DEFEATING CRITICAL THREATS TO CLOUD USER DATA IN TRUSTED EXECUTION ENVIRONMENTS

by

Adil Ahmad

A Dissertation

Submitted to the Faculty of Purdue University

In Partial Fulfillment of the Requirements for the degree of

Doctor of Philosophy

Department of Computer Science

West Lafayette, Indiana

August 2022

THE PURDUE UNIVERSITY GRADUATE SCHOOL
STATEMENT OF COMMITTEE APPROVAL

Dr. Pedro Fonseca, Co-Chair

Department of Computer Science

Dr. Byoungyoung Lee, Co-Chair

Department of Electrical and Computer Engineering, Seoul National University

Dr. Aniket Kate

Department of Computer Science

Dr. Changhee Jung

Department of Computer Science

Dr. Dongyan Xu

Department of Computer Science

Approved by:

Dr. Kihong Park

To my family, both given and chosen.

ACKNOWLEDGMENTS

This dissertation is the culmination of efforts from several individuals in my professional and personal life, to whom I will remain forever indebted.

First and foremost, I would like to express my sincere gratitude to my advisors—Pedro Fonseca and Byoungyoung Lee—for their love and support throughout my doctoral studies. Byoungyoung's kind, fun-loving, and empathetic nature was the major reason I decided to pursue computer security as a field of study, despite pursuing my undergraduate thesis in the area of computer networks. I will remain indebted to him for inculcating in me a love of trusted execution environments. Even though Pedro joined as an advisor after my third year as a doctoral student, I learned a great deal about how to be passionate and meticulous about my research from him. His advice on improving my presentation and paper writing skills have enabled me to bring significant more accessibility to my work.

Over the course of my studies, I was extremely fortunate to undertake several internships at industry research labs like Microsoft Research and NEC Labs America. There is no exaggeration when I say that I found the absolute best mentors in Sangho Lee, Marcus Peinado, and Chung Hwan Kim, at these internships. Sangho's passion for research and knowledge of implementation details was instrumental in helping me complete several research projects. I will always be thankful to him for helping me, not just during the internships, but even in the writing of this dissertation. Marcus has an excellent eye for locating high-impact research problems and carefully cross-examining every small detail in a potential solution. I strongly believe that I improved my own abilities in this important domain by working with Marcus. I remain thankful to Chung Hwan, who was my first mentor outside Purdue, for allowing me the space to grow as a young researcher at NEC Labs.

Perhaps I would never have dreamt of pursuing a doctorate, if it was not for the efforts of my undergraduate thesis advisors—Ihsan Ayyub Qazi and Muhammad Fareed Zaffar—at Lahore University of Management Sciences (LUMS). Ihsan's undergraduate summer research program introduced me to the concept high-quality research, and it was his teaching passion that significantly contributed to my own pursuit of an academic faculty career after the completion of this dissertation. The help from Ihsan and Fareed, during my PhD search,

was monumental in getting a position at Purdue. Fareed also provided me with ample opportunities to establish my credentials in computer security undergraduate research.

I would also like to thank my other thesis committee members—Aniket Kate, Dongyan Xu and, Changhee Jung—for their advice. I remain especially thankful to Dongyan and Aniket for motivating me to apply for academic faculty positions at separate occasions.

The material in this dissertation is spawned from several research projects, none of which would be possible if not for the help I received from many different collaborators. These collaborators include Jaebaek Seo, Kyungtae Kim, Juhee Kim, Yuan Xiao, Muhammad Abubakar, Seounghyun Park, Hyunyoung Oh, Yinqian Zhang, Insik Shin, and Yunheung Park. I want to especially thank Jaebaek, since he was the first to help me navigate research and his examples significantly improved my coding and development skills.

Although I never had the opportunity to work on research projects with other members of my research group—Sishuai Gong, Tapti Palit, Congyu Lu, Dinglan Peng, Paul Pok Hym, and Sruthi Panchapakesan—I greatly enjoyed my time with them. I want to especially express my gratitude to Tapti for spending so much time to help me improve my faculty job talk.

During my doctoral studies, I went to the Fiesta Mexican grill in West Lafayette countless times. The owner's constant happy face and our chats often made my day, especially when I seem defeated by a research matter. I would like to thank him for his chats.

Last but not least, I remain indebted to my given and chosen family. My parents and siblings supported me throughout the process. My partner's constant love and support was the bedrock of my foundation and this dissertation would not be possible without her. My brother's advice was instrumental in improving the quality of my research. My friends—Ali Shan, Muhammad Mustafa, Habiba Farrukh, Bader AlBassam, Aqib Nisar, and Zeeshan Hakim—were my tether and I remain thankful for their constant support, even when I did not deserve it. This dissertation is dedicated to all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

In today's world, cloud machines store an ever-increasing amount of sensitive user data, but it remains challenging to guarantee the security of our data. This is because a cloud machine's system software—critical components like the operating system and hypervisor that can access and thus leak user data—is subject to attacks by numerous other tenants and cloud administrators. Trusted execution environments (TEEs) like Intel SGX promise to alter this landscape by leveraging a trusted CPU to create execution contexts (or *enclaves*) where data cannot be directly accessed by system software. Unfortunately, the protection provided by TEEs cannot guarantee complete data security. In particular, our data remains unprotected if a third-party service (e.g., Yelp) running inside an enclave is adversarial. Moreover, data can be indirectly leaked from the enclave using traditional memory side-channels.

This dissertation takes a significant stride towards strong user data protection in cloud machines using TEEs by defeating the critical threats of adversarial cloud services and memory side-channels. To defeat these threats, we systematically explore both software and hardware designs. In general, we designed software solutions to avoid costly hardware changes and present faster hardware alternatives.

We designed 4 solutions for this dissertation. Our Chancel system prevents data leaks from adversarial services by restricting data access capabilities through robust and efficient compiler-enforced software sandboxing. Moreover, our Obliviate and Obfuscuro systems leverage strong cryptographic randomization and prevent information leakage through memory side-channels. We also propose minimal CPU extensions to Intel SGX called Reparo that directly close the threat of memory side-channels efficiently. Importantly, each designed solution provides principled protection by addressing the underlying root-cause of a problem, instead of enabling partial mitigation.

Finally, in addition to the stride made by our work, future research thrust is required to make TEEs ubiquitous for cloud usage. We propose several such research directions to pursue the essential goal of strong user data protection in cloud machines.

1. INTRODUCTION

Users increasingly send sensitive data, like personally identifiable and healthcare information, to cloud machines for processing. Unfortunately, this data is exposed to attacks from numerous vectors (e.g., untrusted cloud administrators and other machine tenants). As evident in several high-profile data breaches, like the T-Mobile breach that exposed the social security number (SSN) of millions of people, even some of our most crucial and privacy-sensitive data is not secure. Such data breaches affect everyone – governments, businesses, and individual citizens. Moreover, despite strict data protection regulations (e.g., GDPR, CCPA, HIPAA) introduced to reduce breaches, data breaches are still frequent.

Protecting user data on cloud machines is challenging because the system software (e.g., operating system) is *untrusted*, despite having the ability to access and thus leak user data. This lack of trust stems from several factors. In particular, modern system software is exceedingly complex—tens of millions of lines of code [1] needed to implement various functions—leading to numerous vulnerabilities. Unlike personal machines, where a large system software executes too, cloud machines are also readily accessible to numerous tenants. This access lowers the bar significantly for an adversarial tenant to compromise the system software. The most important concern with cloud machines, however, is the fact that *we cannot even trust a vulnerability-free system software implementation* (e.g., like the verified seL4 operating system kernel [2]) because it would be configured by untrusted cloud administrators. Notably, cloud administrators control privileged software (e.g., hypervisors) and build baseline system images used by cloud users (e.g., VM and container images). Hence, it is trivial for administrators to leverage their control over cloud machines and collect sensitive data without a trace visible to users.

1.1 Trusted Execution Environments (TEEs) for Data Protection

One of the most promising solutions to protecting user data in cloud machines is hardware-assisted **trusted execution environments (TEEs)**. These features allow CPUs to create protected execution regions, called **enclaves**, where software can securely compute on sensitive user data. Enclaves cannot be directly accessed by any software executing outside the enclave's

trusted boundary, including privileged software like the operating system, hypervisor, and BIOS. Hence, sensitive user data in enclaves is protected even if all external machine software is malicious. Moreover, computations running in enclaves can execute alongside traditional computations on the same machine. Therefore, cloud providers do not need to purchase extra hardware to support enclaves which makes them cost-effective. Examples of TEEs supported by commodity server CPUs include Intel SGX [3], AMD SEV [4], ARM TrustZone [5], and the upcoming ARM CCA [6].

Due to their desirable properties, TEEs are widely-deployed on cloud machines and increasingly leveraged for sensitive data protection. In particular, major cloud providers, including Microsoft Azure [7], IBM Cloud [8], and Alibaba Cloud [9], allow users to run SGX enclaves on their cloud machines. In addition, many companies are using enclaves to provide privacy-preserving user services, which allows them to standout amongst competitors with traditional solutions. For instance, Signal uses enclaves to enable its private contact discovery service, where users can locate other users of the Signal messenger without leaking their contact information [10]. Other emerging enclave applications include privacy-preserving machine learning analytics on confidential patient data in cloud machines [11].

1.2 Threats to User Data Protected by TEEs

Although TEEs protect user data in cloud machines from several important attacks (e.g., direct memory reads from operating system), there remain critical unprotected attack vectors. In this dissertation, we explore these attack vectors from the perspective of Intel SGX—the most widely-deployed TEE in cloud machines. Nevertheless, these attack vectors are also a problem for other TEE implementations, and our work can be generalized to other TEEs (as discussed in §7). The rest of this section provides a brief overview of the attack vectors.

1.2.1 Adversarial Cloud Services

Enclaves allow users to secure sensitive data provided to their own enclave code on a remote machine. This design excludes an important model where users only send their data to a cloud machine and receive a service from a provider's code—we call this the *cloud service*

model. The machine in this model could be privately-owned by the service provider or rented from a public cloud (refer to §4.1 for a full model description). An example of a cloud service is a genetic analytic service (e.g., 23andMe [12]) which analyzes their user's DNA sequence on cloud machines. If such a service is *adversarial*, it can collect user data (e.g., to sell) even without asking users for permission. Unfortunately, the provider can still trivially collect user data even if they run the service inside an enclave (e.g., to appease user concerns), since enclave code has full control over the provided data.

1.2.2 Memory Side-Channels

Enclaves are appealing because they reduce deployment costs by sharing machine resources (like CPU and memory) with non-enclave software. However, sharing resources also allows untrusted non-enclave software to leak user data through *memory side-channels* borne out of contention of resources [13]–[17] and control of important resources (e.g., page tables [18], [19]). Memory side-channels can reveal a variety of sensitive enclave content from cryptographic keys [14] to database queries [20]. Please refer to §2.5 for a detailed description of memory side-channels and an end-to-end attack. Unfortunately, without effective defense mechanisms, memory side-channels potentially render enclaves inapplicable for high-security use cases.

1.3 Thesis and Research Question

Our thesis is that in the pursuit of strong user data protection on cloud machines, trusted execution environments (TEEs) provide a promising foundation for us to build upon. Naturally, this raises a pertinent question: *how to protect user data in TEEs against the critical threats of adversarial cloud services and memory side-channels?*

1.4 Contributions

In this dissertation, we answer our posed research question by undertaking a systematic exploration of both software and hardware solutions to protect user data in TEEs. Initially, to avoid complex hardware modifications—which might be hard to adopt—we designed several software solutions to protect enclave data from adversarial services and memory side-channels.

However, given the hardware root cause of memory side-channels, more efficient side-channel protection can be achieved using hardware solutions. Hence, we also designed hardware extensions to efficiently protect enclaves against memory side-channels while minimizing costly architectural changes. Through such a systematic study, we believe that this dissertation significantly enhances the promise of enclaves as a cloud user data protection solution.

We contribute the design and implementation of four solutions in this dissertation. Three of these solutions were presented at a top-tier computer security conference, while the last system is being prepared for submission. The rest of this section provides a brief overview of these systems and the conclusions that can be drawn from each research work.

- **Chancel [NDSS 2021]**. Adversarial cloud services running inside enclaves can collect user data because the enclave code has unrestricted access to the provided user data. Our Chancel system addresses this attack vector by restricting data access capabilities of untrusted enclave code using compiler-enforced sandboxing (also called *software fault isolation*). Chancel encrypts all user data sent outside the enclave and prevents indirect data leaks through several covert channels. Moreover, Chancel efficiently supports multi-client programs by enabling secure data sharing between different enclave threads. Through Chancel, we show that robust user data protection under adversarial services is achievable with an order of magnitude higher memory efficiency and performance, compared to existing sandbox implementations.

- **Obfuscuro [NDSS 2019]**. Memory side-channels disclose *distinguishable* traces of an enclave's execution outside the enclave, which reveal sensitive data. Obfuscuro provides principled protection against *all* memory side-channels in a program by making all enclave traces *indistinguishable*. This indistinguishability property does not only apply to different executions of the same program but also holds across different programs. Obfuscuro achieves such principled protection through a scratchpad-based execution model and a strong cryptographic randomization technique called Oblivious RAM (ORAM). This research shows that principled protection for enclaves against memory side-channel attacks is an achievable goal.

- **Obliviate [NDSS 2018]**. File systems form the cornerstone of many important cloud programs that handle sensitive data (e.g., web servers and databases). Unfortunately, enclaves are user-mode and rely on an untrusted file system (controlled by the operating system). Our case-study shows that even though file encryption prevents direct data leak from sensitive files, attackers can indirectly leak data through file system access traces (e.g., observed through system calls and memory side-channels). To avoid this data leak, we designed the OBLIVIATE[1] file system, which transforms all file system operations (e.g., read, write, sync) into *data oblivious* variants. OBLIVIATE also implements several systematic optimizations (e.g., intelligent memory utilization and asynchronous processing) to improve performance of file system operations. Through this work, we both shine a light on a previously unexplored information leak source for enclaves (i.e., file systems) and design a strong protection approach.

- **Reparo [under preparation]**. This shows that a tiny set of architectural extensions to close several memory-based side-channels in Intel SGX and enable efficient side-channel protection. The extensions are designed only using widely supported Intel CPU features to ensure easy future implementation on Intel CPUs. REPARO incurs a performance reduction of only 17% compared to native enclave execution, more than 60× faster than software solutions with strong side-channel protection. Using REPARO, we show that contrary to popular belief, strong architecture-level side-channel protection requires only minor CPU extensions and modest performance costs.

1.5 Outline

The rest of this dissertation is structured as follows. §2 discusses the relevant background and §3 presents an overview of the related work. §4 presents our security enforcement to address the threats posed by adversarial programs in enclaves. §5 introduces the design of a principled software solution to address information leak through memory side-channels inside enclaves. §6 describes novel hardware extensions to enclave designs that efficiently address

[1]↑For brevity, important insights gathered from OBLIVIATE were incorporated in §5 (without full details). Please refer to the OBLIVIATE research paper [20] for full details.

memory side-channels. §8 discusses future research directions this dissertation can spawn and finally the dissertation concludes in §9.

2. BACKGROUND

This chapter provides the background needed to understand this dissertation. It is divided into six parts. The first part provides a brief overview of computer systems and architecture (§2.1), followed by a second part with relevant security concepts (§2.2) Since our work employs trusted execution environments (TEEs) for data protection, the third part provides a brief overview of TEEs (§2.3), followed with a fourth part providing an an in-depth explanation of Intel SGX (§2.4). One focus of this dissertation is towards defeating memory side-channels; hence, the fifth part overviews memory side-channels (§2.5). The sixth and final part describes Oblivious RAM (ORAM) (§2.6), a technique we employ to defeat memory side-channels.

2.1 Computer Architecture and Systems Overview

Subtle architectural and system aspects of modern computers play an important role in the security of TEEs. In general, a computer has two main hardware resources: the processor and memory. The memory contains some information that the processor uses to obtain a computational result through a sequence of coded operations (or *instructions*). The processor is a sealed package that contains one or more Central Processing Unit (CPU) cores, each of which can independently execute instructions. Since these hardware resources must be used by many different tasks, it is the duty of the system software to divide these resources. The remainder of this section provides further relevant details about CPU execution (including modern optimizations) (§2.1.1) and the system software's role (§2.1.2).

2.1.1 CPU Execution and Optimization

CPUs execute instructions in a pipeline that consists of four main stages: fetch, decode, execute, and write-back. In older CPUs, the main bottleneck in this pipeline was the very slow DRAM access for fetch and write-back stages. Modern CPUs implement three major optimizations to mitigate this bottleneck: simultaneous multi-threading (SMT) (also called hyper-threading), caching, and out-of-order execution. Since these optimizations have serious security consequences (described in §2.5.1), we briefly explain them next.

Simultaneous multi-threading (or hyper-threading). In many modern computers (especially Intel and AMD-based), each CPU core contains two or more logical threads (or *hyper-threads* [21]) that can execute two separate stream of instructions simultaneously. If a hyper-thread is stalled at a memory fetch or write-back operation, the other thread uses the CPU core's decode and execution units to maximize the overall utilization of the CPU.

Caching. Modern computers feature several caches where memory is retrieved and stored for small periods to avoid expensive continuous DRAM accesses. These caches are implemented as a hierarchy, where caches closer to CPUs (namely L1 and L2 caches) are smaller but faster, while the cache closer to memory (namely the last-level cache) is larger and slower [22].

Out-of-order execution. CPUs also exploit instruction-level parallelism to execute different instructions *out-of-order* but retire their results to memory *in-order* of their occurrence. This avoids busy-waiting on memory fetch and write-back. However, out-of-order execution encounters a problem when the code contains *branches*—the code jumps to a different set of instructions depending on a certain condition. Traditionally, a CPU could not execute past the branch until its outcome is known. This problem is solved by branch prediction [15] and speculative execution. In particular, a *branch prediction unit* (BPU) predicts the outcome of the branch by keeping a history of previously-taken program branches and speculatively continues execution. If the predicted branch outcome is incorrect (which is rare), the speculated results are rolled back and the execution is repeated with the correct branch.

2.1.2 Privileged System Software

Computers must execute many computations but they cannot all execute at once due to limited CPU and memory resources. Therefore, computers run *privileged system software*, like the operating system and the hypervisor, whose main role is to allocate hardware resources to computations at different times. Unfortunately, both system software can abuse their privilege to leak information through side-channels (refer to §2.5.1). The key privilege that enables system software to open such side-channels comes from the concept of virtualization and one of the major components of virtualization, memory address translation. We explain both in the remaining paragraphs of this section.

Virtualization. Both the operating system and hypervisor heavily rely on the notion of *virtualization* to allocate hardware resources to different computations. The idea is to create a virtual environment where a computation executes as if it were executing on its own physical machine. Due to this virtual environment, developers can ignore other computations while writing software and it minimizes interference during execution. The operating system allocates resources to software computations that we call *processes*, while the hypervisor allocates resources to multiple operating system instances (also called *virtual machines*).

Memory address translation. One of the most important cornerstone of virtualization is *memory address translation*. Each software accesses memory through a set of virtual addresses that are translated to the physical memory address on DRAM modules. This allows isolation between memory contents of different computations. The system software decides how addresses are translated by dividing memory into small regions called *pages* (usually 4 KB) and setting up *page tables*. In modern computers, page tables are operated by hardware units in processors called the memory management units (MMUs) [23]. Such hardware operation ensures efficient and transparent translation.

Memory address translation using page tables also allows a system software to extend a machine's available memory using an external *disk* device. In particular, when the machine runs out of memory, the system software copies pages to disk and sets their corresponding page table entries as invalid. If a software accesses a page whose page table entry is invalid, the MMU raises a *page fault*. The operating system catches this fault and retrieves the required page from disk. Since page swaps are expensive, computers try to minimize them through different heuristics. For instance, a well-known heuristic is the least-recently used (LRU) algorithm—the system software assumes that memory pages recently used by a computation will be used again; hence, it tries not to swap them to disk. To support such heuristics, the MMU marks the memory regions written or read by computations (during address translation) in dirty and access bits inside their page table entries [23].

2.2 Computer Security and Cryptography Overview

Many security properties of TEEs are derived from cryptographic primitives. This section will provide a limited background of major security and cryptography concepts required for understanding this dissertation. Please refer to other sources [24], [25] for a more in-depth explanation of these concepts.

Several concepts we are interested in rely on a primitive called *cryptographic keys*, a piece of information used alongside a cryptographic algorithm that determines the algorithm's operation [26]. Cryptographic keys must be disclosed only according to certain rules. These keys are generated using a key generation algorithm that uses a *random number* to ensure uniqueness. Key generation can be *symmetric* or *asymmetric*. Symmetric key cryptography produces a key called *shared secret key*, which should only be disclosed to trusted participants of a system. An example of a symmetric key algorithm is Advanced Encryption Standard (AES). Asymmetric key cryptography, on the other hand, uses two keys—*public* and *private*. The public key can be disclosed to all participants while the private key should be kept secret by the participant that generated it. An example of an asymmetric key algorithm is Rivest-Shamir-Adleman (RSA).

In computer systems, an important security property provided by cryptographic primitives is *confidentiality*—unauthorized users in a system should not be able to access sensitive contents [27]. This is a useful property when untrusted system participants can watch messages passed between trusted participants (e.g., over a public communication channel). Cryptosystems guarantee confidentiality using keys and the concepts of *encryption* and *decryption*. Encryption is the cryptographic transformation of data (called plaintext) into a form (called ciphertext) that conceals the data's original meaning to prevent it from being known by unauthorized users. The corresponding reversal process is called decryption which is a transformation that restores ciphertext to its original plaintext [28].

In many instances, confidentiality cannot be guaranteed without two additional properties—*integrity* and *freshness*. Integrity is the property that data or information has not been altered in an unauthorized manner [27]. One way of ensuring integrity is using *secure hashing* functions (e.g., SHA) that transform an unbounded input to a small fixed output (called an

integrity hash). The hashing is *one-way*, i.e., an adversary cannot recreate the original input through the integrity hash. A trusted entity can use the integrity hash to check if the data is in its correct state. Freshness is a property that ensures that either the system will have the most updated version of data or it will detect an attack [24]. This ensures that an attacker cannot *replay* old data and revert a system to its old state. Typically, cryptosystems provide freshness by encoding unique information (*nonce*) each time an integrity hash is created.

Finally, apart from general cryptographic concepts, it is important to understand the concept of Trusted Computing Base (TCB). This refers to the cummulative protection mechanisms, including software, hardware and firmware, that are responsible (and must be trusted) for the enforcement of a security property [29].

2.3 Trusted Execution Environments (TEEs)

System software[1] is the most critical part of a machine's software infrastructure because all other software depend on its correctness. However, system software cannot be trusted due to several reasons. In particular, system software has seen an exponential growth over the years due to an increasingly diverse range of expected functions (e.g., supporting many devices). The Linux operating system kernel, for instance, grew from 2.4 million lines of source code in 2001 [1] to a staggering 27.8 million lines of source code in 2020 [30]. Unfortunately, because larger kernels increase the TCB, systems have become increasingly vulnerable to attacks that exploit defects to take complete control of the machine. Moreover, even if we can develop bug-free system software (e.g., formally verified operating systems [2]), we cannot trust such software in cloud machines where it is configured by untrusted administrators.

The distrust of system software has fueled interest in a research domain called Trusted Execution Environments (TEEs). The core idea of TEEs is to have sensitive code and data execute in isolated containers (or *enclaves*) that are inaccessible to even system software. Enclaves share system resources (including CPU and memory) with other software components on the machine. However, the CPU enforces isolation to ensure confidentiality and integrity of sensitive enclave computations, if the system software is malicious. While there are many

[1] ↑From this point onwards, we use system software and operating system interchangeably

different variants of TEEs (some of which are discussed in §3.1), the next section provides details about Intel SGX (the TEE under consideration of this dissertation).

2.4 Intel Software Guard eXtensions (SGX)

Intel SGX [3] is the most widely-deployed TEE implementation in cloud machines. SGX allows a user process to create an enclave region in its virtual address space. This section provides an overview of the critical SGX aspects that are relevant to this dissertation. For the sake of simplicity, we only overview SGX in terms of the operating system and the user process. Please refer to other sources [23], [24], [31], [32] for understanding the interactions between the hypervisor and the operating system and more comprehensive information regarding SGX.

2.4.1 Platform Secret

Each SGX CPU is manufactured with an embedded random and very long secret. This secret is utilized to create a unique *platform secret key*. Intel uses this key to verify its CPUs and allow users to attest that their computations are running inside SGX enclaves. Details about attestation are provided later in this section.

2.4.2 Physical Memory Organization

The SGX CPU reserves a region of physical memory during system boot for assigning pages to enclaves and maintaining internal SGX data structures. This physical memory region is called the processor reserved memory (PRM). The size of the PRM region depends on the computer's BIOS settings. In the earlier versions of SGX, the PRM size was limited to 256 MB, but it can cover a region up to 512 GB in the latest versions [33].

The PRM is reserved using a special CPU register, called the variable memory type range register (vMTRR). The vMTRR contains a base and size component, which reference a contiguous physical region on the DRAM. On each memory access, the CPU's memory management unit (MMU) uses the vMTRR to check if an access falls in the PRM and aborts all access from non-enclave software. Given the contiguous region requirements of the

vMTRR, this check is very efficient [24]. The CPU also protects the PRM from malicious devices by aborting direct memory access (DMA) requests to its memory.

Within the PRM, there are two important sub-regions: the enclave page cache (EPC) and the enclave page cache map (EPCM). The EPC is where SGX allocate enclave pages from. While most pages allocated to an enclave from the EPC are visible to the enclave program (e.g., pages that hold the program's code and data), SGX maintains some pages internally for important per-enclave data structures. One such data structure is the SGX enclave control structure (SECS), containing important information like the enclave measurement for remote attestation (§2.4.5) and setup parameters (e.g., number of allowed threads, size of the heap, etc.). SGX employs the EPCM data structure to track the EPC's status. This structure contains vital information like which EPC page was allocated to an enclave and a *reverse mapping* of enclave virtual addresses to corresponding physical pages.

2.4.3 Instruction Set Architecture (ISA) Extensions

SGX extends the Intel x86 ISA with a set of new instructions (Table 2.1), provided to both privileged software and user computations. Privileged software is allowed to execute SGX supervisor instructions to manage enclaves. This management includes initialization of an enclave within a user process, allocation or deallocation of EPC pages to an enclave, and terminating enclaves. Once the privileged software creates an enclave region for the process, the process executes SGX user instructions to transition between the enclave and non-enclave world. In particular, the user process transitions to the enclave world to start secure computation, while the enclave exits back to the non-enclave world to leverage a privileged software service (e.g., execute a system call, service an interrupt). A complete end-to-end enclave lifecycle is provided in §2.4.4.

2.4.4 Enclave Lifecycle

This section explains an enclave's lifecycle, from initialization to termination.

Initialization. The operating system creates an enclave context inside a user process' virtual address space using `ECREATE`. This instruction requires supplying initial enclave parameters

Table 2.1. A list of supervisor and user instructions added to the Intel x86 ISA by SGX [23], [32]. Within each category, the instructions are ordered alphabetically.

Instruction	Description
Supervisor instructions:	
`EADD`	Allocate an EPC page to an enclave
`EAUG`	Add a page to an initialized enclave
`EBLOCK`	Mark a page in EPC as blocked
`ECREATE`	Create an enclave
`EDBGRD`	Read from a debug enclave
`EDBGWR`	Write from a debug enclave
`EEXTEND`	Measure an EPC page for attestation
`EINIT`	Finalize enclave initialization
`ELDB/ELDU`	Swap a backing store page to the EPC
`EMODPR`	Restrict permissions of an EPC page
`EMODT`	Change the type of an EPC page
`EPA`	Add version array
`EREMOVE`	Remove an EPC page from an enclave
`ETRACK`	Activates EBLOCK checks
`EWB`	Swap an EPC page to backing store
User instructions:	
`EACCEPT`	Accept changes to an EPC page
`EACCEPTCOPY`	Initialize changes to a pending page
`EENTER`	Enter an enclave for the first time
`EEXIT`	Exit an enclave (e.g., interrupt, system call)
`EGETKEY`	Retrieves a cryptographic key
`EREPORT`	Create a cryptographic report
`ERESUME`	Resume an enclave after exit

(e.g., initial virtual address size, number of threads, stack and heap size). These parameters are saved in the SECS (§2.4.2). Once an enclave context is created, the operating system populates the enclave with initial code and data, copied from untrusted memory into an EPC page (allocated to the enclave) using the **`EADD`** instruction.

After populating the initial enclave contents into EPC pages, the operating system must also initiate a measurement of these contents using the **`EEXTEND`** instruction. This instruction generates a SHA-256 integrity hash (§2.2) of the populated EPC page's contents and offset within the enclave. The offset ensures that the operating system creates an enclave memory layout according to the user's specifications. The integrity hash is extended with the hash calculated from any previous page. The final integrity hash—reached after the enclave is fully populated and initialized using **`EINIT`**—is called the enclave measurement (or MRENCLAVE).

The MRENCLAVE is stored in the enclave's SECS. Without generating and finalizing MRENCLAVE, SGX does not allow the enclave to start executing.

Importantly, the initial enclave contents should not have any sensitive data, since these contents are visible to the operating system. It is expected that, once the enclave is initialized, the user will initiate remote attestation (§2.4.5) with the enclave to verify the enclave's correctness (using MRENCLAVE) before sending sensitive data. The user can also implement an in-enclave loader [34], [35] to load sensitive code after remote attestation.

Execution. Once the enclave context is initialized (using `EINIT`), the user process transitions into the enclave (using `EENTER`). The enclave must periodically pause its execution and return to the untrusted non-enclave world (e.g., to let the operating system service an interrupt). The paused execution is later resumed.

An enclave stops its execution using `EEXIT` to let the operating system handle processor interrupts/exceptions (e.g., timer interrupts and page faults) and system calls. Enclave exits to handle processor interrupts and exceptions are called *asynchronous exits* because the enclave has no control over when such an exit happens. We provide additional details about asynchronous exits due to page faults in §2.4.6. System call exits, on the other hand, are *synchronous*. Regardless of the exit type, the SGX CPU saves the enclave's processor state (e.g., registers, flags, etc.) inside the enclave's state save area (SSA), a per-thread reserved enclave page [23]. After the operating system has finished its task, it resumes the user process. At this point the process is executing in a non-enclave state. To context switch into the enclave, the process executes `ERESUME`. On this instruction, the enclave state stored in the SSA is restored. This state save and restore prevents the malicious operating system from manipulating the enclave through its register context [36].

Termination. When the enclave has completed its execution, the user process asks the operating system to terminate the enclave context within the process's address space. The operating system executes `EREMOVE` to reclaim all EPC pages allocated to the enclave. On `EREMOVE`, SGX clears the entire page to ensure that a future enclave cannot access prior enclave's contents.

2.4.5 Remote attestation

Users can attest that their programs are correctly loaded onto SGX enclaves using remote attestation before sending sensitive data. Generally, there are two steps needed for remote attestation. Both steps are explained in this section.

Platform provisioning. Before an SGX CPU can be used for remote attestation, it must be provisioned by Intel. For provisioning, SGX initiates two special enclaves: the provisioning and quoting enclave. These enclaves do not contain user programs, rather they are built by Intel only for attestation purposes. The provisioning enclave executes `EGETKEY` to derive a provisioning key based on the platform secret (§2.4.1) and sends it to an Intel provisioning service [32]. If the key is verifiable, Intel's provisioning service sends an *attestation key* to the provisioning enclave. The provisioning enclave seals the attestation key using a provisioning seal key (also derived from the platform secret) and hands it to the system software for storage. The sealing of the attestation key ensures that platform provisioning is only needed once even when the machine switches owners [24], ensuring machine ownership switch is not known to Intel. The attestation key is then unsealed by the quoting enclave to subsequently attest enclaves created for the user by the operating system.

Local attestation and reporting. Once the platform is provisioned, any enclave can *locally* attest itself to the quoting enclave and receive an attestation digest. The enclave sends the attestation digest to the user for verification and shared secret establishment.

An enclave that wants to be attested executes `EREPORT`. On this instruction, the SGX CPU retrieves the enclave's MRENCLAVE (§2.4.4) and sends it to the quoting enclave. `EREPORT` additionally accepts some data that the enclave wants to include in the report. Typically, this data is the initial secret that the enclave and user will employ to establish a shared secret key (e.g., through diffie-hellman). The quoting enclave uses the information provided by the SGX CPU (through `EREPORT`) and create an attestation digest, signed with the platform's attestation key. The digest is sent back to the enclave that requested attestation.

The enclave will then send this digest to the user for enclave verification and establish a shared secret. On receiving the digest, the user first queries the Intel attestation service to verify the correctness of the signature on the digest. Once the digest is verified, the user

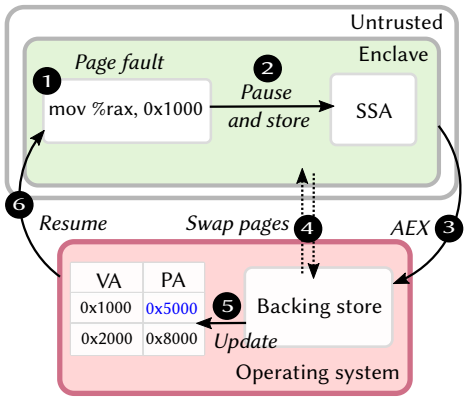


Figure 2.1. SGX page fault handling. The dotted lines denote operations performed using SGX microcode instructions.

recreates the MRENCLAVE on their local machine to validate that initial enclave contents were correctly loaded by the operating system. If MRENCLAVE is valid, the user leverages the secret reported in the digest to establish a shared secret key with the enclave.

2.4.6 Demand paging

SGX allows the operating system to over-subscribe the EPC pages to many enclaves using *demand paging*. On page faults, SGX demand paging requires the operating system to (a) swap enclave pages between a *backing store* in the untrusted memory and EPC and (b) update the enclave page tables to reflect changes.

Importantly, even though the operating system is allowed to implement demand paging, SGX ensures several important security properties. First, SGX ensures the confidentiality, integrity, and freshness for EPC pages that are swapped to the backing store. Confidentiality is achieved by encrypting enclave pages using a per-enclave secret key, while integrity and freshness is achieved using message authentication codes (MAC) and page versions, respectively [24], [32]. Second, SGX ensures that the operating system cannot maliciously modify the memory layout of an enclave (e.g., swap a different physical page to a virtual address) in the process' page tables. This is achieved using the reverse address mapping (virtual to physical address) of each enclave page in the EPC, stored inside the enclave page cache map (EPCM) (§2.4.2). If the operating system maps the wrong physical page to an

enclave virtual address, the SGX CPU will abort on an enclave access to that page because the reverse mapping will fail. When an EPC page is swapped to the backing store, its reverse mapping is also stored alongside the page's contents [24].

Figure 2.1 provides a simplified view of SGX enclave's page fault handling. The enclave tries to access a part of the enclave region whose corresponding physical page is not valid, therefore, a page fault happens (❶). The SGX CPU stops the enclave and saves the enclave's context within the save state area (SSA) (❷). Then, the CPU performs an asynchronous enclave exit (AEX) and transfers execution to to the operating system's page fault handler (❸). If there are no free EPC pages in the EPC, the fault handler uses an SGX microcode instruction, `EWB`, to free an EPC page by copying its encrypted and integrity-protected contents to the backing store. Afterwards, the fault handler uses `ELDU` to load the faulted backing store page into the EPC (❹). On `ELDU`, SGX first verifies the integrity and freshness of the backing store page. Finally, the fault handler updates the page tables to reflect that the page is available (❺) and resumes enclave execution (❻)

2.4.7 Security Overview

SGX promises confidentiality and integrity of enclave computations. Table 2.2 provides an overview of how SGX's promises hold up against various attacks from system software during important SGX operations (e.g., platform provisioning) and all stages of an enclave's lifecycle. Since we already explain different SGX defense mechanisms in previous sections, we refer the reader to previous sections (also marked in Table 2.2).

In addition to protection against malicious system software, SGX also protects against physical attackers (e.g., untrusted cloud administrators) that try to inspect DRAM memory contents (e.g., through cold boot attacks [37]). SGX achieves this protection by encrypting all EPC pages on the DRAM. The encrypted pages are only decrypted within the trusted CPU caches. Thus, a physical attacker is unable to access decrypted contents.

Notably, all of SGX's protections are geared towards attacks that *directly* harm an enclave's confidentiality and integrity. For instance, attacks that allow the operating system to directly access enclave memory and extract sensitive information. While this is a significant step in

Table 2.2. An overview of SGX's defense against attacks from a malicious operating system during all important SGX operations and stages of an enclave's lifecycle. In terms of attack goal, C+I means attacks that can compromise both confidentiality and integrity, C means attacks that only hurt confidentiality, and I means attacks that hurt integrity. In many instances, attacks that hurt integrity can also hurt confidentiality.

Attack goal	Detailed method	SGX defence
During platform provision		
C+I	Steal platform secret	Platform secret is burned into CPU and not revealed to software (§2.4.1)
C+I	Steal provisioning seal key	Key can only be accessed by a signed Intel provisioning enclave (§2.4.5)
C+I	Steal attestation key	Key can only be accessed by a signed Intel quoting enclave (§2.4.5)
During enclave creation		
C	Extract secrets from initial code/data	Initial code/data is not secret, otherwise use an in-enclave loader (§2.4.4)
I	Load malicious code/data	Caught on MRENCLAVE verification during remote attestation (§2.4.4)
I	Load code/data at incorrect offsets	Caught on MRENCLAVE verification during remote attestation (§2.4.4)
I	Extend measurement using non-enclave page	`EEXTEND` only accepts EPC pages assigned to current enclave (§2.4.4)
I	Compromise measurement after initialization	MRENCLAVE is stored in the enclave's SECS, accessible to only SGX (§2.4.4)
During enclave execution		
C	Directly read from enclave pages in EPC	EPC pages are inaccessible outside an enclave (§2.4.2)
C	Directly read from enclave pages in backing store	Backing store pages are encrypted with a key known only to SGX (§2.4.6)
C	Assign enclave pages to a malicious enclave	EPC pages from one enclave cannot be assigned to another (§2.4.4)
I	Directly write to enclave pages in EPC	EPC pages are inaccessible outside an enclave (§2.4.2)
I	Directly write to enclave pages in backing store	Integrity of backing store page is checked before inserting them (§2.4.6)
I	Replay old enclave pages during page faults	Freshness of backing store pages is checked before inserting (§2.4.6)
I	Modify enclave register state on enclave exits	Enclave register state is stored inside the enclave (in SSA) (§2.4.4)
I	Modify enclave virtual memory layout in page tables	Leverage reverse map (EPCM) to ensure correct layout (§2.4.6)
During enclave termination		
C	Assign uncleared EPC pages to a malicious enclave	Pages not cleared (with `EREMOVE`) cannot be assigned (§2.4.4)
During remote attestation		
C+I	Replay old measurement	Random nonces in communication to ensure freshness [24]

the right direction, it excludes all attacks that *indirectly* extract sensitive contents from the enclave memory. In the next section, we describe memory side-channels as a critical example of indirect attacks that SGX does not protect enclaves from. The remaining attacks found against SGX enclaves are described in §3.5.

2.5 Memory Side-Channels

A side-channel is an aspect of a computer system's behavior that can be used to infer secret information [38]. Side-channels arise from the physical implementation of the computer system, rather than a defect in its algorithms. Common computer system aspects that are leveraged in side-channels include timing and power consumption [39].

Amongst side-channels, *memory side-channels* refers to a sub-class that allow an attacker to determine how a target program accesses memory, or *memory access patterns*. This section first explains why memory side-channels exist in enclave computations and provides a taxonomy of different memory side-channels. Then, the section includes a brief demonstration of how memory access patterns can leak sensitive user data from programs.

2.5.1 Root Cause and Classification

The root cause of memory side-channels in SGX is (a) the control of paging by untrusted software (*controlled channels*) or (b) the sharing of microarchitectural resources between enclaves and untrusted software (*shared channels*). Table 2.3 lists known memory SGX side-channels.

Controlled channels. The operating system *controls* all enclave paging functions (§2.4), including handling an enclave's page tables and retrieving pages from a backing store on page faults. By controlling the page tables, the operating system can observe enclave execution—at the granularity of pages—through page access or dirty bits [18], [19], [40]. In particular, whenever an enclave accesses any memory page, the hardware memory management unit (MMU) automatically sets these bits in the page table entry. Moreover, a malicious operating system can invalidate page table entries of an enclave (e.g., remove present bit) and force an

Table 2.3. An overview of known memory side-channels, including the information leaked, and disclosed attacks.

Memory side-channel	Information leaked	Known attacks
Controlled		
Page faults	Page fault address	CCA [18]
Page tables	Page access/dirty bits	Multiple [19], [40]
Shared		
L1/L2 cache	Cache access patterns	Multiple [13], [41]–[43]
Branch target buffer (BTB)	All branch patterns	BranchShadow [15]
Pattern history table (PHT)	Cond. branch patterns	BranchScope [16]
Translation lookaside buffer (TLB)	TLB access patterns	TLBleed [44]
Last level cache	Cache access patterns	Multiple [14], [45]
Memory bus	Cache access patterns	WIHS [17]

enclave to incur a page fault. This allows the operating system to observe the order in which an enclave accesses pages through page faults.

Shared channels. Modern CPUs implement many micro-architectural resources that are shared between enclaves and untrusted software. Since these resources have limited storage space, shared usage results in eviction of contents. If a software needs evicted contents, it must incur an additional delay to populate them again in the micro-architectural resources. An untrusted software can measure this delay to leak an enclave's memory access patterns.

Some resources are shared at the level of hyper-threads—only hyperthreads of the same processor core can simultaneously access the resource—which we will call *per-core* resources. Other resources are shared at the level of processor cores—any processor core can access this resource at any given time—and we will call them *cross-core* resources.

Per-core resources include L1/L2 caches, branch predictor units (namely the branch target buffer, return stack buffer, and pattern history table), and translation-lookaside buffer (TLB). The general approach to exploit side-channels of per-core resources is for an attacker to concurrently execute their malicious code alongside a victim enclave code on hyper-threads of the same processor core. This concurrent execution will result in contention. For instance, both the malicious code and enclave code will try to retrieve contents into the L1/L2 caches. Such contention results in timing differences that the malicious code can measure. In §2.5.2, we provide an end-to-end attack involving a per-core resource.

Cross-core resources include the last-level cache (LLC), CPU ring interconnect, and the memory bus. Side-channel attacks through cross-core resources do not require strict scheduling on same hyper-thread like per-core resources. However, side-channel attacks through cross-core resources are more noisy, because of significant background noise from all shared usage from all other cores.

2.5.2 Attack Case-Study

This section details a case-study on leaking sensitive data from enclaves using a component of the branch predictor unit—the branch target buffer (BTB). This case-study and all example codes have been adapted and simplified from the branch shadowing paper [15]. Please refer to the paper for full details. The rest of this section (a) provides an overview of the BTB, (b) explains how the operating system can deduce an enclave's branches using the BTB, and (c) describes how deduced branches leak secret data from a real-world enclave program.

Branch target buffer (BTB). Branch prediction is a vital performance optimization in modern computers to avoid stalls during out-of-order execution (§2.1). One of the components that CPUs use to perform branch prediction is the branch target buffer (BTB), a *per-core* data structure that resembles a processor cache. The BTB stores information about the branches recently taken by a program. At a high-level, this history is maintained in the form of entries that are referenced using branch virtual addresses. If a program executes a branch, the corresponding branch address is stored in an entry of the BTB. During subsequent out-of-order execution, when the CPU reaches a branch whose outcome is not yet known, the CPU will speculate whether to execute the branch or not based on the address stored in its BTB entry. Please note that the full prediction process using the BTB is more complex and dependent on the CPU model (refer to [15] for more details).

Deducing taken branches from a custom enclave. In a branch shadowing attack, the attackers goal is to determine if a branch was taken and at what virtual address. The program's branching information is highly-sensitive (e.g., it can be used to leak cryptographic keys [15]). This section explains how an attacker can determine the required information from a simplified enclave program by exploiting a side-channel in the BTB.

```
1 if (cond != 0) {
2   ++i;
3   ...
4 }
5 else {
6   --i;
7   ...
8 }
```

(a) Victim enclave code.

```
1 ⋆ if (cond != cond) {
2     nop; // should never be executed
3     ...
4   }
5 ⋆ else {
6 ⋆   nop; // should always be executed
7 ⋆   ...
8 ⋆ }
```

(b) Shadow code aligned with **(a)**.

Figure 2.2. Illustration of a victim enclave code and its shadow code. The lines are assumed to be aligned to the same virtual address in each code. Due to a potential misprediction, the execution time of the correct shadow branch (lines $5-8(*)$) depends on the branching outcome of the victim code.

There are two main assumptions for a successful branch shadowing attack. First, the attacker should know the exact branch addresses in a victim code. This is a realistic assumption because operating system loads an initial program code inside the enclave (§2.4.4); hence, it knows what code is executing inside the enclave. Even if a user leverages an in-enclave loader to hide their code, a significant portion of the sensitive enclave code (e.g., for cryptographic libraries) is standard and open-source [46]. Second, the attacker should create a program with branch conditions aligned to the same virtual address as the victim program's code. Such a program's code is called the *shadow code*. Given knowledge of enclave code, creating a shadow code is not challenging for the operating system.

Once the shadow code is ready, the attacker must carefully execute the victim and shadow code on the same processor core and measure timing differences for the execution of branches. Consider Figure 2.2 for an illustrative example of the branch shadowing attack on a custom enclave program. In the victim enclave code (Figure 2.2a), the execution of the `if` conditional statement block depends on the value of `cond`. If the victim enclave is first executed and the value of `cond` is not zero, then the program will take that `if` block (lines $1-4$) and the BTB will record that the branch was taken for future predictions. When the shadow code (Figure 2.2a) is executed next (on the same processor core), the CPU will predict that its `if` branch (at line $1*$) should also be taken. This prediction, however, is incorrect since the shadow program's `if` condition can never be true. Eventually, when the in-order execution of the CPU catches up, it will realize that the branch was mispredicted and the CPU will

roll-back the predicted changes (as explained in §2.1). The roll-back will increase the time taken to execute the shadow code's branch that should have been taken (lines $5-8(*)$). An attacker can measure this timing increase using several techniques (e.g., `RDTSC` CPU counter, Intel processor trace, etc.).

Leaking cryptographic keys from an mbedTLS enclave. MbedTLS is a well-known cryptographic library that is even hardened against timing side-channels. It is also a popular choice for SGX developers [34], [35], [47] due to its portability. The library uses montgomery multiplication (Figure 2.3) to perform fast modular arithmetic, required by various algorithms like RSA and Diffie-Hellman. The montgomery multiplication algorithm has a side-channel problem, which is that it performs a conditional final subtraction of the modulus based on the input operands (at line 20). MbedTLS mitigates this problem by performing a dummy substraction (lines $24-27$).

Unfortunately, the dummy substraction-based mitigation is not enough to thwart the branch shadowing attack. An attacker can create shadow code for both branches of the montgomery multiplication algorithm and accurately deduce when a dummy branch is taken. This allows an attacker to determine the input operand, which can then be used to leak the entire cryptographic key [48] from a victim enclave.

2.5.3 Relation to Micro-Architectural Defects

In the last few years, the security community has uncovered several *micro-architectural defects* [51]–[55] in CPU processors from several hardware vendors, including Intel, AMD, and ARM. These defects exploit CPU optimizations like speculative execution and out-of-order execution (explained in §2.1) to break hardware-supported isolation between processes on a machine and retrieve forbidden memory contents. For instance, the defects can break isolation between an enclave and the operating system, allowing the operating system to dump the entire enclave memory. Interestingly, to leak information through defects, attackers must also leverage memory side-channels (please refer to [51]). Nevertheless, these defects are inherently different from memory side-channels since they come from buggy or inconsistent hardware implementation of processors. For example, earlier SGX CPUs would enforce EPC access

```
static int mpi_montmul(mbedtls_mpi *A, const mbedtls_mpi *B,
                const mbedtls_mpi *N, mbedtls_mpi_uint mm,
                const mbedtls_mpi *T) {
  size_t i, n, m;
  mbedtls_mpi_uint u0, u1, *d;

  d = T->p; n = N->n; m = (B->n < n) ? B->n : n;

  for (i = 0; i < n; i++) {
    u0 = A->p[i];
    u1 = (d[0] + u0 * B->p[0]) * mm;

    mpi_mul_hlp(m, B->p, d, u0);
    mpi_mul_hlp(n, N->p, d, u1);

    *d++ = u0; d[n+1] = 0;
  }

  if (mbedtls_mpi_cmp_abs(A, N) >= 0) {
    mpi_sub_hlp(n, N->p, A->p);
    i = 1;
  }
  else { // dummy subtraction to normalize timing differences
    mpi_sub_hlp(n, N->p, T->p);
    i = 0;
  }
  return 0;
}
```

Figure 2.3. Montgomery multiplication used by a popular and widely-used cryptographic library, mbedTLS [49]. This code is taken from the file bignum.c in the mbedTLS GitHub repository [50]. It was also previously used by the BranchShadowing paper [15] to demonstrate their attack.

control checks (§2.4.2) during in-order execution, but due to a bug, ignore these checks during out-of-order execution [54]. Therefore, micro-architectural defects are routinely patched by hardware vendors [56]–[58], unlike memory side-channels.

2.6 Oblivious Random Access Memory (ORAM)

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [59], is a cryptographic primitive that enables confidential access to (encrypted) sensitive data, such that access patterns during the computation reveal no secret information [60]. Such a secure access is also called an *oblivious* access. ORAM was originally introduced for a setting where a user

retrieves encrypted data from an untrusted memory region (e.g., cloud machine) and does not want an attacker to learn what data is being retrieved. The untrusted memory region is called the *server*, while the user is called the *client*.

As the name suggests, ORAM securely randomizes accessed memory regions, so that the attacker cannot link a memory access to a prior one. At a high-level, secure randomization is achieved through three steps. Initially, for every memory block to be retrieved, ORAM retrieves multiple blocks to prevent the attacker from determining the needed block. Then, ORAM writes-back each retrieved block to the server at a randomized location. To prevent the attacker from breaking the randomized location of blocks, each block is re-encrypted with a random nonce, making each block look distinct from its previous version. For its operations, ORAM requires access to a small amount of trusted client memory (e.g., on a user's personal machine). This region holds retrieved blocks and stores metadata for the encrypted blocks in untrusted memory (e.g., which block is located where in untrusted memory?). In the next section, we describe the details of Path ORAM, the variant that we used in our work.

2.6.1 Path ORAM

Path ORAM [61] provides a faster ORAM construction compared to the original construction proposed by Goldreich and Ostrovsky. It uses a binary tree-like formation (called an ORAM tree) to store encrypted memory on the server. Each node within the tree is composed of K blocks, where K is a constant defined during initialization. An ORAM tree contains both real blocks with actual client data, and dummy blocks with false data to fool an attacker. The number of real blocks within a tree of L leaf nodes can be at most L in order to ensure secure access patterns.

Using Path ORAM, the client runs an ORAM controller within a small, completely trusted memory region. There are two key data structures for Path ORAM—the position map and the stash. Typically, the position map is an integer array, which links the real block to its corresponding leaf-index within the ORAM tree. Whenever the client needs to access a block, the ORAM controller (a) finds the corresponding leaf from the position map and (b)

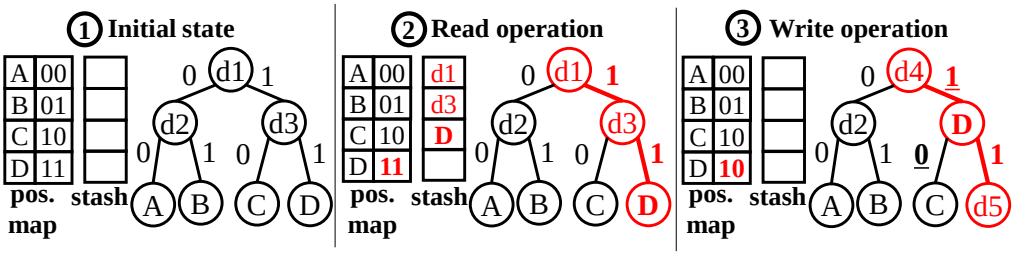


Figure 2.4. A simplified illustration of the Path Oblivious RAM (ORAM) cryptographic protocol.

extracts all nodes (both real and dummy) in the path towards the leaf in the ORAM tree. The extracted blocks are stored inside the stash.

Figure 2.4 illustrates the Path ORAM algorithm. The client attempts to access the block D from the server containing the ORAM tree (①). First, the client looks-up the leaf index corresponding to block D, which is 11 in our example (②). Then, the client extracts the complete path from the root of the tree to the leaf (i.e., $d1, d3, D$) and saves it in the stash. The dummy blocks (i.e., $d1, d3$) are discarded at this point to keep the stash size small. After accessing the block D, the client randomizes its position, i.e., initial leaf was 11 and final leaf is 10, and re-encrypts the block with a random nonce (③). The client then tries to write-back to the tree from the old leaf (11) back to the root. To ensure consistency, the client only writes back a real block on a certain node, iff, that node is the new leaf, i.e., 10, or that node is in the path to the new leaf. If the client does not have a real block to put into the node, it generates dummy data, encrypts it (using random nonce) and writes it to that node. For example, in the figure, $(d4, d5)$ corresponds to the generated dummy data.

3. RELATED WORK

This chapter relates the work presented in this dissertation with previous work. Since this dissertation heavily relies on Intel SGX, it is useful look at the research that led to the development of SGX (§3.1) and the notable secure systems built using SGX (§3.2). Memory side-channels are a critical source of data leak from enclaves (as discussed in §2.5) and one of the main focus of this dissertation. Hence, this section also describes the software (§3.3) and hardware (§3.4) solutions presented to defeat memory side-channels, and compares them to our proposed solutions. Finally, the section concludes with a description of other attacks against enclaves and their proposed defenses (§3.5).

3.1 Towards the Development of SGX

Several solutions [36], [62]–[65] inspired the development of modern TEEs like Intel SGX. This section briefly overviews these solutions and relates them to SGX.

The execute-only memory (XOM) architecture [62] introduced the idea of having sensitive code and data execute in isolated containers, managed by the untrusted operating system. XOM also outlined the mechanisms needed to isolate the container's data from an untrusted operating sytem. For instance, the paper showed that it is critical to save the register state to protected memory before servicing an interrupt. SGX enclaves are similar to XOM's isolated containers and SGX borrows the idea of register state protection from XOM.

Aegis [36] built on the container concept introduced by XOM. Unlike XOM, Aegis creates containers using a trusted software *security kernel* executing at the highest privilege on the machine. Aegis additionally proposed the idea of remote attestation for a container. In particular, Aegis computed a cryptographic integrity hash of the security kernel at boot and extended it with the container's hash at runtime. The combination of these hashes was used to attest to a remote user and establish trust on the container. SGX leverages the key concept of remote attestation from Aegis.

Prior to SGX, the idea of attestation using a trusted hardware was leveraged by the trusted platform module (TPM). The TPM is an auxiliary tamper-resistant chip that can be connected to any machine without requiring CPU changes. The TPM chip contains a tiny

co-processor with a unique secret and a small amount of memory. During system boot-up, the TPM is invoked to measure initial boot contents, including the BIOS, UEFI, and initial OS image. The TPM is now widely deployed on all modern computers and even used by modern operating systems to ensure boot-time security [66]. SGX's provisioning and quoting enclave (§2.4.5) designs are heavily inspired by the TPM [24].

Although the TPM is mainly used to verify boot-time contents, it can also verify contents at runtime. Several systems [63]–[65], [67], [68] used this capability to install and attest a trusted hypervisor on a machine at runtime. This trusted hypervisor acted similar to Aegis' security kernel and ensure that the operating system does not interfere with secure containers. The hypervisor approach was very viable (performance-wise) due to hardware extensions [23] that enabled low-cost virtualization (e.g., nested page tables). Unfortunately, several attacks [69] were found to compromise such trusted hypervisor-based solutions, limiting their adoption. Nevertheless, SGX leverages several key concepts from virtualization solutions (e.g., efficient hardware memory protection of enclaves).

3.2 General Systems Designed using Enclaves

Since the advent of SGX, there has been a consistent significant interest from the academia and industry to build secure systems using enclaves. This section goes over some of the major systems that leverage enclaves.

Haven [70] is the seminal work in SGX research. It implements a library operating system (LibOS) which allows native Windows programs to execute inside enclaves. Haven also showed how to protect enclave programs against several system call-based attacks (discussed in §3.5) and protect files inside the enclave through an in-memory filesystem. Building on the work of Haven, several other library operating systems were introduced for Linux environments too, including Graphene [71], Panoply [72], Scone [73], and Occlum [74].

One of the most attractive use-case of TEEs like SGX is to run secure machine learning and data analytics on sensitive user data in cloud machines. VC3 [75] demonstrated how to run secure distributed MapReduce computations inside enclaves, while providing additional integrity guarantees against bugs in such computations. M^2R [76] provided even stronger

privacy guarantees to MapReduce using ORAM and SGX. Moreover, Ohrimenko et al. [77] described the design of several oblivious machine learning algorithms using SGX. Finally, Opaque [78] created a general distributed data analytics framework using SGX.

In addition to cloud machines, SGX was also found very useful to secure computations on personal machines. Baumann et al. [79] proposed using SGX to protect video game digital rights management (DRM) software from tampering on PCs. BlackMirror [80] (a research work from our group) showed how to defeat the notorious wallhack cheats in video games using SGX. Moreover, PowerDVD, a DVD player software, used SGX to protect its own DRM engine on user machines. Intel also demonstrated how to secure authentication tokens in client web browsers (like Chromium) using SGX [81].

Finally, SGX has also received significant interest from the industry. Fortanix [82], a now successful company, spun as a startup with the goal of securing sensitive computations on enterprise and cloud machines using SGX enclaves. Some of Fortanix's major clients include NEC [83] and UC San Francisco [11]. Microsoft implemented the Open Enclave (OE) SDK [84] to speed-up development for SGX enclaves. Signal, a private internet messenger, uses SGX to implement private contact discovery between its clients [10].

3.3 Software Protection for Enclaves against Memory Side-Channels

Since TEEs like SGX do not trust software outside the enclave, software schemes must be implemented inside enclaves. Such schemes cannot prevent side-channels from disclosing memory access patterns since that is a hardware limitation (§2.5.1). Therefore, software protection schemes either rely on cryptographic protocols—like Oblivious RAM (§2.6)—to obfuscate all memory access patterns (i.e., make all access patterns indistinguishable) or use heuristics to detect attacks. The obfuscation approach—also employed by our Obliviate and Obfuscuro systems (§5)—provides strong protection but at high performance costs. In contrast, the remaining approaches provide weak but efficient protection.

Obliviate obfuscates access patterns related to the file system component of enclaves. Obfuscation of only certain components of the enclave helps reduce performance impact, and it has been adopted by other systems. In particular, ZeroTrace [85] uses ORAM to secure

access to sensitive data structures (e.g., arrays) inside an enclave. Ohrimenko et al. [77] show how to adapt parts of machine learning algorithms to exhibit oblivious memory access patterns. Additionally, T3 [86] obfuscates access patterns on cloud-based bitcoin clients. Compared to OBLIVIATE, the scope of these systems is different and complementary.

Our OBFUSCURO system obfuscates all memory accesses of a program and even ensures that different programs executing in an enclave are indistinguishable (attaining the elusive property of *program obfuscation* discussed in §5). While there is no other enclave system that provides such a guarantee, Raccoon [38] can ensure data-obliviousness—attackers cannot distinguish between different inputs to the same program.

Unlike obfuscation-based systems that provide strong protection against all memory side-channels, several systems use heuristics to provide protection against a few side-channels. Varys [87] is a heuristic-based software defense against side-channels that can partially defeat page table and cache side-channel attacks. Its approach is to monitor the system for known side-channel attack patterns. For instance, Varys detects if an enclave is being stopped frequently (since that could indicate an attack) and aborts the enclave. Unfortunately, this approach only raises the bar for attack and it can result in false positives.

Various defenses [47], [88] were proposed against the page table (or controlled channel) side-channel attacks. In particular, T-SGX [47] uses Transactional Memory (TSX) to run a program without incurring page faults. However, T-SGX is vulnerable to the improved controlled channel attack [19]. Cloak [89] also utilizes TSX as a defense primitive, but its technique can only defeat attacks that exploit the cache side-channel.

Another potential approach to mitigate *passive* side-channel attacks is by randomizing the program periodically [34], [88]. Note that this randomization is not cryptographic in nature like Oblivious RAM; hence, an active adversary (who can observe the entire randomization process) can break it. Finally, while there are other solutions designed for non-enclave environments [90]–[92], they cannot be employed to protect enclave computations since they require support from a trusted operating system or hypervisor.

Table 3.1. An overview of protection against memory side-channels provided by our systems—Obfuscuro and Reparo—and other existing SGX software and hardware defenses. △ means partial protection. Obfuscuro incurs more overhead than others because it additionally provides the strong notion of cryptographic program obfuscation. As discussed in §5.6.1, Obfuscuro is orders of magnitude faster than all other systems that provide cryptographic program obfuscation.

Channel	**Software defenses**			**Hardware defenses**		
	Obfuscuro [our],[93]	Raccoon [38]	Varys [87]	Autarky [94]	InvisiPage [95]	**Reparo** [our]
Controlled						
Page faults	✓	✓	△	✓	✓	✓
Page tables	✓	✓	△	✓	✗	✓
Shared						
L1/L2 cache	✓	✓	△	✗	✗	✓
Branch target buffer	✓	✓	✗	✗	✗	✓
Pattern history table	✓	✓	✗	✗	✗	✓
Translation lookaside buffer (TLB)	✓	✓	△	✓	✓	✓
Last level cache	✓	✓	✗	✗	✗	△
Memory bus	✓	✓	△	✗	✓	✗
Overhead	5100%	2180%	15%	8%	354%	17%

3.4 Hardware Protection for Enclaves against Memory Side-Channels

Given the hardware root cause of memory side-channels, our Reparo research provides a solution to close a diverse range of memory side-channels in SGX environments with minimal hardware changes and high performance (§6). Other researchers have also proposed ways to update the hardware and close memory side-channels. Note that the basic way to close a memory side-channel remains the same across many different systems. For instance, cache side-channels can only be fully defeated by partitioning the use of caches between enclave and non-enclave computations. The difference between different research is how these ideas are implemented in the hardware. In general, Reparo requires fewer hardware changes to SGX than existing approaches, while closing more side-channels.

The closest research work to Reparo is Sanctum [96]. It proposed a software-hardware co-design for a TEE that is similar to the SGX model. Sanctum also closes several side-channels by providing control of page tables to the enclave and partitioning/invalidating the

caches. Compared to REPARO, the set of side-channels addressed by Sanctum is still limited and many of its proposed implementations require substantial hardware changes (e.g., a new address translation mechanism alongside a redesigned last-level cache).

The ideas proposed by Sanctum—page tables controlled by enclaves and cache partitioning—have also been implemented by other academic TEE solutions like Komodo [97], Penglai [98], and Keystone [99]. However, these systems design entirely new TEE solutions that require substantial changes to the SGX model for deployment. For instance, all of these systems requires enclave implementations that have both privileged and unprivileged software layers, which is incompatible with the SGX model.

Autarky [94] and InvisiPage [95] close some attack vectors for controlled channels in SGX, namely page faults and page table access bits, using enclave-operating system collaborative paging. This collaborative approach does not fully address the root cause of controlled channels—the operating system is still required to retrieve pages from the backing store using paging instructions. In particular, to securely retrieve pages without leaking access patterns, these defenses must still obfuscate patterns using the expensive ORAM.

The HOP [100], Ascend [101], and Phantom [102] secure processors implemented ORAM techniques in the CPU's memory controller to provide full hardware protection against memory side-channels. While these approaches can defeat side-channels even at the physical bus layer (which is beyond REPARO's scope), they incur very high performance costs compared to REPARO. These costs hinder their potential deployment.

Finally, there are many other proposed hardware schemes [103]–[109] that can defeat a subset of side-channel attacks with high performance. However, given hardware complexity, it is challenging in practice to combine diverse hardware defenses for full protection.

3.5 Other Attacks against Enclaves and Proposed Protection Schemes

In addition to memory side-channel attacks (§2.5), SGX enclave computations are vulnerable to other new and traditional attack vectors against computation. This section describes all such attack vectors and proposed protection schemes.

SGX enclave computations rely on the operating system for handling page tables and other privileged functionalities (e.g., accessing disk through system calls). While page table handling by the operating system opens up the controlled channel attacks (discussed in §2.5.1), system call handling opens up another attack vector, IAGO attacks [110]. In IAGO attacks, the operating system returns malicious results to enclave system calls (e.g., incorrect memory address in response to the `mmap` system call). These malicious results can trigger an enclave to inadvertently reveal sensitive information (e.g., cryptographic keys).

IAGO attacks can be defeated by rigorously sanitizing the results of system calls. For instance, the `mmap` system call result should be checked to ensure that it does not overlap protected memory. Fortunately, the SGX SDK [111] (provided by Intel) already protects against some IAGO attacks by implementing internal memory allocation. Other library operating systems (LibOS)—Graphene-SGX [71], [112], Haven [70], and Occlum [74]—implemented for SGX to port native Linux/Windows applications provide more comprehensive protection against IAGO attacks by sanitizing most/all system call results.

SGX was also found to be vulnerable to voltage manipulation attacks, namely PlunderVolt [113]. In this attack, the operating system operates the enclave-running CPU core at a low voltage, inducing faults in the enclave computation. Such faults can be carefully and predictably inserted into an enclave's computation to leak cryptographic keys and induce memory-safety bugs in bug-free enclave code. Intel addressed this attack by restricting the undervolting capabilities presented to the operating systems (through model specific registers) in SGX CPUs [57]. However, physical variants of this attack are still possible.

Unlike PlunderVolt, which exploits physical properties of the CPU package, rowhammer attacks [114] leverage the physical DRAM properties to inject faults during an enclave computation. In particular, due to high-density packaging of memory cells in modern DRAMs, it is possible for memory cells to interact electrically between themselves. Such electrical interactions lead to *bit flips* in neighbouring memory rows that are not addressed by the current memory access. An attacker can craft special memory access patterns that lead to predictable bit flips and leak sensitive information.

Although there exist software protection schemes [115] against rowhammer attacks, they are incompatible with SGX (since they require trusted privileged software). The previous

version of SGX provided full protection against rowhammer attacks through Merkle Hash Trees (MHT) with root protected on the CPU chip [116]. However, MHT-based protection is very expensive, and the current version of SGX (implemented on server CPUs) relies on error-correcting code (ECC) memory for partial mitigation of rowhammer.

Finally, like non-enclave computations, enclave code is also vulnerable to traditional memory corruption and safety vulnerabilities, like return-oriented programming (ROP) [117]–[119] and buffer overflows. Many traditional defenses (e.g., memory randomization) against these attacks cannot directly be leveraged inside enclaves. Nevertheless, there exist many systems built for SGX to address these attacks. In particular, SGX-Shield [34] implements memory randomization inside SGX enclaves. Moreover, SGX-Bounds [120] implements pointer bounds checking for SGX. Our CHANCEL system (§4) also protects against memory corruption vulnerabilities in untrusted programs running inside enclaves.

4. SOFTWARE SANDBOXING TO DEFEAT ADVERSARIAL CLOUD SERVICES

Traditionally, TEEs like SGX assume that the program running inside the enclave is trusted. Unfortunately, this assumption is not always valid because applications may contain bugs or may have been built by malicious programmers (i.e., *adversarial programs*). Hence, users cannot trust remote programs because they may leak confidential data, even if protected by SGX. For example, consider a scenario where a service provider rents a cloud machine, with SGX capabilities, to provide a service to its clients. In this scenario, SGX will protect client data from a malicious cloud provider, but cannot prevent the data from being collected by the service provider using an adversarial program. Therefore, clients must trust the service provider to not leak their confidential data.

The significant value of private user data makes it an appealing target for collection by service providers. For example, consider a popular messaging platform, Signal [121], which supports private contact discovery [122]. Signal keeps an offline database of its users and periodically updates it on an SGX machine. The users connect to the SGX machine to discover which contacts use Signal. The private contact discovery is meant to prevent Signal from extracting a social graph, i.e., determine which contacts know each other. However, although Signal's source code is available for inspection, it is non-trivial to determine that it satisfies such security properties. Moreover, unlike Signal, many companies do not disclose their proprietary algorithms which further aggravates the problem by requiring users to blindly trust service providers.

This chapter introduces Chancel, a sandbox environment that restricts adversarial programs from leaking user data provided to enclaves. Chancel leverages *software fault isolation* (SFI) [123] principles for its sandbox. In particular, Chancel implements on a novel Multi-Client SFI (MCSFI) scheme to securely and efficiently handle requests from different clients, within a single enclave (§4.2). Multi-Client SFI ensures *thread isolation*, i.e., sensitive client data and other thread content is only available within the thread's context, and *shared memory enforcement*, i.e., threads can only use shared memory that is protected against data leakage or tampering. Furthermore, Chancel ensures the confidentiality of computational

results by encrypting all outgoing data using a shared secret key with each client. Lastly, CHANCEL provides various in-enclave functionalities (e.g., an in-memory filesystem) and offers practical protections against covert channel attacks.

To ease program development for CHANCEL, we implement a compiler toolchain using LLVM [124] (§4.3). Our evaluated programs required no manual changes to build using our toolchain and few additional code to utilize CHANCEL's runtime services.

We evaluate CHANCEL on SGX hardware to assess its security impact and performance. Our security evaluation (§4.4) confirms that CHANCEL can prevent a wide range of attack scenarios, including attempts to compromise the instrumentation checks, perform code injection attacks, and leak client data. Our performance evaluation (§4.5) shows that CHANCEL has significantly higher performance than the baseline sandbox approach (e.g., Ryoan [125]). CHANCEL showed a $4.06 - 53.70\times$ performance improvement in micro-benchmarks and $0.02 - 21.18\times$ improvement across a range of important real-world workloads. Finally, the average overhead of CHANCEL compared to native execution is only 12.43% on the nbench benchmark [126], which demonstrates CHANCEL's applicability to a wide range of scenarios.

4.1 Motivation

This section overviews the multi-client system model that CHANCEL is concerned with (§4.1.1) and enumerates examples of critical services (e.g., private information retrival and product recommendation services) that are relevant to the system model (§4.1.2).

4.1.1 System Model

Our work considers a computing model in which a server program runs on a machine to provide a service (database, intrusion detection, etc.) to multiple clients (shown in Figure 4.1). This model has three main entities: the service provider, the clients, and the cloud provider. The underlying assumption is that none of the parties trust each other. In the following, we explain the role of each participant in the system.

- **Service provider.** The service provider builds and deploys a *possibly adversarial* program that serves many clients. The program uses multi-threaded programming abstractions

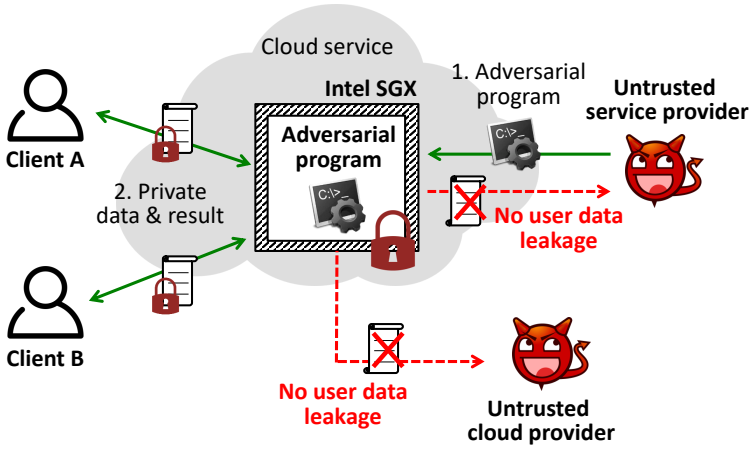


Figure 4.1. CHANCEL's system model with three participants: the clients, the service provider, and the cloud provider.

to handle multiple clients efficiently. Importantly, each thread handles a request which involves confidential data from *one client* at a time while accessing information from a *shared read-only* region (e.g., database).

The service provider is honest but curious—it provides a functionally correct service but is tempted to collect its client's data for monetary purposes (e.g., advertisements). In most cases, dishonesty is easy for clients to detect using redundancy (e.g., ask two providers for the same service and compare results). Furthermore, dishonesty may result in lower service quality, prompting the clients to terminate their contracts with the provider. In contrast, clients cannot detect secrecy violations; hence, secrecy violations are a more significant concern. Finally, despite the provider's curiosity, we expect the service provider will still deploy security mechanisms (e.g., SGX) due to client privacy concerns and to observe governmental regulations (e.g., GDPR [127]).

- **Client.** The client issues requests, containing confidential data (e.g., database query, internet history, etc.) to the program to utilize the provided service. The client wants to ensure that their confidential data is not leaked from the program. Note that the client might intentionally share some data (e.g., passwords) with the service provider, before accessing the service. Such intentionally shared data is not considered confidential.

- **Cloud provider.** A third entity, the cloud provider, *may* exist if the service provider rents hardware from third-parties such as Microsoft Azure [7]. In such scenarios, we assume that the cloud provider is also honest but curious, i.e., it will not deny a client access to the service but desires to extract the client's sensitive data.

4.1.2 Examples of Target Scenarios

Many critical real-world scenarios follow this chapter's system model (§4.1.1), including private information retrieval, intrusion detection systems, and product recommendation services. This section describes how an efficient multi-client sandbox (such as Chancel) can be beneficial in such cases.

Private information retrieval. Consider a company that provides health-care suggestions (e.g., drug information [128]) based on a client's provided information, such as previous medical history. The company can determine the health condition of its clients by observing their queries. A multi-client sandbox could serve many clients in a single sandbox, allow each client to query a shared database, and get relevant information without revealing their health conditions. Other examples include Signal's private contact discovery [10], navigation services, and web servers that serve sensitive pages (e.g., prohibited political content).

Intrusion detection systems. Intrusion detection systems (IDS) analyze packet payloads to detect trojans, viruses, and malware based on a pre-defined signature dictionary. Since dictionaries are huge and inspection can take many computational cycles, cloud-based systems [129]–[131] allow uploading files which are scanned on cloud machines and a report is provided to the client. However, such services inspect unencrypted sensitive files, potentially from many untrusted users simultaneously. Multi-client (and multi-threaded) sandboxes can enable secure, efficient, and parallel inspection of many files from the same or different clients, within isolated threads, using a common dictionary.

Product recommendation services. Modern product recommendation services [132] use machine learning algorithms on their product's catalog and a user's search history to predict the products that users are most likely to buy. However, purchase or search history is sensitive, so many companies provide the option of anonymizing such history but at the

cost of less relevant recommendations. A multi-client sandbox enables secure servicing of many users' previous history on the service provider's common catalog and provides relevant recommendations.

4.2 Chancel Design

This section begins with an overview of Chancel (§4.2.1) and an operation workflow related to the service provider and clients (§4.2.2). Then, it describes Chancel's implementation of Multi-Client SFI (MCSFI) within an SGX enclave (§4.2.3). Finally, it details shared data initialization mechanisms (§4.2.4), and Chancel's runtime services (§4.2.5).

4.2.1 Overview

Chancel implements Multi-Client SFI, an efficient and scalable multi-client isolation scheme under adversarial programs to guarantee client data confidentiality within an SGX enclave. In particular, Chancel guarantees the following properties:

- **P1. Program isolation.** The program is isolated from the untrusted world to prevent direct external leakage.
- **P2. Thread isolation.** Each thread handles sensitive data from a single client and is isolated to protect the data from leakage across thread boundaries.
- **P3. Shared memory enforcement.** All threads share a memory region, holding common non-sensitive data, and enforced with *read-only* permissions during servicing.
- **P4. Encrypted outgoing communication.** All data leaving the enclave is encrypted with a shared secret key known only to the client (of a specific thread).
- **P5. Mediated interactions.** All interactions between the program and the untrusted world are fully mediated to prevent leakage through such channels.

To ensure the properties mentioned above, Chancel runs a sandboxed execution environment, SecureLayer, inside the enclave. SecureLayer bounds the program's load and store

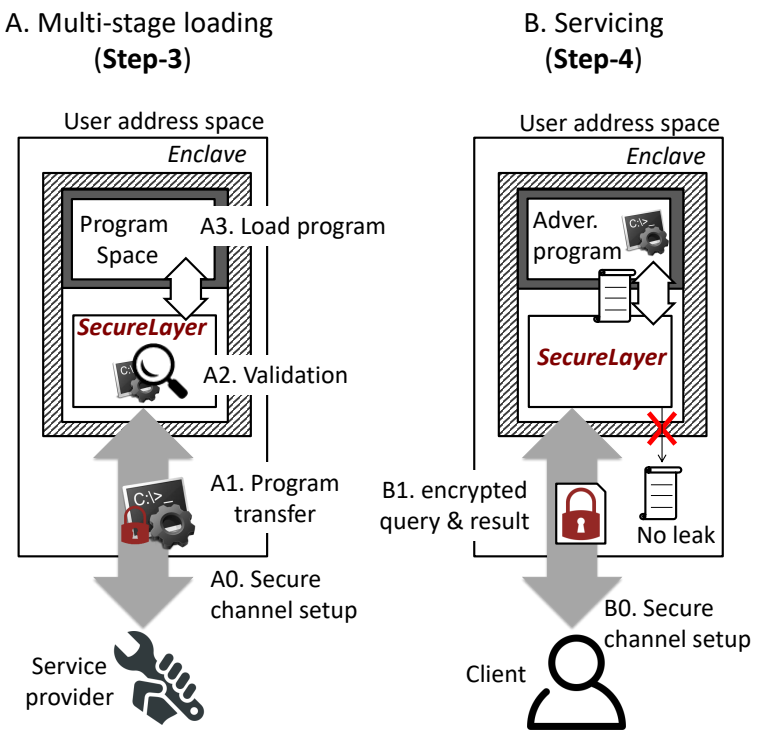


Figure 4.2. CHANCEL workflow. A) An enclave containing SecureLayer is created, which then validates and loads a program provided by the service provider. B) The runtime behavior of the program is restricted, and SecureLayer mediates all interactions originating from it to avoid any security threat.

operations according to Multi-Client SFI, with three rules. First, reading or writing outside the enclave by the program is prohibited (P1). Second, each thread can read and write to its private memory region (P2). Third, each thread can only read from memory shared with other threads of the program (P3).

Besides, SecureLayer provides a shielded communication interface to allow the program to obtain data from the client and return its results securely. SecureLayer encrypts all outgoing data with a corresponding client's session key so that only the client can decrypt it (P4). Lastly, SecureLayer provides a set of required functionalities, e.g., in-enclave filesystem and dynamic memory allocation. Through this, CHANCEL ensures that all interactions are handled internally to the enclave and mediated by SecureLayer (P5).

4.2.2 Workflow

The workflow of CHANCEL consists of five steps. The first two steps (Step-1 and Step-2) can be performed anytime before running a program, and the following three steps are performed while the program is running (Figure 4.2).

Step-1. Agreement (Offline). Both parties, a service provider and a client, inspect and agree on the implementational details of SecureLayer. In particular, they ensure that the source code of SecureLayer satisfies their security requirements. Then, each party computes a SHA-256 hash of SecureLayer, which acts as the trust anchor of their agreement and is required for verification during SGX remote attestation (Step-3 and Step-4).

Step-2. Program Building (Offline). The service provider builds a program using CHANCEL's development toolchain (i.e., compiler and compatible libraries). CHANCEL's compiler enforces its security requirements (§4.2.3) and outputs a binary which will be loaded into the enclave (Step-3).

During program building, some service providers obfuscate program binaries to protect their intellectual property. However, CHANCEL obviates the need for such obfuscation by guaranteeing stronger protection using end-to-end binary encryption between the service provider and CHANCEL's enclave (as we explain in Step-3). Note that since the service provider already inspected CHANCEL's implementation (in Step-1), it verified that CHANCEL would not leak program contents.

Step-3. Multi-stage Loading (Online). This step involves loading two binaries into the enclave, SecureLayer and the target program (Step-2).

Initially, the service provider creates an enclave containing SecureLayer, either locally or on a remote machine (e.g., cloud machines). Then, SecureLayer obtains the target program's binary from the service provider and loads it into the enclave. More specifically, the second loading phase involves the following: (a) concerning remote enclaves, SecureLayer and the service provider mutually authenticate using SGX remote attestation and the pre-computed hash value; (b) the service provider sends the encrypted program built during Step-2; (c) SecureLayer decrypts the program and ensures (using a binary disassembler) that all desired

security properties are applied; (d) if validation is successful, SecureLayer loads the program and jumps to the program's entry point.

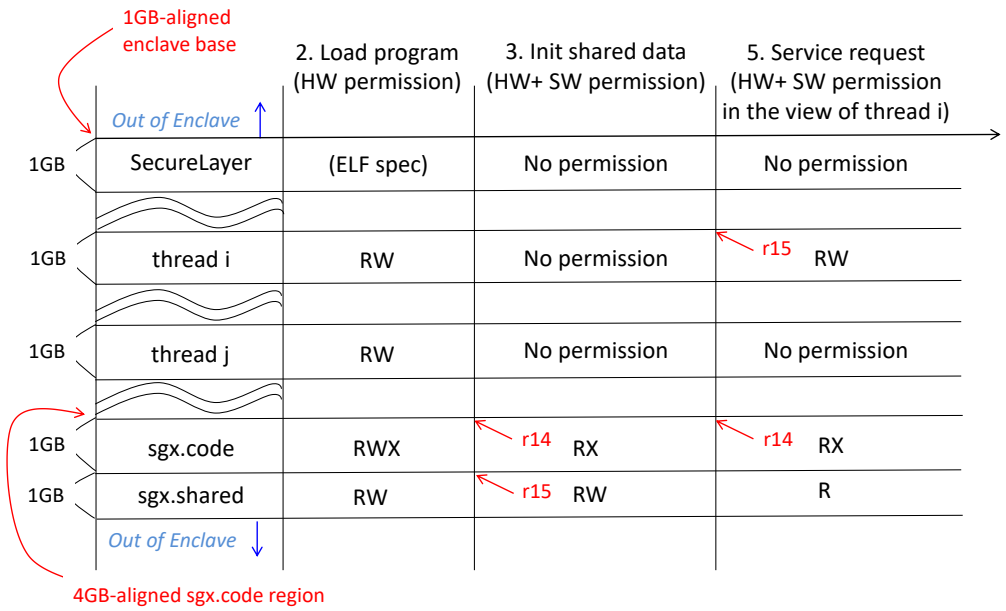
Step-4. Servicing (Online). At this point, SecureLayer is waiting for client requests. Upon receiving a request, SecureLayer provisions an available thread which authenticates to the client using SGX remote attestation and establishes a secure channel with the client (e.g., using Diffie-Hellman key exchange). Then, the client's data is securely transmitted to the thread. After receiving the data, all external interactions are encrypted through SecureLayer using the exchanged session key, thereby restricting the visibility of all results only to the connected client. Lastly, to stop covert channels, CHANCEL normalizes communication patterns (see §4.2.5 for more details).

Step-5. Cleanup (Online). After servicing, SecureLayer clears the thread's private region to avoid potential misuse by the next user of the thread. In particular, SecureLayer cleans up the thread context, including registers and memory contents. In some cases, the thread contains some initialization data, i.e., data put there by the program before servicing a client. In those cases, SecureLayer restores the data to its original (unmodified) version for the next request.

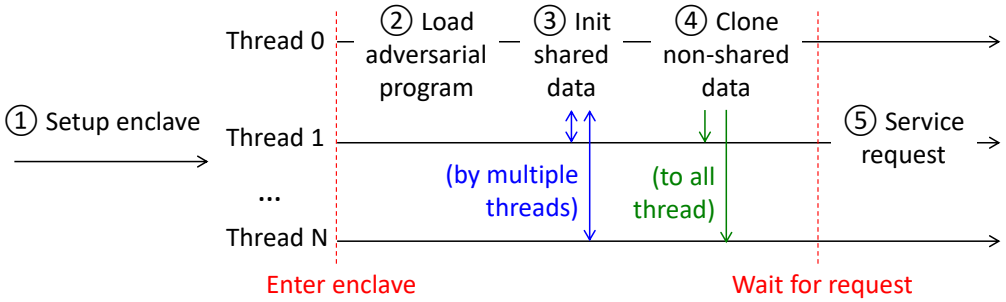
4.2.3 Multi-Client SFI (MCSFI)

This section presents CHANCEL's design of SFI for multiple clients, which supports thread isolation and enforced sharing of memory between threads in an enclave, using only low-overhead compiler instrumentation. CHANCEL's MCSFI requires a custom memory layout, a mechanism to enforce and modify permissions during various stages of its executions, and compiler instrumentation.

Memory layout. Apart from an enclave's native memory segments (e.g., `.code`, `.bss`), MCSFI requires the following additional segments: (a) a *private* region dedicated to each thread, (b) a *shared* region available to all threads, and (c) an extra executable region. In particular, the private region is used by threads to store client's sensitive data, the shared region is provisioned with non-sensitive memory (e.g., a map database), and the executable



(a) Memory Layout



(b) Timeline

Figure 4.3. CHANCEL's memory layout and permissions enforced during 5 stages of its execution.

region holds the code of the target program. Based on these requirements, CHANCEL sets up a suitable enclave layout.

Figure 4.3a shows the enclave's memory segments and the size of their reserved addresses. The SecureLayer memory region contains the attestation and validation code (and data) required to load the target program correctly. The rest of the enclave memory contains per-thread private memory segments (with guard regions in between), the executable `sgx.code` segment, and the `sgx.shared` segment, which contains data shared by all threads. The upper limit on the size of each segment (apart from the guard regions) is configurable as $2^n\times$

4 KB (in our experiments we use 1 GB segments). The limit must be fixed to ensure bounded data access and code execution (more details in §4.2.3-(c)).

Like other SGX SFI schemes [74], [125], MCSFI requires static allocation of thread regions and their reserved addresses at compile-time. In fact, SGX itself requires the specification at compile-time of the maximum number of threads in an enclave and SGX1 only allows statically-allocated enclave memory. While SGX2 allows dynamically increasing the enclave memory, permitting such allocation would allow the program to leak confidential information directly through the pages or covertly through the allocation. Therefore, MCSFI's required memory layout fits well with SGX's design philosophy and ensures strong security properties.

Furthermore, while MCSFI can theoretically support an arbitrary number of threads, it is bound by the limited virtual address space of SGX (i.e., 64 GB). Therefore, considering the described memory layout (Figure 4.3a), CHANCEL can support 60 threads when configured with 1 GB segments per thread (4 GB is reserved for SecureLayer, `sgx.code`, `sgx.shared`, and guard regions). Nevertheless, the size of thread regions can be configured, e.g., 512 MB segments allow 120 threads.

Timeline and permissions. Figure 4.3b shows the timeline of CHANCEL's execution and the permissions enforced at various execution stages are shown in Figure 4.3a.

Before loading the target program (①), only hardware permissions are enforced in the enclave (second column of Figure 4.3a). Therefore, SecureLayer can install a received target program binary (after validation) in `sgx.code` (②). After loading, both hardware and software-based mechanisms (i.e., MCSFI) enforce permissions. At this stage, CHANCEL allows the program to initialize `sgx.shared` (③) (explained in §4.2.4). For example, a health service could install its drug database in `sgx.shared` before handling user queries. However, before allowing the program to execute, CHANCEL ensures that all segments of SecureLayer and `sgx.code` are *non-writable*, preventing modification of its code or the (validated) program code. After initializing shared data, the program returns to SecureLayer, which then clones per-thread data (e.g., private global data) to each thread's private region (④).

Finally, CHANCEL allows the program to execute in different threads and service client requests (refer to §4.2.5) (⑤). At this stage, CHANCEL enforces permissions (illustrated

```
1  ; Before instrumentation
2  movq rax, [rdx]        ; rax = *(rdx)
3
4  ; After instrumentation
5  leal r13, [rdx]        ; r13 = rdx & (4GB - 1)
6  cmpq r13, r14
7  jge  READ_SGX_SHARED   ; If r13 >= r14, read sgx.shared
8                           ; Otherwise, read from thread i
9  READ_THREAD_LOCAL:
10 andl r13d, 0x3fffffff ; r13 = r13 & (1GB - 1); masking
11 movq rax, [r15 + r13] ; rax = *(r15 + r13)
12 jmp  DONE              ; jump to DONE
13
14 READ_SGX_SHARED:
15 movq rax, [r13]        ; rax = *(r13)
16
17 DONE:
18 ...
```

Figure 4.4. Software enforcement on an indirect memory load instruction. CHANCEL first checks if destination is less than the base of `sgx.code` (i.e., `r14`), as shown in line `6`. If yes, CHANCEL uses `r15` as a base address to read the thread region (lines `9-12`). Otherwise, the target is greater than `sgx.code`, i.e., cannot be SecureLayer or another thread region, and is allowed since it can only be `sgx.shared` (lines `14-15`). `r13` is a temporary register that is assumed to either already be available at this point or spilled before use.

in the last column of Figure 4.3a) as follows: (a) *read-only* permissions on `sgx.shared` to ensure that malicious threads cannot tamper with the service or leak their sensitive data, (b) *read-or-execute* permissions on `sgx.code` to prevent code injection, and (c) *read-or-write* permissions on each thread's private region only to avoid malicious writes outside a thread context.

Compiler instrumentation. This section explains how CHANCEL enforces MCSFI during its execution (§4.2.3-(b)) through compiler instrumentation. Since the target program must both initialize `sgx.shared` (i.e., stage ③) and service user requests from various thread regions (i.e., stage ⑤), CHANCEL must dynamically enforce permissions based on the execution stage and thread context.

CHANCEL takes advantage of per-thread general-purpose registers, i.e., `r14` and `r15`, to achieve dynamic permissions. In particular, CHANCEL reserves `r14` to hold the base address of the program's code (`sgx.code`). Furthermore, CHANCEL reserves `r15` to hold either (a)

```
1 ; Before instrumentation
2 movq [rdx], rax       ; *(rbx) = rax
3
4 ; After instrumentation
5 leal r13d, [rdx]      ; r13 = rdx & (4GB - 1)
6 andl r13d, 0x3fffffff ; r13 = r13 & (1GB - 1); masking
7 movq [r15 + r13], rax ; *(r15 + r13) = rax
```

Figure 4.5. Software enforcement on an indirect memory store instruction. The line `6` clears the upper `34 bits` of `r13`. As a result, `r13` becomes an offset within the thread region. Then, `r15+r13` in line `7` becomes an address in the thread region. It is assumed that `r13` is an available register (or spilled beforehand) and thus used as a temporary register.

```
1 ; Before instrumentation
2 subq  rsp, 0x30         ; rsp = rsp - 0x30
3
4 ; After instrumentation
5 subl  esp, 0x30         ; rsp = (0xffffffff & rsp)-0x30
6 leaq  [r15 + rsp], rsp  ; rsp = r15 + rsp
```

Figure 4.6. Updating `rsp` register. SecureLayer safeguards direct updates to `rsp` and `rbp`, ensuring they stay within a thread's private region.

```
1 ; Before instrumentation
2 call rax
3
4 ; After instrumentation
5 andl eax, 0x3fffffe0  ; i.e., mask and align
6 leaq rax, [r14 + rax] ; rax = r14 + rax
7 call rax
```

Figure 4.7. Software enforcement on an indirect branch instruction. The line `5` clears the upper `34 bits` of `rax` and aligns it with `32 bytes`, similar to the indirect branch enforcement of Native Client [133]. It prevents the program from bypassing CHANCEL's instrumentation checks or jumping outside `sgx.code`.

the base address of shared region (`sgx.shared`) during shared data initialization (the third column of Figure 4.3a) or (b) the base address of each thread's private region during servicing (the last column of Figure 4.3a). Then, CHANCEL enforces permissions by instrumenting the program's control-flow and data-flow instructions using these registers. The following paragraphs explain CHANCEL's MCSFI instrumentation concerning the servicing stage only but permissions during initialization are enforced in the same way.

Figure 4.4 shows how CHANCEL instruments load instructions to bound them to a thread's private region and `sgx.shared`. In particular, if the target of the load instruction (`r13`) is less than the base of `sgx.code` (`r14`), the destination is masked to point to the thread's private region (see lines `10-12`). Otherwise, CHANCEL allows accessing the original target, since the target must be `sgx.shared`.

Figure 4.5 depicts how CHANCEL instruments store instructions to bound them to a thread's private region only. In particular, line `6` masks the destination to set `r13` as the distance from the base of the thread region (`r15`). Therefore, the destination of the store in line `7` (`r15+r13`) points to the thread region. Note that this instrumentation also prevents the program from rewriting the enclave's executable regions, `sgx.code` and SecureLayer.

However, CHANCEL does not need to instrument all load and store instructions. In particular, CHANCEL confines data-flow concerning stack objects (e.g., local variables), by ensuring that the stack registers, `rsp` and `rbp`, point to a thread's private region (Figure 4.6). The guard region between segments (shown in Figure 4.3a) prevents a malicious stack incursion on another thread's private region or code regions.

Moreover, to ensure that the program does not bypass CHANCEL's instrumented data-flow checks or execute code that contains no checks (e.g., SecureLayer), CHANCEL aligns the target program's code and instruments every indirect branch instruction including `call`, `jmp`, and `ret` (Figure 4.7). In particular, each valid call target in the program is aligned with 32 bytes (using `nop` instructions). Therefore, an indirect branch's target is also forced to be aligned with 32 bytes (line `5`), which ensures each transfer is a valid starting address of CHANCEL's instrumented indirect branch sequence, i.e., not a direct jump to the `call` instruction. Then, the indirect branch's target is masked to 1 GB (line `5`) and redirected using `r14` (line `6`), ensuring that the target is within `sgx.code`.

Importantly, unlike other control-flow checks that are vulnerable under multi-threaded execution (e.g., shadow stack [134]), CHANCEL's checks are thread-safe. In particular, CHANCEL loads the indirect branch address into a register, performs all transformations on the register, and jumps to the final target stored in the register (Figure 4.7). Since a thread cannot manipulate another thread's registers, it cannot divert the other thread's control-flow.

Finally, direct memory access (i.e. using an absolute or `rip`-relative address) can be abused to read memory belonging to other threads during execution or overwrite executable pages. Therefore, CHANCEL validates (§4.2.2) that the program binary does not contain direct memory access instructions.

4.2.4 Shared Data Initialization

The *read-only* data, shared between threads, may belong to global variables, the enclave's heap, or shared (in-enclave) files. The service provider specifies the shared data using annotation (for global objects), run-time specification (for heap objects) and load-time specification (for shared files). Furthermore, CHANCEL allows the initialization of shared data both after program loading and servicing all client requests.

Shared global objects. CHANCEL's compiler provides an attribute, `annotate("sgx.shared")`, to indicate that a certain global variable is shared. During program loading, CHANCEL moves the marked global data into `sgx.shared`.

Shared heap objects. CHANCEL initializes a heap region in `sgx.shared` and allows the program to use heap allocation routines (e.g., `malloc`, `calloc`) to allocate and subsequently initialize shared data within the heap. Importantly, the shared heap is meant only to share *read-only* data. While servicing user requests, each thread writes to an internal heap initialized at the thread's private memory region (§4.2.5).

Shared files. The program developer informs CHANCEL (during enclave creation) about the program's required files and their permissions (i.e., *read-only* or *writable*). CHANCEL loads the read-only files into the `sgx.shared` segment and exposes file system routines (§4.2.5) to permit initialization.

In the future, CHANCEL can automate shared data initialization using static data-flow analysis to determine shared objects and files, without developer annotation, similar to an existing SGX automated compartmentalization scheme [135]. Furthermore, if the resulting analysis is too imprecise, it can be improved through dynamic analysis with a representative workload [136], [137]. Importantly, developer-assisted identification does not pose security threats. In particular, shared objects are read-only during servicing and cannot harm

```
// Receive data from the corresponding client.
bool recv(void* buf, uint buflen);
// Send data to the corresponding client.
bool send(void* buf, uint buflen);
// Notify the end of data migration
void end_migration();
// Terminate the thread.
void exit();
```

Figure 4.8. Some runtime interfaces supported by SecureLayer.

CHANCEL's goals. Hence, imprecise or malicious identification only reduces performance since redundant data exists in thread regions.

4.2.5 Runtime Services

SecureLayer provides three runtime services to the program: in-enclave file system, dynamic memory allocation, and shielded client communication.

In-enclave file system. CHANCEL implements an in-enclave file system for application compatibility since many applications, like web servers, extensively use file system abstractions for operation. To use the file system, the program developer specifies a list of files, which SecureLayer loads into the enclave during initialization. In particular, SecureLayer loads the read-only files into the `sgx.shared` segment and the writable files into private thread regions. While servicing client requests, SecureLayer exposes the POSIX file system routines (e.g., `open, read`) to access these files. Finally, after servicing, the writable files, in each thread's private region, are restored to their original contents to avoid the leakage of confidential data through overwritten files.

Dynamic memory allocation. CHANCEL provisions each thread with a private heap, initialized at the thread's private region. The size of the internal heap is configurable but must be specified by the program developer, similar to the heap in native enclave programs. While servicing user requests, SecureLayer exposes heap allocation routines (e.g., `malloc`, `calloc`) to allow the thread to dynamically allocate memory from its internal heap. After servicing a client's request, the thread's private heap contents are cleared to avoid confidential data leakage.

Shielded client communication. CHANCEL mediates the entire communication between a client and their connected thread, to avoid direct and covert confidential data leakage through this communication. Initially, SecureLayer validates that the program binary does not contain instructions to exit the enclave, i.e., `EEXIT` instructions required for SGX system calls (OCALLs). Then, SecureLayer provisions two API functions, `recv()` and `send()` (Figure 4.8), allowing a thread to receive or send client data, respectively. However, to ensure confidentiality, SecureLayer encrypts all outgoing data with a shared key established with the concerned client (refer to §4.2.2). Finally, to stop covert channels created by the service provider (i.e., encode client data into either size or timing of outgoing data), SecureLayer transmits a fixed size of data at every predefined time intervals, similar to prior work [125].

Note that, despite encryption, a program could try to encode sensitive information in the output if it knows the encrypted output. However, the encrypted output is generated by SecureLayer, as mentioned previously. Hence, such encoding-based attempts to exfiltrate sensitive information from the enclave are unsuccessful.

4.3 Implementation

This section describes CHANCEL's development toolchain and procedure to build the target program (§4.3.1), as well as SecureLayer's components, executing in the enclave (§4.3.2).

4.3.1 Program Development Toolchain

The toolchain compiles and instruments a target program and its shared libraries into a binary (`.so`) file. The components of the toolchain are not included in the trusted computing base (TCB) since SecureLayer validates the instrumentation of the output binary during program loading. Note that SecureLayer is not developed using this toolchain.

Compiler. The CHANCEL compiler is based on the LLVM backend [138] with 1,162 lines of code changes and 94 lines of linker scripts. The backend instruments the program according to MCSFI, whereas the linker script provisions a MCSFI-aware memory layout (§4.2.3).

Supported C libraries. CHANCEL supports Linux programs that are built using either `tlibc` [139], a minimalistic C library provided by Intel, or `musl libc` [140], a robust C library

Table 4.1. CHANCEL's components included in the enclave.

Component	KLoc	Base
SecureLayer		
Validator	53	`Capstone` [141]
Loader	1.3	-
Runtime services library	0.5	-
C library	15 / 66	`tlibc` [139] / `musl` [140]
Crypto library	23	`tcrypto` [46]
Total	92.8 / 143.8	

which simplifies program development. For CHANCEL's `musl libc`, we statically removed all routines requiring system calls and redirected all supported system call functionality (e.g., file system) to SecureLayer.

Required routines. CHANCEL expects the program to have two additional routines, `shared_init`, which initializes shared data, and `service`, which services a client's request. These routines are not unique since they are required for any program that processes client requests. CHANCEL's only additional requirement is that the `service` routine must use the shielded client communication (§4.2.5) to send and receive data.

4.3.2 SecureLayer Components

SecureLayer runs within the enclave, loads and validates a provided binary, and enables runtime services. Therefore, the SecureLayer constitutes CHANCEL's TCB. Table 4.1 provides a breakdown of SecureLayer's components alongside other libraries included within the enclave.

Validator. SecureLayer includes an x86 disassembler, based on Capstone [141], which validates that the provided binary is correctly instrumented (§4.2.3).

Loader. The loader relocates program symbols, enforces MCSFI (i.e., using `r14` and `r15`), and provisions the shared (i.e., `sgx.shared`) and per-thread data (e.g., private global data).

Runtime services library. This library supports the services mentioned in §4.2.5. For heap allocation, it includes wrapper functions (e.g., `malloc`) but reuses the SGX SDK's [111] heap allocation logic to initialize and maintain a heap in private thread regions. Furthermore, for secure communication channels with clients, it uses Intel's cryptographic library, `tcrypto`.

Table 4.2. CHANCEL's defenses against various attack vectors. Instr. means instrumentation.

Attack goal	Detailed attack method	Defence
Instr. bypass using existing code		
Jump after checks	`jmp`, `call`, or `ret` using register	Aligned (32-bytes) indirect branch transfers (Figure 4.7)
Jump outside `sgx.code`	`jmp`, `call` using register	Mask branch target and redirect to `sgx.code` (Figure 4.7)
	An invalid return (`ret`)	Instrument `pop`, mask target, and redirect to `sgx.code` (Figure 4.7)
Modify `r14` or `r15`	Assembly instructions	Caught during validation (§4.2.2)
	Use SecureLayer code [142]	Not located in `sgx.code`; therefore, not executable
Instr. bypass using injected code		
Modify SecureLayer or `sgx.code`	Write using register	Instrumented to target thread region only (Figure 4.5)
	Update `rsp, rbp` and `push`	Ensure `rbp, rsp` point to thread region (Figure 4.6); Guard page before thread regions
Add new code pages	Use `SGX2` instructions (`EACCEPT`)	Caught during validation (§4.2.2)
Extract confidential client data		
Read from other threads	Read using register	Instrumented to target thread region and `sgx.shared` (Figure 4.4)
	Update `rsp, rbp` and `pop`	Instrumented to ensure `rbp, rsp` point to thread region (Figure 4.6); Guard page after thread regions
Write outside the enclave	Write using register	Instrumented to target thread region only (Figure 4.5)
	`EEXIT` and leak values in registers	`EEXIT` is caught during validation (§4.2.2)
Others	Save thread data and leak it later	Clear thread region after servicing each user (§4.2.2)
	Absolute or PC-relative access	Caught during validation (§4.2.2)

4.4 Security Analysis

This section first elaborates on CHANCEL's defenses against attempts to bypass its instrumentation either using existing code or by injecting code. Then, it describes how CHANCEL's instrumentation prevents the extraction of sensitive client data. Finally, this section presents our validation results based on several implemented attacks. Table 4.2 provides an overview of all attacks and defenses.

Prevent instrumentation bypass using existing code. The attacker can attempt code reuse attacks to bypass CHANCEL's instrumentation checks (e.g., jump directly to a `mov` instruction), execute code that does not contain instrumentation checks (e.g., SecureLayer), or modify general-purpose registers (e.g., `r14`, `rbp`) to render instrumentation checks ineffective.

CHANCEL prevents all such attempts by restricting the code that the attacker can execute. During compilation, CHANCEL aligns all valid call targets in the program with 32 bytes. Then, CHANCEL instruments control-flow instructions (e.g., `call`, `jmp`, `ret`) to ensure that indirect branch targets are aligned with 32 bytes to prevent the attacker from jumping to the middle of an instrumentation sequence (Figure 4.7). The control-flow instrumentation

also ensures that the attacker can only execute code within `sgx.code`, preventing code reuse attacks involving the remaining enclave code (i.e., SecureLayer and others).

Furthermore, CHANCEL protects instrumentation-critical registers, i.e., `r14`, `r15`, `rbp`, and `rsp`. In particular, the target program is not allowed to contain instructions to modify `r14` or `r15`, while all explicit updates to `rbp` and `rsp` are instrumented to ensure they remain within the thread's region (Figure 4.6). Finally, while some pre-loaded enclave code (e.g., `asm_oret`) can also modify these registers [142], all such code is part of SecureLayer, and is not executable by the program.

Prevent instrumentation bypass using injected code. The attacker might try to inject malicious code into either of the two executable regions, i.e., SecureLayer or `sgx.code`. However, CHANCEL instruments memory writes (Figure 4.5), which ensures that the program cannot modify these regions. Furthermore, CHANCEL prevents stack operations (e.g., `push`) from overflowing to code regions using guard pages. Finally, the attacker might use SGX2 instructions (e.g., `EACCEPT` and `EMODPE`) to add additional executable pages, during execution, and inject their malicious code. Such instructions are forbidden and caught during validation.

Prevent extraction of confidential client data. The previous sections explain how CHANCEL ensures that its instrumentation is not bypassed. This section explains how CHANCEL's instrumentation prevents the extraction of confidential data.

CHANCEL prevents the program from leaking sensitive client data by ensuring that all memory writes target each thread's private region (Figure 4.5). CHANCEL also prevents malicious writes using stack operations (e.g., `push`) through guard pages (Figure 4.3a) and instrumentation of explicit updates to stack registers (Figure 4.6), ensuring all stack operations are in the thread's private region. Furthermore, to prevent the program from disclosing memory outside the enclave through SGX system calls (i.e., OCALLs), CHANCEL validates that the program binary does not contain `EEXIT` instructions. Hence, all outside communication is through CHANCEL's shielded service (§4.2.5), which ensures confidentiality through encryption using a shared key with the concerned client.

CHANCEL also instruments load instructions (Figure 4.4) to prevent a thread from directly reading another thread's private memory region and leaking the sensitive client data belonging

to that thread. Finally, guard pages between thread regions and instrumented updates to stack pointers prevent the abuse of `pop` instructions to read data from other threads.

4.5 Performance Evaluation

In this section, we provide a performance evaluation of CHANCEL with the goal of answering the following questions:

- How does CHANCEL compare to a multi-process sandbox (§4.5.1)?
- What is the overhead of CHANCEL on benchmarking applications (§4.5.2)?
- How does CHANCEL perform for real-world target scenarios (§4.5.3)?

Experimental setup. All experiments were conducted on an Intel (R) Core (TM) i7-6700K CPU 3.40GHz (4 cores and 8 threads) and 64GB RAM (128 MB for EPC). The machine ran a 64-bit Ubuntu 16.04.5 LTS with Linux version 4.4.207. We ran our SGX enclaves using Intel SGX SDK v2.2 [111] and Intel SGX driver v2.6 [143].

Terminology. For each experiment, we compare (a) NATIVE, referring to an enclave running the target application in multiple threads, without CHANCEL's instrumentation, (b) CHANCEL, which refers to an enclave running CHANCEL's multi-client sandbox using multiple threads, and (c) CHANCEL-MP, which refers to enclaves running CHANCEL but with different enclave processes rather than threads.

4.5.1 Improvement over Multi-Process Sandbox

This section analyzes the performance of CHANCEL in comparison with a traditional multi-process sandbox approach such as Ryoan [125].

Settings. Our benchmark application allocated a large in-enclave memory region and sequentially accessed the memory region. For enclaves running NATIVE and CHANCEL, the memory was allocated once in `sgx.shared` and was read by different enclave threads. However, the memory was cloned to each enclave process with CHANCEL-MP. The benchmark application executed as follows: read 8 bytes k times in each 512 KB region of the m MB memory chunk from each thread or process. Furthermore, we also measured the number of

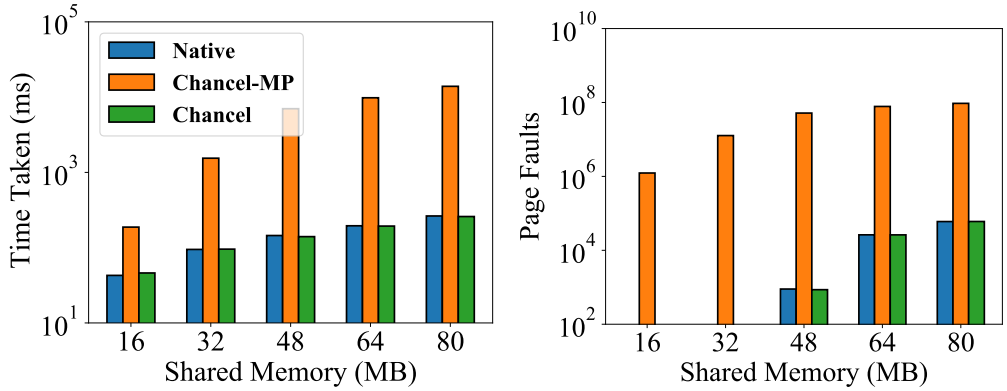


Figure 4.9. (a) Average completion time and (b) the total number of EPC page faults when the amount of memory shared increases linearly. CHANCEL is $4.06 - 53.70\times$ faster than CHANCEL-MP and incurs a slowdown of only $0.8 - 7.5\%$ over NATIVE.

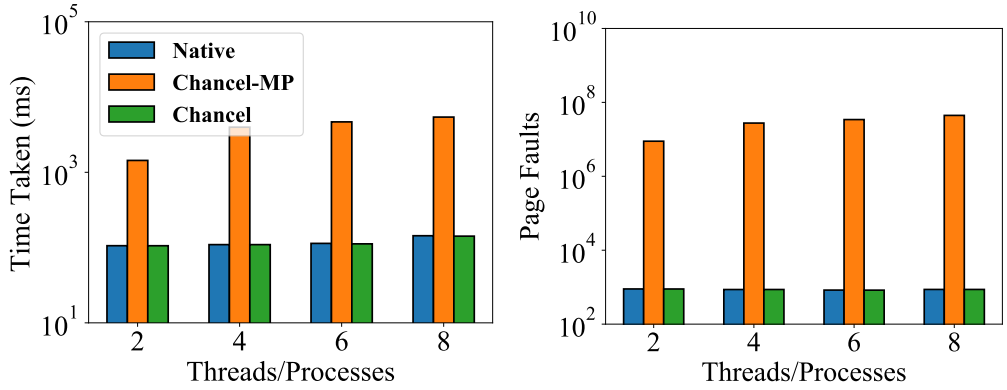


Figure 4.10. (a) Average completion time and (b) the total number of EPC page faults when the number of processes/threads increases linearly. CHANCEL is $13.59 - 41.73\times$ faster than CHANCEL-MP and incurs an overhead of only $0.2 - 1.0\%$ over NATIVE.

EPC page faults by hooking the SGX page fault handler. Finally, we ran each experiment 50 times and report the average.

Results. Figure 4.9 shows the impact on completion time and number of page faults while varying the amount of memory accessed (m). We configured all runs as $n = 8$ processes/threads and $k = 16$. The figure shows that CHANCEL out-performs CHANCEL-MP by $4.06 - 53.70\times$. As far as CHANCEL-MP is concerned, the memory chunk is cloned to each process. Therefore, it exerts a high memory pressure on the limited EPC, evident from the considerable increase in page faults, as we increase m. On the other hand, NATIVE and

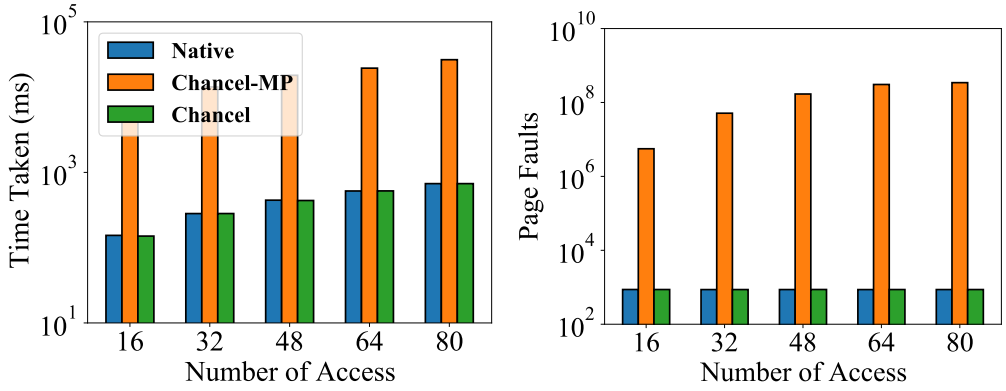


Figure 4.11. (a) Average completion time and (b) the total number of EPC page faults when the number of memory accesses to each EPC page increases linearly. CHANCEL is $37.88 - 48.08\times$ faster than CHANCEL-MP and incurs an overhead of only $0.1 - 0.8\%$ compared to NATIVE.

Table 4.3. Nbench [126] running inside CHANCEL. The table shows slowdown incurred and additional instructions executed.

Benchmark	**Native**	**Chancel**	
		Slowdown	**Additional instr.**
	(iterations/sec)	(%)	(%)
NUMERIC SORT	906.28	17.09	39.07
BITFIELD	4.55×10^8	24.89	40.23
STRING SORT	669.77	22.76	39.99
FP EMULATION	94.72	16.28	38.18
FOURIER	53470.00	2.39	4.21
ASSIGNMENT	23.52	3.44	11.72
IDEA	2962.20	0.63	7.45
HUFFMAN	2535.40	23.40	22.60
NEURAL NET	36.94	12.54	24.78
LU DECOMPOSITION	1066.50	0.91	6.99
Average	-	12.43	23.52

CHANCEL scale nicely, incurring no page faults for smaller memory chunks and fewer page faults otherwise. Note that we observe page faults starting from 48 MB since some memory is allocated for SecureLayer components and runtime services (refer to §4.3.2).

Furthermore, we show how the performance scales while increasing the number of enclave threads versus enclave processes. Figure 4.10 shows the results while increasing the process/thread count (n) from 2 to 8 while keeping $m = 48$ MB and $k = 16$. Despite increasing the number of processes, CHANCEL shows very similar completion times (i.e., less than 5%

overhead for 8 threads compared to 2 threads) and a similar number of page faults due to the sharing of memory. Note that our benchmark application uses negligible per-thread memory; therefore, there is no noticeable increase in page faults when increasing the number of threads. In contrast, each new enclave process incurs additional page faults and degrades the performance of CHANCEL-MP.

Figure 4.11 depicts the average completion time and number of EPC page faults when the number of memory accesses (k) increases with $n = 8$ processes/threads and $m = 48$ MB. In particular, as we increase the number of memory accesses for CHANCEL-MP, each enclave process accesses an enclave page for a longer duration, resulting in more contention on the EPC memory, more page faults, and longer completion time. In contrast, CHANCEL shares the shared memory page with other threads; therefore, there is no noticeable increase in page faults even as we increase k. However, we observe a longer completion time for CHANCEL and NATIVE as we increase k because each additional memory access adds latency.

Takeaway. CHANCEL outperforms a multi-process sandbox by $4.06-53.70\times$ when increasing the amount of shared memory, number of accesses, or number of threads.

4.5.2 Overhead of Chancel

We calculate the overhead of CHANCEL using a popular benchmarking application, nbench [126]. The application executes various CPU and memory-intensive tasks including sorting algorithms, bit manipulation, and floating point emulation. Each task is executed for a fixed amount of time and nbench outputs the average number of iterations it executed per-second (i.e., throughput). Nbench has previously been used in the evaluation of similar SGX systems [144], [145].

Settings. We ran nbench in a non-enclave setting but using CHANCEL's program loader and validator, i.e., the setting was the same as it would be in an enclave. A non-enclave execution allows us to ascertain the actual cost of CHANCEL without amortization due to EPC page faults. We ran each test 50 times and report the average.

Results. Table 4.3 reports both the throughput slowdown and the number of additional instructions executed (determined using `perf` [146]). The performance overhead was 0.91 – 24.89%, averaging at 12.43%.

CHANCEL adds overhead due to two reasons: (a) reserving `r14` and `r15`, which results in more memory-spills due to fewer available registers and (b) executing additional instructions to enforce Multi-Client SFI, which increases overall computational cycles. Concerning the latter, since data accesses generally outnumber control-flow transfers, CHANCEL's major overhead is from the instrumentation checks on data accesses. Therefore, memory-intensive benchmarks (e.g., NUMSORT and STRINGSORT) are more affected by CHANCEL and exhibit a higher overhead.

Note that for some applications (e.g., IDEA and FOURIER), the number of additional instructions executed by CHANCEL is minimal; therefore, they exhibit negligible overheads. Such scenarios happen for two reasons. First, the application is CPU-intensive; i.e., executes fewer instrumentation checks for data access. Second, the application mostly performs stack-based data access (e.g., allocate an array on the stack and perform computation on the array). In the latter case, CHANCEL has to execute fewer checks since data access targeting the stack is protected by ensuring that `rsp` and `rbp` always point to a thread's private memory region (refer to §4.2.3).

Therefore, under many scenarios, CHANCEL's overhead is low. Importantly, CHANCEL's performance overhead is comparable or superior to existing SGX SFI implementations [125], [145]. In particular, MPTEE [145] reported an overhead of 0.4 – 34% on nbench, while Ryoan [125] reported an (emulated) overhead of 12 – 100% on its evaluated real-world applications. In contrast, even in highly memory-intensive scenarios, CHANCEL exhibits a worst-case overhead of less than 25%. Hence, we expect that CHANCEL is applicable to a wide range of scenarios.

4.5.3 Performance with Real-world Programs

Based on CHANCEL's target scenarios (§4.1.2), we evaluate five real-world programs—DrugBank [128], [147], OSSEC [148], Recommender [149]), ShieldStore [150], and Snort [151].

Table 4.4. Real-world evaluated program statistics. The table shows each program's NATIVE code (`.text` section) size and its increase due to CHANCEL's instrumentation. The table also shows the total instrumented binary size (including code and static data) and its loading time.

Application	Code		Chancel Binary	
	Native (KB)	Chancel (+%)	Size (MB)	Load time (ms)
OSSEC (IDS)	26.17	49.66	123.51	529.94
DrugBank (PIR)	15.22	23.40	2.12	82.12
Recommender (PRS)	55.35	89.23	2.29	84.17
ShieldStore (PIR)	11.65	76.62	2.07	81.23
Snort (IDS)	40.12	100.57	2.04	80.89

Common settings. We ran the programs using four and eight clients. The four client setting exhibits CHANCEL's realistic performance on our machine when hyper-threading (HT) is disabled to defeat SGX micro-architectural defects, as recommended by Intel [152]. In contrast, the eight client setting shows CHANCEL's performance with more capable current SGX CPUs that support eight processor cores even when hyper-threading is disabled.

Moreover, we ran each program under two workload types, light (less than 128 MB) and heavy (greater than 128 MB). This distinction considers whether the workloads are small enough to fit within our machine's EPC memory (128 MB) entirely or not. Each thread was allocated 8 MB of private memory for the NATIVE and CHANCEL experiments. Finally, we ran each experiment 50 times and report the average.

Table 4.4 shows the overall statistics, including code size and its increase, instrumented binary size and loading time, for each application.

OSSEC (intrusion detection system). We evaluate OSSEC [148], which is a famous and widely-used IDS, using CHANCEL. OSSEC analyzes packet payloads to detect trojans and viruses, based on a dictionary of pre-defined signatures.

Settings. We initialized OSSEC using a database of virus signatures from ClamAV [153]. In the case of NATIVE and CHANCEL, OSSEC initialized its internal dictionary on the shared heap. Throughout the experiments, we gradually inserted a different number of signatures to increase the size of its dictionary. Then, we analyzed 100 packets (60 bytes each) from each thread or process, to check for malicious content.

Table 4.5. Average delay (and the number of page faults in the parenthesis) for inspecting a payload with regex matching in OSSEC. The overhead imposed by CHANCEL over NATIVE is $2.7 - 13.1\%$.

`sgx.shared`	Native	Chancel-MP	Chancel	Improv.
Four clients (HT off)				
Light workloads				
18 MB	29.48 ms (130K)	38.04 ms (718 K)	31.41 ms (130 K)	0.21×
36 MB	53.51 ms (136K)	169.38 ms (1668 K)	58.13 ms (136 K)	1.90×
72 MB	92.11 ms (146K)	650.86 ms (4442 K)	100.77 ms (146 K)	6.45×
Heavy workloads				
144 MB	724.70 ms (2655 K)	1437.23 ms (11463 K)	769.02 ms (2655K)	0.87×
288 MB	1314.70 ms (4794 K)	2713.26 ms (19415 K)	1335.82 ms (4797K)	1.03×
576 MB	2625.10 ms (9106 K)	6169.55 ms (35752 K)	2653.58 ms (9112K)	1.32×
Eight clients (HT on)				
Light workloads				
18 MB	34.80 ms (497 K)	178.55 ms (7185 K)	38.10 ms (501K)	3.69×
36 MB	66.07 ms (503 K)	418.51 ms (8705 K)	69.56 ms (508K)	5.02×
72 MB	111.34 ms (514 K)	1090.45 ms (13072 K)	125.94 ms (538K)	7.66×
Heavy workloads				
144 MB	934.70 ms (2655 K)	2560.97 ms (24211 K)	961.69 ms (2696K)	1.66×
288 MB	1794.70 ms (4794 K)	6192.66 ms (46837 K)	1948.20 ms (4815K)	2.18×
576 MB	2925.10 ms (9106 K)	12656.20 ms (89373 K)	3185.80 ms (9948K)	2.97×

Results. Table 4.5 shows the results obtained on dictionaries of size $18 - 576$ MB. In particular, CHANCEL shows a performance improvement of $0.21 - 7.66\times$ over CHANCEL-MP.

Since OSSEC performs regular expression (regex) matching to compare a query (packet) with each signature in its dictionary, a query's working set and analysis time should increase proportionally to the dictionary size. We observe that CHANCEL and NATIVE incur few page faults under light workloads; hence, they show proportional analysis time increase relative to the dictionary size. However, CHANCEL-MP shows a disproportional increase in analysis time due to a significant number of page faults.

Interestingly, under heavy workloads, CHANCEL's improvement against CHANCEL-MP reduces ($0.87 - 1.32\times$ with four clients). In particular, even multi-threaded execution over the EPC limit incurs many page faults because OSSEC must access a large amount of memory for reach request. Hence, we see a significant jump in analysis time even for NATIVE and CHANCEL. Nevertheless, CHANCEL's improvement increases with the increase in dictionary size of heavy workloads and the number of clients (up to $1.32\times$ and $2.97\times$ with four and eight clients, respectively). Judging by the observed trend, we expect more improvement with additional clients and heavier workloads. Hence, CHANCEL suits IDS applications such as OSSEC in both light and heavy workloads.

DrugBank (private information retrieval). We use a C hash map application [147] to act as a secure database for a company providing drug recommendations. The application uses CRC32-based hashing to insert and retrieve entries.

Settings. We populated the hash map using a drug database obtained from the famous DrugBank website [154]. For NATIVE and CHANCEL, the program used the shared heap to allocate its backing store. During the experiment, we inserted a varying number of entries from the database into the hash map. Then, we searched $2,000,000$ drug-related queries from the hash map using each thread or process.

Results. Table 4.6 shows the results obtained while increasing the hash map size from $30 - 480$ MB. The application shows an improvement of $0.02 - 10.01\times$. Unlike OSSEC, the DrugBank application exhibits a continuous rising trend in improvement, even with heavy workloads. The DrugBank program has a minimal working set for each query. In particular, due to hashing, the application retrieves a small set of enclave pages for each query.

Under light workloads, due to DrugBank's minimal query working set, CHANCEL-MP shows reasonable performance—CHANCEL improves only up to $0.76\times$. However, under heavy workloads, the competition for EPC memory increases due to larger per-enclave hash maps; hence, CHANCEL-MP naturally incurs more page faults. In contrast, the shared hash map alongside the minimal query working set ensures that even under heavy workloads, CHANCEL incurs few additional page faults. Therefore, CHANCEL further improves over CHANCEL-MP

Table 4.6. Average delay (and the number of page faults in the parenthesis) to search 2,000,000 queries in DrugBank. The overhead imposed by CHANCEL over NATIVE is $0.2 - 11.4\%$.

`sgx.shared`	Native	Chancel-MP	Chancel	Improv.
Four clients (HT off)				
Light workloads				
30MB	411.64 ms (82K)	420.79 ms (556K)	412.51 ms (82K)	0.02×
60MB	414.80 ms (90K)	526.95 ms (651K)	419.81 ms (90K)	0.12×
90MB	414.90 ms (99K)	564.49 ms (670K)	422.92 ms (99K)	0.27×
Heavy workloads				
180 MB	413.02 ms (322 K)	805.98 ms (3227 K)	423.44 ms (324K)	0.90×
360 MB	416.39 ms (346 K)	2422.80 ms (10073 K)	424.09 ms (350K)	4.71×
480 MB	418.39 ms (429 K)	3150.80 ms (25389 K)	425.82 ms (429K)	6.41×
Eight clients (HT on)				
Light workloads				
30 MB	571.12 ms (299 K)	861.18 ms (5940 K)	616.08 ms (299K)	0.38×
60 MB	569.15 ms (309 K)	943.80 ms (6015 K)	621.50 ms (312K)	0.52×
90 MB	558.84 ms (309 K)	1094.59 ms (5706 K)	622.36 ms (316K)	0.76×
Heavy workloads				
180 MB	580.53 ms (343 K)	2579.74 ms (18894 K)	627.95 ms (345K)	3.11×
360 MB	581.87 ms (408 K)	5666.67 ms (40650 K)	628.47 ms (418K)	8.02×
480 MB	582.75 ms (447 K)	6960.90 ms (45338 K)	631.71 ms (451K)	10.01×

under heavy workloads (up to 6.41× for four clients). Finally, like OSSEC, increasing the clients emphasizes CHANCEL's improvement (up to 10.01× for eight clients).

Hence, applications like DrugBank, with a minimal query working set, benefit modestly from CHANCEL under light workloads but considerably under heavy workloads.

Recommender (product recommendation service). Recommender [149] is an open-source tool that uses Collaborative Filtering (CF) to suggest products. The tool builds a model based on a user's past behavior and the behavior extrapolated from other users to provide highly accurate suggestions.

Settings. We used a benchmark that creates a set of clients and populates their history of purchases. Similar to other experiments, the benchmark used the shared heap to allocate a product catalog under CHANCEL. Then, each thread/process used the randomly populated client information to search through and recommend products from the catalog.

Results. Table 4.7 shows the results as we increase the product catalog size from 28 MB to 504 MB. We notice a pattern similar to OSSEC but with CHANCEL having even more significant performance improvement over CHANCEL-MP (up to 17.20×) under light workloads. We expect these results since recommender uses CF on each product in the catalog; hence, its query working set depends on the catalog size, like OSSEC. Furthermore, under heavier workloads, CHANCEL's performance improvement reduces but remains significant and shows an upwards trend with increasing catalog size (up to 2.01×). Finally, with eight clients, CHANCEL's performance improvement expands to 21.18× and 4.02×, under light and heavy recommender workloads, respectively.

ShieldStore (private information retrieval). ShieldStore [150] is an optimized key-value store that reports up to 20× better performance than memcached [155] in SGX enclaves, through various key-based optimizations.

Settings. We populated the store using a provided benchmark that inserts random 16B key-value pairs. Then, we implemented a custom benchmark to retrieve $100,000$ keys at fixed offsets relative to the number of populated keys—if $1,000,000$ keys were populated initially, the test retrieved every tenth key.

Results. Table 4.8 shows the results obtained with various stores ranging from 16 to 384 MB in size. CHANCEL exhibits a performance improvement over CHANCEL-MP of $0.68 - 16.32\times$. The observed trend is similar to OSSEC and Recommender—CHANCEL's performance improvement, with four clients, in light workloads (up to 11.44×) is better than on heavy workloads (up to 1.20×). We believe that is because our benchmark deliberately accesses keys at fixed intervals; therefore, it retrieves a large portion of the store. Consequently, CHANCEL and NATIVE also incur many page faults on heavy workloads, which reduces performance.

Table 4.7. Average delay (and the number of page faults in the parenthesis) to access a recommendation result. The overhead imposed by CHANCEL over NATIVE is $1.3 - 13.1\%$.

`sgx.shared`	Native	Chancel-MP	Chancel	Improv.
Four clients (HT off)				
Light workloads				
28 MB	1.46 ms (94 K)	2.22 ms (204 K)	1.56 ms (94 K)	0.41×
56 MB	3.94 ms (100 K)	13.03 ms (619 K)	4.19 ms (100 K)	2.11×
112 MB	8.45 ms (114 K)	157.98 ms (2472 K)	8.68 ms (114 K)	17.20×
Heavy workloads				
252 MB	420.17 ms (2063 K)	945.41 ms (12884 K)	429.54 ms (2064 K)	1.20×
378 MB	621.21 ms (2765 K)	1645.87 ms (17324 K)	639.23 ms (2778 K)	1.56×
504 MB	838.54 ms (3787 K)	2545.92 ms (21400 K)	842.78 ms (3793 K)	2.01×
Eight clients (HT on)				
Light workloads				
28 MB	2.17 ms (351 K)	4.24 ms (2980 K)	2.37 ms (353K)	0.78×
56 MB	4.66 ms (357 K)	41.57 ms (3926 K)	5.11 ms (358K)	7.13×
112 MB	9.67 ms (364 K)	217.56 ms (4412 K)	9.78 ms (370 K)	21.18×
Heavy workloads				
252 MB	460.49 ms (2081 K)	1879.63 ms (24200 K)	465.40 ms (2092K)	3.04×
378 MB	682.28 ms (2784 K)	3137.23 ms (35619 K)	702.36 ms (2891K)	3.47×
504 MB	916.97 ms (3820 K)	4699.94 ms (52573 K)	936.38 ms (3836K)	4.02×

Nevertheless, like previous programs, eight clients further improves CHANCEL's performance, compared to CHANCEL-MP, by up to 16.32× and 2.92× for light and heavy workloads, respectively. Furthermore, CHANCEL exhibits an upward improvement trend, against CHANCEL-MP, in heavy workloads and servicing more clients. Hence, in real-world settings, where many clients and heavier workloads are normal, CHANCEL should significantly outperform CHANCEL-MP for applications like ShieldStore.

Table 4.8. Average delay (and the number of page faults in the parenthesis) to search 100,000 queries using ShieldStore [150]. The overhead imposed by CHANCEL over NATIVE is 1.1 − 8.4%.

`sgx.shared`	Native	Chancel-MP	Chancel	Improv.
Four clients (HT off)				
Light workloads				
16 MB	57.11 ms (419 K)	99.42 ms (1144 K)	59.11 ms (419K)	0.68×
32 MB	61.15 ms (420 K)	343.02 ms (5868 K)	62.15 ms (420K)	4.52×
64 MB	64.21 ms (429 K)	823.34 ms (15003 K)	66.21 ms (429K)	11.44×
Heavy workloads				
128 MB	508.74 ms (3648 K)	976.25 ms (33595 K)	518.74 ms (3648K)	0.88×
256 MB	967.50 ms (13222 K)	1930.33 ms (76968 K)	997.06 ms (13275K)	0.94×
384 MB	1107.10 ms (22406 K)	2465.58 ms (108189 K)	1118.93 ms (22431K)	1.20×
Eight clients (HT on)				
Light workloads				
16 MB	90.47 ms (595 K)	434.33 ms (7401 K)	94.71 ms (298K)	3.59×
32 MB	95.45 ms (304 K)	488.47 ms (11739 K)	97.80 ms (305K)	3.99×
64 MB	99.31 ms (311 K)	1739.63 ms (32121 K)	100.44 ms (314K)	16.32×
Heavy workloads				
128 MB	696.67 ms (3204 K)	2029.59 ms (81130 K)	719.84 ms (3230K)	1.82×
256 MB	1245.70 ms (12325 K)	3891.65 ms (196678 K)	1261.72 ms (12434K)	2.09×
384 MB	1383.82 ms (21536 K)	5664.49 ms (303476 K)	1446.32 ms (21542K)	2.92×

Snort (intrusion detection system). Snort [151] is a widely-deployed and open-source network intrusion detection system that is capable of real-time traffic analysis and logging. Snort routinely publishes its set of rules that aid its detection of malicious network activity.

Settings. We divided Snort's official published rules into different sizes and populated the rules in Snort's internal malware database. Then, we examined 3,000 randomly-created network packets using snort.

Results. Table 4.9 shows the results obtained while using various databases of sizes 32 to 1047 MB. In general, CHANCEL exhibits a performance improvement over CHANCEL-MP of up to 4.14× and 5.24× on four and eight clients, respectively. The Snort experiment

Table 4.9. Average delay (and the number of page faults in the parenthesis) to inspect 3,000 packets using Snort [151]. The overhead imposed by CHANCEL over NATIVE is $0.5 - 11.8\%$.

`sgx.shared`	Native	Chancel-MP	Chancel	Improv.
Four clients (HT off)				
Light workloads				
32 MB	1.23 ms (589 K)	1.64 ms (2764 K)	1.32 ms (688K)	0.24×
57 MB	1.46 ms (688 K)	2.05 ms (2876 K)	1.47 ms (694K)	0.39×
92 MB	1.63 ms (705 K)	2.46 ms (2985 K)	1.67 ms (706K)	0.47×
Heavy workloads				
572 MB	1.85 ms (857 K)	4.93 ms (3694 K)	1.97 ms (857K)	1.50×
868 MB	1.94 ms (950 K)	8.75 ms (3952 K)	2.17 ms (950K)	3.03×
1047 MB	2.25 ms (1011 K)	12.00 ms (4176 K)	2.34 ms (1013K)	4.14×
Eight clients (HT on)				
Light workloads				
32 MB	2.08 ms (701 K)	2.73 ms (5623 K)	2.15 ms (702K)	0.27×
57 MB	2.21 ms (705 K)	3.36 ms (5703 K)	2.30 ms (710K)	0.46×
92 MB	2.24 ms (714 K)	4.20 ms (5776 K)	2.40 ms (716K)	0.75×
Heavy workloads				
572 MB	2.95 ms (870 K)	8.10 ms (6991 K)	3.11 ms (872K)	1.61×
868 MB	3.19 ms (968 K)	13.19 ms (8453 K)	3.48 ms (979K)	2.79×
1047 MB	3.28 ms (1026 K)	22.57 ms (10738 K)	3.62 ms (1032K)	5.24×

closely resembles DrugBank—minor performance improvement on light workloads due to a minimal query working set and a significant improvement on heavy workloads. Based on the observed trend, we expect CHANCEL's performance to improve over CHANCEL-MP as heavier workloads are employed or more clients are serviced. Hence, these results further emphasize our previous finding that minimal query working set applications, like Snort, greatly benefit from CHANCEL, especially under heavy workloads and many clients.

Key takeaways. In realistic scenarios and with a modest number of clients $(4 - 8)$, CHANCEL outperforms CHANCEL-MP by $0.02 - 21.18\times$ while only incurring a performance

overhead of $0.2 - 13.1\%$ over NATIVE. Importantly, we observe that a query working set and the number of clients factor considerably in CHANCEL's performance improvement over CHANCEL-MP. We summarize our findings below.

- Programs with a robust query working set (e.g., OSSEC and Recommender) show substantial performance improvements for CHANCEL over CHANCEL-MP (up to 21.18×) with a light workload due to fewer page faults. However, such programs under heavy workloads incur high slowdown even for NATIVE due to many page faults. Hence, CHANCEL yields lesser yet significant improvements under heavy workloads (up to 4.02×).

- Programs with a minimal query working set (e.g., DrugBank) show modest performance improvements for CHANCEL over CHANCEL-MP on light workloads. However, the minimal query working set ensures few page faults even under heavy workloads. Hence, CHANCEL exhibits huge improvement over CHANCEL-MP under heavy workloads and a minimal query working set (up to 10.01×).

- Regardless of the type of query working set and workload, servicing more clients increases CHANCEL's performance improvement over CHANCEL-MP, because the latter adds enclave processes that incur more page faults.

4.6 Discussion

This section describes how CHANCEL compares to related work and how it can be extended to support multi-hop cloud programs and provide further covert channel protection.

4.6.1 Comparison with Other Enclave SFI Schemes

CHANCEL (§4) leverages Multi-Client SFI to prevent adversarial enclave code from collecting user data. This section provides a comparison between CHANCEL and other systems that provide security enforcement through SFI.

Ryoan [125] is the closest system to CHANCEL. It also protects user data from adversarial code inside the enclave using the Native Client SFI [133]. However, Ryoan does not consider multi-client scenarios in a *single* enclave. In particular, Ryoan lacks thread isolation, requiring

Table 4.10. A comparison between CHANCEL and related schemes that implement SFI in enclaves. For Occlum [74] and MPTEE [145], both SGX and MPX are required hardware features.

System	Scope		Multi-client		Requirements
	Adversarial program	Unintended bugs	Thread isolation	Shared mem. and enforce.	
Ryoan [125]	✓	✓	✗	✗	SGX2
Occlum [74]	✗	✓	✓	✗	SGX1/SGX2 + MPX
MPTEE [145]	✗	✓	✗	✓	SGX1/SGX2 + MPX
Chancel [our],[35]	✓	✓	✓	✓	SGX1/SGX2

multiple clients to be isolated in different enclave processes. Multiple enclave processes are inefficient—they significantly increase memory consumption (due to redundant copies of common data) and reduce performance (§4.5.1). Moreover, Ryoan requires advanced memory protection features—dynamic page permissions—which are only implemented in a few SGX CPUs that support SGXv2. This makes it challenging to deploy Ryoan. In contrast, CHANCEL implements thread isolation to efficiently handle multiple clients in a single enclave and it can be deployed without advanced hardware features.

Occlum [74] and MPTEE [145] are also systems that implement SFI inside SGX enclaves. Unlike CHANCEL, these systems does not consider adversarial code in the program, instead they only protect data from unintended bugs introduced by benign program developers. Hence, adversarial code can divulge sensitive client data through direct disclosure (e.g., transmitting information outside the enclave) or covert channels. Moreover, neither of these systems efficiently support multiple clients and their SFI implementations depends on a deprecated hardware feature, Intel Memory Protection eXtensions (MPX) [156], [157].

VC3 [75] aims to securely process data under the Hadoop [158] framework. It offers a compiler invariant of SFI to prevent data leakage through unsafe memory accesses but only addresses benign mistakes rather than intentional memory leakage since it does not consider adversarial programs. Also, unlike CHANCEL, it is not designed for multi-client scenarios in a single enclave, making it inefficient like Ryoan and other systems. Furthermore, Rohit et al. [159] introduce a runtime library that offers an interface to communicate outside the

enclave securely. They provide a framework to automatically verify applications and ensure that they meet confidentiality guarantees. However, their model does not cater to covert channels or consider multi-client scenarios.

SGX-SHIELD [34] also implements a custom SFI implementation to enable in-enclave Address Space Layout Randomization (ASLR). However, SGX-SHIELD is concerned with a different threat model (i.e., memory vulnerabilities). Therefore, it does not prevent data leakage from adversarial programs.

Beyond enclaves, SFI has been used in many different contexts. The idea itself was introduced by Wahbe et al. [123] as a novel way to isolate faults in a software-based manner. XFI, BGI, and LXFI [160]–[162] design further SFI mechanisms to isolate Windows kernel modules. Among non-enclave SFI research, the design of CHANCEL is inspired by Native Client (NaCl) [133], [163]. However, CHANCEL identifies and overcomes various challenges to propose an SFI scheme that is both SGX-compatible and supports multi-client isolation in a single principle, without relying on strict hardware requirements (e.g., Intel MPX).

4.6.2 Supporting Multi-Hop Adversarial Programs

The current implementation of CHANCEL assumes a self-contained program that only communicates with the client (except during setup and update steps as mentioned in §4.2.2). However, many cloud-based programs contain multiple process hops, where results from one application process is passed to another process before returning the result to a client. For instance, imagine a web server that is designed to obtain additional pages from a separate database. Transforming such programs to a self-contained version could be non-trivial. In such cases, CHANCEL can support such programs by having each application process sandboxed in CHANCEL enclaves and implementing information-flow tracking labels between sandboxes [125]. Such an extension also requires carefully mitigating covert channels through information passed between sandboxes (e.g., normalize size and timing of messages).

4.6.3 Strengthening Protection against Covert Channels

Chancel addresses many high-bandwidth channels (e.g., size and intervals of encrypted messages). Nevertheless, covert channels remain a long-standing problem for all computer systems and merit additional investigation for Chancel. In particular, it would be very useful if a future system could monitor different channels (e.g., micro-architectural components, system calls, encrypted messages, etc.) and rank the amount of data leak through them. This could help Chancel focus on the most critical covert channels.

4.7 Summary

Chancel ensures protects user data from adversarial services using Multi-Client SFI (MCSFI). Chancel supports thread isolation and shared memory enforcement, thereby ensuring secure servicing of multiple clients in different threads of an enclave while permitting the secure sharing of non-sensitive data. Our evaluation showed that Chancel outperforms a multi-process sandbox by $4.06 - 53.70\times$ while providing strong security guarantees.

5. SOFTWARE OBFUSCATION TO DEFEAT MEMORY SIDE-CHANNELS

Cryptographic program obfuscation [164], [165] is a popular construct with important cloud applications towards protecting sensitive user data, as well as the intellectual property of software owners. Under program obfuscation, a sender, who owns a program, transforms it to create an obfuscated version of the program which is: (a) functionally identical to the original version, and (b) runs for a fixed time before returning an output. The sender then sends this obfuscated program to a receiver. The receiver runs the obfuscated program within a black box-like environment — the receiver cannot see (or infer) intermediate computational results and/or footprints from the obfuscated program. Consequently, even though the receiver can run the obfuscated program using any input of their choice, they will *learn nothing about the program or be able to distinguish the program from another that is similarly obfuscated.* Therefore, as far as the attacker is concerned, they are interacting with a virtual black box, which takes an input and gives the intended output.

In the past, there has been significant theoretical research [166]–[169] in achieving program obfuscation, but with crippling performance overheads. Recently, there has been a systematic breakthrough, HOP [100], in achieving program obfuscation through relaxed assumptions of trust on the underlying hardware. However, HOP relies on special-purpose hardware, severely limiting its deployment potential.

We asked ourselves the question: *can we leverage trusted execution environments (TEEs) like Intel SGX to achieve the strong guarantees of program obfuscation on commodity hardware?* As it turns out, it is non-trivial to support program obfuscation using Intel SGX. In particular, SGX suffers from critical memory side-channels (§2.5) that break a key assumption in program obfuscation—the program should be run in a black-box-like environment—allowing attackers to observe both code execution and data access patterns within the SGX enclave. Moreoever, enclave environments do not have access to a secure clock (the CPU timer information is controlled by the operating system), making it challenging for a computation to execute for a fixed amount of time, another key requirement of program obfuscation schemes.

This chapter addresses the aforementioned challenges to take a significant stride towards commodity program obfuscation. Initially, this chapter outlines a program obfuscation approach that can be readily-adopted for legacy programs written for SGX environments (§5.1). Using this approach, we designed OBFUSCURO (§5.2). Our thorough security analysis (§5.4) of OBFUSCURO shows that it prevents information leak through both access pattern-based and timing-based side-channels—achieving program obfuscation. On the performance side, OBFUSCURO incurs an average overhead of 51× over native SGX execution for our custom benchmarks (§5.5), several orders of magnitude faster than existing cryptographic research that achieves program obfuscation on commodity hardware (§5.6.1).

5.1 Approaching Obfuscation using Scratchpads and Instrumentation

SGX provides a *partial* black-box environment for program obfuscation—the attacker cannot directly access contents of programs running inside enclaves (§2.4.7). However, SGX suffers from memory side-channel limitations (§2.5) and lacks access to a trusted timer source. Therefore, any system attempting to realize program obfuscation using SGX must answer the following three questions — (a) how to execute a target program's code inside an enclave without leaking memory access patterns?, (b) how to provide secure access to data regions (e.g., stack, heap etc.) without leaking memory access patterns?, and (c) how to ensure that the program leaks no information through its execution time?

The answer to (a) and (b) lies in the design of fixed scratchpad regions for code execution and data access—each code block in the program is executed from a code scratchpad, while data block is accessed from a data scratchpad. The scratchpad regions are both a single cache-line (i.e., 64 B), which is the smallest observable granularity of non-branching side-channels (e.g., using caches, page tables, etc.). To secure the code scratchpad against branching side-channels (e.g., Branch Shadowing [15]), OBFUSCURO ensures that all branches to/from the scratchpad are at fixed locations. All these scratchpad protections nullify memory side-channels because from a memory side-channel attacker's perspective, *each code execution and data access from a scratchpad region looks indistinguishable.*

The scratchpad design raises two important questions — (i) how to support code execution at the granularity of cache-line and normalized branches within the scratchpad?, and (ii) how to securely fetch code and data blocks onto the scratchpads without leaking information through side-channels? Scratchpad code execution is supported by instrumenting (e.g., using a compiler) the target program's code into 64 B basic blocks with branch instructions at fixed offsets within each basic block. To securely fetch code and data blocks onto the scratchpad regions without leaking information through side-channels, a system should utilize Oblivious RAM (ORAM) (§2.6) to hide access patterns. As shown by several previous work [20], [38], [85], the ORAM controller must be further provisioned to avoid leaking information through side-channels (more details in §5.2.1).

Finally, to counter the threat of timing channels and consequently answer (c), the execution time of the target programs is fixed by extending the program's execution using dummy (but indistinguishable) code blocks. The enclave stops only after the a fixed number N of code blocks have been executed. As we show in §5.4.2, each code block execution takes the same time, resulting in execution-time-normalization for the program.

5.2 Obfuscuro Design

Leveraging the approach outlined in §5.1, we designed Obfuscuro, the first system to achieve program obfuscation on commodity SGX hardware. The core design features of Obfuscuro can be summarized as follows.

- **Secure ORAM scheme**. Obfuscuro implements its ORAM controller using data oblivious algorithms, protecting itself from side-channel attacks (§5.2.1). Also, Obfuscuro implements a *register-based* stash which improves on the existing side-channel resilient ORAM implementations [20], [85].

- **Repurposing native programs**. Obfuscuro transforms native programs (§5.2.2) through memory layout transformation and virtual address translation in order to bridge the semantic gap between native program execution and ORAM-based operations.

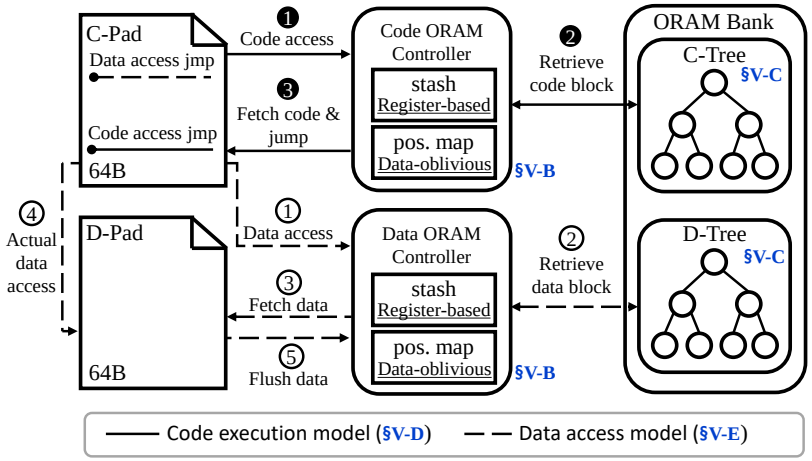


Figure 5.1. OBFUSCURO's system-level overview.

- **Code execution model**. OBFUSCURO ensures that the code execution (of a target program) is exclusively performed within a fixed location, C-Pad (§5.2.3). All instructions are loaded onto the scratchpad using ORAM operations and executed from the start to the end of the scratchpad (❶∼ ❸).

- **Data access model**. OBFUSCURO ensures that all data access is performed at a data scratchpad, D-Pad, a fixed memory location updated using ORAM operations (§5.2.4). The target program's read and write operations are performed at the same memory location regardless of execution context (①∼⑤). OBFUSCURO also ensures that the data access is always performed once per C-Pad, normalizing the number of data accesses patterns.

- **Start-to-end obfuscation**. OBFUSCURO ensures that the target program continues executing till a predefined time to mitigate timing-based channels (§5.2.5). OBFUSCURO achieves this by instrumenting the target application to introduce dummy memory blocks, after the termination of the intended logic.

Workflow. The input to OBFUSCURO is the source code of a target enclave application. Using the input, OBFUSCURO produces an instrumented executable, fully loaded with a runtime library (containing the ORAM controller). During initialization, the runtime library *populates* the code and data blocks into different ORAM trees. Afterwards, the ORAM controller extracts the first code block to be executed, loads it onto the code scratchpad,

and ensures execution starts from the beginning of code scratchpad. When the code block performs a branch instruction, the branch instruction is replaced with new jump instruction to the ORAM controller for codes. Then, the ORAM controller loads the required code block onto the code scratchpad using ORAM operations, and jumps back to the beginning of the code scratchpad. While accessing data (i.e., global/heap/stack objects), the access instruction is replaced with new jump to the ORAM controller for data. The ORAM controller for data always loads the corresponding data block onto the data scratchpad using ORAM operations, and returns the appropriate address (i.e., base address of D-Pad + access offset). Finally, Obfuscuro ensures that the program keeps executing till a certain time period has elapsed before returning an output to the user thereby ensuring complete start-to-end obfuscation.

5.2.1 Secure ORAM Scheme

This section explains how Obfuscuro designs a secure ORAM scheme to ensure oblivious program execution. First, Obfuscuro places both the ORAM controller and trees within an SGX enclave. Second, in response to side-channel threats against SGX enclaves, Obfuscuro secures working mechanisms of its ORAM controller, i.e., ensuring that each operation is branch-free (to mitigate the risk of branch-prediction) and data-independent (to mitigate the risk of page table and cache attacks). In this regard, Obfuscuro constructs two stash designs: `CMOV`-based and register-based stash for the ORAM controller. Furthermore, Obfuscuro employs a data-oblivious population scheme to securely populate the ORAM trees.

ORAM controller. Obfuscuro secures the two main data structures of the ORAM controller, i.e., position map and stash, against access-pattern leakage. By securing access onto these data structures, Obfuscuro also ensures that its code is devoid of conditional branches (i.e., secure against branch-prediction attacks).

Oblivious position map. The position map contains sensitive information regarding ORAM blocks, i.e., mapping from block-id to the leaf in ORAM tree. An attacker can leak sensitive information about program execution by observing the access patterns onto the position map. Obfuscuro employs data oblivious access mechanism to prevent information leakage from the position map. The key security primitive of this mechanism is in leveraging

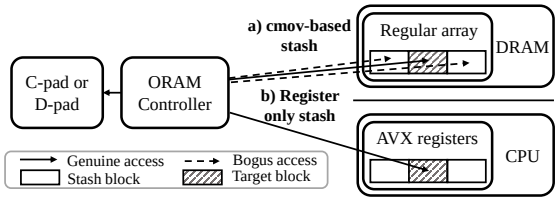


Figure 5.2. Register-based stash versus `CMOV`-based stash. `CMOV`-based stash has to access an entire array placed in DRAM whereas register-based stash can directly retrieve an item from CPU's AVX registers.

`cmov` instruction in x86 to stream through the entire data structures. Similar to Raccoon [38], we devise a wrapper function for the `cmov` instruction to add additional bogus memory access. Depending on the flag value provided to the wrapper function of the `cmov` instruction, the function performs either the actual memory write (if the flag is true) or a bogus memory access without writing (if the flag is false).

Next, we describe how OBFUSCURO secures access onto the stash. Naively accessing the stash would leave memory traces that can be used to distinguish between real and dummy blocks in the extracted ORAM tree path. OBFUSCURO can utilize two different stash designs, `CMOV`-based stash and a novel *register-based* stash. While both completely secure stash accesses, it imposes different performance characteristics depending on the underlying hardware architecture.

CMOV-based stash. OBFUSCURO can use data-oblivious access (using `CMOV`) to stream through the complete stash memory region (Figure 5.2-a), similar to previous schemes [20], [85]. As a result, the `CMOV`-supported access guarantees that the attacker learns nothing from the leaked access patterns as the attacker observes accesses onto all stash indices. One caveat of this approach is that the stash is a large memory region, i.e., $>= Blog_2N$ bytes; where B is the block-size in bytes and log_2N is the size of the ORAM tree containing N nodes. Therefore, using `CMOV` within the stash can result in performance overhead as noted by previous works and reported in §5.5. Figure 5.3a shows a code snippet illustrating how the `CMOV`-based stash functions.

Register-based stash. OBFUSCURO also designs a novel register-based stash, which leverages Advanced Vector Extensions (AVX) instruction set along with the XMM and

```
void retrieve_from_stash_cmov(void* cpad, int required_blk) {
  bool flag = false;

  for (int i = 0; i < NUM_STASH_BLOCKS; i++) {
    // Check the validity of the condition, i.e.,
    // is this the block to retrieve from the stash
    flag = ((stash[i].blocknum == required_blk));

    // Based on the flag, either perform a real or a dummy copy
    x86_cmov(cpad, stash[i].memblk, flag);
  }
}
```

(a) CMOV-based stash

```
; %rsi points to the base address of ORAM tree block.
movaps (%rsi), %xmm0
vinserti128 $0x0, %xmm0, %ymm5, %ymm5
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x1, %xmm0, %ymm5, %ymm5
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x0, %xmm0, %ymm6, %ymm6
add $16, %rsi
movaps (%rsi), %xmm0
vinserti128 $0x1, %xmm0, %ymm6, %ymm6
```

(b) Register-based stash

Figure 5.3. Implementation snippets of OBFUSCURO's stash access: (a) OBFUSCURO obliviously retrieves a block from the stash using `CMOV`; and (b) OBFUSCURO leverages YMM registers to obliviously access stash indices. As can be observed, there are no conditional branches and/or data-dependent access in both cases.

YMM registers. We collectively refer to these registers as AVX registers. The key idea is to reserve these registers for ORAM stash only and restrict the program and associated libraries from using them. An operation performed on any CPU register does not imprint traces on memory-related units (cache, TLB/MMU, DRAM etc.) and is therefore oblivious to even privileged attackers such as the OS (Figure 5.2-b). Therefore, OBFUSCURO copies each tree block onto a set of AVX registers and performs all required operations on these registers. This limits the involvement of `CMOV` and therefore provides a performance improvement of $30-40\%$ as compared to the `CMOV`-based stash as shown in §5.5. Figure 5.3b shows an example of where the memory located at `rsi` is moved in chunks of 32-bytes into `ymm5` and `ymm6`.

However, there are two things to consider while opting for the register-based stash over the `CMOV`-based stash. Firstly, the register-based stash limits the involvement of AVX registers for other important operations such as AES-NI instruction set and if the enclave program requires these operations, it would be better suited to use the `CMOV`-based stash. Secondly, current desktop hardware only supports AVX2 [170] which provides 16 YMM registers of 32 B memory each, totaling to 512 B of memory for the stash. This size is enough for small ORAM tree size (e.g., 4-8KB) but is insufficient for larger tree sizes. However, the AVX-512 [171] instruction set architecture introduces larger AVX registers (ZMM registers), currently present on high-end hardware [172], [173]. The ZMM registers are 32 registers in total, with each being 512-bit wide and can support a total stash size of 2-kilobytes which increases our tree size that can be supported from 8KB to 256MB.

Workflow. We now illustrate how Obfuscuro performs a secure ORAM access. First, Obfuscuro uses `CMOV` to scan through the whole position map to find the required ORAM block. Then, Obfuscuro sequentially copies the tree blocks to either memory (if `CMOV`-based stash is used) or the registers (if the register-based stash is used). Afterwards, Obfuscuro performs an oblivious retrieval of the required block from the stash. In the case of `CMOV`-based stash, it performs a sequential `CMOV` access on each individual stash index and in the case of register-based stash, it performs an inline assembly move operation to move it from the register to the memory. After performing the relevant tasks on the ORAM block, we rewrite the block back using similar approach as mentioned above.

ORAM bank. Obfuscuro places the ORAM bank, comprising of the ORAM trees, within the enclave memory. Obfuscuro performs secure ORAM tree population to mitigate side-channel leakage.

Allocation. The ORAM trees are allocated as global arrays within the enclave program's memory space (i.e., within the EPC). Obfuscuro can avoid encrypting ORAM trees, which is an important step in the ORAM protocol, because the *Memory Encryption Engine* (MEE) in SGX [174] implicitly performs the encryption. There are two things to note here: (a) the allocation step does not leak any important information to the attacker apart from the location of the ORAM tree (which is public information in the ORAM attack model) and (b)

the size of the code and data trees should be carefully considered prior to allocation since as per Path ORAM's design, the size of the trees cannot be dynamically adjusted.

Population. As per Path ORAM's requirement, the population of each block into the ORAM tree should be performed as a regular ORAM access. To further illustrate, the population of code and data blocks in C-Tree and D-Tree respectively, is carried out as follows: (a) Obfuscuro picks a block which is to be added to the ORAM tree. (b) Obfuscuro determines a random position to store the block within the ORAM tree. The random position is determined using the `RDRAND` hardware instruction, which only involves the trusted CPU. (c) Obfuscuro performs an ORAM access onto the path that corresponds to the selected position. At first glance, this might leak some information to the attacker. However, since this is an ORAM access, the final destination of the block will be randomized within the path once more which ensures strong secrecy. (d) Obfuscuro repeats the above steps until all real blocks are populated to the ORAM tree.

5.2.2 Repurposing Native Programs

In order to bridge the semantic gap between native and oblivious execution, Obfuscuro transforms the target program's memory layout into an ORAM-compatible memory layout, provides virtual address translation to support dynamic memory relocation, and introduces scratchpad regions for code execution and data access.

Memory layout transformation. Obfuscuro separates the target program into two sections, i.e., code and data, and allocates a dedicated ORAM tree for each section, namely C-Tree for code and D-Tree for data. Obfuscuro can estimate the size of the C-Tree since the program's code size remains static. Since the size of dynamically allocated data (e.g., heap and stack) cannot be precisely estimated, Obfuscuro sets a maximum limit on the size of the D-Tree. This is not a limitation since SGX programs themselves are initialized with a user-provided stack and heap size. Code blocks are prepared during the compilation phase, where the code is divided into blocks of the same size and filled with instrumented instructions by Obfuscuro (more details in §5.2.3). During program initialization, Obfuscuro populates both the code blocks and data blocks into the C-Tree and D-Tree respectively. The initialized

data objects (i.e., global variables) are filled in their corresponding blocks whereas the blocks corresponding to uninitialized data blocks are zero-initialized.

Virtual address translation. All memory accesses in a traditional program are realized through virtual addresses, while ORAM operations deal in blocks of the ORAM tree. To reconcile this, OBFUSCURO performs *on-the-fly* translation of virtual addresses into ORAM block indices. OBFUSCURO linearly maps the virtual address space of a program into ORAM blocks and performs bitwise right-shift to secure translation.

Heap management. Since SGX enclaves do not have support for dynamic memory allocation, the maximum heap size required for the application has to be decided at compilation time. OBFUSCURO provides a wrapper for the `malloc` and `free` function calls, i.e., `malloc_ob` and `free_ob`, which are responsible for managing the heap memory (alongside the metadata) requested by the enclave program. In particular, `malloc_ob` *obliviously* picks a block from the D-Tree which is already provisioned with blocks to handle heap memory requests during program initialization. The wrapper function returns the virtual address corresponding to the selected block. Later, when `free_ob` is called, it deallocates the heap memory region, figures out which blocks from the D-Tree are now free and tags them as such.

Scratchpad. In traditional ORAM, the program can simply access the extracted block from the stash. However, doing so within the SGX environment will leak a considerable amount of information. To deal with this problem, OBFUSCURO prepares two fixed locations (determined during program initialization) of fixed size (one cache line, i.e., 64 B) to access code and data blocks, called C-Pad and D-Pad respectively. These memory regions are provisioned with SGX-specific defenses (refer to §5.2.3 and §5.2.4). After OBFUSCURO performs oblivious operation and locates a target block in stash, OBFUSCURO copies the target block in stash to scratchpad. Note that this copy from stash is oblivious (as described in §5.2.1). Therefore, by normalizing access location and size through scratchpads, OBFUSCURO can successfully hide actual memory location and the attacker can not infer that information. We provide more details as to how this is accomplished in the next two sections.

5.2.3 Code Execution Model

OBFUSCURO ensures the following three security properties in its code execution model: C1) Code execution is always performed within the C-Pad[1]; C2) Code access instructions (i.e., branch instructions which impact the control-flow of a program, including call, return, unconditional branch, and conditional branch instructions) are only executed at a fixed location (i.e., the end of the C-Pad); C3) All code access instructions are replaced with an instruction jumping to a runtime function (i.e., `code_oram_controller`), which performs an ORAM operation to fetch the code block required.

The above mentioned security properties of OBFUSCURO protect code execution from access-based side-channel attacks. Since the size of the C-Pad is the same as the minimum granularity of page table and cache-based attacks (i.e., 64 B), C1 prevents these attacks from gaining any meaningful information. C2 and C3 prevent a branch prediction attack, because all the control-flow changes are made from the same location (i.e., the end of C-Pad as specified by C2) to the same destination (i.e., `code_oram_controller` as specified by C3), irrespective of the semantics of the original branch instruction.

To meet the property C1, OBFUSCURO restricts all basic blocks to be at the size of C-Pad (i.e., 64 B) during the compilation phase. Specifically, OBFUSCURO breaks up larger basic blocks into smaller ones equaling the size of the C-Pad. If the size of the basic block is smaller than the C-Pad, OBFUSCURO inserts `nop` instructions to fill the space. To meet the properties C2 and C3, OBFUSCURO replaces all branch instructions with a sequence of equivalent instructions invoking `code_oram_controller`. This invocation is always performed using `jmp` instruction to `code_oram_controller`, which is aligned at the end of the basic block.

For example, Figure 5.4a shows how OBFUSCURO replaces a unconditional branch instruction. Given the original `jmp` instruction, OBFUSCURO first instruments an instruction storing the virtual address of the jump target in `R15`. Then, OBFUSCURO inserts a `jmp` instruction to the `code_oram_controller`. The code ORAM controller computes the ORAM block index using the virtual address stored in `R15` (as mentioned in §5.2.2), and retrieves the

[1] ↑The C-Pad is a writable and executable region but it can be secured against attacks by employing SFI similar to SGX-Shield [34] and/or dynamic page protection available in SGXv2.

```
; Before
  jmp jump_target

; After
  mov R15, jump_target        ; Pass jump_target through R15
  jmp code_oram_controller    ; code_oram_controller loads the code
                              ; block to C-Pad and then jumps to the
                              ; beginning of C-Pad.
```

(a) Unconditional branch (code access)

```
; Before
  mov 4(RAX), RBX           ; Store RBX at where (RAX + 4) points to

; After
  lea R15, 4(RAX)           ; Pass the store address through R15
  mov R14, after_fetch      ; Pass the return address through R14
  jmp data_oram_controller  ; data_oram_controller fetches data block
                            ; and returns address of (D-Pad + offset)
                            ; through R15
after_fetch:
  mov (R15), RBX            ; Write a value RBX to (D-Pad + offset)
```

(b) Store (data access)

Figure 5.4. Instrumentation on code and data access.

required code block from the C-Tree through an ORAM access. Afterwards, Obfuscuro overwrites C-Pad using the obtained code block and resumes execution from the beginning of C-Pad. In this manner, Obfuscuro translates all types of control flow instructions, including conditional jump, function call, return.

5.2.4 Data Access Model

Obfuscuro ensures the following security properties in the data access model: D1) Data access is always performed within the D-Pad of size 64 B; D2) Data access instructions are only executed once per C-Pad at a fixed location (i.e., the beginning of the C-Pad); and D3) All data access instructions are replaced with an instruction jumping to a runtime function, `data_oram_controller`, which performs an ORAM operation to load the corresponding data block onto the D-Pad. Similar to the code execution model (§5.2.3), these properties prevent cache and page table attacks. This is because attackers will always observe the same data access patterns onto D-Pad.

One thing to note here is that D2 enforces each code block to perform a single jump to the `data_oram_controller`. This restriction is partly due to the constraint of the 64-byte code

block. In particular, OBFUSCURO's data access instructions take 28-bytes and the code access instructions (mentioned in §5.2.3) take 20-bytes. Since a code block requires at least one code access instruction, i.e., to access the next code block, it leaves room for only a single data access. However, as a result of this, OBFUSCURO ensures that there is a normalized number of data access per code block, which cannot be exploited by an attacker. OBFUSCURO also prevents branch-prediction attacks by placing the data access instruction at a fixed location. If a certain code block does not require a data access, OBFUSCURO performs a dummy data access in order to portray the same memory footprints for each block.

Unlike the code execution model, the data access model allows offset-based access within the D-Pad such that a memory access can be directly performed at any location within D-Pad. This offset-based access is secure against memory-based side-channel attacks since the D-Pad is the size of the minimum granularity of attack resolution, i.e., 64 B. In order to reflect changes made by the enclave code on the D-Pad back to the ORAM tree, OBFUSCURO flushes the extracted data block after performing required memory access.

For example, Figure 5.4b illustrates how OBFUSCURO instruments the store instruction. Similar to the code execution model, OBFUSCURO uses the reserved `R15` register to pass the virtual address (i.e., the memory operand of a store instruction) to the `data_oram_controller`. Then the `data_oram_controller` translates the virtual address into the corresponding ORAM block index, and updates D-Pad after extracting the data block using an ORAM access. Afterwards, the `data_oram_controller` returns the virtual address through `R15`, which points within D-Pad (i.e., `p1` + `p2`, where `p1` is the base address of D-Pad and `p2` is the offset within the D-Pad). Therefore, the enclave program correctly performs the store instruction using `R15`, and the data block is later flushed back into the D-Tree.

5.2.5 Start-to-End Obfuscation

In the previous subsections, we explain how OBFUSCURO ensures that the target program's code blocks perform a normalized sequence of operations, irrespective of their original logic. However, that is not enough for complete obfuscation. In particular, there is one further distinguishing factor in the program, i.e., execution time of the program. For example,

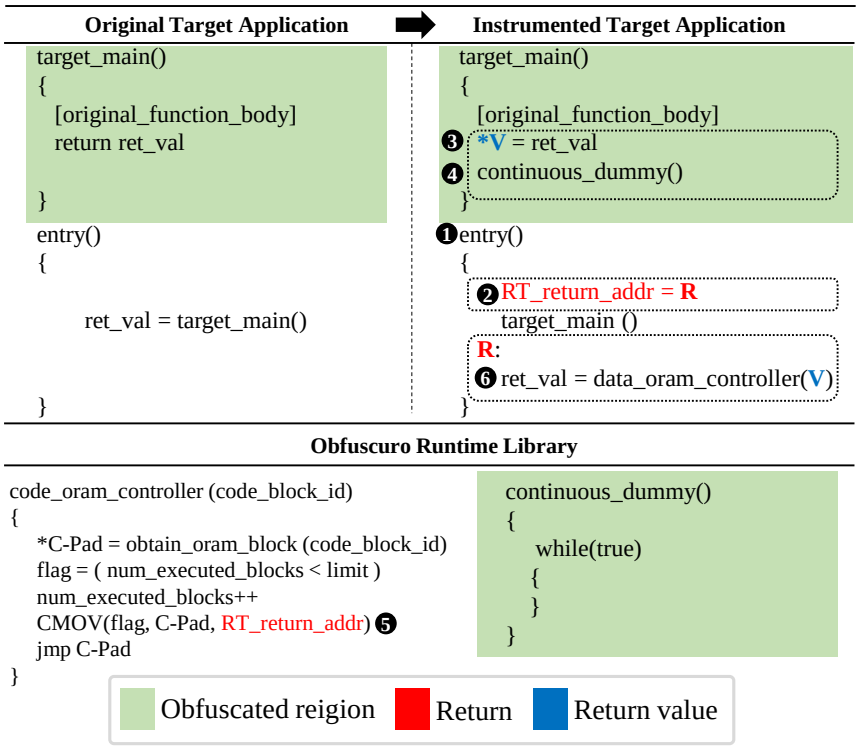


Figure 5.5. OBFUSCURO's continuous execution.

running different programs or just running the same programs with different inputs can result in drastically different execution times, which can be abused by an attacker.

OBFUSCURO handles both of these cases to ensure that, irrespective of program logic, the obfuscated execution always terminates after a fixed amount of time. In order to fix the execution time, OBFUSCURO inserts dummy code blocks within a native program's code ensuring that the program keeps executing even after completing the intended program logic. OBFUSCURO instruments the target application as shown in Figure 5.5. As shown in the figure, OBFUSCURO injects a dummy function called `continuous_dummy` into the program. The dummy function is meant to execute a `while` loop indefinitely, ensuring that program will not terminate of its own will. As mentioned in §5.2.3, each code access will go through the `code_oram_controller`. Therefore, OBFUSCURO can stop the program execution after a certain predefined number of code blocks, even if the dummy function never stops executing. However, to do so and provide the required output back, OBFUSCURO needs an address to jump to after reaching the limit on code blocks.

Now, we explain the workflow of the instrumented target program. The application code is defined as `target_main` whereas the enclave officially starts execution from the `entry` function (❶). At the start of the `entry`, OBFUSCURO ensures that the return address `R` is passed to the runtime library by writing `RT_return_addr` (❷). Afterwards, OBFUSCURO starts running the `target_main` function and writes its output to a global memory within the program (❸). It is worth noting that this write will also be achieved through an ORAM access (as per all data access mentioned in §5.2.4) and is therefore oblivious to the attacker. Then, OBFUSCURO invokes `continuous_dummy` (❹), ensuring that the program continues executing.

As the program executes, it will jump to the `code_oram_controller` on each code access. At this time, OBFUSCURO checks that the predefined limit on the number of code blocks has been reached or not. If the limit has been reached, the program jumps back to `RT_return_addr` instead of jumping to the C-Pad (❺). At this point, we completed the execution of original program logic but have not obtained the output. To get the output, OBFUSCURO calls the `data_oram_controller` to extract the output from the D-Tree (❻). Through the above mentioned steps, OBFUSCURO ensures that there is a *start-to-end* obfuscation of the target program, which always executes the same number of code blocks and thus terminates after a fixed amount of time.

5.3 Implementation

We prototyped OBFUSCURO using the LLVM compiler suite. We modified the LLVM backend to emit 64B of code blocks as well as to instrument code and data access instruction. Additionally, we implemented a compiler runtime library for ORAM controllers. In the LLVM backend, especially the assembly emitter, we arranged a new code emitter to measure the size of instructions in parallel with default emitter. We also utilized built-in machine code builder to redirect the codes and data accesses to the runtime ORAM controllers. The compiler runtime library includes the implementation of data-oblivious ORAM, and interfaces for LLVM backend and applications to employ it. The oblivious stash access is implemented with `vinserti128`, and `vextracti128` AVX register manipulating instructions in the assembly language level. The oblivious position map access is based on the `CMOV` instruction, and we

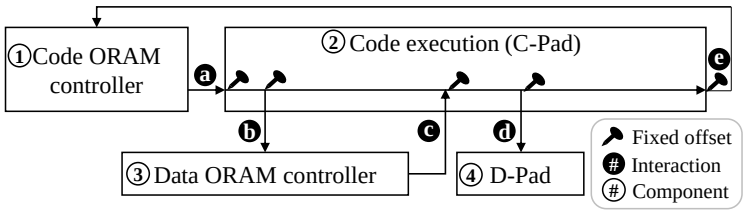


Figure 5.6. Data oblivious execution cycle of OBFUSCURO

Table 5.1. Security analysis of secure ORAM implementation used by the code and data controller.

ORAM operations	Sensitive info.	Obfuscuro defense	Observed traces by adversaries
1. Locating corresponding pos-map element	Pos-map offset	`CMOV`-scanning read	Sequential read traces on pos.map
2. Extracting requested ORAM path to stash	-	-	Sequential copy traces from requested ORAM path to stash
3-a. Copying ORAM block in stash to scratchpad (`CMOV`-based)	Stash offset	`CMOV`-scanning copy	Sequential copy traces from stash to scratchpad
3-b. Copying ORAM block in stash to scratchpad (Register-based)	Stash offset	Register operations	No traces since registers are oblivious to memory
4. Updating pos-map with new leaf number	Pos-map offset	`CMOV`-scanning write	Sequential write traces on pos.map
5-a. Writing back scratchpad to stash (`CMOV`-based)	Stash offset	`CMOV`-scanning write	Sequential write traces from scratchpad to stash
5-b. Writing back scratchpad to stash (Register-based)	Stash offset	Register operations	No traces since registers are oblivious to memory
6. Writing back stash to requested ORAM path	-	-	Sequential write traces from stash to requested ORAM path

generalized its operation to variable lengths. We also changed the enclave loader of the Intel SGX SDK to make C-Pad using SGX's `EADD` instruction.

In total, OBFUSCURO introduces 3,117 LoC in LLVM backend, 2,179 LoC in compiler runtime library, and 25 LoC in Intel SGX SDK. Its implementation is available at https://github.com/adilahmad17/Obfuscuro.

5.4 Security Analysis

This subsection provides a security analysis of OBFUSCURO. In general, there are two ways an attacker can steal information from SGX enclaves using side-channels. Firstly, an attacker can abuse observed access-patterns to infer some information about the program and/or its input. Secondly, an attacker can perform timing-based attacks to leak some

information. We provide a systematic security analysis of OBFUSCURO against both of these attack avenues.

5.4.1 Access Pattern Attacks

As OBFUSCURO is composed of multiple components to realize obfuscated program execution, we start by showing the security properties of the individual components of OBFUSCURO. Then we show how these components interact with each other and show that these interactions are completely oblivious as well. Finally, we present the results of an empirical study showing that OBFUSCURO achieves access pattern obliviousness.

Obliviousness of individual components. OBFUSCURO introduces new components to legacy programs in order to achieve obfuscated execution, as shown in Figure 5.6. In the figure, we show the four components of OBFUSCURO (labeled as ① ∼ ④). We comment on each component individually in the following.

① ***Code ORAM controller*:** The code ORAM controller takes the virtual address of next required code block as input, and it places the corresponding code block on the C-Pad. An attacker cannot decipher the virtual address because OBFUSCURO performs secure computation based on this address. In particular, the address is first translated to a specific ORAM block using data oblivious right-shift operation (§5.2.2), which returns the corresponding block number in the ORAM tree. Then, OBFUSCURO finds the corresponding leaf for this block through sequential `CMOV`-based scanning of the position map.

For the stash, OBFUSCURO uses two variants, a `CMOV`-based and a register-based. The `CMOV`-based stash performs `CMOV`-based memory access similar to how OBFUSCURO shields the position map. This includes both (a) while copying the required block from the stash to the C-Pad or D-Pad and (b) while writing back the blocks from the C-Pad or D-Pad to the stash. For the register-based stash, the AVX registers are retrofitted as stash space. Since all operations on the AVX registers are oblivious to the underlying system, we can perform a direct memory access to/from a specific register while ensuring that no information is leaked. Please refer to Table 5.1 for detailed operations performed by the code controller.

② ***C-Pad***: OBFUSCURO ensures that the C-Pad has a fixed location (determined at the program loading) and a fixed size (i.e., 64B), and ensures that all oblivious code execution occurs from this location. Since 64B is the cache-line size (i.e., the finest visible granularity through access pattern-based side-channel attacks), the attacker learns no useful information to infer semantics during the C-Pad execution. In other words, as OBFUSCURO runs the target program, the attacker will keep observing the same memory activity over C-Pad, which is completely independent of the code block being executed.

③ ***Data ORAM controller*:** The data ORAM controller takes the virtual address of data objects as input, and places the corresponding data block to D-Pad. The data controller follows the exact same workflow of the code controller except that it operates on the D-Tree instead of the C-Tree. As previously shown for the code controller, the data controller also does not leak any sensitive information.

④ ***D-Pad***: The D-Pad is functionally and structurally similar to the C-Pad, except that data access is performed on it and not code execution. Similar to the C-Pad, it has a fixed location and the same size, thereby showing the same memory activity for each data access.

Oblivious interactions between components. The aforementioned components perform five interactions between them (labeled as ⓐ ∼ ⓔ). We illustrate below how each of these interactions is secure against access pattern-based attacks.

ⓐ ***Jump from Code ORAM controller to C-Pad after fetching code block*:** After obliviously extracting a block from the C-Tree and copying it to C-Pad, the code controller performs a single jump to the start of the C-Pad. This step only reveals that some code block of a target program will now be executed, which entails no semantics behind the code block being executed.

ⓑ ***Jump from C-Pad to Data ORAM controller for fetching data block*:** Each code block (executing within the C-Pad) is strictly enforced to perform a single jump to the data controller, because OBFUSCURO normalizes the number of data access within each code block to be exactly one (refer §5.2.4). Moreover, this jump is performed at a fixed offset within C-Pad to mitigate the risk of branch prediction attacks. The target address of this jump is also fixed, i.e., the start of the data controller's logic.

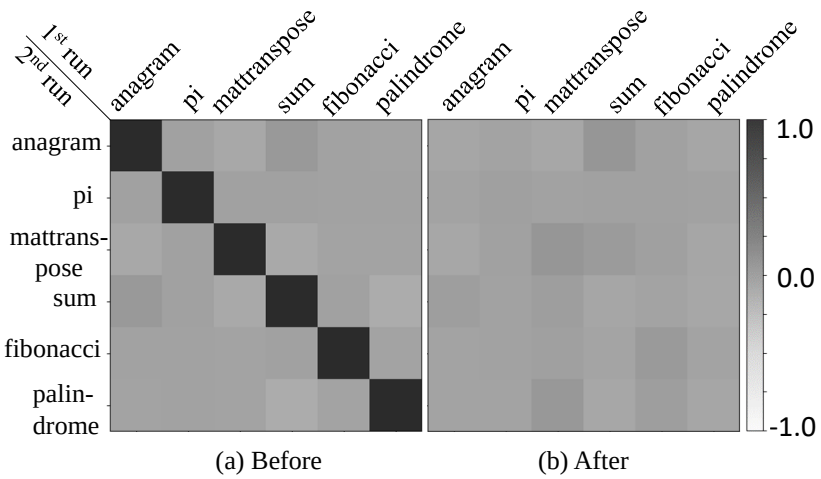


Figure 5.7. Confusion matrix for native access patterns vs. obfuscated patterns shown by OBFUSCURO.

ⓒ ***Return from Data ORAM controller to C-Pad*:** There is only a single jump from the data controller to the C-Pad at a fixed offset within the C-Pad, after fetching/updating the required data block on the D-Pad.

ⓓ ***Single D-Pad access*:** There is only a single access to the D-Pad per code block. Since the size of the D-Pad is 64B, this access does not reveal offset information either.

ⓔ ***Jump from C-Pad to Code ORAM controller*:** Finally, OBFUSCURO enforces that there is only one jump from C-Pad to the code controller at a fixed address. The target address of this jump is also fixed at the start of the code controller logic.

Empirical study. We also present the results of our empirical study on native and obfuscated memory traces exhibited by various applications (Figure 5.7). For this study, we chose six target applications: `anagram`, `pi`, `mattranspose`, `sum`, `fibonacci`, and `palindrome`. These applications were chosen due to the diversity of their computational complexity. We measured ten runs for each application. For each run, we collected a sequence of the addresses accessed by the application. Using this data, we calculated the Pearson correlation value between the test applications and populate a confusion matrix.

Figure 5.7-(a) shows the confusion matrix formed while comparing native execution of the applications. Consider the $(anagram, anagram)$ cell in Figure 5.7-(a), the Pearson correlation value is very close to 1 because this cell is comparing the memory traces between

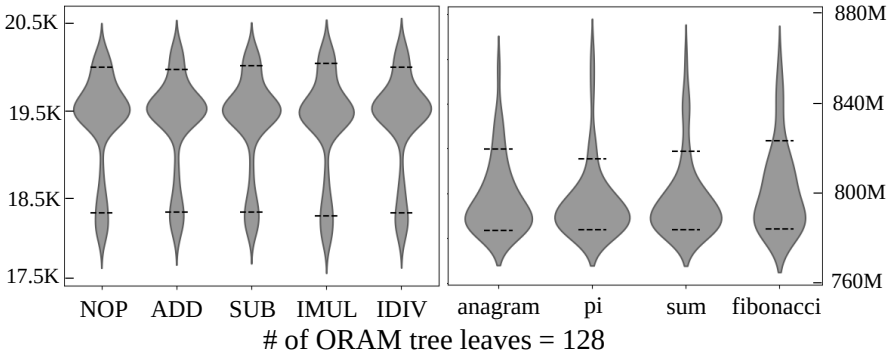


Figure 5.8. (a) Distributions of code execution cycles of different types of code blocks (y-axis) with 10%∼90% percentile intervals. (b) Distributions of total execution cycles of various test programs (y-axis) with 10%∼90% percentile intervals.

two native executions of the same program. On the other hand, the correlation value in the $(anagram, pi)$ cell is nearly 0 because their (native) access patterns are unique.

Figure 5.7-(b) shows the confusion matrix formed while comparing the execution of obfuscated application (using OBFUSCURO). In general, we observed no correlation between native and obfuscated memory traces of the same application nor any correlation between two executions of an obfuscated application.

5.4.2 Timing-based Attacks

Apart from access pattern attacks, a privileged attacker can also break program obfuscation within Intel SGX by abusing timing channels. In particular, we expect following two ways in which an attacker can abuse timing channels to leak information from OBFUSCURO—(a) observing the time it takes for individual code blocks (in C-Pad) to execute, and (b) observing the total time it takes for an obfuscated program to execute. We individually show the infeasibility of each of these timing channels.

C-Pad execution time. Timing differences in executing each code block (i.e., C-Pad) can leak information about the execution semantic of the program. We statistically show that this side channel is infeasible within OBFUSCURO's execution. The reason for this is that the execution time for the data ORAM access (which is performed exactly once per C-Pad) dominates the entire execution time of the C-Pad, and the time taken to perform the ORAM

access is independent to which data block it accesses. We conducted a statistical experiment measuring CPU cycles in executing different classes of code blocks. We constructed five different code blocks, including `NOP`, `ADD`, `SUB`, `IMUL`, `IDIV` code blocks. Each code block initially jumps to the data controller to fetch a data block and the remaining space is filled using one of the instruction type. Furthermore, we impose data dependencies within the instructions to prevent out-of-order execution. We accumulated the execution times for each class over 10,000 repetitions, and the distribution is shown in Figure 5.8-(a). As illustrated, the 10%∼90% percentile intervals for each type (marked as two broken lines) largely overlap, which is hardly possible for an attacker to distinguish.

Program execution time. ObfuscuRO ensures that a program continues executing until its number of executed code blocks reaches a fixed user-configured limit. If the program's logic terminates before that number is reached, ObfuscuRO continues executing dummy code blocks to complete the number of C-Pad executions.

In order to show that this results in a uniform execution time irrespective of the target program being executed, we performed an experiment on a diverse set of applications as shown in Figure 5.8-(b). In the experiment, we fixed the total number of C-Pad executions for each of these applications to $30,000$ and measured the total execution time. We accumulate 100 executions for each program, and plot the distributions of them. As shown in the figure, the ranges of total execution times for the chosen evaluation set largely overlaps, despite computational diversity of these applications. The reason for this is that each C-Pad execution, as illustrated before, is bounded at very similar execution times irrespective of the underlying CPU instructions. Therefore, it is expected that the program execution time (with same number of C-Pad executions) will also be very similar.

5.5 Performance Evaluation

This section reports ObfuscuRO's performance using micro-benchmarks, custom benchmarks, and a real-world program (OpenSSL [175]).

Experimental setup. All our evaluations were performed on Intel(R) Core(TM) i7-6700K CPU @ 3.40GHz with 64 GB RAM (128 MB for EPC). Our system ran Ubuntu 16.04 with

Table 5.2. Performance improvement achieved by using the AVX2 register extensions as the ORAM stash compared to `CMOV`-based stash.

Data Size (Bytes)	CMOV (cycles)	AVX (cycles)	Improvement
1,024	272M	206M	32%
2,048	521M	388M	34%
4,096	1,044M	741M	41%
8,192	2,050M	1,481M	38%

Linux v4.4.0.59. We performed our experiments using Intel SGX SDK [111] and the Intel SGX drivers [143]. Due to the current unavailability of AVX-512 for SGX-enabled computers, most of our experiments (having large code and data sizes) used `CMOV`-based stash. However, we experimented with AVX2 registers to find the expected benefit of using the *register-based* stash and have accordingly simulated the performance improvement achieved by *register-based* stash on our target applications.

Micro-benchmark (comparison between CMOV-based and Register-based stash). Through this micro-benchmark, we answer the question — what is the performance benefit attained by using register-based stash over the CMOV-based stash? One caveat is that all our experiments are based on the AVX2 registers but we expect the performance benefits to be similar while using the AVX-512 registers. Table 5.2 attempts to illustrate the performance benefit achieved by AVX extensions over `CMOV` while accessing data of variable size through ORAM. The improvement is between 30–40%. Compared to the `CMOV`-based stash, since the register-based stash performs just a single oblivious access onto the AVX registers, it outperforms the `CMOV`-based stash.

Custom benchmarks. We ported benchmarking applications to Obfuscuro in order to show the feasibility of obfuscated execution using commodity hardware such as Intel SGX. In particular, we ported a diverse set of applications from simple ones like finding the maximum within a given array to complex binary searching. Figure 5.9-(a) shows the performance shown by Obfuscuro while running the test set of applications described above. We also simulate the performance of Obfuscuro-AVX (the version of Obfuscuro which uses register-based stash. These simulated results are based on the experiments we performed on AVX2. In general, the performance overhead of Obfuscuro-CMOV is on average 83×

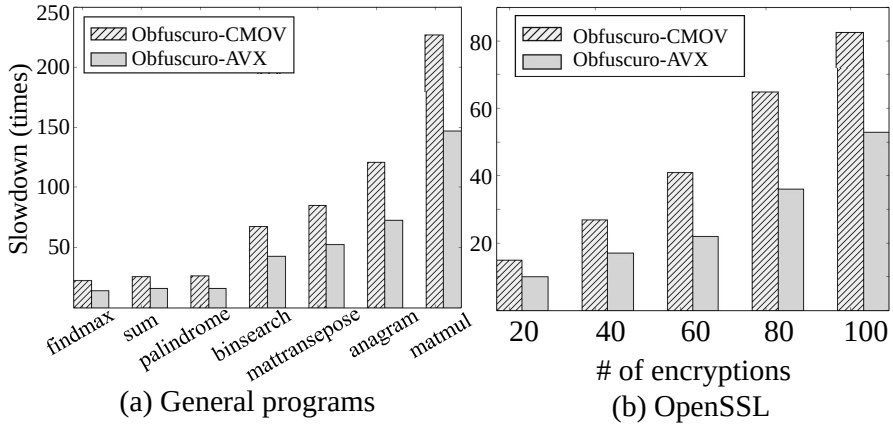


Figure 5.9. Performance benchmarks from our test applications. The average performance overhead of OBFUSCURO-CMOV is 83× and for OBFUSCURO-AVX (simulated) is 51×.

and OBFUSCURO-AVX is 51× The overhead is attributed to: (a) code access control especially dealing with branch-alignment, (b) data access normalization and (c) side-channel-resistant ORAM-based access inside Intel SGX. Of these three causes, (c) is the dominant cost.

Real-world program (OpenSSL). OpenSSL is a popular cryptographic library; hence, we ported it to OBFUSCURO. Figure 5.9-(b) shows OpenSSL's performance—with and without OBFUSCURO—on a variable number of consecutive encryption operations using the AES algorithm. As the number of encryptions increase, the difference between the performance of OBFUSCURO and native also increases. The reason is that OBFUSCURO must perform a fixed number of ORAM operations which adds significant overhead per-encryption whereas the per-encryption overhead of native execution is very small.

5.6 Discussion

This section compares OBFUSCURO to cryptographic program obfuscation solutions and describes how to selectively apply its protections to ensure more efficient performance.

5.6.1 Comparison with Cryptographic Program Obfuscation

This section compares OBFUSCURO with theoretical program obfuscation techniques, which also construct a virtual black box (VBB). We note that, unlike OBFUSCURO which performs hardware-assisted secure remote computation, theoretical program obfuscation techniques [176], [177] do not rely on specific architectural characteristics and thus are designed to be resistant to memory-based side-channel attacks.

Two well-known cryptographic primitives, fully-homomorphic encryption (FHE) and garbled circuits are combined to achieve program obfuscation, but both of them are limited in terms of either performance and generality. In the case of FHE [178], its performance overhead is in twelve orders of magnitude scale in string search [179] without ensuring integrity. On the other hand, garbled circuits [180] incur a performance overhead of around four orders of magnitude. Moreover, they cannot be used for generic programs (i.e., a loop structure in a program cannot be supported), and the integrity cannot be guaranteed similar to FHE. To ensure integrity, verifiable computing techniques can be adopted but verifiable computing itself imposes huge overheads (i.e., about 10^4 times [181]).

In contrast to cryptographic solutions, OBFUSCURO leverages memory protection and remote attestation mechanisms of SGX. OBFUSCURO is a more practical solution since it imposes two orders of magnitude performance overhead, as opposed to twelve and four orders in the case of FHE and circuit representation, respectively. OBFUSCURO also supports generic programs since it retains the form of the host-architecture instruction.

5.6.2 Automating Efficient Application of Obfuscation

Instead of cryptographic program obfuscation, for many enclave use-cases, a lesser property where the program does not exhibit data-dependant access patterns can suffice. This can be efficiently achieved by applying OBFUSCURO's protection only to certain sensitive parts of a program. However, this raises two important questions: (a) how to automatically determine what parts are sensitive? and (b) how to connect sensitive and non-sensitive parts?

In principle, a system can automatically determine what parts of a program are sensitive through a combination of static and dynamic analysis. In particular, pointer analysis [182] can

be employed at compile-time to determine what parts of the code sensitive user data interacts with. The inaccuracy of pointer analysis can be overcome by tracking data at runtime using well-known techniques (e.g., taint tracking [183], hardware memory protection [184]). Connecting sensitive and non-sensitive parts of a program requires to accurately track how information flows between these parts. If the program is partitioned at well-defined points (e.g., function entry or exit), this will be less challenging. We leave the study of the feasibility of this approach to future work.

5.7 Summary

Obfuscuro is the first system to achieve cryptographic program obfuscation on commodity hardware by leveraging the functionality of SGX alongside a principled scratchpad and instrumentation approach. Obfuscuro systematically protects the SGX enclave against information leakage through all side-channels, thereby neutralizing all memory and timing footprints to create a virtual black box program execution. Our evaluation shows that Obfuscuro is significantly faster than existing cryptographic schemes and more deployment-friendly than existing system-based solutions.

6. HARDWARE EXTENSIONS TO DEFEAT MEMORY SIDE-CHANNELS

Despite the hardware root cause of memory side-channels and its impact to SGX, side-channel protection is left to program developers [185]. The assumption is that the hardware design changes to SGX and performance costs to address these side-channels are too high. Unsurprisingly, program developers must choose between (a) overcoming the hardware inadequacies through slow but strong cryptographic protection (like our OBFUSCURO system does in §5) or (b) efficient but weak selective protection (discussed in §3.3).

This chapter questions the assumption that closing critical memory side-channels in SGX requires significant hardware changes and performance costs using REPARO, a small set of SGX model extensions. SGX CPUs can support REPARO with minor architectural modifications, consisting of microcode updates and widely-used hardware components. Importantly, REPARO retains compatibility with legacy SGX—users can run REPARO and legacy enclaves together on future machines. Moreover, even under conservative estimates, REPARO incurs a geometric mean slowdown of only 17% across SPEC CPU 2006 on the latest SGX server machines.

To design REPARO, we extensively analyzed existing hardware memory side-channel protection proposals (§6.1). Since these systems were mostly built for non-SGX contexts, in SGX contexts, they require intrusive architectural changes (e.g., break backwards-compatibility). Moreover, they sometimes provide partial protection or remain inefficient. In light of these problems, our work answers three questions: (a) can we adapt existing approaches to provide strong protection, compatibility, and efficiency in SGX contexts?, (b) what concrete changes to SGX CPUs are required to support our design?, and (c) how do we conclusively demonstrate our design's performance on real SGX hardware?

Surprisingly, SGX CPUs can support REPARO (§6.2) with only a small set of architectural modifications. In fact, four out of five changes (§6.3) can be applied using microcode updates without CPU hardware design changes. Note that CPUs routinely implement new features at both the microcode and hardware design levels [186]. REPARO only requires hardware design changes in the form of a *single* new bound check, a well-understood feature that is efficiently implemented in SGX CPUs, such as in the memory protection extensions (MPX).

Importantly, Reparo can be flexibly enabled for only highly-sensitive enclaves, allowing users to run Reparo and SGX enclaves on the same machine.

We built an emulator, Reparo-EMU, which employs virtual machine extensions to *conservatively* emulate Reparo on real SGX machines (§6.4). We ran both benchmarks and real-world programs—the SPEC CPU 2006 integer suite [187], Redis [188], and Lighttpd [189]. Reparo incurred a geometric mean performance overhead of 17% on SPEC and only up to 11% across remaining programs.

The findings of this chapter suggest that strong and efficient hardware protection against SGX memory side-channels only requires modest architectural and performance costs. Therefore, we urge the community to consider hardware protection against critical memory side-channels, especially in SGX contexts where software approaches are inefficient.

6.1 Approaching Hardware Protection against Memory Side-Channels

This section analyzes several possible approaches to implement hardware protection against controlled and shared memory side-channels (§2.5). Most of these approaches were proposed in non-SGX contexts (e.g., clean-slate enclave designs). The section describes each approach's trade-off in terms of protection, architectural changes required for SGX adoption, and efficiency.

6.1.1 Protection against Controlled Channels

Closing controlled channels requires handing the control of demand paging and page tables to a trusted software component (e.g., enclaves).

Controlling Demand Paging

Under current enclave design philosophy (i.e., unprivileged processes), enclaves can only *partially* control demand paging through collaboration with the operating system. However, privileged enclaves can fully control demand paging or enclaves can rely on an external trusted privileged software.

Collaborating between enclaves and untrusted software. An enclave and operating system can coordinate page fault-handling to secure demand paging. In particular, page faults are sent to the enclave, which then tells the operating system what page to retrieve. The enclave can leverage ORAM [59] to securely retrieve multiple pages and avoid leaking the faulting address. Such collaborative demand paging is proposed by several systems [94]–[96].

Limitation: partial protection and inefficiency. While this approach does not require invasive changes [94], [95], it does not fully close controlled channels. In particular, the operating system still controls paging instructions, which it uses to swap pages. The operating system can abuse these instructions to leak enclave data through Foreshadow, even when the enclave is stopped (i.e., no collaboration) [54]. Even if we ignore Foreshadow (for which partial hardware mitigations exist [56]), this approach significantly increases the performance cost incurred for the already expensive enclave page faults [190] since enclaves and OS must coordinate with each other.

Elevating enclave privileges or leveraging an external privileged software. If enclave privileges are elevated, they can fully handle page faults without any involvement from the operating system [99]. Alternatively, an external (fully-trusted) privileged software (e.g., trusted hypervisor [191], [192]) can be used to handle page faults. With either design, enclave pages can be swapped to a protected disk.

Limitation: invasive changes. Both approaches provide full protection and would be efficient. However, they require completely rethinking the SGX design, and implementing several new functionality (e.g., protected IO for enclaves [193]). Such a redesign will hurt backwards compatibility.

> **Key takeaway.** Secure and efficient enclave demand paging requires redesigning SGX.

Controlling Page Tables

A hardware scheme could install page tables inside the enclave where the operating system cannot observe page table bits or make changes. An alternate scheme could keep page tables

in the operating system's memory but allow the enclave to track (and verify) all critical changes.

Splitting page tables and protecting in enclave memory. Address translation can be divided into enclave and untrusted page tables [96]. The hardware implements protection checks during translation to ensure each page table can only reference its own region. Only enclave page tables are installed in the enclave. This prevents enclaves from abusing page tables to launch attacks (e.g., corrupt system memory), and it allows the operating system to manage untrusted memory.

A major challenge is to efficiently implement protection checks. Sanctum [96] implements them efficiently by dividing the DRAM into large *fixed* regions and ensuring enclaves occupy an entire region.

Limitation: invasive changes. This approach is both secure and efficient. The latter is because enclaves can easily implement any page permission changes (needed for several SGX tasks [34], [35], [70], [71], [74], [125]). However, Sanctum's implementation is incompatible with SGX. In particular, since SGX allocates memory using traditional (4 KB) pages, implementing fixed DRAM regions hurts backward compatibility. DRAM region-based allocation is also inefficient. It only allows (a) fixed size enclaves at multiples of DRAM regions and (b) a limited number of enclaves (e.g., 8 enclaves [96]).

Hiding and tracking page tables in untrusted memory. If page tables are kept in the operating system's memory, the hardware can be updated to hide sensitive information of enclave address translation in page tables (e.g., avoid the setting of page access and dirty bits). However, the operating system can still make changes to the page tables (e.g., induce page faults by unsetting a page table valid bit). To prevent malicious changes, the hardware can help the enclave track all critical changes made by the operating system (refer to [94] for details).

Limitation: inefficiency. Despite all these hardware changes, an enclave still must *redundantly* implement memory management functionality. Moreover, since the operating system controls the page tables, every change (e.g., updating page permissions) still requires coordinating with the operating system.

Key takeaway. Efficiently controlling page tables require direct enclave control, but existing implementation requires invasive changes to SGX.

6.1.2 Protection against Shared Channels

Closing a side-channel through a shared microarchitectural resource requires isolating the use of that resource. Isolation can be achieved by (a) redesigning the hardware to automatically enable isolation or (b) leveraging a trusted privileged software layer to manually isolate resource usage.

Redesigning hardware components. We could redesign system memory components (e.g., caches, branch predictors) to automatically isolate their use amongst different computations. Several systems [96], [103]–[109] have proposed such secure memory components.

Limitation: invasive changes. This requires a fundamentally new silicon design (for each insecure hardware component) and oftentimes changes to address translation components [96] to support the new hardware design.

Leveraging a trusted privileged software. Modern CPUs implement several microarchitectural resource isolation features. A privileged software can enforce isolation of resources between computations using these features [192], [194]. For instance, Time Protection [194] uses a trusted operating system to partition the last-level cache between computations [195].

Limitation: invasive changes. Implementing a privileged trusted software layer requires significant architectural modifications to SGX (refer to §6.1.1).

Key takeaway. Many isolation features are already implemented in SGX CPUs, but they require configuration through a trusted privileged software.

6.2 Reparo Design

Reparo is a set of SGX model extensions, namely *enclave-controlled paging* and *microarchitectural resource isolation*, that close information leakage through controlled and shared channels, respectively. We carefully designed these extensions by adapting previous

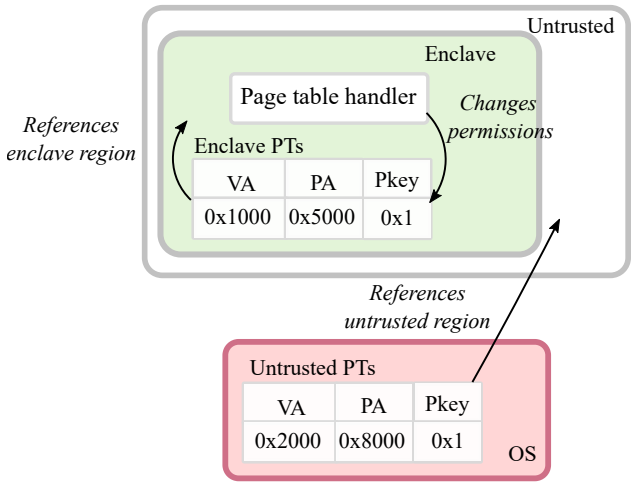


Figure 6.1. REPARO's enclave-controlled paging.

approaches (§6.1) to avoid significant architectural changes, ensure strong protection, and enable efficiency. In §6.3, we explain how future SGX CPUs can enable REPARO alongside existing (legacy) SGX enclaves.

6.2.1 Enclave-Controlled Paging

REPARO restricts the operating system from demand paging enclave memory and enables enclaves to maintain page tables without any help from the operating system. The following sections explain our rationale behind these changes and provide further details.

Restricted enclave demand paging. REPARO does not allow demand paging on enclave memory, but the enclave can implement user-level paging [190]. We chose this design given limitations of enclave demand paging approaches. In particular, the only architecturally-feasible demand paging approach, collaborative demand paging (§6.1.1), provides partial protection and does not offer performance advantages compared to user-level paging (as we explain in the paragraphs below). Moreover, Intel's latest server machines have a significant amount of EPC memory (§2.4).

In cases where an enclave requires additional trusted memory, it can employ user-level paging—use software mechanisms to implement custom protection of untrusted memory pages and self-page them. Many existing SGX research proposals [20], [73], [190] have already

implemented user-level paging. In the future, user-level paging can be included in software development kits to avoid burdening developers (like [94] did).

User-level paging has several advantages over collaborative demand paging. First, it is secure since the operating system does not control any aspect of paging. Second, user-level paging is faster than traditional enclave demand paging (e.g., up to 2.2× more throughput in memcached [190]) since enclave page faults are very costly. Finally, ORAM can protect access patterns on user-level paged memory [20], [85] (like how it is used in collaborative enclave demand paging).

SGX-compatible split page tables. Since Reparo restricts insecure demand paging, there is no need for the operating system to access page table entries for enclave regions. Therefore, Reparo divides the address translation task into enclave and untrusted page tables (discussed in §6.1.1). Then, Reparo stores enclave page tables in protected enclave memory and allows them to be controlled by the enclave. Figure 6.1 illustrates split enclave and untrusted page tables.

Importantly, Reparo keeps enclave memory allocation at the granularity of pages instead of fixed-size DRAM regions used by Sanctum's split page tables (§6.1.1). Page-based allocation is crucial to maintain (a) backwards-compatibility with current SGX versions (which also use pages for allocation) and (b) enclave flexibility—enclaves can be arbitrary sizes and an unlimited number of enclaves are possible.

One challenge with page-based allocation is efficiently ensuring that enclave and untrusted page tables only reference their own regions. This is important to prevent attacks from malicious enclaves and operating systems (§6.1.1). Reparo efficiently ensures this requirement using *contiguous* enclave physical pages and reusing two existing SGX checks. Contiguous memory checks are efficient and many forms of such checks are implemented in SGX CPUs (refer to §6.3.2).

Enclave page tables are created by the operating system, and installed inside enclave memory during initialization. Since SGX verifies enclave address translation (§2.4), the operating system cannot prepare malicious enclave page tables (e.g., map a virtual address to the wrong physical address). During enclave execution, an enclave can update its own page tables.

For instance, the enclave can change its page permissions using Intel Memory Protection Keys [23]. Permission changes are needed for many security tasks (e.g., implementing address space layout randomization [34] and isolating untrusted code [35], [74], [125]).

Finally, REPARO secures enclave virtualization by ensuring that enclave page tables cannot be nested, hence, the hypervisor cannot observe enclave access patterns using nested page tables. Importantly, despite lacking nested page tables, enclave regions inside a virtual machine can be correctly translated because REPARO ensures there is (1) a direct mapping between guest physical and host physical EPC addresses and (2) a static partitioning of EPC to each virtual machine at creation, which is also the norm in current data centers that provide SGX [7], [8], [196], [197]. Furthermore, since REPARO ensures that enclave page tables can only reference an enclave region, there is no risk of an enclave misusing these page tables to access another virtual machine's memory.

6.2.2 Microarchitectural Resource Isolation

REPARO enables the SGX microcode to isolate microarchitectural resources using existing and new hardware isolation features. Since these features can also be configured by the system software, REPARO restricts their configuration from system software when enclave computations execute.

Isolation can either be spatial or temporal. Spatial isolation guarantees that untrusted software and enclaves access different parts of the resource at the same time. In contrast, temporal isolation ensures that untrusted software and enclaves do not access shared hardware resources at the same time. For each resource, REPARO adopts one of the two policies by considering the resource's sharing property: cross-core or per-core. Cross-core resources (e.g., last level cache) can be accessed from other processor cores while the enclave executes. In contrast, per-core resources (e.g., L1/L2) can only be accessed after the enclave stops (if hyper-threading is disabled).

Cross-core spatial isolation. REPARO spatially isolates the last-level cache (LLC) using Intel Cache Allocation Technology (CAT) [23], [195], a widely-implemented feature in Intel

CPUs. In current CPUs, only the system software can configure CAT. With REPARO, CAT is fully-controlled by the trusted SGX microcode when an enclave executes (§6.3.4).

The number of supported enclave partitions by CAT depends on the number of *ways* in the LLC (refer to [195] for details). Current LLCs contain 12–20 ways [192], [195], and a CAT partition needs at least a 1 way allocation. Since a partition must be reserved for the untrusted software (e.g., the operating system) to execute, current CPUs can support 11–19 mutually-untrusted enclaves. Note that REPARO allows unlimited mutually-trusted enclaves (e.g., from the same user) if they share a partition (as we describe in §6.3.4).

We expect that future SGX CPUs can introduce additional LLC ways (or an alternative partitioning mechanism) to support more tenants. This is likely since caches are increasingly large due to new 3D stacking technologies. For example, AMD recently introduced their 3D-stacked CPU, which tripled the LLC over its predecessor with minor increase to CPU power consumption [198]. Intel has also announced similar future 3D-stacked designs [199]. Further strengthening this proposal is our evaluation, which shows that even tiny (<1 MB) cache partitions do not significantly reduce performance in many realistic scenarios (§6.5.3).

Per-core temporal isolation. By design, it is infeasible to spatially isolate per-core resources between hyper-threads of a processor core in SGX CPUs. Hence, REPARO temporally isolates these resources by disabling hyper-threading, ensuring enclaves have sole access to them during execution. Once the enclave exits, REPARO invalidates microarchitectural resources to clear enclave execution traces from them.

Modern SGX CPUs [200] implement hardware control to disable hyper-threading on each CPU core. REPARO leverages this control to ensure that enclave-running CPU cores are not allowed to run hyper-threads. Importantly, since hyper-threading can be disabled on each core separately, this reduces its performance impact (as we show in §6.5.4).

SGX CPUs already implement mechanisms to invalidate several micro-architectural resources. In particular, on enclave exits, REPARO uses the cache invalidation instruction (`WBINVD`) to clear the processor caches, TLB, and internal data buffers. While current SGX CPUs do not provide mechanisms to invalidate some branch predictors—branch target buffer

Table 6.1. Architectural support required by REPARO and the CPU components involved.

	Component	Description
Protection enablement (§6.3.1)	Microcode	Set a CPU bit to enable/disable REPARO; report protection in SECS
Enclave-controlled paging (§6.2.1)		
Address translation checks (§6.3.2)	MMU	One new bound check for enclave region; two existing bound checks
Physical page management (§6.3.3)	Microcode	Update EPC instructions (`EADD` and `EAUG`) to allocate contiguous pages
Hardware resource isolation (§6.2.2)		
Cache partition (§6.3.4)	Microcode	Enforce partitions at enclave start (e.g., `EENTER`); update `WRMSR`
Branch predictor invalidation (§6.3.5)	Microcode	Two new entries in IA32_FLUSH_CMD for invalidating BTB/PHT

and pattern history table—they can easily be implemented using features like Intel has used in the past (§6.3.5).

6.3 Architectural Support for Reparo

This section describes how SGX CPUs can support REPARO with small architectural changes. Implementing new architectural components is a complex task that holds back adoption. To avoid this hurdle, this section proposes REPARO's required support *only* using components previously implemented in SGX CPUs (summarized in §6.3.6). All components we built on are well-documented in several Intel manuals [23], [201] and technical reports [24], [202], [203]. Table 6.1 provides an overview of the necessary support.

6.3.1 Protection Enablement

REPARO implements one additional bit in the SGX Enclave Control Structure (SECS) for enabling memory side-channel protections when an enclave is created. The SECS is a structure that holds several enclave attributes (e.g., 64-bit mode, debug enabled, etc.), and it is protected inside the enclave. If the REPARO bit is set, a CPU *flag* is raised, and the changes mentioned in the remaining subsections apply. Otherwise, enclaves execute in legacy (insecure)

mode. The measurement of the SECS is sent to a user during remote attestation [23]; hence, the user can determine whether their enclave computations are protected against memory side-channels.

6.3.2 Address Translation Checks

Enclave-controlled paging requires that enclave and untrusted page tables should only reference their own regions (§6.2.1). Reparo enforces this rule through the memory management unit (MMU), a CPU component that performs virtual to physical address translation by walking the page tables using a finite state machine (FSM) implementation. In fact, SGX also uses the MMU to enforce rules (e.g., reserve EPC [24]). Importantly, Reparo's rule can be enforced with a *single* new bounds check, which complements the two bounds checks already present in SGX CPUs (Figure 6.2).

First, at the start of translation, the MMU checks if the enclave tried to access an enclave virtual address (❶). If yes, the MMU translates it using enclave page tables whose base is stored in an additional control register, enclave CR3 (eCR3). SGX MMUs already implement this check to ensure the untrusted host process receives an abort transaction when it tries to access an enclave virtual address [24]. In particular, this check is efficiently implemented using a variable memory type range register (MTRR) [23], which requires that enclave virtual addresses are contiguous and their base and size is aligned to the same power of two.

Second, while walking each level of the enclave page tables (from eCR3), the MMU checks if the obtained physical address belongs to the enclave's physical region or not (❷). If it does not, the MMU aborts the translation since an enclave is trying to access untrusted regions using enclave page tables. To implement this check, Reparo requires enclaves to have contiguous EPC physical pages (§6.2.1), allowing an efficient bounds check on the address. This bounds check cannot be implemented using a variable MTRR, because an MTRR limits where an enclave could exist within the EPC based on enclave region size, thus, it is not suitable for allocating enclaves at arbitrary locations.

Fortunately, CPUs have other efficient ways to check arbitrary contiguous memory bounds. One example is using memory protection extensions (MPX), which verifies if an address

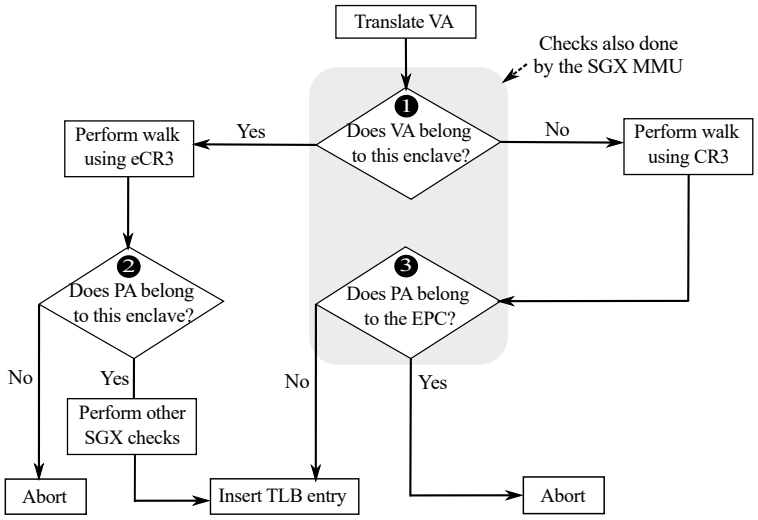


Figure 6.2. REPARO's simplified FSM for a 1-level page table.

falls within an arbitrary upper and lower bound stored in registers to implement access control [23]. Importantly, MPX checks are efficient—research has shown that both upper and lower bounds can be checked in a single cycle [203]. Please refer to §6.4 for more details.

Note that check ❷ does not verify if the final translation is correct—is this the correct physical address corresponding to the starting virtual address? Fortunately, SGX MMUs already verify this using the enclave page cache map (EPCM) before inserting TLB entries (§2.4). While EPCM can also perform check ❷ [204], it would incur a higher performance cost to read EPCM on each page table walk.

Lastly, while walking the untrusted page tables (from CR3), the MMU checks if the physical address belongs to the EPC (❸). If yes, the MMU aborts since enclave regions should only be translated using enclave page tables. SGX already implements this check using a variable MTRR to prevent the operating system from accessing the EPC [24].

6.3.3 Physical Memory Management

REPARO updates SGX instructions to ensure, for REPARO-protected enclaves, the operating system (a) allocates contiguous EPC pages and (b) can only reclaim all EPC pages. Moreover, REPARO allows enclaves to invalidate TLB entries and enforce enclave page table

updates. All these changes only require microcode updates—Intel has shown that microcode updates can change instructions (e.g., `VERW` [205]) even to perform completely different tasks.

During enclave creation, the operating system uses `EADD` to allocate EPC pages to an enclave. In SGX version 2, `EAUG` is used to allocate additional EPC pages to an enclave that is already executing. Both instructions are provided with the EPC page physical address [23], making it trivial to ensure contiguous pages. It is the enclave's duty to ensure that runtime allocation using `EAUG` does not leak information.

The operating system can only kill the enclave and reclaim all its pages, instead of reclaiming pages. To ensure this, Reparo updates the `EWB` instruction (which swaps an EPC page to untrusted memory) to ensure that Reparo-protected enclave pages are not swapped out. When an enclave is killed and its pages are reclaimed (using `EREMOVE`), those pages are already cleared by the SGX microcode.

Finally, Reparo allows enclaves to execute `INVLPG` (TLB entry invalidation instruction) to enforce changes to enclave page table entries (e.g., page permissions). Since enclaves can only update their own (enclave) page tables, there is no harm in allowing enclaves to invalidate TLB entries.

6.3.4 Cache Partition

Reparo partitions the last-level cache (LLC) using Cache Allocation Technology (CAT) (§6.2.2). The paragraphs below describe how Reparo ensures CAT is only configured by the SGX microcode and creates CAT partitions for enclaves.

CAT is configured by system software using model-specific registers (MSRs) [23]. Once enclaves start executing, privileged software (e.g., operating system) is not allowed to use CAT. This is fulfilled by modifying the microcode instruction that updates MSRs—`WRMSR`—ensuring it does not change CAT-related MSRs when enclaves are running [204].

During enclave initialization, Reparo creates an enclave partition and coordinates with other processor cores (using inter-processor-interrupts) to ensure no other core uses that partition. This coordination is a *one-time* cost during enclave initialization since CAT configuration cannot be changed by untrusted software.

When an enclave executes on a processor core, Reparo updates the core's MSR to use its dedicated enclave partition. On enclave exits, Reparo switches the processor core back to an untrusted partition. Each time a partition is changed, the cache is invalidated to enforce those changes [23].

Finally, enclave owners can specify if their enclaves are mutually-trusted so that they can share a partition. Like SGX, Reparo determines ownership using MRSIGNER, a SHA-256 hash of the enclave owner's public key in the enclave's initialization token [23].

6.3.5 Branch Predictor Invalidation

Reparo invalidates two branch predictor units, the branch target buffer (BTB) and pattern history table (PHT), on enclave exits. While SGX CPUs do not provide mechanisms to invalidate these units, they can be implemented as new entries in IA32_FLUSH_CMD, an MSR that SGX CPUs use to invalidate processor structures. Intel has previously issued microcode updates to add new entries to the MSR (e.g., to allow L1D cache invalidation [206]). Additionally, since the structures of these units are standard—PHT is a finite state machine [16], and BTB is an associative cache [15]—it is not challenging to implement such mechanisms [204].

6.3.6 Summary of Architectural Support

Previous subsections demonstrate that defeating several critical memory side-channels only involves a small set of architectural changes. In fact, four out of five changes (§6.3.1 and §6.3.3∼§6.3.5), only require microcode updates to current SGX CPUs. These microcode updates only make small changes that directly employ *existing features* used by SGX, such as instructions (e.g., EPC allocation and paging) and mechanisms (e.g., L1D invalidation). Reparo additionally needs a minor change to the CPU hardware design (§6.3.2), essentially a *single* new bounds check, such as those of MPX, which is already used in SGX CPUs [23], [203]. Note that Intel routinely introduces new features at both the microcode and hardware level each year [186]. Without going to the point of actually building the chip (which is only possible for Intel), all our findings strongly suggest Reparo's feasibility on SGX CPUs.

6.4 Implementation

We built Reparo-EMU to conservatively *emulate* Reparo on Intel's latest machines (§6.5). Reparo-EMU leverages virtual machine extensions (VMX) [23] to *mimic* the microcode. In particular, it intercepts all transitions between the enclave and the untrusted world. On intercepts, Reparo-EMU implements Reparo's protections using hardware features and software. Reparo-EMU is built by adding 1,578 code lines to the Bareflank extensible hypervisor [207].

While Reparo's page walk checks (§6.3.2) cannot be emulated by Reparo-EMU, these checks should have a *negligible* performance impact. In particular, the untrusted page table check uses a variable MTRR, while the enclave page table check verifies an MPX upper and lower bound. Both checks can be evaluated in a *single cycle* [24], [203]. Since page walks can take up to many hundreds of cycles and are very infrequent due to TLB caches, an additional single cycle latency only has a 0.01% impact on system performance [96].

Reparo-EMU intercepts an enclave entry using the extended page tables (EPT), a VMX feature that can revoke execute permissions from any physical memory region. Using this feature, Reparo-EMU revokes execute permissions from the enclave page cache (EPC). Hence, when a process enters an enclave region and executes an instruction, the machine raises a protection trap to Reparo-EMU.

Reparo-EMU also intercepts when an enclave stops executing (generally called an enclave exit). Traditionally, enclave exits occurred for three reasons: system calls, page faults, or interruptions. However, modern enclaves use the SGX switchless system call feature [73], [208] and background threads to service system call without exits. Furthermore, due to enclave-controlled paging (§6.2.1), Reparo ensures there are no page fault-based exits. Hence, Reparo-EMU only needs to trap enclave exits due to interrupts. This is achieved by setting the interrupt-exiting bit in the x86 virtual machine control structure, a feature of VMX [23].

When an enclave is created, Reparo-EMU (a) creates a dedicated LLC partition for that enclave using Intel CAT and (b) sends an inter-processor-interrupt to ensure that no processor core uses that partition. The partitions are created using model-specific-registers:

Table 6.2. REPARO's defense for memory side-channels.

Channel	Reparo defense
Controlled	
Page fault	Restrict insecure demand paging (§6.2.1)
Page table entry	Keep page tables in enclave region (§6.2.1)
Paging instructions	OS cannot execute the instructions (§6.3.3)
Shared	
Last-level cache	Partition LLC during execution (§6.3.4)
L1/L2 cache	Invalidate L1/L2 on enclave exit (§6.3.4)
Branch target buffer	Invalidate BTB on enclave exit (§6.3.5)
Pattern history table	Invalidate PHT on enclave exit (§6.3.5)
TLB	Invalidate TLB on enclave exit (§6.2.2)
Internal data buffers	Invalidate buffers on enclave exit (§6.2.2)

IA32_L3_MASK_{0-N}. When a processor core starts executing an enclave (i.e., enclave entry), REPARO-EMU enforces the enclave's dedicated CAT partition by setting IA32_-PQR_ASSOC. The processor core uses the partition until there is an enclave exit, at which point, REPARO-EMU switches the processor core to an untrusted partition. Each time a partition is changed, the cache is invalidated using `WBINVD` to enforce the changes.

Since current SGX CPUs do not implement mechanisms for branch predictor invalidation, REPARO-EMU implements a *software* branch predictor flush on enclave exits. The idea behind our flush is to leverage reverse-engineered structures and addressing mechanisms of the predictors [15], [16] to fill them with bogus entries. Our flush executes 49,152 conditional branches and 4,096 other branch instructions to fill the PHT and BTB, respectively. The correctness of this manual flush is dependent on the reverse-engineered addressing mechanisms of the branch predictors. However, given that this flush is shown to be suitable for high-resolution branch predictor side-channel attacks and a lack of alternatives, we chose this mechanism. In all likelihood, our flush is *over-estimating* the performance impact, rather than under-estimating, because hardware invalidation is considerably more efficient than filling branch predictors with bogus data.

Table 6.3. Machine platforms used for evaluation.

	Desktop	Server
Hardware		
CPU model	i7-8700	Xeon Gold 6348
CPU sockets	1	2
Cores × threads	6 × 2	28 × 2
Clock speed	3.20GHz	2.60GHz
Cache (L1/L2/LLC)	64KB/256KB/12MB	64KB/1.2MB/42MB
LLC ways	16	12
RAM size	16GB	512GB
EPC size	128MB	128GB
Software		
Linux kernel	5.4	5.11
SGX SDK	2.3	2.15
SGX driver	Legacy 2.6	DCAP 1.41

6.5 Performance Evaluation

This section describes REPARO's performance through custom benchmarks and diverse real-world programs.

6.5.1 Setup

We evaluated REPARO using both desktop and server machines. Although SGX is deprecated in desktop machines [209], we also used this machine because it has a very small last-level cache. This allows us to estimate REPARO's performance on future CPU iterations where enclaves are provided smaller LLC partitions to support more untrusted enclaves. Table 6.3 lists the specifications of our machines. Both machine kernels were configured as *tickless* [210] to reduce enclave exits due to timer interrupts [87].

6.5.2 Micro-Benchmarks

This section describes the direct overhead of REPARO's cache and branch predictor protections at enclave exits (§6.2.2).

Settings. We used REPARO-EMU (§6.4). We ran a benchmark enclave program that continuously writes to a large 256 MB buffer on both machines. We ran the enclave continuously for 60 seconds and measured protection overheads (Table 6.4) at enclave exits.

Extra emulation overhead. The framework adds a tiny overhead (∼0.003 milliseconds) to intercept enclave entries and exits. Although this is extra overhead that is not incurred in a real hardware implementation, it is significantly smaller than the overhead of other protections (mentioned below). Thus, its impact on our real-world program results is negligible.

Cache partitioning and invalidation. REPARO's cache defenses require (a) partitioning the last-level cache (LLC) and switching partitions during enclave execution and (b) writing back and invalidating the caches at enclave exits.

Partitioning the LLC and switching partitions is fast: it takes ∼200 cycles to update a model-specific register (using `WRMSR`). Cache write-back and invalidation time depends on the state and size of the cache. On the desktop machine, we noticed that it took up to 3.35 ms, whereas its lower bound (through consecutive invalidations) was 0.08 ms. Cache invalidation took from 1.25 ms to 7.48 ms on the server.

Despite a smaller cache, invalidation on the desktop is not much faster than the server. The reason is that, unlike server machines where SGX does not implement hardware memory integrity [98], the desktop enforces integrity using a Merkle tree. This tree is updated on each cache-line flush [116], incurring 6 additional memory accesses. Notably, invalidating non-enclave memory on the desktop machine took only up to 0.81 ms.

Branch predictor invalidation. We executed our custom branch predictor flush to invalidate the BTB and PHT (§6.4). The lower bound for both invalidations together was 0.04 and we saw an upper bound of 0.30 milliseconds. This is expected given the small size of these units and it is consistent with prior simulation results [15], [211].

6.5.3 Real-world Enclave Programs

This section discusses REPARO's performance on diverse real-world programs, including an assorted program suite, a key-value store, and a web server.

Table 6.4. REPARO's isolation overhead in micro-benchmarks.

Invalidation	Time (kcycles) Min	Max	Time (ms) Min	Max
Desktop				
Cache (L1/L2 + LLC)	247	10560	0.08	3.35
Branch (PHT + BTB)	120	844	0.04	0.30
Total	367	11404	0.12	3.65
Server				
Cache (L1/L2 + LLC)	3240	19454	1.25	7.48
Branch (PHT + BTB)	373	494	0.14	0.19
Total	3613	19947	1.39	7.67

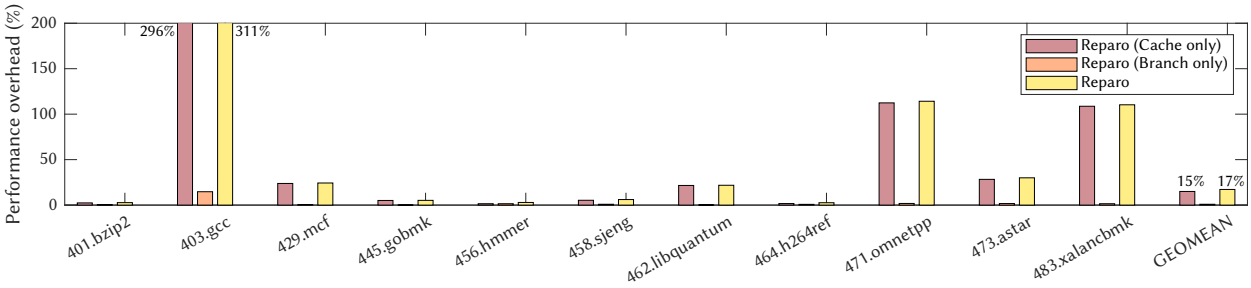


Figure 6.3. SPEC 2006 performance with REPARO using the reference dataset on the server machine. The enclave partition was 1/12*LLC. For this test, the enclave exits per-second were: 3, 105, 4, 3, 3, 2, 2, 1, 11, 1, 8, from left to right.

Common settings. Since running complex Linux programs inside enclaves is challenging, we used the Occlum and Graphene library OSs [74], [112]. Unless noted otherwise, we ran programs on the server machine using an enclave partition size of 1/12*LLC, the smallest allowed partition on the server machine (the server machine's LLC is divided into 12 ways). This setting allows the machine to be shared amongst the most number of users, highly desirable in cloud machines. We ran each program 5 times and report the average.

Since it is challenging to reason about background workloads in diverse cloud settings, we ran no heavy background workloads. This setting essentially compares the performance of the *worst-case* of REPARO with the *best-case* of SGX. In practice, background workloads significantly reduce SGX performance through normal cache contention. These workloads do

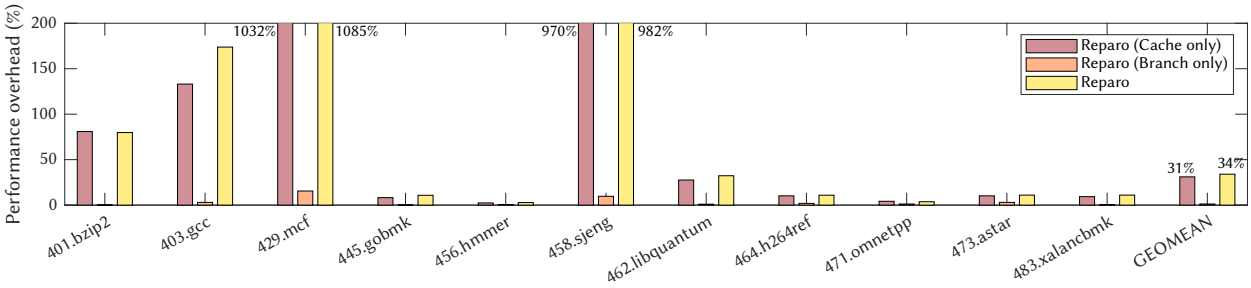


Figure 6.4. SPEC CPU 2006 performance with REPARO using the test dataset on the desktop machine. The enclave partition was 1/8*LLC. For this test, the enclave exits per-second were 29, 2184, 3071, 965, 25, 3444, 417, 298, 60, 22, 91, from left to right.

not impact REPARO since its last-level cache is isolated. Thus, this setting is very *conservative* and we chose it to estimate REPARO's upper-bound overheads.

Assorted (SPEC CPU 2006). SPEC CPU 2006 is a collection of well-known CPU and memory-intensive programs, hence, it is useful to assess real-world system performance.

Figure 6.3 illustrates REPARO's performance across SPEC programs using reference datasets on the server machine. Encouragingly, most programs incurred a modest performance overhead—7 out of 11 incurred less than 20% slowdown, and the geometric mean slowdown was 17%. Across all programs, the biggest slowdown factor was cache protections at enclave exits. Since our experiments used switchless system calls and tickless kernels (§6.4), most programs incurred very few enclave exits and showed modest performance overhead. Nevertheless, the smaller enclave LLC partition had a considerable effect (e.g., 311%) on the performance of highly memory-intensive programs like gcc and omnetpp.

We also ran SPEC programs on the desktop machine using test datasets to estimate performance in even more cache-constrained scenarios. Since reference workloads require significant memory, it is infeasible to run them on the desktop. In Figure 6.4, we show REPARO's performance on the desktop using 1/8*LLC, the closest to fair sharing for each core. We also show desktop performance using 1/16*LLC, the smallest possible partition, and 1/4*LLC (Figure 6.5).

The geometric mean overhead was 34% on the desktop machine with 1/8*LLC. A factor out of our control on this machine was demand paging using page faults, which is restricted

under REPARO. In particular, even with test datasets, the SPEC programs required between 150 MB - 1 GB of memory. Since the desktop machine only has a 128 MB EPC, demand paging was inevitable. Thus, the programs incurred many more enclave exits because of page faults and performance was (expectedly) lower than the server machine. We noticed two programs, mcf and sjeng, incurred a very high overhead. We found that their test datasets required up to 1 GB of memory, hence their enclaves incurred the most exits (due to page faults) per-second.

Finally, even with a 1/16*LLC enclave partition, enclaves used only 0.75 MB of the desktop's LLC, SPEC showed an 86% geometric mean overhead. We expect this performance is acceptable in high-security use-cases. Indirectly, this shows that server CPUs with large caches can further divide the cache to support additional enclave partitions, while maintaining acceptable performance. For instance, if our server machine divides its cache into 0.75 MB partitions, it would support 110 enclave partitions (for both sockets).

Key-value store (Redis). Key-value stores like Redis [188] are widely used in cloud environments. We evaluated Redis using its official redis-benchmark, which tests 20 different key-value store operations including GET, SET, MSET, and POP. We ran each operation for 100,000 iterations using the default settings of 50 parallel clients.

REPARO reduced throughput by 4–21% (geometric mean was 11%) across these operations, compared to native SGX. We observed 27 enclave exits per-second during the benchmark's 119 second execution. Given a low number of enclave exits and the fact that Redis is highly memory-intensive, the major factor behind its throughput reduction was the program executing on a very restricted LLC partition.

Web server (Lighttpd). Webservers like Lighttpd [212], handle sensitive queries to fetch webpages, and hence are a good fit for SGX. We ran Lighttpd with 8 worker threads because this setting maximized throughput. From a separate server machine (average latency between machines was 0.09 ms), we used ApacheBench to send 10,000 requests for a 10 KB file from up to 256 concurrent clients.

REPARO's geometric mean throughput reduction across the test was only 5%. Interestingly, requests from a single client incurred a 65% throughput reduction, while requests from 256

concurrent clients incurred only 1% reduction. The reason is that the worker threads go to sleep when there are no requests and they are awakened through inter-processor-interrupts, hence they incur additional enclave exits. With greater concurrency, the workers are always busy handling requests, thus they do not go to sleep and incur fewer exits. In practice, the performance for single client scenarios can be improved using spinlocks to ensure threads never sleep.

6.5.4 Non-Enclave Programs

Reparo addresses the root cause of critical side-channels by avoiding resource sharing. This inevitably reduces non-enclave (and non-Reparo enclave) performance since there are performance advantages of insecurely sharing hardware resources between processes. Generally, Reparo impacts their performance due to (a) smaller LLC partition and (b) limited hyper-threading. Note that Intel also recommends disabling hyper-threading [56] to defeat foreshadow. We evaluate this impact in the paragraphs below.

Settings. Since the desktop machine has a smaller cache and fewer cores, its non-enclave impact is likely greater, hence we used it for this test. We ran memcached [155] as a non-enclave program which serves requests using 12 threads. From a separate machine (average latency was 0.32 ms), we executed the official memaslap [213] benchmark to request keys with a 9:1 split of GET:SET for 60 seconds using 12 threads. In the background, we ran up to 5 memory-intensive enclaves (since the machine only has 6 cores) using native SGX and Reparo. Each enclave allocated and accessed 16 MB of memory continuously. For Reparo, we tested with both disjoint 1/8*LLC and 1/16*LLC partitions for each enclave. For both SGX and Reparo, we progressively disabled hyper-threading on each spawned enclave's processor core.

Results. Intriguingly, memcached performed better alongside Reparo than legacy SGX enclaves under 1 to 4 enclaves (Figure 6.6). The reason is that Reparo's CAT partitioning ensures that the memory-intensive background enclaves cannot flush memcached's cache contents. In fact, this is precisely the reason why CAT was introduced [202].

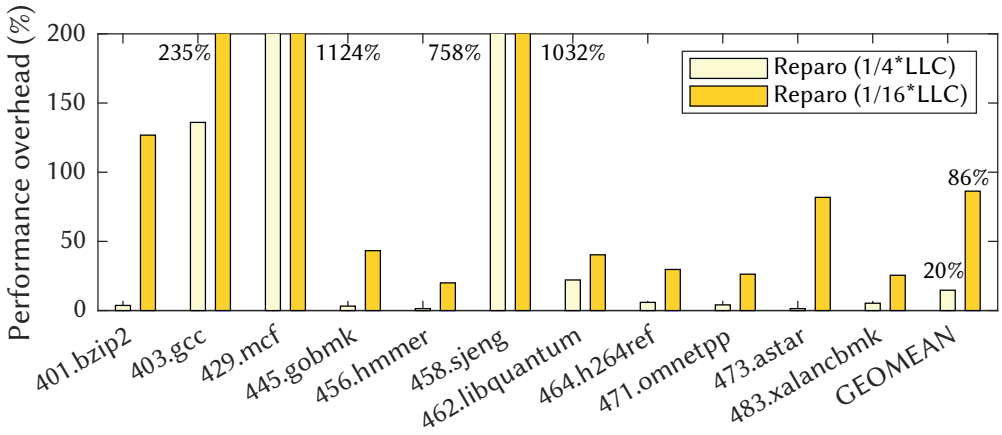


Figure 6.5. SPEC CPU 2006 performance with REPARO using different enclave LLC partitions on the desktop machine.

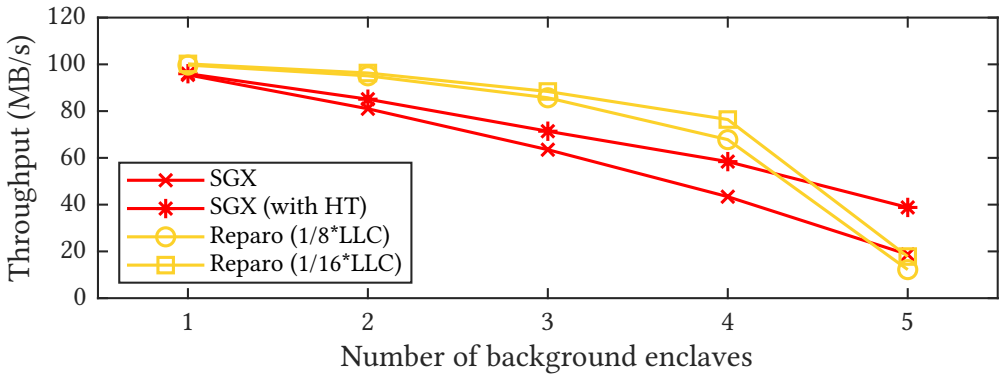


Figure 6.6. Memcached's performance as a non-enclave program, alongside SGX and REPARO, on the desktop machine.

With 5 background enclaves, the combined effect of reduced hyperthreading and a tiny LLC partition available to memcached produced a throughput reduction of 5% compared to SGX. If we consider SGX with hyper-threading (not recommended by Intel), the reduction is 54%. In this scenario, the machine is significantly over-exerted, hence it is unlikely to be a common use-case. In general, we expect non-enclave impact can be minimized by load-balancing computations.

6.6 Discussion

This section discusses how REPARO can be optimized and describes REPARO's protection against other shared micro-architectural resources. It concludes by discussing the limitations of the isolation guarantees provided by CAT.

Efficiency optimizations. If additional architectural changes are feasible, REPARO can improve enclave memory efficiency and performance using two techniques.

First, REPARO could employ the enclave page cache map (EPCM), instead of the MPX check to secure enclave page tables (§6.3.2). This would allow an enclave's pages to be allocated anywhere in the EPC, avoiding memory fragmentation. However, there would be an additional performance penalty, since accessing the EPCM would require one more memory access at each page walk.

Second, REPARO could decouple CAT partition switching from `WBINVD` and introduce an L1/L2 cache flush instruction to reduce enclave exit cost. Currently, CAT uses `WBINVD` to switch partitions, but this instruction also flushes the last-level cache, which is not required since it is partitioned. However, this extension might require changes to CAT.

Other micro-architectural resources. Apart from the memory resources directly targeted by REPARO, we briefly discuss other micro-architectural resources below.

Per-core computational resources. Hyper-threads in a CPU core share various non-memory computational resources, including the execution ports [214] and functional units [215]. REPARO closes side-channels through these resources by disabling hyper-threading for enclave-running cores. Since these resources are *stateless* (no residual state when enclave stops), invalidation is not needed.

Memory bus. CPU cores share a memory bus to retrieve contents from DRAM. The limited bus bandwidth creates a contention-based side-channel [17]. Unfortunately, the memory bus is a fundamental architectural component of modern CPUs, and securing it requires architectural redesigns [216]. We leave its study to future work.

CPU ring interconnects. Intel *desktop* CPUs (where SGX is deprecated) use a ring to connect processor cores, which can be used as a side-channel [217]. Intel server CPUs do not use a CPU ring, rather they use a mesh interconnect [218].

CAT isolation properties. Although CAT enables isolation and protection against an extensive range of last-level cache attacks [192], [195] (especially in SGX scenarios where memory is not shared), it does not offer full isolation against subtle attacks [219]. There are well-known proposals [220] to extend last-level caches with full isolation, small changes, and similar performance to CAT. In the future, hardware vendors should consider these proposals to fully-protect last-level caches.

6.7 Summary

Protection against memory side-channels is a task left for enclave developers because it is assumed to require significant architectural and performance penalties. In this chapter, we question this implicit assumption. We present Reparo, a set of SGX model extensions that close critical memory side-channels with very few architectural modifications and a geometric mean performance overhead of only 17%. Given these insights, we urge the community to reconsider hardware protection against memory side-channels, especially in enclave scenarios where software solutions are prohibitively slow or provide weak protection.

7. GENERALIZATION TO OTHER ENCLAVES

All solutions proposed in this dissertation are built for SGX enclaves, because SGX is the most widely-adopted TEE in cloud machines at the time of this dissertation. While the design of our hardware extensions (Reparo) is tied to the SGX hardware, our built systems—Chancel, Obfuscuro, and Obliviate—are not specific to SGX. Instead, these systems can be generalized to other TEEs that provide *process-based* enclaves, including Keystone [99], Sanctum [96], and ARM Confidential Computing Architecture (CCA) [6]. The rest of this section explains why we expect this generalization to be feasible and how to approach it.

Chancel. Chancel's Multi-Client SFI does not require architecture-specific features, unlike other SGX SFI (e.g., Occlum [74] requires advanced memory protection provided by Intel MPX). In particular, MCSFI only requires general-purpose registers. Hence, by carefully porting its logic, MCSFI can be extended to any potential architecture. This is similar to how the Native Client SFI was ported to ARM [221]. The major porting effort required would be to carefully reason about how an enclave interacts with the untrusted world (e.g., enclave exits to the operating system) on the platform being ported to defeat data leaks.

Obfuscuro. Obfuscuro's scratchpad-based program obfuscation approach (§5.1) is not specific to the SGX hardware. Instead, the approach carefully considers the information leaked from memory side-channels opened by different micro-architectural resources (§2.5.1). Generally, these resources behave in a similar manner even across hardware platforms and leak information in the same way (e.g., all platforms leak data access patterns at the cache-line granularity). Finally, although the ARM instruction set does not have a `CMOV` instruction or AVX registers—needed by Obfuscuro's secure ORAM implementation (§5.2.1)—it has conditional execution primitives [222] which we believe can be used to create a similarly secure ORAM implementation.

Obliviate. The Obliviate file system implementation follows the POSIX system call standards and leverages ORAM for data obliviousness (please refer to the paper [20]). As we discussed in the previous section, ORAM can be securely implemented in different platforms; hence, Obliviate can also be extended to such platforms.

8. FUTURE RESEARCH DIRECTIONS

While the work described in previous sections significantly enhances the promise of enclaves for user data protection on cloud machines, there is still a long path ahead to ensure ubiquitous data protection in all cloud scenarios. This section provides a non-exhaustive list of research directions that can be spun from this dissertation.

8.1 Securing Interactions between Enclaves and Devices

Enclave solutions like SGX confine the trusted boundary of a protected computation to the CPU and do not implement techniques to securely interact with external devices. This is a significant limitation because increasingly cloud computations rely on external device accelerators like GPUs and FPGAs. For instance, deep learning computations on sensitive user data (e.g., patient data collected from hospitals [11]) rely extensively on GPUs. Hence, it is vital for enclave computations to securely connect to external devices. The rest of this section discusses several ways to create a secure communication channel between SGX enclaves and external devices, while discussing various trade-offs.

Leveraging trusted devices. We can design new external devices that are retrofitted with mechanisms to allow enclave-to-device attestation, like a secret key burned in the device e-fuses during manufacturing. A user can supply the device's public key to an enclave and let the enclave attest the device through standard attestation protocols. After attestation, the device and enclave can establish a shared secret key for secure communication. In fact, although not formally part of this dissertation, our group has shown how to design FPGA [223] and USB [224] devices with attestation capabilities using commodity hardware components. Another research work showed how to create a trusted GPU device [225]. Given the diversity of devices and device manufacturers, it is challenging to deploy all devices with such trusted attestation primitives. Moreover, the device should have some basic computational capability (e.g., for attestation), which is not feasible for all devices (e.g., tiny sensors).

Retrofitting enclaves with a trusted software privileged layer. The SGX design can be extended to implement trust on a software privileged layer (external to the enclave), which ensures a trusted path between the enclave program and the device [226]. This privileged

layer must control the device's configuration and interrupts. The advantage of this approach is that it maintains compatibility with current or future devices. A straightforward candidate for a trusted privileged layer is the VMX (or hypervisor) layer [23], which has the capability to fully control device communication on a machine. Unfortunately, it is challenging to trust the hypervisor in already virtualized cloud environments, relegating the use of this approach only to baremetal SGX deployments [196]. Another possibility is to install a trusted software in the system management mode (SMM), a privileged layer present in all SGX CPUs to handle system-wide operations like power management and hardware-assisted debugging. Recently, a research work leveraged the SMM layer to debug the enclave memory [227]. Finally, future Intel server machines are implementing a new trusted privileged layer for their new security technologies (e.g., Intel TDX [228]). This layer could also be a potential candidate.

Implementing architectural extensions. A third approach could be to extend the architectural design of CPUs and allow enclaves to directly contact external devices. One way to implement this approach is to change the PCIe interconnect and MMU design to secure a device's configuration and map its memory-mapped I/O (MMIO) regions to a computation [193]. However, this implementation is only suitable for PCIe devices and alternate mechanisms would need to be implemented for legacy device interconnect technologies. Like the trusted device approach, the significant advantage of such architectural extensions is that they would minimize trusting software external to the enclave. The extension approach also does not require encryption/decryption during enclave-to-device data transfer.

8.2 Leveraging Virtual Machine Enclaves for Data Protection

In recent years, AMD Secure Encrypted Virtualization (SEV) [4] introduced a new trusted execution environment (TEE) that protects entire virtual machines—including the operating system and multiple applications—within the enclave boundary. The *virtual machine enclave* approach has both benefits and drawbacks. The benefit is that applications can natively execute within virtual machine enclaves, unlike SGX enclaves which require rewriting or a library operating system [70]. Unfortunately, this software compatibility comes at the cost of a very large trusted computing base (TCB) (e.g., the entire Linux operating system with

millions of code lines). In practice, however, the advantages of virtual machine enclaves seem to outweigh the drawbacks, which is why they are increasingly being adopted by other hardware vendors [6], [228]. The rest of this section discusses how to address the large TCB challenges raised by virtual machine enclaves.

Minimizing software trusted computing base (TCB). The software TCB of virtual machine enclaves can be minimized by (a) reducing the operating system codebase (e.g., micro-kernels or software debloating [137]) or (b) implementing a higher privileged tiny security monitor to protect sensitive applications. We find the latter approach more interesting; hence, we discuss it in the next paragraph.

We can use a *security monitor* to protect trusted computations inside a virtual machine enclave. Significant previous research [63], [68] implement such security monitors using a trusted hypervisor, which is incompatible with SEV. An alternate method to implement such a security monitor is through software fault isolation (SFI) or newly-introduced intra-guest hardware privilege isolation mechanisms. In particular, like CHANCEL, a robust SFI technique can reduce the privileges of the untrusted guest operating system and prevent it from accessing a trusted computation's memory regions [191], [192]. Modern processors have also introduced two new hardware privilege isolation mechanisms, namely virtual machine privilege levels (VMPL) [229] and protection keys for supervisors (PKS) [23]. These features allow a trusted supervisor-level software to control page access/execution permissions. Unlike the VMX layer, both mechanisms are still available to implement a security monitor inside a guest machine and protect trusted computations.

Monitoring the virtual machine state. Another approach to increase trust on the correctness of a virtual machine enclave is to continuously log sensitive operations [224] (e.g., network-related system calls) and analyze logs to determine the virtual machine's state before running sensitive computations. This can be used as a form of *dynamic attestation* of the enclave. A key challenge with a logging approach is the protection of system logs—once the operating system is compromised, it can trivially destroy logs. The privileged security monitor (described in the previous section) is one possible way of implementing log protection.

9. CONCLUSION

This dissertation makes a significant stride towards the ideal of strong user data protection on cloud machines. We achieve such protection by leveraging Trusted Execution Environments (TEEs) like Intel SGX while defeating the critical threats of adversarial cloud services and memory side-channels through software and hardware approaches. All leveraged approaches provide principled protection—address the underlying root cause of a problem instead of providing partial mitigation. Finally, we outline several future research directions to explore in the pursuit of ubiquitous data protection in cloud machines using TEEs.

REFERENCES

[1] Tom's Hardware, *Linux kernel grows past 15 million lines of code*, https://www.tomshardware.com/news/Linux-Linus-Torvalds-kernel-too-complex-code,14495.html.

[2] G. Klein, K. Elphinstone, G. Heiser, *et al.*, "seL4: Formal verification of an OS kernel," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[3] F. McKeen, I. Alexandrovich, A. Berenzon, *et al.*, "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, Jun. 2013.

[4] D. Kaplan, "AMD x86 memory encryption technologies," https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/kaplan, Austin, TX: USENIX Association, Aug. 2016.

[5] D. Cerdeira, N. Santos, P. Fonseca, and S. Pinto, "Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted TEE systems," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2020.

[6] ARM, *Arm confidential compute architecture*, https://www.arm.com/architecture/security-features/arm-confidential-compute-architecture, 2022.

[7] Microsoft, *Azure confidential computing*, https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 2018.

[8] IBM, *Data-in-use protection on IBM Cloud using Intel SGX*, https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/, 2018.

[9] Alibaba Cloud, *ECS bare metal instances*, https://www.alibabacloud.com/product/ebm.

[10] Signal, *Technology preview: Private contact discovery for Signal*, https://signal.org/blog/private-contact-discovery/.

[11] Fortanix, *UCSF, Fortanix, Intel, and Microsoft Azure utilize privacy-preserving analytics to accelerate AI in healthcare*, https://www.fortanix.com/company/pr/2020/10/ucsf-fortanix-intel-and-Microsoft-azure-utilize-privacy-preserving-analytics-to-accelerate-ai-in-healthcare/.

[12] 23andMe, *23andMe: Dna genetic testing and analysis*, 2017. [Online]. Available: https://www.23andme.com.

[13] J. Götzfried, M. Eckert, S. Schinzel, and T. Müller, "Cache attacks on Intel SGX," in *Proceedings of the 10th European Workshop on Systems Security (EUROSEC)*, Belgrade, Serbia, Apr. 2017.

[14] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Proceedings of the 14th Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA)*, Bonn, Germany, Jul. 2017.

[15] S. Lee, M. 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, BC, Aug. 2017.

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

[17] Z. Wu, Z. Xu, and H. Wang, "Whispers in the hyper-space: High-speed covert channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[18] 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.

[19] 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, BC, Aug. 2017.

[20] A. Ahmad, K. Kim, M. I. Sarfaraz, and B. Lee, "Obliviate: A data oblivious file system for Intel SGX," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[21] Intel, *What is hyper-threading? - intel*, https://www.intel.com/content/www/us/en/gaming/resources/hyper-threading.html.

[22] 7-CPU, *Intel haswell*, https://www.7-cpu.com/cpu/Haswell.html.

[23] Intel, "Intel 64 and ia-32 architectures software developer's manual," *Volume 3A: System Programming Guide*, 2016.

[24] V. Costan and S. Devadas, "Intel SGX explained.," *IACR Cryptology ePrint Archive*, 2016.

[25] J. Katz and Y. Lindell, *Introduction to Modern Cryptography*. Chapman and Hall/CRC Press, 2007, ISBN: 978-1-58488-551-1.

[26] W. Newhouse, M. Bartock, J. Cichonski, *et al.*, "Derived personal identity verification (piv) credentials," *NIST Special Publication*, Aug. 2019.

[27] M. Scholl, K. Stine, J. Hash, *et al.*, "An introductory resource guide for implementing the health insurance portability and accountability act (HIPAA) security rule," *NIST Special Publication*, Oct. 2008.

[28] K. Stouffer, J. Falco, K. Scarfone, *et al.*, "Guide to industrial control systems (ICS) security," *NIST Special Publication*, May 2015.

[29] M. Nieles, K. Dempsey, V. Y. Pillitteri, *et al.*, "An introduction to information security," *NIST Special Publication*, Jun. 2017.

[30] Phoronix, *The linux kernel enters 2020 at 27.8 million lines in git but with less developers for 2019*, https://www.phoronix.com/scan.php?page=news_item&px=Linux-Git-Stats-EOY2019#:~:text=The%20Linux%20Kernel%20Enters%202020,Less%20Developers%20For%202019%20%2D%20Phoronix&text=As%20of%20this%20morning%20in,in%20at%2027.8%20million%20lines!.

[31] F. McKeen, I. Alexandrovich, I. Anati, *et al.*, "Intel® software guard extensions (Intel® SGX) software support for dynamic memory allocation inside an enclave," in *Proceedings of the 5th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, Seoul, South Korea, Jun. 2016.

[32] Intel, *Intel software guard extensions programming reference*, https://www.intel.com/content/dam/develop/external/us/en/documents/329298-002-629101.pdf.

[33] Intel, *Intel 3rd gen xeon scalable processors (ice lake)*, https://www.storagereview.com/news/intel-3rd-gen-xeon-scalable-processors-ice-lake.

[34] J. Seo, B. Lee, S. Kim, *et al.*, "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.

[35] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "Chancel: Efficient multi-client isolation under adversarial programs," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Event, 2021.

[36] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "Aegis: Architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 28th ACM International Conference on Supercomputing (ICS)*, Munich, Germany, Jun. 2014.

[37] J. A. Halderman, S. D. Schoen, N. Heninger, *et al.*, "Lest we remember: Cold-boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul. 2008.

[38] 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.

[39] P. Grassi, M. Garcia, and J. Fenton, "Digital identity guidelines," *NIST Special Publication*, Jun. 2017.

[40] W. Wang, G. Chen, X. Pan, *et al.*, "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)*, Dallas, TX, Oct. 2017.

[41] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proceedings of the 11th USENIX Workshop on Offensive Technologies (WOOT)*, Vancouver, BC, 2017.

[42] D. Gruss, C. Maurice, K. Wagner, and S. Mangard, "Flush+ flush: A fast and stealthy cache attack," in *in Proceedings of the 15th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Paris, France, Jun. 2016.

[43] Y. Shin, H. C. Kim, D. Kwon, J. H. Jeong, and J. Hur, "Unveiling hardware-based data prefetcher, a hidden source of information leakage," in *Proceedings of the 25th ACM Conference on Computer and Communications Security (CCS)*, Toronto, ON, Oct. 2018.

[44] B. Gras, K. Razavi, H. Bos, and C. Giuffrida, "Translation leak-aside buffer: Defeating cache side-channel protections with TLB attacks," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[45] 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.

[46] Intel, *Linux-sgx/sdk/tlibcrypto/*. [Online]. Available: %5Chowpublished%7Bhttps://github.com/intel/linux-sgx/tree/master/sdk/tlibcrypto%7D.

[47] 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.*

[48] D. Brumley and D. Boneh, "Remote timing attacks are practical," in *Proceedings of the 12th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2003.

[49] *Mbedtls*, https://tls.mbed.org, 2017.

[50] *Mbedtls/bignum.c at archive/mbedtls-2.3 - mbed-tls/mbedtls - github*, https://github.com/Mbed-TLS/mbedtls/blob/archive/mbedtls-2.3/library/bignum.c.

[51] P. Kocher, J. Horn, A. Fogh, *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Jose, CA, May 2019.

[52] M. Lipp, M. Schwarz, D. Gruss, *et al.*, "Meltdown: Reading kernel memory from user space," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Jul. 2018.

[53] S. van Schaik, A. Milburn, S. Österlund, *et al.*, "RIDL: Rogue in-flight data load," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[54] J. V. Bulck, M. Minkin, O. Weisse, *et al.*, "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.

[55] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Proceedings of 4th IEEE European Symposium on Security and Privacy (EuroS&P)*, Stockholm, Sweden, Jun. 2019.

[56] Intel, *L1 terminal fault / cve-2018-3615 , cve-2018-3620,cve-2018-3646 / intel-sa-00161*, https://software.intel.com/security-software-guidance/software-guidance/l1-terminal-fault.

[57] Intel, *Intel® processors voltage settings modification advisory*, https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00289.html.

[58] Intel, "Intel analysis of speculative execution side channels," 2018. [Online]. Available: https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf.

[59] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious rams," in *Journal of the ACM (JACM)*, 1996.

[60] X. Wang, H. Chan, and E. Shi, "Circuit oram: On tightness of the goldreich-ostrovsky lower bound," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS)*, Denver, Colorado, Oct. 2015.

[61] E. Stefanov, M. van Dijk, E. Shi, *et al.*, "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.

[62] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," *in ACM SIGARCH Computer Architecture News*, Nov. 2000.

[63] D. Champagne and R. B. Lee, "Scalable architectural support for trusted software," in *Proceedings of the 16th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Bangalore, India, Jan. 2010.

[64] 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.

[65] Intel, *Intel Trusted eXecution Technology (TXT)*, https://software.intel.com/content/www/us/en/develop/articles/intel-trusted-execution-technology.html?wapkw=TXT.

[66] Microsoft, *Windows 11 system requirements*, https://www.microsoft.com/en-us/windows/windows-11-specifications?r=1.

[67] 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 ACM SIGOPS Operating Systems Review (SOSR)*, vol. 42, May 2008.

[68] X. Chen, T. Garfinkel, E. C. Lewis, *et al.*, "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)*, Houston, TX, Mar. 2013.

[69] L. Zhao, H. Shuang, S. Xu, *et al.*, *Sok: Hardware security support for trustworthy execution*, 2019. [Online]. Available: https://arxiv.org/abs/1910.04957.

[70] 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.

[71] C.-c. Tsai, D. E. Porter, and M. Vij, "Graphene-sgx: A practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, Jun. 2017.

[72] S. Shinde, D. Le 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.

[73] S. Arnautov, B. Trach, F. Gregor, *et al.*, "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.

[74] Y. Shen, H. Tian, Y. Chen, *et al.*, "Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX," in *Proceedings of the 25th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, 2020.

[75] F. Schuster, M. Costa, C. Fournet, *et al.*, "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.

[76] T. T. A. Dinh, P. Saxena, E.-C. Chang, 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.

[77] O. Ohrimenko, F. Schuster, C. Fournet, *et al.*, "Oblivious multi-party machine learning on trusted processors.," in *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, 2016.

[78] 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.

[79] E. Bauman and Z. Lin, "A case for protecting computer games with SGX," in *Proceedings of the 1st Workshop on System Software for Trusted Execution (SYSTeX)*, Trento, Italy, 2016.

[80] S. Park, A. Ahmad, and B. Lee, "BlackMirror: Preventing wallhacks in 3D online FPS games," in *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Oct. 2020.

[81] Intel, *Hardening authentication tokens in web browsers using intel software guard extensions*, https://www.intel.com/content/dam/develop/external/us/en/documents/tokenbinding-whitepaper-final.pdf.

[82] Fortanix, *Fortanix: Data-first multi-cloud security*, https://fortanix.com/.

[83] Fortanix, *Nec partners with fortanix for confidential computing*, https://www.fortanix.com/resources/case-study/nec-partners-with-fortanix-for-confidential-computing.

[84] Microsoft, *Open source solutions to build enclave applications*, https://docs.microsoft.com/en-us/azure/confidential-computing/enclave-development-oss#oe-sdk.

[85] S. Sasy, S. Gorbunov, and C. W. Fletcher, "Zerotrace: Oblivious memory primitives from Intel SGX," in *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2018.

[86] D. V. Le, L. T. Hurtado, A. Ahmad, M. Minaei, B. Lee, and A. Kate, "A tale of two trees: One writes, and other reads. optimized oblivious accesses to large-scale blockchains," in *Proceedings of the Privacy Enhancing Technologies Symposium (PETS)*, Virtual Event, 2020.

[87] 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, Jun. 2018.

[88] S. Shinde, Z. 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.

[89] 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 27th USENIX Security Symposium (Security)*, Vancouver, BC, 2017.

[90] Z. Zhou, M. K. Reiter, and Y. Zhang, "A software approach to defeating side channels in last-level caches," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[91] T. Kim, M. Peinado, and G. Mainar-Ruiz, "Stealthmem: System-level protection against cache-based side channel attacks in the cloud," in *Proceedings of the 21st USENIX Security Symposium (Security)*, Bellevue, WA, Aug. 2012.

[92] 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.

[93] A. Ahmad, B. Joe, Y. Xiao, Y. Zhang, I. Shin, and B. Lee, "Obfuscuro: A commodity obfuscation engine for Intel SGX," in *Proceedings of the 2019 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2019.

[94] M. Orenbach, A. Baumann, and M. Silberstein, "Autarky: Closing controlled channels with self-paging enclaves," in *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)*, Virtual Event, 2020.

[95] S. Aga and S. Narayanasamy, "Invisipage: Oblivious demand paging for secure enclaves," in *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Phoenix, AZ, 2019.

[96] 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.

[97] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)*, Shanghai, China, Oct. 2017.

[98] E. Feng, X. Lu, D. Du, *et al.*, "Scalable memory protection in the PENGLAI enclave," in *Proceedings of the 15th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Virtual Event, Jul. 2021.

[99] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the 15th ACM European Conference on Computer Systems (EuroSys)*, Virtual Event, 2020.

[100] K. Nayak, C. Fletcher, L. Ren, *et al.*, "Hop: Hardware makes obfuscation practical," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2017.

[101] C. W. Fletcher, M. v. Dijk, and S. Devadas, "A secure processor architecture for encrypted computation on untrusted programs," in *Proceedings of the 7th ACM workshop on Scalable trusted computing (STC)*, ACM, Raleigh, NC, Oct. 2012.

[102] M. Maas, E. Love, E. Stefanov, *et al.*, "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.

[103] D. Skarlatos, Z. N. Zhao, R. Paccagnella, C. W. Fletcher, and J. Torrellas, "Jamais vu: Thwarting microarchitectural replay attacks," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual Event, 2021.

[104] D. Evtyushkin, T. Benjamin, J. Elwell, J. A. Eitel, A. Sapello, and A. Ghosh, "Computing with time: Microarchitectural weird machines," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual Event, 2021.

[105] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka City, Japan, Oct. 2018.

[106] C. Sakalis, S. Kaxiras, A. Ros, A. Jimborean, and M. Själander, "Efficient invisible speculative execution through selective delay and value prediction," in *Proceedings of the 46th ACM/IEEE International Symposium on Computer Architecture (ISCA)*, Phoenix, AZ, 2019.

[107] Q. Tan, Z. Zeng, K. Bu, and K. Ren, "Phantomcache: Obfuscating cache conflicts with localized randomization," in *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS)*, Virtual Event, 2021.

[108] G. Dessouky, T. Frassetto, and A.-R. Sadeghi, "Hybcache: Hybrid side-channel-resilient caches for trusted execution environments," in *Proceedings of the 29th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2020.

[109] J. Behrens, A. Cao, C. Skeggs, A. Belay, M. F. Kaashoek, and N. Zeldovich, "Efficiently mitigating transient execution attacks using the unmapped speculation contract," in *Proceedings of the 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov. 2020.

[110] 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.

[111] 01org, *Intel(r) software guard extensions for linux* os (source code)*, https://github.com/01org/linux-sgx, 2016.

[112] C.-C. Tsai, K. S. Arora, N. Bandi, *et al.*, "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.

[113] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (Oakland)*, Virtual Event, May 2020.

[114] Y. Xiao, X. Zhang, Y. Zhang, and R. Teodorescu, "One bit flips, one cloud flops: Cross-VM row hammer attacks and privilege escalation," in *Proceedings of the 25th USENIX Security Symposium (Security))*, Austin, TX, Aug. 2016.

[115] R. K. Konoth, M. Oliverio, A. Tatar, *et al.*, "ZebRAM: comprehensive and compatible software protection against rowhammer attacks," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[116] S. Gueron, *A memory encryption engine suitable for general purpose processors*, Cryptology ePrint Archive, Report 2016/204, https://eprint.iacr.org/2016/204, 2016.

[117] 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.

[118] 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.

[119] J. Lee, J. Jang, Y. Jang, *et al.*, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proceedings of the 26th USENIX Security Symposium (Security)*, Vancouver, BC, Aug. 2017.

[120] D. Kuvaiskii, O. Oleksenko, S. Arnautov, *et al.*, "Sgxbounds: Memory safety for shielded execution," in *Proceedings of the 12th ACM European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, Apr. 2017.

[121] Signal, *Signal » home*, https://signal.org, 2019.

[122] Signal, *Technology preview: Private contact discovery for signal*, https://signal.org/blog/private-contact-discovery/, 2019.

[123] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review (SOSR)*, 1994.

[124] LLVM, *The LLVM compiler infrastructure*, 2016. [Online]. Available: http://llvm.org/.

[125] 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.

[126] Uwe F. Mayer, *Linux/unix nbench*. [Online]. Available: https://www.math.utah.edu/~mayer/linux/bmark.html.

[127] Proton AG, *General data protection regulation compliance guidelines*, https://gdpr.eu.

[128] D. S. Wishart, C. Knox, A. C. Guo, *et al.*, "Drugbank: A comprehensive resource for in silico drug discovery and exploration," *Nucleic Acids Research*, vol. 34, 2006.

[129] OPSWAT, *MetaDefender Cloud*, https://metadefender.opswat.com/?lang=en.

[130] VirusTotal, *Virustotal*, https://www.virustotal.com/gui/home/upload.

[131] Jotti, *Jotti malware scan*, https://virusscan.jotti.org/en-US/scan-file.

[132] Amazon, *Amazon personalize*, https://aws.amazon.com/personalize/.

[133] B. Yee, D. Sehr, G. Dardyk, *et al.*, "Native client: A sandbox for portable, untrusted x86 native code," in *Proceedings of the 30th IEEE Symposium on Security and Privacy (Oakland)*, Oakland, CA, May 2009.

[134] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, May 2019.

[135] J. Lind, C. Priebe, D. Muthukumaran, *et al.*, "Glamdring: Automatic application partitioning for Intel SGX," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, 2017.

[136] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *Proceedings of 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.

[137] M. Abubakar, A. Ahmad, P. Fonseca, and D. Xu, "Shard: Fine-grained kernel specialization with context-aware hardening," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[138] LLVM, *Writing an llvm backend*, http://llvm.org/docs/WritingAnLLVMBackend.html, 2017.

[139] Intel, *Linux-sgx/sdk/tlibc/*, https://github.com/intel/linux-sgx/tree/master/sdk/tlibc.

[140] Musl-Libc, *Musl-libc*, https://www.musl-libc.org, 2017.

[141] Capstone, *Capstone. the ultimate disassembler*, http://www.capstone-engine.org, 2017.

[142] 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.

[143] 01org, *Intel(r) software guard extensions for linux* os (linux sgx driver)*, https://github.com/01org/linux-sgx-driver, 2016.

[144] J. Seo, B. Lee, S. Kim, *et al.*, "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.

[145] W. Zhao, K. Lu, Y. Qi, and S. Qi, "MPTEE: bringing flexible and efficient memory protection to Intel SGX," in *Proceedings of 12th ACM European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, Apr. 2020.

[146] *Perf wiki*, https://perf.wiki.kernel.org/index.php/Main_Page.

[147] P. Warden, *C_hashmap*, https://github.com/petewarden/c_hashmap.

[148] OSSEC, *Ossec - world's most widely used host intrusion detection system - hids*, http://ossec.github.io, 2017.

[149] Ghamrouni, *Recommender is a c library for product recommendations/suggestions using collaborative filtering (cf)*, 2016. [Online]. Available: http://ghamrouni.github.io/Recommender/index.html.

[150] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "ShieldStore: shielded in-memory key-value storage with SGX," in *Proceedings of the 14th European Conference on Computer Systems (EuroSys)*, Dresden, Germany, 2019.

[151] SNORT, *Snort - Network Intrusion Detection & Prevention System*, 2016. [Online]. Available: https://www.snort.org/.

[152] *Engineering new protections into hardware.* [Online]. Available: %5Chowpublished%7Bhttps://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html%7D.

[153] ClamAV, *ClamAVNet*, https://www.clamav.net/, 2018.

[154] DrugBank, *Drugbank online*, 2017. [Online]. Available: http://www.drugbank.ca.

[155] memcached, *Memcached - a distributed memory object caching system*, https://memcached.org/.

[156] Phoronix, *Intel MPX support is dead with Linux 5.6*, https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead.

[157] Phoronix, *Intel MPX support removed from GCC 9*, https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9.

[158] Apache, *Apache hadoop project*, 2017. [Online]. Available: http://hadoop.apache.org.

[159] R. Sinha, M. Costa, A. Lal, *et al.*, "A design and verification methodology for secure isolated regions," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2016.

[160] Ú. Erlingsson, M. Abadi, M. Vrable, M. Budiu, and G. C. Necula, "XFI: software guards for system address spaces," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, Nov. 2006.

[161] M. Castro, M. Costa, J.-P. Martin, *et al.*, "Fast byte-granularity software fault isolation," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[162] Y. Mao, H. Chen, D. Zhou, X. Wang, N. Zeldovich, and M. F. Kaashoek, "Software fault isolation with API integrity and multi-principal modules," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[163] D. Sehr, R. Muth, C. Biffle, *et al.*, "Adapting software fault isolation to contemporary CPU architectures," in *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[164] B. Barak, O. Goldreich, R. Impagliazzo, *et al.*, "On the (im)possibility of obfuscating programs," *in Journal of the ACM (JACM)*, vol. 59, May 2012.

[165] S. Hada, "Zero-knowledge and code obfuscation," in *Proceedings of the 6th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Kyoto, Japan, Dec. 2000.

[166] N. Bitansky, R. Canetti, S. Goldwasser, S. Halevi, Y. T. Kalai, and G. N. Rothblum, "Program obfuscation with leaky hardware," in *Proceedings of the 17th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Seoul, South Korea, Dec. 2011.

[167] K.-M. Chung, J. Katz, and H.-S. Zhou, "Functional encryption from (small) hardware tokens," in *Proceedings of the 19th International Conference on the Theory and Application of Cryptology and Information Security (ASIACRYPT)*, Bangalore, India, Dec. 2013.

[168] N. Döttling, T. Mie, J. Müller-Quade, and T. Nilges, "Basing obfuscation on simple tamper-proof hardware assumptions," *In IACR Cryptology ePrint Archive*, 2011.

[169] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *Proceedings of the 7th IACR Theory of Cryptography Conference (TCC)*, Zurich, Switzerland, Feb. 2010.

[170] Intel, "Overview: Intrinsics for intel (r) advanced vector extensions 2 (intel(r) avx2) instructions," 2014. [Online]. Available: https://software.intel.com/en-us/node/523876.

[171] Intel, *Intel(r) advanced vector instructions 512 (AVX-512) overview*, https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

[172] Intel, "Intel (r) core (tm) i9-7980xe extreme edition processor," 2017. [Online]. Available: https://ark.intel.com/products/126699/Intel-Core-i9-7980XE-Extreme-Edition-Processor-24%5C_75M-Cache-up-to-4%5C_20-GHz.

[173] Intel, "Intel (r) core (tm) i9-7970x processor," 2017. [Online]. Available: https://ark.intel.com/products/126697/Intel-Core-i9-7960X-X-series-Processor-22M-Cache-up-to-4%5C_20-GHz.

[174] S. Gueron, *A memory encryption engine suitable for general purpose processors*, Cryptology ePrint Archive, 2016. [Online]. Available: https://eprint.iacr.org/2016/204.

[175] OpenSSL Software Foundation, *OpenSSL: Cryptography and SSL/TLS Toolkit*, https://www.openssl.org/, 2017.

[176] N. Döttling, T. Mie, J. Müller-quade, and T. Nilges, "Basing obfuscation on simple tamper-proof hardware assumptions.," in *IACR Cryptology ePrint Archive*, 2011.

[177] V. Goyal, Y. Ishai, A. Sahai, R. Venkatesan, and A. Wadia, "Founding cryptography on tamper-proof hardware tokens," in *Proceedings of the 7th IACR Theory of Cryptography Conference (TCC)*, D. Micciancio, Ed., Zurich, Switzerland, Feb. 2010.

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

[179] R. A. Popa, C. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, Oct. 2011.

[180] O. Goldreich, "Cryptography and cryptographic protocols," *Distributed Computing*, 2003.

[181] J. Thaler, *Verifiable computing: Between theory and practice*, https://people.eecs.berkeley.edu/~alexch/docs/pcpip_thaler.pdf, 2017.

[182] Y. Sui and J. Xue, "SVF: interprocedural static value-flow analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CCC)*, Barcelona, Spain, Mar. 2016.

[183] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: Protecting sensitive data leaks using application-level taint tracking," *in ACM SIGOPS Operating Systems Review (SOSR)*, vol. 45, 2011.

[184] A. Ahmad, S. Lee, P. Fonseca, and B. Lee, "Kard: Lightweight data race detection with per-thread memory protection," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Virtual Event, Apr. 2021.

[185] Intel, *Intel sgx and side-channels*, https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sgx-and-side-channels.html.

[186] A. Baumann, "Hardware is the new software," in *Proceedings of the 16th Workshop on Hot Topics in Operating Systems (HotOS)*, Whistler, BC, May 2017.

[187] SPEC, *SPEC CPU 2006*, https://www.spec.org/cpu2006/.

[188] Redis, *Redis*, https://redis.io/.

[189] NGINX, *Advance load balancer, web server, and reverse proxy - nginx*, https://www.nginx.com/.

[190] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: Exitless OS services for SGX enclaves," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, Belgrade, Serbia, Apr. 2017.

[191] J. Criswell, N. Dautenhahn, and V. Adve, "Virtual ghost: Protecting applications from hostile operating systems," in *Proceedings of the 19th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Salt Lake City, Utah, 2014.

[192] X. Dong, Z. Shen, J. Criswell, A. L. Cox, and S. Dwarkadas, "Shielding software from privileged side-channel attacks," in *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2018.

[193] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *Proceedings of the 24th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Providence, RI, 2019.

[194] Q. Ge, Y. Yarom, T. Chothia, and G. Heiser, "Time protection: The missing OS abstraction," in *Proceedings of the 14th ACM European Conference on Computer Systems (EuroSys)*, Dresden, Germany, Mar. 2019.

[195] F. Liu, Q. Ge, Y. Yarom, *et al.*, "Catalyst: Defeating last-level cache side channel attacks in cloud computing," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Barcelona, Spain, Mar. 2016.

[196] Microsoft, *Intel SGX based confidential computing VMs now available on Azure Dedicated Hosts*, https://azure.microsoft.com/en-us/updates/intel-sgx-based-confidential-computing-vms-now-available-on-azure-dedicated-hosts/.

[197] Packet, *Packet platform features*, https://www.packet.com/cloud/features/.

[198] AnandTech, *AMD demonstrates stacked 3d v-cache technology: 192 mb at 2 tb/sec*, https://www.anandtech.com/show/16725/amd-demonstrates-stacked-vcache-technology-2-tbsec-for-15-gaming.

[199] Electronic Design, *Intel proposes new path for moore's law with 3d stacked transistors*, https://www.electronicdesign.com/technologies/embedded-revolution/article/21183706/electronic-design-intel-proposes-new-path-for-moores-law-with-3d-stacked-transistors.

[200] Intel, *Product brief 11th gen intel(r) core(tm) desktop processors*, https://newsroom.intel.com/wp-content/uploads/sites/11/2021/03/11th-gen-product-brief.pdf.

[201] Intel, *Speculative execution side-channel mitigations*, https://www.intel.com/content/dam/develop/external/us/en/documents/336996-speculative-execution-side-channel-mitigations.pdf.

[202] Intel, *Introduction to Cache Allocation Technology in the Intel(r) Xeon(r) Processor E5 v4 Family*, https://software.intel.com/content/www/us/en/develop/articles/introduction-to-cache-allocation-technology.html.

[203] O. Oleksenko, D. Kuvaiskii, P. Bhatotia, P. Felber, and C. Fetzer, "Intel MPX explained: A cross-layer analysis of the Intel MPX system stack," in *Proceedings of the ACM Conference on Measurement and Analysis of Computing Systems (SIGMETRICS)*, Irvine, CA, Jun. 2018.

[204] *Personal correspondence with Intel*, Mar. 2021.

[205] The kernel development community, *18. Microarchitectural Data Sampling (MDS) mitigation – The Linux Kernel Documentation*, https://www.kernel.org/doc/html/latest/x86/mds.html.

[206] *L1TF - L1 Terminal Fault*, https://projectacrn.github.io/latest/developer-guides/l1tf.html.

[207] Bareflank, *Bareflank/hypervisor*, https://github.com/Bareflank/hypervisor.

[208] Intel, *Switchless enclave example*, https://github.com/intel/linux-sgx/tree/master/SampleCode/Switchless.

[209] Techspot, *Intel's SGX deprecation impacts DRM and ultra hd blu-ray support*, https://www.techspot.com/news/93006-intel-sgx-deprecation-impacts-drm-ultra-hd-blu.html.

[210] Wikipedia, *Tickless kernel*, https://en.wikipedia.org/wiki/Tickless_kernel.

[211] J. Dean, *Latency numbers every programmer should know*, https://gist.github.com/jboner/2841832.

[212] Lighttpd, *Home - lighttpd - fly light*, https://www.lighttpd.net/.

[213] B. Aker, *Memaslap - load testing and benchmarking a server*, http://docs.libmemcached.org/bin/memaslap.html.

[214] A. C. Aldaya, B. B. Brumley, S. ul Hassan, C. Pereida García, and N. Tuveri, "Port contention for fun and profit," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[215] Z. Wang and R. B. Lee, "Covert and side channels due to processor architecture," in *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, Miami, FL, Dec. 2006.

[216] S. Aga and S. Narayanasamy, "Invisimem: Smart memory defenses for memory bus side channel," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, Toronto, ON, Canada, 2017.

[217] R. Paccagnella, L. Luo, and C. W. Fletcher, "Lord of the ring(s): Side channel attacks on the CPU on-chip ring interconnect are practical," in *Proceedings of the 30th USENIX Security Symposium (Security)*, Virtual Event, Aug. 2021.

[218] Intel, *New Intel (r) mesh architecture: The 'superhighway' of the data center*, https://silix.com.br/pdf/Intel/Intel_Mesh_Whitepaper.pdf.

[219] M. Yan, R. Sprabery, B. Gopireddy, C. Fletcher, R. Campbell, and J. Torrellas, "Attack directories, not caches: Side channel attacks in a non-inclusive world," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2019.

[220] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer, "Dawg: A defense against cache timing attacks in speculative execution processors," in *Proceedings of the 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka City, Japan, Oct. 2018.

[221] Google, *Chromium blog: Native client support on Arm*, https://blog.chromium.org/2013/01/native-client-support-on-arm.html.

[222] Arm, *Conditional execution in Arm state*, https://developer.arm.com/documentation/dui0473/c/condition-codes/conditional-execution-in-arm-state?lang=en.

[223] H. Oh, A. Ahmad, S. Park, B. Lee, and Y. Paek, "Trustore: Side-channel resistant storage for SGX using Intel hybrid CPU-FPGA," in *Proceedings of the 2020 ACM Conference on Computer and Communications Security (CCS)*, Virtual Event, Oct. 2020.

[224] A. Ahmad, S. Lee, and M. Peinado, "Hardlog: Practical tamper-proof system auditing using a novel audit device," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[225] S. Volos, K. Vaswani, and R. Bruno, "Graviton: Trusted execution environments on GPUs," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Carlsbad, CA, Oct. 2018.

[226] Z. Zhou, V. D. Gligor, J. Newsome, and J. M. McCune, "Building verifiable trusted path on commodity x86 computers," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2012.

[227] L. Zhou, X. Ding, and F. Zhang, "Smile: Secure memory introspection for live enclave," in *Proceedings of the 43rd IEEE Symposium on Security and Privacy (Oakland)*, San Francisco, CA, May 2022.

[228] Intel, *Intel trust domain extensions*, https://software.intel.com/content/dam/develop/external/us/en/documents/tdx-whitepaper-final9-17.pdf.

[229] AMD, *AMD SEV-SNP: strengthening VM isolation with integrity protection and more*, https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.