# CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs

Adil Ahmad† Juhee Kim§ Jaebaek Seo* Insik Shin‡ Pedro Fonseca† Byoungyoung Lee§
†*Purdue University* §*Seoul National University* **Google* ‡*KAIST*

***Abstract*—Intel SGX aims to provide the confidentiality of user data on untrusted cloud machines. However, applications that process confidential user data may contain bugs that leak information or be programmed maliciously to collect user data. Existing research that attempts to solve this problem does not consider multi-client isolation in a single enclave. We show that by not supporting such in-enclave isolation, they incur considerable slowdown when concurrently processing multiple clients in different enclave processes, due to the limitations of SGX.**

**This paper proposes CHANCEL, a sandbox designed for multi-client isolation within a single SGX enclave. In particular, CHANCEL allows a program's threads to access both a per-thread memory region and a shared *read-only* memory region while servicing requests. Each thread handles requests from a single client at a time and is isolated from other threads, using a Multi-Client Software Fault Isolation (MCSFI) scheme. Furthermore, CHANCEL supports various in-enclave services such as an in-memory file system and shielded client communication to ensure complete mediation of the program's interactions with the outside world. We implemented CHANCEL and evaluated it on SGX hardware using both micro-benchmarks and realistic target scenarios, including private information retrieval and product recommendation services. Our results show that CHANCEL outperforms a baseline multi-process sandbox by $4.06 - 53.70\times$ on micro-benchmarks and $0.02 - 21.18\times$ on realistic workloads while providing strong security guarantees.**

## I. INTRODUCTION

Intel SGX guarantees the confidentiality and integrity of a program without trusting software components such as the OS or hypervisor, and protects against specific hardware attacks [1]. SGX also supports remote attestation to ensuring that programs are correctly installed on remote machines. Thus, SGX is a promising candidate to ensure the confidentiality of sensitive data running on remote machines in the cloud.

However, the traditional assumption behind SGX's implementation is that the program running inside an enclave is trusted. 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 service providers. For example, consider a popular messaging platform, Signal [2], which supports private contact discovery [3]. 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.

Existing research concerning client data protection in remote machines (e.g., Ryoan [4], VC3 [5]) does not support *multi-client* isolation within a single enclave. Considering the Signal example, the application could be designed such that multiple threads (handling requests from individual clients) directly share a contact database provided by Signal. However, lacking such support, prior approaches require a dedicated enclave process for each client. Although lacking a single process multi-client sandbox is not critical in non-SGX environments, it is a significant drawback in SGX environments. In particular, the lack of secure memory sharing between enclaves and limited enclave memory (i.e., only $128 - 256$ MB), results in inefficient use of the limited memory and prohibitive overheads when different processes have distinct copies of shared data.

To further elaborate, SGX does not allow the sharing of enclave pages between different enclave processes; therefore, common data must either be shared through untrusted interfaces (e.g., OS-controlled IPC), which is insecure, or be cloned to each enclave process. However, since SGX has limited trusted memory, cloning common data for each process/enclave would waste precious memory resources and eventually slowdown the entire system due to the expensive page-swaps between trusted and untrusted memory [6]. Our experiments on a real-world product recommendation application suggest that under a four client scenario with 112 MB of product catalog memory, a multi-process sandbox (i.e., the catalog is cloned to 4 processes) incurs almost $20\times$ more page-swaps, and an overhead of more than $17\times$, compared to a multi-client sandbox (i.e., the catalog

*The author worked on this project as a PhD student at KAIST.

Network and Distributed Systems Security (NDSS) Symposium 2021
21-24 February 2021
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24057
www.ndss-symposium.org

is shared by 4 threads) (§VIII-C).

This paper proposes CHANCEL[1], a multi-client sandbox that enables multiple threads to securely and efficiently handle requests from different clients, within a single enclave. CHANCEL relies on a novel Multi-Client SFI (MCSFI) scheme to enable *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.

Existing SFI techniques are not designed for multi-client scenarios and require advanced hardware features to implement in SGX enclaves. In particular, Native Client (NaCl) [7] considers the program's data to belong to a single client; therefore, it provisions a single memory view for each thread and does not support thread isolation. Furthermore, Multi-Domain SFI [8, 9] supports thread isolation but lacks shared memory (or its enforcement) between the threads, crucial to ensure efficient memory usage, and hence high performance for SGX enclaves. Finally, SGX implementations of SFI techniques [4, 9, 10] require hardware features that are not widely available, e.g., SGX2 [11], or have been discontinued, e.g., Memory Protection eXtensions (MPX) [12, 13], significantly hindering deployment.

Therefore, to enable efficient multi-client separation, we propose Multi-Client SFI (MCSFI), inspired by Native Client [7]. In particular, MCSFI allows each thread to access two memory regions: (a) per-thread *private* data region, (b) *shared* read-only region for all threads of the program. Each thread stores sensitive data obtained from a single client in its private region, inaccessible to other threads. Furthermore, all threads can access a common region to share non-sensitive data (e.g., Signal's contact database). However, to ensure that malicious threads cannot abuse the shared region to leak their sensitive contents or tamper with the service, MCSFI ensures read-only enforcement of the shared region, except during initialization or after servicing all client requests.

CHANCEL enforces a specific enclave memory layout and uses compiler instrumentation to enforce MCSFI without requiring advanced or discontinued hardware functions. Specifically, CHANCEL reserves two general-purpose registers (i.e., `r14` and `r15`) to hold executable and accessible addresses for each thread to restrict the program's control-flow and data-flow depending on the execution stage and thread context. For example, CHANCEL's data-flow instrumentation ensures that the program can write to the shared memory region, during initialization, and only write to a per-thread private region, while serving client requests. CHANCEL also instruments the program's control-flow to ensure it can only execute the target program's code and cannot bypass the instrumentation checks, thereby preventing attacks that rely on binary rewriting or arbitrary code execution. Our evaluation case studies (§VIII-C) show that MCSFI is applicable to many classes of real-world scenarios, including private information retrieval and product recommendation services.

To ease program development for CHANCEL, we implement a compiler toolchain using LLVM [14]. The service provider must utilize CHANCEL's toolchain to build an instrumented binary, which is loaded into CHANCEL's enclave after instrumentation validation using an x86 disassembler, `capstone` [15]. During program execution, CHANCEL provides various services, i.e., dynamic memory allocation, file system, and shielded client communication, and also protects against covert channel attacks (e.g., encoding information in the output size). Our evaluated programs required no manual changes to build using our toolchain and few additional code (less than 10 lines) to utilize CHANCEL's runtime services.

We evaluate CHANCEL on SGX hardware to assess its security impact and performance. Our security evaluation 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.

Furthermore, our evaluation shows that CHANCEL has significantly higher performance than the baseline secure multi-process sandbox approach (e.g., Ryoan [4]). 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—intrusion detection systems (OSSEC [16] and Snort [17]), private information retrieval (DrugBank database [18, 19] and ShieldStore [20] key-value store), and product recommendation systems (Recommender [21]). Finally, the average overhead of CHANCEL compared to a native execution is only $12.43\%$ on the nbench benchmark [22], which demonstrates CHANCEL's applicability to a wide range of scenarios with only modest overheads.

## II. BACKGROUND ON INTEL SGX

Intel SGX [23] allows a user process to create a protected memory region in its virtual address space, called an *enclave*. Privileged software, including the OS and hypervisor, is prevented by the hardware from accessing the enclave's runtime execution context and memory. The enclave can arbitrarily access any memory location within its process's unprotected memory. This section provides a brief overview of the critical SGX aspects that are relevant to this work and defers the reader to other sources for more comprehensive information regarding SGX [23, 24].

**Remote attestation.** The user can attest that their programs are correctly loaded onto SGX enclaves using remote attestation. In particular, the CPU creates a SHA-256 digest of the enclave memory, signed using the CPU's secret key, and sent to the user for verification.

**Memory management.** SGX dedicates a portion of the physical memory, Enclave Page Cache (EPC), to store enclave pages. The EPC memory is limited—128 and 256 MB for SGX1 and SGX2, respectively. If this memory is exhausted, the Linux OS can perform page-swaps to extend the enclave memory on page faults. However, these page-swaps are expensive due to the encryption of enclave pages and context switch overheads (e.g., TLB and L1D flushes) [6, 25]. Finally, concerning SGX1, the

[1]CHANCEL is the part of a church near the altar that is reserved for clergy and choir, and separated from the main hall

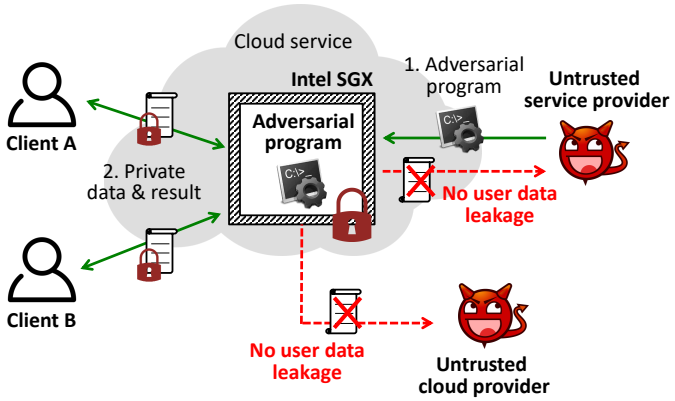



**Fig. 1:** CHANCEL's system model with three participants: the clients, the service provider, and the cloud provider. CHANCEL must prevent the service provider's adversarial program that uses multi-threading to concurrently serve many clients, from leaking confidential client data.

entire enclave memory must be statically allocated at compile-time, but SGX2 permits dynamic allocation of enclave memory during execution.

**Multi-threading.** SGX supports multi-threaded execution in an enclave, but the maximum number of parallel threads must be specified in the enclave's configuration file.

## III. MOTIVATION

This section provides an overview of the multi-client system model that CHANCEL is concerned with (§III-A), enumerates examples of critical services (e.g., private information retrival and product recommendation services) that are relevant to the system model (§III-B), and discusses the limitations of existing approaches concerning multi-client scenarios (§III-C).

### A. System Model

This paper 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 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 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). The rationale behind such an adversary is that, 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 [26]).

- **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.** An *optional* third entity, the cloud provider, may exist if the service provider rents hardware from third-parties such as Microsoft Azure [27]. 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.

### B. Examples of Target Scenarios

Many critical real-world scenarios follow this paper's system model (§III-A), 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 [18]) 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 (mentioned in §I), navigation services (e.g., Google Maps), 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 [28–30] 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 [31] 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.

| System | Scope | | Multi-client | | Requirements |
|---|---|---|---|---|---|
| | Adversarial program | Unintended bugs | Thread isolation | Shared mem. and enforce. | |
| Ryoan [4] | ✓ | ✓ | ✗ | ✗ | SGX2 |
| Occlum [9] | ✗ | ✓ | ✓ | ✗ | SGX1/SGX2 + MPX |
| MPTEE [10] | ✗ | ✓ | ✗ | ✓ | SGX1/SGX2 + MPX |
| CHANCEL | ✓ | ✓ | ✓ | ✓ | SGX1/SGX2 |

**TABLE I:** A comparison between CHANCEL and closely related schemes. Please refer to §X for a discussion on other SGX permission enforcement schemes. For Occlum [9] and MPTEE [10], both SGX and MPX are required hardware features.

### C. Limitations of Existing Approaches

The effective handling of multi-client scenarios in adversarial programs requires enforcing restrictions during execution. For example, each thread must have a different view over the enclave memory to prevent data extraction between threads. Hence, the closest related work to CHANCEL are SGX systems that use SFI for in-enclave permission enforcement [4, 9, 10]. However, none of the existing work can efficiently tackle multi-client scenarios (§III-A) because they suffer from limited protection scope and inefficient multi-client support. Furthermore, existing approaches require uncommon hardware features that are either discontinued [12, 13] or not widely-supported [11]; therefore, they are challenging to deploy. Table I compares CHANCEL and relevant SFI implementations in SGX.

**Limited protection scope.** Several existing systems [9, 10] do not consider adversarial programs. In particular, Occlum [9] runs untrusted application binaries in isolated threads of a single enclave while MPTEE [10] provides general-purpose memory protection for SGX enclaves. Both systems consider the program's code and data to belong to the same entity; therefore, they only prevent attacks involving untrusted inputs and unintended bugs (e.g., buffer overflows) in the program. Hence, they do not prevent an adversarial program from divulging sensitive client data through direct disclosure (e.g., transmitting information outside the enclave) or covert channels.

**Inefficient multi-client support.** None of the existing systems considers multi-client scenarios in a *single* enclave. In particular, Ryoan [4] and MPTEE [10] lack thread isolation; which requires multiple clients to be inefficiently isolated in different enclaves. While Occlum [9] supports thread isolation, it does not (a) allow directly shared memory (i.e., shared global or heap objects) and (b) enforce permissions on indirectly shared memory (i.e., shared files) between threads. On the one hand, lacking directly shared memory is contrary to general multi-threaded program development patterns, also evident from our evaluated programs (§VIII-C). On the other hand, lacking enforcement on indirectly shared memory allows malicious threads to leak sensitive client data or compromise the service for other clients (e.g., overwrite contained data).

Therefore, considering existing systems, commonly shared data (e.g., a database) must be cloned to each enclave process or thread. Since EPC memory is limited (i.e., $128 - 256$ MB), CHANCEL's secure multi-client sandbox outperforms such schemes by upto $53.70\times$ (refer to §VIII-A), without compromising on its security goals.

**Substantial hardware requirements.** All related systems that implement SFI in SGX enclaves require advanced hardware features, in particular SGX2 and MPX (Memory Protection eXtensions). Such requirements are likely to hinder deployment. In particular, SGX2 (required by Ryoan [4]) is supported by few machines and currently lacks cloud deployment [11]. In comparison, the original SGX1 is supported by all Intel desktop machines from Skylake onwards and is deployed in all major public clouds [27, 32–34]. Furthermore, MPX support (required by Occlum [9] and MPTEE [10]) has been discontinued from Linux 5.6 [13] and GCC 9 [12]. Therefore, building enclave programs with MPX is no longer possible using the latest Linux kernel and GCC compiler. Finally, Intel Memory Protection Keys (MPK) [35], can provide protections similar to MPX but requires kernel privileges.

## IV. THREAT MODEL

CHANCEL prevents the leakage of client data from a remote program that uses different threads to serve different client (i.e., one thread services only one client). To this end, the program and all its executing threads are considered malicious. In practice, they can be malicious because: (a) the service provider deliberately wrote the program to extract users' confidential data, or (b) the cloud provider or third-party hackers gain control of the program (e.g., using ROP attacks [36, 37]). However, we assume that a malicious thread will not provide the wrong output since clients could catch such actions, but will attempt to leak client data through the following strategies.

1) **Leak its contained confidential data.** Each thread contains data from its respective client, which is confidential. The thread can try to leak this data by (a) writing outside its thread context, (b) writing to the shared region between threads, and (c) passing some information through covert channels (e.g., system-calls).
2) **Leak another thread's confidential data.** A malicious thread can try to leak sensitive data belonging to *other* threads of the program. In addition to all the leakage techniques mentioned above, a malicious thread can try to leak such data by sending it to its connected client, who may be distinct from the client who provided the victim thread's confidential data.

**Out-of-scope.** SGX does not guarantee the availability of the service. The program can choose not to service a user's request, or the enclave might not be scheduled by a malicious OS.

Digital side-channels [38–42] and micro-architectural defects [43–45] are outside this paper's scope. A malicious thread can also abuse these issues to covertly pass information outside the enclave. CHANCEL should be used together with existing countermeasures [46–50] and the latest patched processors [51] to address these hardware limitations. We briefly discuss digital side-channels and micro-architectural defects in §IX. Other side-channels, e.g., power monitoring, electromagnetic attacks, and hardware snooping, are also beyond our scope since they are generally costly to launch and require physical access.

While CHANCEL addresses covert channels due to software interfaces such as system calls and network communication, there are some covert channels that it cannot adequately protect, e.g., the execution time of the enclave. CHANCEL mitigates information leakage through this channel by ensuring that all output is sent back at fixed intervals and not in between. The

issue can be further addressed by normalizing the execution time of programs [49, 52]. Nevertheless, this is a low bandwidth channel, i.e., once every request.

## V. Design

This section begins with an overview of CHANCEL (§V-A) and an operation workflow related to the service provider and clients (§V-B). Then, it describes CHANCEL's implementation of Multi-Client SFI (MCSFI) within an SGX enclave (§V-C). Finally, it details shared data initialization mechanisms (§V-D), and CHANCEL's runtime services (§V-E).

### A. 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 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).

### B. 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 2).

**Step-1. Agreement (Offline).** Both parties, a service provider and a client, inspect and agree on the implementational details

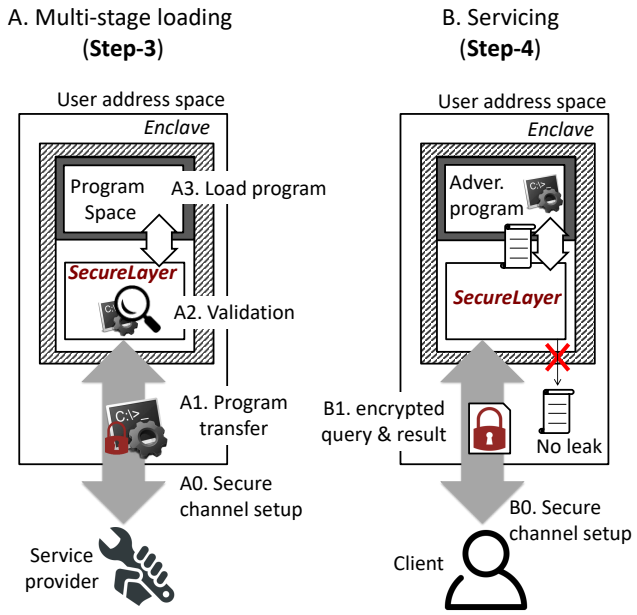



**Fig. 2:** Overall workflow of CHANCEL. A) An enclave containing SecureLayer is created, which then validates and loads a program provided by the service provider. B) The runtime behavior of the program is restricted, and SecureLayer mediates all interactions originating from it to avoid any security threat.

of SecureLayer. In particular, they ensure that the source code of SecureLayer[2] 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 (§V-C) 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

[2]To be made available for inspection

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 §V-E 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.

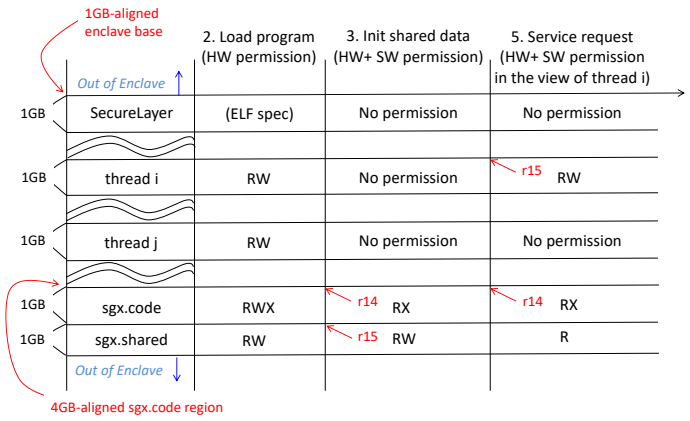
### C. Multi-Client SFI (MCSFI)

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.

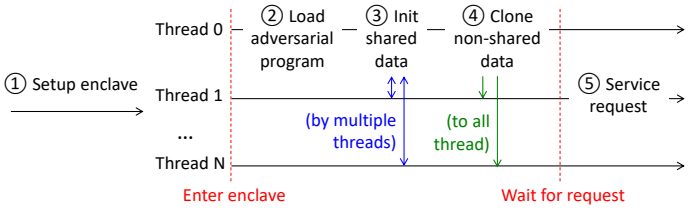
**Memory layout.** Apart from an enclave's native memory segments (e.g., `.code`, `.bss`), MCSFI requires the following additional segments: (a) a *private* region dedicated to each thread, (b) a *shared* region available to all threads, and (c) an extra executable region. In particular, the private 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 region holds the code of the target program. Based on these requirements, CHANCEL sets up a suitable enclave layout.

Figure 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 §V-C-(c)).

Similarly to other SGX SFI schemes [4, 9], 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.




**(a)** Memory Layout




**(b)** Timeline

**Fig. 3:** CHANCEL's memory layout and permissions enforced during 5 stages of its execution.

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 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. In real-world scenarios though, CHANCEL does not need to be configured to support many threads because the latest SGX processor only supports up to 10 hardware threads [53].

**Timeline and permissions.** Figure 3b shows the timeline of CHANCEL's execution and the permissions enforced at various execution stages are shown in Figure 3a.

Before loading the target program (①), only hardware permissions are enforced in the enclave (second column of Figure 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 §V-D). 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*,

```
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 ...
```

**Fig. 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`). It is assumed that `r13` is an available register (or spilled beforehand) and thus used as a temporary register.

```
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
```

**Fig. 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.

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 §V-E) (⑤). At this stage, CHANCEL enforces permissions (illustrated in the last column of Figure 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 (§V-C-(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) the base address of shared region

```
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
```

**Fig. 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
```

**Fig. 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 [7]. It prevents the program from bypassing CHANCEL's instrumentation checks or jumping outside `sgx.code`.

(`sgx.shared`) during shared data initialization (the third column of Figure 3a) or (b) the base address of each thread's private region during servicing (the last column of Figure 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 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 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 6). The guard region between segments (shown in Figure 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 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 [54]), 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 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 (§V-B) that the program binary does not contain direct memory access instructions.

### D. 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 (§V-E).

**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 (§V-E) 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 [55]. Furthermore, if the resulting analysis is too imprecise, it can be improved through dynamic analysis with a representative workload [56, 57]. Importantly, developer-assisted identification does not pose security threats. In particular, shared objects are read-only during servicing and cannot harm CHANCEL's goals. Hence, imprecise or malicious identification only reduces performance since redundant data exists in thread regions.

### E. Runtime Services

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

```
1 // Receive data from the corresponding client.
2 bool recv(void* buf, uint buflen);
3 // Send data to the corresponding client.
4 bool send(void* buf, uint buflen);
5 // Notify the end of data migration
6 void end_migration();
7 // Terminate the thread.
8 void exit();
```

**Fig. 8:** Some runtime interfaces supported by SecureLayer.

**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 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 §V-B). 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 [4].

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.

## VI. IMPLEMENTATION

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

### A. 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 [58] 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 (§V-C).

**Supported C libraries.** CHANCEL supports Linux programs that are built using either `tlibc` [59], a minimalistic C library provided by Intel, or `musl libc` [60], a robust C library 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 (§V-E) to send and receive data.

### B. 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 II provides a breakdown of SecureLayer's components alongside other libraries included within the enclave.

**Validator.** SecureLayer includes an x86 disassembler, based on `Capstone` [15], which validates that the provided binary is correctly instrumented (§V-C).

**Loader.** The loader is responsible for relocating program symbols, enforcing MCSFI (i.e., using `r14` and `r15`), and provisioning 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 §V-E. For heap allocation routines, it includes wrapper functions (e.g., `malloc`) but reuses the SGX SDK's [61] heap allocation logic to initialize and maintain a heap in private thread regions. Furthermore, to create secure communication channels with clients, it uses Intel's cryptographic library, `tcrypto`.

## VII. 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 III provides an overview of all attacks and defenses.

| Component | KLoc | Base |
|---|---|---|
| **SecureLayer** | | |
| Validator | 53 | `Capstone` [15] |
| Loader | 1.3 | - |
| Runtime services library | 0.5 | - |
| **C library** | 15 / 66 | `tlibc` [59] / `musl` [60] |
| **Crypto library** | 23 | `tcrypto` [62] |
| **Total** | 92.8 / 143.8 | |

**TABLE II:** CHANCEL's components included in the enclave.

**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 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), not located in `sgx.code`.

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 6). Finally, while some pre-loaded enclave code (e.g., `asm_oret`) can also modify these registers [37], 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 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 caught during validation since they are forbidden.

**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 from the enclave.

CHANCEL prevents the program from leaking sensitive client data by ensuring that all memory writes target each thread's private region (Figure 5). CHANCEL also prevents malicious writes using stack operations (e.g., `push`) through guard pages (Figure 3a) and instrumentation of explicit updates to stack registers (Figure 6), ensuring all stack operations are in the thread's private region. Furthermore, to prevent the

| 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 7) |
| Jump outside `sgx.code` | `jmp`, `call` using register<br>An invalid return (`ret`) | Mask branch target and redirect to `sgx.code` (Figure 7)<br>Instrument `pop`, mask target, and redirect to `sgx.code` (Figure 7) |
| Modify `r14` or `r15` | Assembly instructions in target binary<br>Use SecureLayer code (e.g., `asm_oret` [37]) | Caught during validation (§V-B)<br>Not located in `sgx.code`; therefore, not executable |
| **Instr. bypass using injected code** | | |
| Modify SecureLayer or `sgx.code` | Write using register<br>Update `rsp`, `rbp` and `push` | Instrumented to target thread region only (Figure 5)<br>Ensure `rbp`, `rsp` point to thread region (Figure 6);<br>Guard page before thread regions |
| Add new code pages | Use `SGX2` instructions (`EACCEPT`) | Caught during validation (§V-B) |
| **Extract confidential client data** | | |
| Read from other threads | Read using register<br>Update `rsp`, `rbp` and `pop` | Instrumented to target thread region and `sgx.shared` (Figure 4)<br>Instrumented to ensure `rbp`, `rsp` point to thread region (Figure 6);<br>Guard page after thread regions |
| Write outside the enclave | Write using register<br>`EEXIT` and leak values in registers | Instrumented to target thread region only (Figure 5)<br>`EEXIT` is caught during validation (§V-B) |
| Others | Save thread data and leak it later<br>Absolute or PC-relative access | Clear thread region after servicing each user (§V-B)<br>Caught during validation (§V-B) |

**TABLE III:** CHANCEL's defenses against various attack vectors. Instr. means instrumentation.

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 (§V-E), which ensures confidentiality through encryption using a shared key with the concerned client.

Furthermore, CHANCEL instruments load instructions (Figure 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.

**Defense validation.** To confirm that CHANCEL can stop these attacks, we implemented and attempted the attacks mentioned in Table III. Our results indicate that all specified attacks are prevented either because (a) SecureLayer detects incorrect instrumentation during program validation; (b) hardware permissions prevent exploitation (e.g., overflows on guard pages, jumps to non-executable pages etc.); or (c) clearing of thread-specific data during cleanup.

## VIII. 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 (§VIII-A)?
- What is the overhead of CHANCEL on benchmarking applications (§VIII-B)?
- How does CHANCEL perform for real-world target scenarios (§VIII-C)?

**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 [61] and Intel SGX driver v2.6 [63].

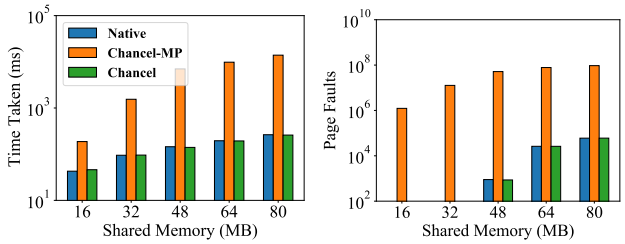



**Fig. 9:** (a) Average completion time and (b) the total number of EPC page faults when the amount of memory shared increases linearly. CHANCEL is $4.06 - 53.70\times$ faster than CHANCEL-MP and incurs a slowdown of only $0.8 - 7.5\%$ over NATIVE.

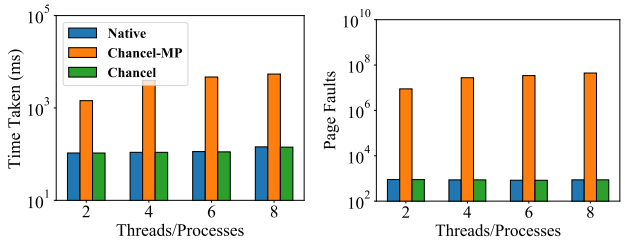



**Fig. 10:** (a) Average completion time and (b) the total number of EPC page faults when the number of processes/threads increases linearly. CHANCEL is $13.59 - 41.73\times$ faster than CHANCEL-MP and incurs an overhead of only $0.2 - 1.0\%$ over NATIVE.

**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.

### A. Improvement over Multi-Process Sandbox

This section analyzes the performance of CHANCEL in comparison with a traditional multi-process sandbox approach

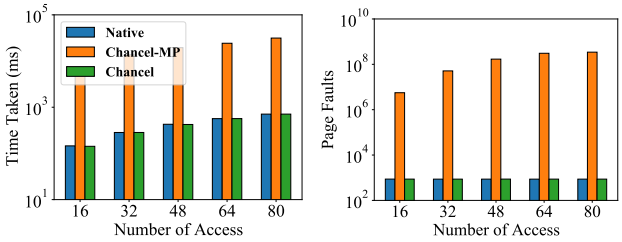



**Fig. 11:** (a) Average completion time and (b) the total number of EPC page faults when the number of memory accesses to each EPC page increases linearly. CHANCEL is $37.88 - 48.08\times$ faster than CHANCEL-MP and incurs an overhead of only $0.1 - 0.8\%$ compared to NATIVE.

| Benchmark | NATIVE | CHANCEL | |
|---|---|---|---|
| | (iterations/sec) | Slowdown (%) | Additional instr. (%) |
| NUMERIC SORT | 906.28 | 17.09 | 39.07 |
| BITFIELD | $4.55 \times 10^8$ | 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 |

**TABLE IV:** Nbench [22] running inside CHANCEL. The table shows slowdown incurred and additional instructions executed.

such as Ryoan [4]. Unfortunately, Ryoan [4] used QEMU full-system emulation because SGX2 CPUs were unavailable at the time of its publication; hence, its implementation cannot run on real SGX CPUs. Given these limitations, this section employs CHANCEL-MP, a variant of CHANCEL that isolates clients in different enclave processes like Ryoan, for comparison. The main difference between CHANCEL-MP and Ryoan is that the former uses CHANCEL's SFI while the latter employs Native Client [7]. Nevertheless, CHANCEL's SFI out-performs Ryoan's reported Native Client performance, as we discuss in §VIII-B. Hence, CHANCEL-MP is a suitable alternative multi-process sandbox for comparison purposes.

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

**Results.** Figure 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 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 §VI-B).

Furthermore, we show how the performance scales while increasing the number of enclave threads versus enclave processes. Figure 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 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.

Our experiments show that 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.

### B. *Overhead of* CHANCEL

We calculate the overhead of CHANCEL using a popular benchmarking application, nbench [22]. 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 [10, 64].

**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 IV reports both the throughput slowdown and the number of additional instructions executed (determined using `perf` [65]). 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

| 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 |

**TABLE V:** 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.

available registers and (b) executing additional instructions to enforce Multi-Client SFI, which increases overall computational cycles. Concerning the latter reason, since data accesses generally outnumber control-flow transfers, CHANCEL's major overhead is by 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 §V-C).

Therefore, under many scenarios, CHANCEL's overhead is low. Importantly, CHANCEL's performance overhead is comparable or superior to existing SGX SFI implementations [4, 10]. In particular, MPTEE [10] reported an overhead of $0.4 - 34\%$ on nbench, while Ryoan [4] 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.

### C. Performance with Real-world Programs

Based on CHANCEL's target scenarios (§III-B), we evaluate five real-world programs—DrugBank [18, 19], OSSEC [16], Recommender [21]), ShieldStore [20], and Snort [17].

**Common settings.** We ran the real-world experiments using both 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 [51]. 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

| `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× |

**TABLE VI:** 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 $2.7 - 13.1\%$.

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 V 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 [16], 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 [66]. 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.

*Results.* Table VI 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.

| 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× |

**TABLE VII:** 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 $0.2 - 11.4\%$.

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 [19] 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 [67]. 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 VII shows the results obtained while increasing the hash map size from $30 - 480$ MB. 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 under heavy workloads (up to $6.41\times$ for four clients). Finally, like OSSEC, increasing the clients emphasizes CHANCEL's improvement (up to $10.01\times$ 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 [21] 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 retrofit an included benchmark that creates a set of clients and populates their history of product 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 VIII 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 $17.20\times$) 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 $2.01\times$). Finally, with eight clients, CHANCEL's performance improvement expands to $21.18\times$ and $4.02\times$, under light and heavy recommender workloads, respectively.

**ShieldStore (private information retrieval).** ShieldStore [20] is an optimized key-value store that reports up to $20\times$ better performance than memcached [68] 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 IX 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 $11.44\times$) is better than on heavy workloads (up to $1.20\times$). 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.

| 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× |

**TABLE VIII:** 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** | | | | |
| 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× |

**TABLE IX:** Average delay (and the number of page faults in the parenthesis) to search $100,000$ queries using ShieldStore [20]. The overhead imposed by CHANCEL over NATIVE is $1.1 - 8.4\%$.

Nevertheless, like previous programs, eight clients further improves CHANCEL's performance, compared to CHANCEL-MP, by up to $16.32\times$ and $2.92\times$ 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.

**Snort (intrusion detection system).** Snort [17] 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 X 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 $4.14\times$ and $5.24\times$ on four and eight clients, respectively. The Snort experiment 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, ShieldStore, and Recommder) show substantial performance improvements for CHANCEL over CHANCEL-MP (up to $21.18\times$) 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 $4.02\times$).
- Programs with a minimal query working set (e.g., DrugBank and Snort) 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 $10.01\times$).
- 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.

## IX. DISCUSSION

**Improvement detection.** CHANCEL's performance improvement over a multi-process sandbox is significant under heavy workloads. However, the performance gain can be modest

| 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× |

**TABLE X:** Average delay (and the number of page faults in the parenthesis) to inspect $3,000$ packets using Snort [17]. The overhead imposed by CHANCEL over NATIVE is $0.5 - 11.8\%$.

(less than $0.5\times$) under light workloads given a minimal program query working set (refer to §VIII-C). Therefore, to automatically determine CHANCEL's utility, a developer could establish their program's query working set. Alternatively, a developer could manually compare CHANCEL's performance against CHANCEL-MP without much effort.

A developer can automatically establish their program's query working set using tools that can monitor the page tables [69, 70] during a program's execution. Such tools log and periodically reset a page table entry's referenced (or access) bit to determine what and how many pages were accessed by the program. Using such tools, a developer can execute their program (even a non-SGX version) under representative workloads to determine its query working set.

Alternatively, a developer can empirically analyze the performance of CHANCEL-MP and CHANCEL without significant porting efforts. In particular, CHANCEL-MP (like CHANCEL) supports many native Linux applications, including OSSEC [16], Snort [17], and Recommender [21], with no existing code changes and minimal code additions (refer to §VI-A). Hence, a developer can trivially run many applications with CHANCEL and CHANCEL-MP to establish performance.

Importantly, even when a developer establishes that a multi-process sandbox provides good performance for their use-cases, they might prefer to employ CHANCEL-MP over Ryoan [4]. In particular, compared to Ryoan, CHANCEL-MP provides higher performance (refer to §VIII-B), is compatible with all SGX CPUs, and can even be extended to other TEE implementations that, like SGX1, lack mechanisms to enforce memory protection dynamically (as we discuss later in this section).

**Defects and side-channels.** Although digital side-channels and micro-architectural defects are out of this work's scope (as described in §IV), it is nevertheless helpful to discuss their potential impact and mitigation.

*Micro-architectural defects.* Micro-architectural defect-based attacks (e.g., Foreshadow [44]) are defended by microcode patches and Intel's latest hardware [51]. Since SGX remote attestation can verify that CHANCEL executes on a patched machine, CHANCEL is secure against these defects.

*Cross-thread side-channels.* Exploiting side-channels internally to CHANCEL's enclave, i.e., from another concurrently executing thread, is non-trivial, under CHANCEL's sandboxing. Specifically, an enclave thread can only perform non-privileged side-channel attacks, involving resource contention (e.g., cache attacks) and, to be practical, a reliable, fine-grained timer source. However, such timer sources are unavailable within the enclave since CHANCEL (a) does not permit hardware timers, which are unsupported in SGX1 and caught during validation in SGX2, (b) does not allow collaboration between threads, which would otherwise allow relative measurement of time [41], and (c) controls (and normalizes) a thread's network interactions. Therefore, CHANCEL should not be worse than a multi-process sandbox regarding such attacks.

*Host-based side-channels.* CHANCEL requires appropriate counter-measures to defeat side-channel attacks that are launched externally from the host. Such attacks might involve collusion between the host OS and the untrusted enclave program to extract information. The known host-based side-channel attacks against SGX enclaves are page-table [38, 39], cache [40, 41, 71], branch-prediction [42, 72], port [73] and TLB [74] attacks. However, port and TLB attacks require hyper-threading; since a user can ensure that hyper-threading is disabled using SGX remote attestation, also needed to defeat Foreshadow [51], these attacks are defeated.

Existing SGX research has extensively analyzed and implemented defenses [25, 42, 46, 47, 52, 75–78] against the remaining host-based side-channel attacks. In particular, Varys [46] prevents page-table and cache attacks by intercepting enclave exits to obfuscate the page and cache-set access patterns leaked to the OS. Additionally, to defend branch prediction side-channels, Lee et. al proposed zigzagger [42], a trampoline solution that obfuscates all branches taken by the enclave program. Both defenses are compiler-enforced and report modest overheads (15% on average). In the future, CHANCEL can be extended to automatically enable these defenses.

**Non-x86 TEE extension.** CHANCEL can be extended to other Trusted Execution Environments (TEEs) since it does not require architecture-specific features, unlike other SGX SFI implementations (e.g., Occlum [9] requires MPX). In particular, MCSFI only requires general-purpose registers; hence it can be extended to TEEs based on RISC-V CPUs, e.g., Sanctum [79] or Keystone [80]. Furthermore, CHANCEL can also be extended to ARM CPUs, like Native Client's extension [81], but existing ARM TEEs lack hardware memory encryption [82], crucial to ensure client data confidentiality on cloud machines.

Importantly, CHANCEL's extension to RISC-V TEEs like Sanctum or Keystone would benefit both parties. On the one hand, CHANCEL would benefit from the robust and high-performance side-channel defenses implemented by Sanctum

and Keystone. On the other hand, these TEEs would benefit from CHANCEL in adversarial scenarios since they also provide limited trusted memory, like SGX. For example, Sanctum builds on an SGX-like memory encryption engine that provides hardware integrity; hence, it can only support $128-256$ MB trusted memory. Similarly, Keystone provides full confidentiality and integrity protection only to enclaves that reside on the caches—only a few MBs in current machines.

**Multiple services.** CHANCEL can handle scenarios where the program must send client data or intermediate computational results to other services. For instance, a recommendation system needs to retrieve a product's image file from a backend database. To handle such scenarios, all services must execute in different CHANCEL instances, attest each other using SGX remote (if different machines) or local attestation, and then share secret keys and encrypted client data or intermediate results.

**Client endpoints.** Service providers that employ CHANCEL but require an application installed on the client's machine to access the service should publicly release the application's source code, like Signal [2]. Hence, the client (or third-parties) can inspect the application to ensure data confidentiality. Note that this is not an issue in many instances where the client endpoint is trusted (e.g., web-browser-based services).

## X. RELATED WORK

**Other SGX permission enforcement schemes.** This section discusses other SGX permission enforcement schemes aside from those already discussed in §III-C.

VC3 [5] aims to securely process data under the Hadoop [83] framework by providing software-based self-integrity mechanisms. It also offers a compiler invariant 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. Furthermore, Rohit et al. [84] 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 [64] enables in-enclave Address Space Layout Randomization (ASLR) using executable data pages and a software data execution prevention technique (using `r15`). However, SGX-SHIELD is concerned with a different threat model (i.e., memory vulnerabilities). Therefore, it does not prevent data leakage from adversarial programs, and does not support thread isolation or shared memory enforcement.

**General-purpose SGX research.** Library OS schemes [9, 85–88] allow running unmodified applications in an enclave. CHANCEL is compatible with the library OS model. In particular, a library OS can be installed within SecureLayer to provide more generic functionality to the enclave. Furthermore, some library OS [86, 87] use program loaders, similar to CHANCEL, but do not enforce W⊕X on executable data pages, making them vulnerable to code-injection attacks. Such schemes can benefit from CHANCEL's MCSFI enforcement.

**Iago attacks.** SGX is vulnerable to Iago attacks [89] since it relies on the OS for ring-0 operations. CHANCEL prevents filesystem-related Iago attacks by providing in-memory filesystem functionality at runtime. In the future, it can be integrated with LibOS schemes [85, 87, 88] to provide full protection.

**General SFI research.** Wahbe et al. [90] suggested a novel way to isolate faults in a software-based manner. XFI, BGI, and LXFI [91–93] design further SFI mechanisms to isolate Windows kernel modules. Among SFI research, the design of CHANCEL is inspired by Native Client (NaCl) [7, 94]. 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).

**Computation on encrypted data.** Homomorphic encryption [95, 96] and order-preserving encryption [97] can protect client data on untrusted machines. Various systems [98, 99] allow secure computation using such techniques but suffer from high overheads and limited scenarios, unlike CHANCEL.

## XI. CONCLUSION

This paper presents CHANCEL, a system ensuring efficient multi-client isolation under adversarial programs. CHANCEL uses Multi-Client SFI (MCSFI), which 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.

## ACKNOWLEDGMENT

We would like to thank the anonymous reviewers for their valuable comments and suggestions. This work was partly supported by a National Research Foundation (NRF) of Korea grant funded by the Korea government MSIT (No. NRF-2019R1C1C1006095) and an Institute for Information and communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No. 2020-0-01840, Analysis on technique of accessing and acquiring user data in smartphone). The Institute of Engineering Research at Seoul National University provided research facilities for this work.

## REFERENCES

[1] J. A. Halderman, S. D. Schoen, N. Heninger, W. Clarkson, W. Paul, J. A. Calandrino, A. J. Feldman, J. Appelbaum, and E. W. Felten, "Lest we remember: cold-boot attacks on encryption keys," in *Proceedings of the 17th USENIX Security Symposium (Security)*, San Jose, CA, Jul.–Aug. 2008.

[2] "Signal » home," 2019. [Online]. Available: {https://signal.org}

[3] "Technology preview: Private contact discovery for signal," 2019. [Online]. Available: {https://signal.org/blog/private-contact-discovery/}

[4] 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.

[5] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich, "Vc3: Trustworthy data analytics in the cloud using sgx," in *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, May 2015*.

[6] 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.

[7] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, "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.

[8] N. Boucher, "Multi-domain sfi." [Online]. Available: {https://github.com/nickboucher/Multi-Domain-SFI/blob/master/paper/Multi%20Domain%20SFI.pdf}

[9] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel sgx," in *Proceedings of the 23th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Lausanne, Switzerland, Apr. 2020.

[10] W. Zhao, K. Lu, Y. Qi, and S. Qi, "Mptee: Bringing flexible and efficient memory protection to intel sgx," in *Proceedings of 12th European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, 2020.

[11] "Sgx-hardware." [Online]. Available: {https://github.com/ayeks/SGX-hardware}

[12] "Intel mpx support removed from gcc 9," https://www.phoronix.com/scan.php?page=news_item&px=MPX-Removed-From-GCC9.

[13] "Intel mpx support is dead with linux 5.6," https://www.phoronix.com/scan.php?page=news_item&px=Intel-MPX-Is-Dead.

[14] "The llvm compiler infrastructure," 2016. [Online]. Available: http://llvm.org/

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

[16] "Ossec hids," 2017, http://ossec.github.io.

[17] "Snort ids," 2016. [Online]. Available: https://www.snort.org/

[18] D. S. Wishart, C. Knox, A. C. Guo, S. Shrivastava, M. Hassanali, P. Stothard, Z. Chang, and J. Woolsey, "Drugbank: A comprehensive resource for in silico drug discovery and exploration," *Nucleic acids research*, vol. 34, no. suppl 1, pp. D668–D672, 2006.

[19] "petewarden/c_hashmap." [Online]. Available: {https://github.com/petewarden/c_hashmap}

[20] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "Shieldstore: Shielded in-memory key-value storage with sgx," in *Proceedings of the 11th European Conference on Computer Systems (EuroSys)*, Dresden, Germany, Apr. 2019.

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

[22] "Linux/unix nbench." [Online]. Available: http://www.tux.org/~mayer/linux/bmark.html

[23] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative instructions and software model for isolated execution." in *HASP@ ISCA*, 2013, p. 10.

[24] V. Costan and S. Devadas, "Intel sgx explained." *IACR Cryptology ePrint Archive*, vol. 2016, p. 86, 2016.

[25] 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.*

[26] "General data protection regulation." [Online]. Available: {https://gdpr-info.eu/}

[27] M. Azure, "Azure confidential computing," https://azure.microsoft.com/en-us/blog/azure-confidential-computing/, 2018.

[28] "Metadefender cloud." [Online]. Available: {https://metadefender.opswat.com/?lang=en}

[29] "virustotal." [Online]. Available: {https://www.virustotal.com/gui/home/upload}

[30] "Jotti virus scan." [Online]. Available: {https://virusscan.jotti.org/en-US/scan-file}

[31] "Amazon personalize." [Online]. Available: {https://aws.amazon.com/personalize/}

[32] P. Karnati, "Data-in-use protection on ibm cloud using intel sgx," https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx.

[33] "Packet platform features," https://www.packet.com/cloud/features/.

[34] "Ecs bare metal instances," https://www.alibabacloud.com/product/ebm.

[35] S. Park, S. Lee, W. Xu, H. Moon, and T. Kim, "libmpk: Software abstraction for intel memory protection keys (intel mpk)," in *Proceedings of the 2019 USENIX Annual Technical Conference (ATC)*, Renton, WA, Jun. 2019.

[36] 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.

[37] 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.*

[38] 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.

[39] 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.

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

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

[42] 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), Baltimore, MD, Aug. 2017.*

[43] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *Proceedings of the 40th IEEE Symposium on Security and Privacy (Oakland), San Jose, CA, May 2019.*

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

[45] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "Sgxpectre attacks: Leaking enclave secrets via speculative execution," *CoRR*, vol. abs/1802.09085, 2018.

[46] 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.

[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] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena, "Preventing your faults from telling your secrets: Defenses against pigeonhole attacks," *arXiv preprint arXiv:1506.04832*, 2015.

[49] 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.

[50] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity," in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2015.

[51] "Engineering new protections into hardware." [Online]. Available: {https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html}

[52] 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.

[53] "Intel® core™ i9-10900k processor." [Online]. Available: {https://www.intel.com/content/www/us/en/products/processors/core/core-vpro/i9-10900k.html}

[54] N. Burow, X. Zhang, and M. Payer, "Sok: Shining light on shadow stacks," *IEEE Symposium on Security and Privacy (SP)*, May 2019.

[55] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for intel SGX," in *Proceedings of the 2017 USENIX Annual Technical Conference (ATC)*, Santa Clara, CA, 2017.

[56] 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.

[57] 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)*, Vancouver, BC, Aug. 2021.

[58] "Writing an llvm backend," 2017, http://llvm.org/docs/WritingAnLLVMBackend.html.

[59] "linux-sgx/sdk/tlibc/." [Online]. Available: {https://github.com/intel/linux-sgx/tree/master/sdk/tlibc}

[60] "musl-libc," 2017, https://www.musl-libc.org.

[61] 01org, "Intel(r) software guard extensions for linux* os (source code)," 2016, https://github.com/01org/linux-sgx.

[62] "linux-sgx/sdk/tlibcrypto/." [Online]. Available: {https://github.com/intel/linux-sgx/tree/master/sdk/tlibcrypto}

[63] 01org, "Intel(r) software guard extensions for linux* os (linux sgx driver)," 2016, https://github.com/01org/linux-sgx-driver.

[64] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim, "Sgx-shield: Enabling address space layout randomization for sgx programs," in *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2017*.

[65] "Perf wiki." [Online]. Available: {https://perf.wiki.kernel.org/index.php/Main_Page}

[66] ClamAV, "ClamAV," https://www.clamav.net/, 2018.

[67] "Drugbank," 2017. [Online]. Available: http://www.drugbank.ca

[68] "memcached - a distributed memory object caching system." [Online]. Available: {https://memcached.org/}

[69] "How to measure the working set size on linux." [Online]. Available: {http://www.brendangregg.com/blog/2018-01-17/measure-working-set-size.html#:~:text=The%20Working%20Set%20Size%20(WSS,capacity%20planning%20and%20scalability%20analysis.}

[70] "Dynamorio." [Online]. Available: {https://dynamorio.org/}

[71] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using sgx to conceal cache attacks," *arXiv preprint arXiv:1702.08719*, 2017.

[72] 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 Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

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

[74] 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.

[75] 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, 2016.

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

[77] 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*.

[78] 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, USA, Oct. 2020.

[79] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," Cryptology ePrint Archive, Report 2015/564, 201 5. http://eprint. iacr. org, Tech. Rep.

[80] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proceedings of the 12th European Conference on Computer Systems (EuroSys)*, Heraklion, Greece, Apr. 2020.

[81] "Native client on arm." [Online]. Available: {NativeClientsupportonARM}

[82] 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.

[83] "Apache hadoop project," 2017. [Online]. Available: http://hadoop.apache.org

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

[85] 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.

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

[87] C. che 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.

[88] 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*.

[89] 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.

[90] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5. ACM, 1994, pp. 203–216.

[91] 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.

[92] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, Oct. 2009.

[93] 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.

[94] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures." in *Proceedings of the 19th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2010.

[95] Z. Brakerski and V. Vaikuntanathan, "Fully homomorphic encryption from ring-lwe and security for key dependent messages," in *Proceedings of the 31st Annual Conference on Advances in Cryptology (CRYPTO)*, 2011.

[96] C. Gentry, *A fully homomorphic encryption scheme*. Stanford University, 2009.

[97] A. Boldyreva, N. Chenette, Y. Lee, and A. O'neill, "Order-preserving symmetric encryption," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009.

[98] 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*.

[99] R. Bost, R. A. Popa, S. Tu, and S. Goldwasser, "Machine learning classification over encrypted data." in *Proceedings of the 2015 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA, Feb. 2015*.