusted operating system," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[135] M. Maas, E. Love, E. Stefanov, M. Tiwari, E. Shi, K. Asanović, J. Kubiatowicz, and D. Song, "PHANTOM: Practical oblivious computation in a secure processor,"

in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[136] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, Aug. 2016.

[137] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[138] G. Irazoqui, T. Eisenbarth, and B. Sunar, "S$A: A shared cache attack that works across cores and defies VM sandboxing—and its application to AES," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2015.

[139] C. Liu, A. Harris, M. Maas, M. Hicks, M. Tiwari, and E. Shi, "GhostRider: A hardware-software system for memory trace oblivious computation," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[140] L. Guan, J. Lin, B. Luo, and J. Jing, "Copker: Computing with private keys without RAM," in *Proceedings of the 21st Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2014.

[141] P. Colpa, J. Zhang, J. Gleeson, S. Suneja, E. de Lara, H. Raj, S. Saroiu, and A. Wolman, "Protecting data on smartphones and tablets from memory attacks," in *Proceedings of the 20th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Istanbul, Turkey, Mar. 2015.

[142] N. Zhang, K. Sun, W. Lou, and Y. T. Hou, "CaSE: Cache-assisted secure execution on ARM processors," in *Proceedings of the 37th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2016.

[143] T. Müller, F. C. Freiling, and A. Dewald, "TRESOR runs encryption securely outside RAM," in *Proceedings of the 20th USENIX Security Symposium (Security)*, San Francisco, CA, Aug. 2011.

[144] J. Götzfried and T. Müller, "Armored: CPU-bound encryption for Android-driven ARM devices," in *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES)*, 2013.

[145] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. A. Waldspurger, D. Boneh, J. Dwoskin, and D. R. Ports, "Overshadow: A virtualization-based approach

to retrofitting protection in commodity operating systems," in *Proceedings of the 13th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Seattle, WA, Mar. 2008.

[146] U. Steinberg and B. Kauer, "NOVA: A microhypervisor-based secure virtualization architecture," in *Proceedings of the 5th European Conference on Computer Systems (EuroSys)*, Paris, France, Apr. 2010.

[147] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig, "TrustVisor: Efficient TCB reduction and attestation," in *Proceedings of the 31th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2010.

[148] L. Martignoni, P. Poosankam, M. Zaharia, J. Han, S. McCamant, D. Song, V. Paxson, A. Perrig, S. Shenker, and I. Stoica, "Cloud Terminal: Secure access to sensitive applications from untrusted systems," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC)*, Boston, MA, Jun. 2012.

[149] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "MiniBox: A two-way sandbox for x86 native code," in *Proceedings of the 2014 USENIX Annual Technical Conference (ATC)*, Philadelphia, PA, Jun. 2014.

[150] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel, "Sego: Pervasive trusted metadata for efficienctly verified untrusted system services," in *Proceedings of the 21st ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Atlanta, GA, Apr. 2016.

[151] Xen, *Xen security advisories*, `http://xenbits.xen.org/xsa/`.

[152] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for TCB minimization," in *Proceedings of the 3rd European Conference on Computer Systems (EuroSys)*, Glasgow, Scotland, Mar. 2008.

[153] D. Levin, J. R. Douceur, J. R. Lorch, and T. Moscibroda, "TrInc: Small trusted hardware for large distributed systems," in *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Apr. 2009.

[154] A. M. Azab, P. Ning, and X. Zhang, "SICE: A hardware-level strongly isolated computing environment for x86 multi-core platforms," in *Proceedings of the 18th ACM Conference on Computer and Communications Security (CCS)*, Chicago, Illinois, Oct. 2011.

[155] K. Sun, J. Wang, F. Zhang, and A. Stavrou, "SecureSwitch: BIOS-assisted isolation and switch between trusted and untrusted commodity OSes," in *Proceedings of the*

19th Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2012.

[156] E. Owusu, J. Guajardo, J. McCune, J. Newsome, A. Perrig, and A. Vasudevan, "OASIS: On achieving a sanctuary for integrity and secrecy on untrusted platforms," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS)*, Berlin, Germany, Oct. 2013.

[157] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[158] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," in *Proceedings of the 18th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Houston, TX, Mar. 2013.

[159] P. Jain, S. Desai, S. Kim, M.-W. Shih, J. Lee, C. Choi, Y. Shin, T. Kim, B. B. Kang, and D. Han, "OpenSGX: An Open Platform for SGX Research," in *Proceedings of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2016.

[160] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves," in *Proceedings of the 21th European Symposium on Research in Computer Security (ESORICS)*, Heraklion, Greece, Sep. 2016.

[161] S. Arnautox, B. Tarch, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux containers with Intel SGX," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[162] R. Pires, M. Pasin, P. Felber, and C. Fetzer, "Secure content-based routing using Intel Software Guard Extensions," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.

[163] S. Brenner, C. Wulf, M. Lorenz, N. Weichbrodt, D. Goltzsche, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper using Intel SGX," in *Proceedings of the 16th Annual Middleware Conference (Middleware)*, 2016.

[164] R. Sinha, S. Rajamani, S. Seshia, and K. Vaswani, "Moat: Verifying confidentiality of enclave program," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[165] R. Sinha, M. Costa, A. Lal, N. P. Lopes, S. Rajamani, S. A. Seshia, and K. Vaswani, "A design and verification methodology for secure isolated regions," in *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Santa Barbara, CA, Jun. 2016.

[166] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, Nov. 2016.

[167] M. Meinardi, *How to Develop a Business Case for the Adoption of Public Cloud IaaS*, https://www.gartner.com/en/documents/3893869, 2018.

[168] J. Mangalindan, *Is User Data Safe in the Cloud?* http://tech.fortune.cnn.com/2010/09/24/is-user-data-safe-in-the-cloud, Sep. 2010.

[169] AMD, *AMD Secure Encrypted Virtualization (SEV)*, https://developer.amd.com/sev/.

[170] N. Porter, *Introducing Asylo: an open-source framework for confidential computing*, https://cloud.google.com/blog/products/gcp/introducing-asylo-an-open-source-framework-for-confidential-computing, 2018.

[171] Microsoft, *Open Enclave SDK*, https://openenclave.io/sdk/, 2019.

[172] Microsoft Azure, *Azure Confidential Computing*, https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2019.

[173] S. Johnson, *Intel SGX and Side-Channels*, https://software.intel.com/en-us/articles/intel-sgx-and-side-channels.

[174] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin, "SGX-LAPD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, 2017.

[175] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting privileged side-channel attacks in shielded execution with déjá vu," in *Proceedings of the 12th ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, Abu Dhabi, UAE, Apr. 2017.

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

[177] A. Pardoe, *Spectre mitigations in MSVC*, https://devblogs.microsoft.com/cppblog/spectre-mitigations-in-msvc/, 2018.

[178] D. Megyesi, *AWS vs GCP vs on-premises CPU performance comparison*, https://medium.com/infrastructure-adventures/aws-vs-gcp-vs-on-premises-cpu-performance-comparison-1cb3e91f9716, 2018.

[179] W. Kim, *Fun with Intel Transactional Synchronization Extensions*, https://software.intel.com/en-us/blogs/2013/07/25/fun-with-intel-transactional-synchronization-extensions, 2013.

[180] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An oblivious and encrypted distributed analytics platform," in *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, Mar. 2017.

[181] WebAssembly Community Group, "WebAssembly Specification: Release 1.0," Tech. Rep., May 2019.

[182] A. Jangda, B. Powers, A. Guha, and E. Berger, "Mind the Gap: Analyzing the Performance of WebAssembly vs. Native Code," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, Jul. 2019.

[183] *Mirror of the spec testsuite*, https://github.com/WebAssembly/testsuite, 2019.

[184] W. Wang, B. Ferrell, X. Xu, K. W. Hamlen, and S. Hao, "SEISMIC: SEcure in-lined script monitors for interrupting cryptojacks," in *European Symposium on Research in Computer Security*, Springer, 2018, pp. 122–142.

[185] D. Lehmann and M. Pradel, "Wasabi: A Framework for Dynamically Analyzing WebAssembly," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, Apr. 2019.

[186] Intel, *Intel software guard extensions: Intel attestation service API*, https://software.intel.com/sites/default/files/managed/3d/c8/IAS_1_0_API_spec_1_1_Final.pdf.

[187] Greg, *SGX Attestation results in CONFIGURATION_NEEDED*, https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/798777, 2018.

[188] ——, *GROUP_OUT_OF_DATE - what is the most recent microcode version?* https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/755769, 2018.

[189] Clemens Hammacher, *Liftoff: A new baseline compiler for webassembly in v8*, https://v8.dev/blog/liftoff, 2018.

[190] M. Poletto and V. Sarkar, "Linear scan register allocation," *ACM Trans. Program. Lang. Syst.*, vol. 21, no. 5, pp. 895–913, Sep. 1999.

[191] G. Chaitin, "Register allocation and spilling via graph coloring," *SIGPLAN Not.*, vol. 39, no. 4, pp. 66–74, Apr. 2004.

[192] *emscripten*, https://emscripten.org/, 2015.

[193] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux applications with SGX enclaves," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[194] L. Clark, *Standardizing WASI: A system interface to run WebAssembly outside the web*, https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019.

[195] P. Kocher, *Spectre Mitigations in Microsoft's C/C++ Compiler*, https://www.paulkocher.com/doc/MicrosoftCompilerSpectreMitigation.html, 2018.

[196] B. C. Xing, M. Shanahan, and R. Leslie-Hurd, "Intel software guard extensions (Intel SGX) software support for dynamic memory allocation inside an enclave," in *Proceedings of the 5th Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[197] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. R. Pietzuch, "SGX-LKL: Securing the Host OS Interface for Trusted Execution," *CoRR*, vol. abs/1908.11143, 2019. arXiv: 1908.11143.

[198] *PolyBench*, http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/, 2015.

[199] Intel, *Control-flow Enforcement Technology Specification*, https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf, 2019.

[200] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer, "SGXBounds: Memory Safety for Shielded Execution," in *Proceedings of*

the 12th European Conference on Computer Systems (EuroSys), Belgrade, Serbia, Apr. 2017.

[201] C. Disselkoen, T. Garfinkel, D. Stefan, and C. W. and, "Trestle: Bridging the Performance and Safety Divide in WebAssembly," in *Workshop on Principles of Secure Compilation (PriSC)*, 2019.

[202] C. Watt, J. Renner, N. Popescu, S. Cauligi, and D. Stefan, "CT-Wasm: Type-Driven Secure Cryptography for the Web Ecosystem," in *Proceedings of the 46th ACM Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, Jan. 2019.

[203] E. Bauman, H. Wang, M. Zhang, and Z. Lin, "SGXElide: Enabling enclave code secrecy via self-modification," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO)*, Vienna, Austria, Feb. 2018.

[204] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A Commodity Obfuscation Engine on Intel SGX," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[205] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," in *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, London, UK, Nov. 2019.

[206] W. Qiang, Z. Dong, and H. Jin, "Se-Lambda: Securing privacy-sensitive serverless applications using SGX enclave," in *International Conference on Security and Privacy in Communication Systems (SecureComm)*, 2018.

[207] G. A. Cohen, J. S. Chase, and D. L. Kaminsky, "Automatic Program Transformation with JOIE," in *Proceedings of the 1998 USENIX Annual Technical Conference (ATC)*, Jun. 1998.

[208] G. Kniesel, P. Costanza, and M. Austermann, "JMangler – A Framework for Load-Time Transformation of Java Class Files," in *Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation*, IEEE, 2001, pp. 98–108.

[209] E. Bruneton, R. Lenglet, and T. Coupaye, "ASM: a code manipulation tool to implement adaptable systems," *Adaptable and extensible component systems*, vol. 30, no. 19, 2002.