Stockade: Hardware Hardening for Distributed Trusted Sandboxes

Joongun Park[1], Seughyo Kang[1], Sanghyun Lee[1], Taehoon Kim[2],
Jongse Park[1], Yongjin Kwon[1], and Jaehyuk Huh[1]

[1]School of Computing, KAIST
[2]ETRI

Abstract

Recent studies showed that a cloud application consists of multiple distributed modules provided by mutually distrustful parties. For trusted services, such applications can use trusted execution environments (TEEs) communicating through software-encrypted memory channels. Such an emerging TEE execution model requires a new type of bi-directional protection: protecting the rest of the system from the enclave module with sandboxing and protecting the enclave module from third-party modules and the operating system. However, the current TEE model cannot efficiently represent such distributed sandbox applications. To overcome the lack of hardware supports, this paper proposes an extended TEE model called STOCKADE*, which supports distributed sandboxes hardened by hardware.* STOCKADE *proposes new three key techniques. First, it extends the hardware-based memory isolation in SGX to confine a user software module only within its TEE (enclave). Second, it proposes a trusted monitor enclave that filters and validates systems calls from enclaves. Finally, it allows hardware-protected memory sharing between a pair of enclaves for efficient protected communication without software-based encryption. Using an emulated SGX platform with the proposed extensions, this paper shows that distributed sandbox applications can be effectively supported with small changes of SGX hardware.*

1. Introduction

Hardware-based trusted execution environments (TEEs) enabled the strong isolation of execution contexts in remote clouds, even when the servers are exposed to potential vulnerability in privileged software and physical attacks. Among recent TEE supports, Intel Software Guard Extension (SGX), a commercial incarnation of TEEs, provides isolated execution environments called *enclaves* protected by the CPU hardware. The CPU hardware isolates each enclave from the operating system. Its code and data are encrypted and integrity-verified while they reside in the external DRAM.

The introduction of commercially available TEEs has been accelerating the exploration of application scenarios utilizing their strong isolation capability. One important cloud-oriented scenario is to provide function-as-a-service or software-as-a-service on clouds, running a function or software in each enclave [1, 2, 55]. In such applications, it is critical not only to protect user-provided functions from the potentially vulnerable cloud system but also to secure the hosting cloud system by sandboxing the user-provided functions or software, as they cannot be fully trusted from the perspective of the hosting system. Besides, an application task is composed of multiple functions communicating with each other [57]. Figure 1 presents such a distributed sandboxed application. Such an application consists of software modules from multiple software providers, which may not entirely trust the other providers. With multiple participants, each module must be protected from other modules or the hosting system, and modules must also be confined to prevent any exploitation of system vulnerability. A key software technique for such distributed secure applications is *software sandboxing* which prevents the codes in an enclave from accessing the memory beyond the protected enclave memory and validates system calls.

The distributed sandboxed applications reveal the limitations of the current SGX model. First, the codes inside an enclave can freely access the remaining untrusted memory of the process. Such uni-directional protection can endanger the rest of the system if the enclave code is malicious. To address such vulnerability, the prior study proposed to employ a heavy software sandboxing running with user codes inside an enclave [28, 50, 57, 68, 82]. Second, enclaves require to use operating system services via system calls, but the secure interaction via system calls must be considered. Not only such system call requests must be verified to protect the hosting system [96], but return values must be checked to prevent Iago attacks against the enclave [30, 47, 85, 91]. Finally, the communication channel among enclaves is not provided by the hardware mechanism. For secure inter-enclave communication, a pair of enclaves must share an untrusted memory region, and each message must be encrypted and integrity-protected by the software running inside the enclaves. Such software-based encrypted communication not only increases the communication latency but also can cause a vulnerability [34, 85].

To overcome the limitations of the current TEE model, this study proposes an extension of the enclave model, called STOCKADE. STOCKADE provides efficient hardware-supported solutions for the three limitations. First, instead of using software-based sandboxing, STOCKADE blocks accesses from enclaves to the untrusted world. We call the sandboxed enclave *bi-enclave*. By simply extending the pre-existing memory validation mechanism in SGX hardware, a bi-enclave can not only be protected from the untrusted world but also be prevented from accessing the untrusted context.

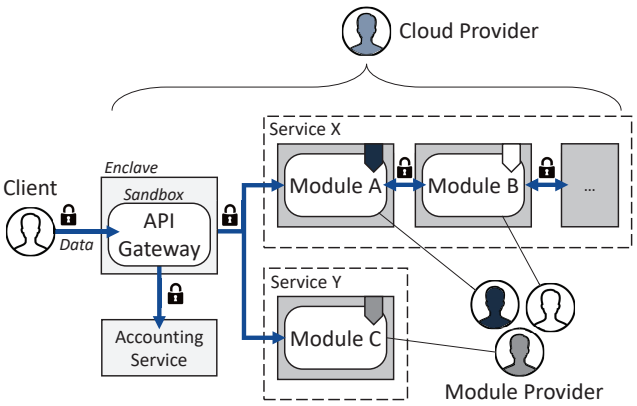

Figure 1: A distributed sandboxes with SGX. Gray boxes are enclaves running modules from different providers

Such bi-directional isolation enables solid sandboxing support for each bi-enclave without any extra software layer.

The second mechanism is to provide a hardened interaction between a bi-enclave and the operating system. The interaction of the bi-enclave and operating system can be forced to go through the monitor enclave to process the system calls only if they are valid. The key difference from the prior approaches [28, 50, 57, 68, 82, 96] is that the monitor is isolated both from the bi-enclave and from the operating system, which provides stronger protection for the system call verification and return value validation. The codes running in the monitor enclave are attested by both the bi-enclave and operating system, providing verified monitoring operations by the two entities. With the neutral monitor enclave, STOCKADE can provide a temper-proof accounting service of system resources such as file I/Os and network usages, as both the cloud users and providers can trust the monitor enclave.

The final mechanism allows sharing of trusted memory pages between two enclaves. The hardware provides an interface for sharing the protected pages between two enclaves, and the memory isolation mechanism is extended to allow two enclaves to access the shared pages. By sharing the hardware-protected pages, the communication between the enclaves does not require costly software-based encryption and integrity protection.

To show the effectiveness of the new enclave extensions, we ported several application scenarios on an emulated SGX runtime with the extended interface. The experimental results show that minor hardware extensions can improve the efficiency and security of distributed sandbox applications on clouds. Compared to the prior SW-based sandboxing, it provides 1.4∼19.5% performance improvements. This study hardens the distributed sandbox applications with hardware extensions. To the best of our knowledge, it is the first study to extend the execution model and hardware for bi-directional protection with the protected system call monitor. The new contributions of this paper are as follows:

- It proposes bi-directional isolation between an enclave and its untrusted environment. The design shows that a simple extension of the existing memory access control mechanism in SGX can provide efficient isolation for both ways.
- It proposes a hardware-protected monitoring mechanism for handling system call filtering and accounting operations for each enclave.
- It proposes a shared trusted memory between two enclaves. With a careful design, a designated part of the protected memory of an enclave can be shared with the other enclave.

The rest of the paper is organized as follows. Section 2 presents the background of distributed sandbox applications. Section 3 discusses the motivations of three extensions, and Section 3.4 discusses the related works. Section 4 presents the proposed hardware extensions. Section 5 presents the security analysis, and Section 6 provides four application scenarios using bi-enclave and their performance on an emulated SGX runtime. Section 7 concludes the paper.

2. Background

2.1. Intel Software Guard Extensions (SGX)

Intel SGX provides a user-level trusted execution environment called an *enclave*. The context of an enclave is protected by the hardware mechanism. The protected memory region of enclaves is created in Enclave Page Cache (EPC). Part of physical memory, Processor Reserved Memory (PRM), is reserved for SGX and is protected by the hardware memory encryption engine (MEE). PRM contains the EPC pages in addition to other security meta-data for SGX. Although EPC pages are in the external DRAM, their confidentiality and integrity are guaranteed under direct physical attacks on DRAM and system interconnection components. The attestation support allows a user to verify the identity and measured digest of an enclave and platform setting where the enclave runs.

The memory isolation for each enclave is done during the address translation step for each memory access. A mode transition between the enclave mode and untrusted mode requires flushing Translation Lookaside Buffers (TLBs). For each TLB miss, the validity of access is verified by the CPU hardware logic. A key internal data structure for verification is Enclave Page Cache Map (EPCM) which is stored in PRM. An EPCM entry has information about a physical page that belongs to the EPC region. It contains the owner's enclave ID and its virtual address in the enclave memory space, in addition to other status information. Even though page tables are still managed and updated by the operating system, the EPCM table is accessible only by the hardware, and the page table entry for EPC can be verified using EPCM. The crucial invariant for the correctness of memory isolation is that *TLB must contain only verified translations*.

SGX controls enclave through a set of instructions. After an enclave is created, *EINIT* initializes it to be ready for protected execution. The virtual address of protected memory region for an enclave is fixed during the initialization of the enclave. The virtual address range for an enclave should be a single contigu-

ous region called Enclave Linear Address Range (ELRANGE). The context information of an enclave is stored in its SGX Enclave Control Structures (SECS). SECS are allocated in EPC pages for its safety against the malicious operating system. SGX includes instructions for switching modes between enclave context and unprotected context: *EENTER* to enter enclave mode, and *EEXIT* to exit enclave mode.

2.2. Sandboxing

Sandboxing confines an application in its own environment. By isolating untrusted applications, sandboxing protects the kernel and host environment against potential attacks from the applications. Sandboxing is widely adopted for runtime protection against third-party applications, such as web browsers running plugins written by unauthorized developers [8, 31], and testbeds for third-party developers migrating their applications to the production system [2, 6, 10, 16, 17, 18, 21].

An application running in a sandbox must not be allowed to directly access the memory outside of the sandbox. In addition, the application control should never reach beyond the designated sandbox, neither directly nor indirectly during its runtime. To provide sandboxing, fault isolation confines control transfer and data access within a sandbox, and system call filtering validates system call requests from sandbox applications.

Fault Isolation: Fault isolation provides a logically isolated compartment by enforcing its confinement policy on memory and control transfer. Software-based fault isolation provides such confinement by binary instrumentation or compiler support [38, 54, 78, 95, 100]. Using binary instrumentation, Google Native Client (NaCl) restricts memory accesses from untrusted applications, by masking target addresses with memory boundary before the binary execution. Such software-based isolation needs to execute extra instructions for access validation, adding performance overheads. In addition, the instruction-based bound checking is potentially vulnerable to the Spectre attacks [37, 60, 61, 66, 69]. Other fault isolation techniques rely on CPU hardware supports for confinement [68, 79, 96, 101]. With hardware supports such as Intel Memory Protection Keys (MPK) [72], or ARM Domain [15], they provide sandboxes to separate modules from each other. However, the current MPK uses page tables to track memory domains, and thus the domain information can be changed by OS.

System Call Monitoring: In addition to the memory access control, the interaction with the operating system must also be regulated by sandboxing. Although the operating system is protected with privilege separation and system call interfaces, system vulnerabilities via system calls have been continuously reported [3, 11, 13, 14]. A naive way to alleviate this problem is not allowing untrusted applications to make any system calls. However, many real-world applications are relying on system call interfaces such as POSIX to use network supports and file management. Therefore, the sandbox must provide controlled system functionalities by verifying system calls from the untrusted application. Seccomp-bpf [44] interposes system call requests by filtering system call with ID and arguments. In addition to filtering system calls, by manipulating return values of system calls, a malicious operating system can leak the application's secret or break the execution integrity known as the Iago Attack [40]. To prevent Iago attacks, return values also need to be validated [56, 62].

2.3. Cloud Applications and Trusted Execution

Cloud services have evolved to use a more complex task model, where many different software modules are interacting with each other. A single cloud application may rely on multiple modules from different parties. As shown in Figure 1, a recent advancement of function-as-a-service or software-as-a-service has enabled cloud applications to be composed of small functions or modules. Each module is implemented by a different party, and thus, its trustworthiness is not fully guaranteed from the perspective of the other module providers. In addition, the cloud provider must protect the system from modules and clients. To support such new application scenarios, the trusted execution model needs to evolve.

Recent studies investigated applying trusted computing to such distributed cloud applications consisting of multiple modules with different providers [29, 51, 57, 89, 94]. They proposed to run each module in an enclave, but additionally sandboxing is combined with the module running in an enclave. The sandbox library blocks access to untrusted memory from an enclave, which confines the access boundary of an enclave only to its own EPC region. With such software sandboxing, it attempts to prevent the functions in an enclave from exploiting potential vulnerabilities of the system. In addition, it uses software-encrypted communication via shared memory between enclaves to allow the coordination and data transfer of multiple modules.

Such cloud applications require extensions of the current SGX model: First, TEE not only needs to protect the context in an enclave but also must confine accesses from enclaves, if necessary. Second, a software module in an enclave often needs to access the system resource via system calls. How to control the system call access must also be considered in the SGX model. Third, multiple software modules must efficiently interact with each other. However, the current SGX is not designed for facilitating inter-enclave communication.

3. Motivation

3.1. Bi-directional Isolation with Enclave

In distributed sandboxed applications, an enclave execution must be protected, but it must also be prevented from accessing memory beyond its own EPC region. In the current SGX model, the in-enclave execution is freely allowed to access the rest of its process memory, which is confined only by the operating system. There are two different ways of providing confinement supports for the current SGX enclave model.

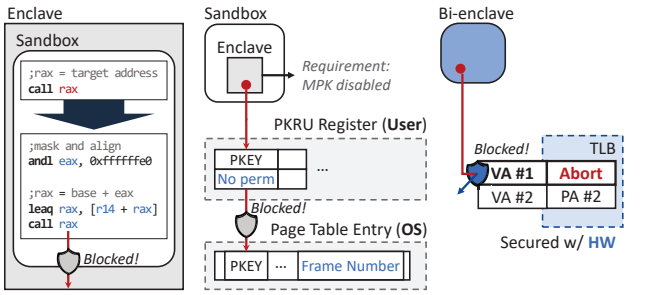

Figure 2: Three confinement approaches: SW, HW+OS (MPK), and HW-only (Bi-enclave)

The software-based confinement for the enclave is to include the instrumented application binary and sandbox library codes together in each enclave. In such an approach, the sandbox library, as well as application binary, must be trusted. Figure 2 (a) describes the software-based approach (SW). Combining the sandbox runtime with the application binary in a single enclave increases the Trusted Computing Base (TCB) of the enclave. When a vulnerability exists in the sandbox library [11], the application code can potentially exploit the vulnerability to bypass the confinement. In addition to the increased TCB, the memory access occurring in the enclave must be verified by instrumented instructions, causing extra performance costs. A recent study [28] reports that the software-based confinement incurs a slowdown of an average of 12.43%, up to 24.89% compared to native execution because it requires 23.52% more instructions.

Recent hardware supports for memory confinement such as Intel Memory Protection Keys (MPK) can mitigate the weaknesses of software-only approaches. However, the current hardware-assisted mechanism relies on page tables for tracking the isolated memory domains. Since page tables can be modified by any privileged access, the confinement assumes that the operating system is trusted. Figure 2 (b) shows the hardware-assisted approach with MPK (HW+OS). A limitation of MPK is that the PKRU registers which defines the permission for each domain are user-accessible. Therefore, the user application in an enclave must be verified not to update the PKRU registers. In addition, the current HW+OS approach relies on the security of page tables. However, recent studies showed that page tables can be vulnerable to various attacks including rowhammer attacks [42, 53, 77, 93, 98].

To overcome the limitations of the current software-only and hardware-assisted sandbox designs, this paper proposes to extend the SGX memory access control mechanism. Unlike MPK, it does not use page tables to store critical domain information. Figure 2 (c) shows the pure hardware-enforced approach of STOCKADE (HW-only).

3.2. OS Interactions with Bi-enclave

A bi-enclave is prohibited to access the memory outside of its own EPC, but it should be allowed to issue system call requests

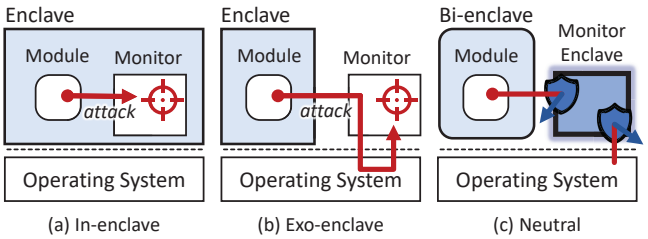

Figure 3: Three syscall monitor approaches: in-enclave, exo-enclave, and neutral enclave

if the system call requests can be verified for their safety. Therefore, to provide fully functioning sandbox enclaves, it is necessary to support a safe mechanism to verify system call requests and forward the filtered requests to the operating system. In addition, the returned value from the untrusted operating system may need to be checked. On the contrary, in the prior distributed sandbox approach [57], a confined module is not allowed to use any system call directly.

There are two different approaches to provide system call services to enclave execution: system call emulation and system call delegation. First, the system call emulation approach imports the entire library OS [73, 75, 90] and C standard libraries [9] inside an enclave [30, 32, 68, 74, 82, 91]. With the intra-enclave libOS, porting efforts for existing applications to SGX is minimized. In addition, with a carefully designed shim layer, it helps to minimize the exposed interfaces between the host and enclave which is the main attack surface of the enclave [59]. However, this approach adds the entire software stack within an enclave, increasing the TCB of an enclave significantly. Figure 3 (a) shows the in-enclave monitor approach. It assumes that the monitor code can be completely isolated from the application module by a software confinement layer. However, as discussed in the prior subsection, the vulnerability in SW-only confinement may not guarantee the protection of the monitor.

Second, the system call delegation approach relies on the underlying OS itself, thereby reducing TCB drastically [85, 96]. Rather than including a large libOS stack within each enclave, it includes a much smaller codebase for system call interposition and verification after execution. It delegates system calls to the non-enclave mode and performs its system call. The returned results from the system call are verified inside the enclave to prevent malicious intervention. In both cases, the enclave must be able to interact with the untrusted context for system calls. Figure 3 (b) shows the exo-enclave monitor approach. It assumes that the monitor code exists as a process in unprotected environments accessible from the OS kernel. Therefore, this approach is vulnerable to attacks like privilege escalation [12, 13, 48].

To allow the controlled interaction from the sandbox enclave and operating system, we propose to add a *monitor enclave* that can be coupled with one or more bi-enclaves. A monitor enclave is a conventional enclave, and thus it can jump to the system call function in the untrusted region. Be-

Related Work	Confinement		System call filter		Protection against				Multi module support	
	Type	Method	Type	Method	SW Sandbox Vulnerability	Privilege Escalation	PTE Corruption	Iago attack	3rd Party Module	Protected Channel
Ryoan [57]	SW	Native Client	In-enclave	Libc	✗	✓	✓	✓	✓	Inter-enclave (SW)
Chancel [28]	SW	LLVM	In-enclave	Libc	✗	✓	✓	✓	✗	Intra-enclave
AccTEE [50]	SW	WebAssembly	In-enclave	LibOS	✗	✓	✓	✗	✗	✗
Occlum [82]	HW+OS	MPX ‡	In-enclave	LibOS	✗	✓	✓	✗	✗	Intra-enclave
EnclaveDom [68]	HW+OS	MPK †	In-enclave	LibOS	✗	✓	✗	✗	✓	Intra-enclave
SGXJail [96]	HW+OS	MPK †	Exo-enclave	Seccomp	✓	✗	✗	✗	✗	✗
STOCKADE	**HW-only**	**SGX**	**Neutral**	**Monitor**	✓	✓	✓	✓	✓	**Inter-enclave (HW)**

✓: Considered / Secure ✗: Not considered / Vulnerable ‡Deprecated [7, 70] †HW modification required

Table 1: Comparing STOCKADE to prior work

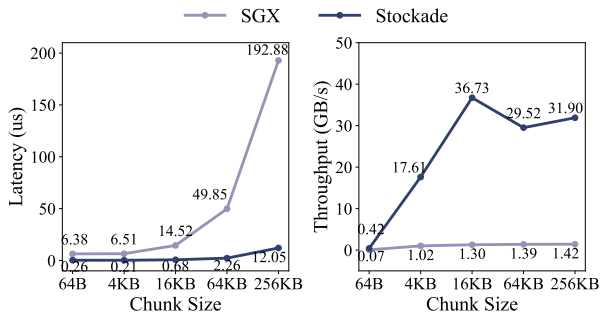

Figure 4: Comparing inter-enclave communication between SGX and STOCKADE

tween the bi-enclave and monitor enclave, a protected memory channel is created, and the bi-enclave sends a request to the monitor enclave. As shown in figure 3 (c), STOCKADE takes a different approach to others. STOCKADE locates the monitor in a position-neutral enclave. In the approach, the monitor is protected both from the user enclave and OS kernel.

Sandboxes using system call delegation have to consider races between sandboxes [49]. In addition, a prior study [47] observed that a Iago attack can occur across multiple components, and thus checking the return value within each enclave individually is not enough to prevent such an attack. In STOCKADE, the monitor can track global states among bi-enclaves thus can prevent Iago attacks against connected bi-enclaves. In addition, the design helps developer not to add Iago attack protection in every bi-enclave.

Tamper-proof resource accounting: One of the requirements for the trusted cloud service is tamper-proof resource accounting [50, 58, 88]. For each user, the system resource usages must be securely tracked and reported. To support such tamper-proof accounting which can be trusted by both cloud users and service providers, it is necessary to track system resource usages by a mutually trusted entity. As a monitor enclave can be isolated from both users and OS, it can act as a neutral accountant, recording the file and network I/Os.

3.3. Interactions between Enclaves

Intel SGX supports function-like interfaces, ecall and ocall, between an enclave and unprotected side for context switching. However, SGX does not provide APIs for inter-enclave communication. The prior work used a software-based reliable messaging mechanism with shared untrusted memory between enclaves [57]. The messages are encrypted with a shared key. Message authentication code and counters ensure the integrity and freshness of the message. The protection mechanism can be further improved by padding or truncating the messages for making the attacker hard to guess the original message size. The software module for cryptography must be trusted and verified, as it is included in the enclave binary. However, Panoply [85] showed that an attacker can abort the application silently and make the application misjudge its state by dropping messages even the communication channel is encrypted.

In this study, we use a similar message-based interaction model, but propose to extend the hardware memory access control to allow the protected memory channel. We propose a new secure memory sharing model between two enclaves, which uses a designated part of EPC memory region. The memory region is accessible only by the two enclaves, and the hardware memory access control and memory encryption engine protect it. It eliminates the need for software-based memory encryption, which increases the latency and may cause potential vulnerabilities as exploited by Panoply. Figure 4 shows the communication latency and bandwidth between the current SGX and proposed bi-enclave communication. The SGX model uses software-based encryption(AES-GCM) via untrusted shared memory, while the bi-enclave configuration directly shares the hardware protected memory. As shown in the figure, the latency for transferring data is much higher with SGX than STOCKADE, and the difference increases as the chunk size increases. In addition, the throughput of STOCKADE communication exhibits much higher bandwidth than the software-encrypted channel.

3.4. Comparison to the Prior Work

There have been several prior work for sandboxing within a TEE. Minibox [65] presents the first two-way sandbox for native x86 code, providing secure file I/O and Iago attack protection. Ryoan [57] modifies Native Client [100] for its software sandboxing. Similar to STOCKADE, Ryoan decomposed a cloud application into distributed enclaves with the software sandboxing and SW-encrypted channels. Chancel [28] proposes multi-client software fault isolation through binary instrumentation and read-only shared memory between threads. It supports multiple isolated threads within an enclave. Several studies accommodate hardware features (Intel MPX or MPK) under software control for multi-domain SFI scheme

[28, 68, 81, 82]. Occlum [82] and Enclavedom [68] provide isolated compartments within a sandboxed enclave using Intel MPX or MPK. SGXJail [96] isolates an enclave instance for each process, and the system call filtering is provided by the seccomp filters in each process.

Table 1 presents the confinement mechanism and monitor locations of the prior works compared to STOCKADE. Only STOCKADE can provide comprehensive protection against four attack types, SW sandbox vulnerability, privilege escalation, rowhammer on PTE, and Iago attack. In addition, STOCKADE allows secure hardware-protected inter-enclave communication.

Other related work: There are studies [30,32,82,91] provide trusted library OSes running in the enclave, and enable unmodified application execution in the enclave. Panoply [85] reduces TCB by delegates syscalls to OS and verify later. Nested Enclave [71] presents static sharing enclave and communication via the outer enclave. STOCKADE provides dynamic EPC sharing with page granularity.

3.5. Threat model

STOCKADE shares the basic threat model and trusted computing base (TCB) of SGX. The SGX-enabled processor package is trusted. Privileged software such as the operating system and hypervisor can be compromised by its vulnerability or any person who obtains the privilege permission. Moreover, attackers can wield direct physical attacks on on-board interconnections and external DRAM.

A different assumption of this work over SGX is in the trustworthiness of software modules running in enclaves. In SGX, a software module running in an enclave is trusted, and potential attacks from the module itself to the rest of the system are not considered. We assume that each module does not fully trust the other modules, even when they are used together to build an application. In our model, the code running in the monitor enclave is trusted, and the monitor enclave and a bi-enclave are mutually protected from each other.

Out of scope: Architecture defects [36, 52, 60, 66, 97], side channel attacks [33, 35, 64, 86, 87, 92, 99], and availability are not considered in this work. For such attacks, prior patches [26] and protections [27, 46, 59, 76, 80, 83, 84] can be used as orthogonal measures. STOCKADE does not support resiliency to code reuse attacks [33, 63] and arbitrary API invocation (e.g. COIN attack [59]).

4. Architecture

4.1. Overview

Figure 5 presents STOCKADE's distributed sandboxing model. STOCKADE architecture has two types of enclave, *bi-enclave* and *monitor enclave* as shown in (a). Bi-enclave inherits all the properties of SGX enclave, and additionally, a bi-enclave blocks all accesses to the non-enclave memory with the hardware support. The code running in a bi-enclave is not allowed

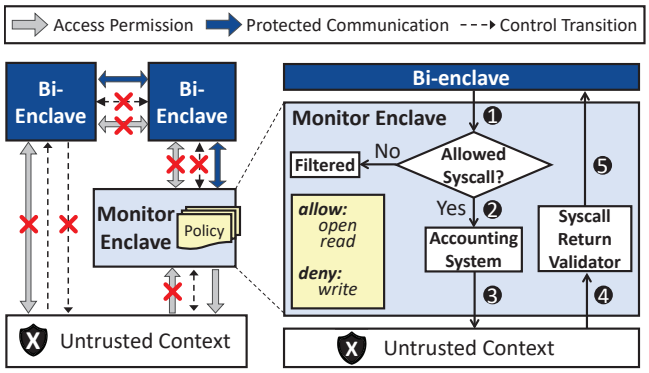

Figure 5: STOCKADE architecture. The control flow from the untrusted context to bi-enclave is only allowed at launching

Target	Description
SECS*	New field (1 bit) for bi-enclave flag (§4.1)
EPCM entry*	New field (52 bits) for phsycal address of Co-owner's SECS (§4.2)
EINIT†	Set bi-enclave flag in SECS on initialization (§4.1)
EEXIT†	Abort EEXIT when bi-enclave flag is set in SECS (§4.2)
ESADD†	Establish shared EPC to target enclave (Ring 3) (§4.3)
ESACCEPT†	Accept shareable EPC from ESADD (Ring 3) (§4.3)
TLB‡	Fill entry on Co-onwer access / Abort on forbidden access (§4.2)

*Data Structure †Instruction ‡Access checks on TLB miss

Table 2: Summary of hardware changes

to read, write, and execute contents outside the bi-enclave memory. In addition, the control of a bi-enclave cannot be directly transferred to the non-enclave context, but it must go through the monitor enclave to interact with the rest of the system. STOCKADE allows the monitor enclave to communicate with the operating system (OS).

The monitor enclave works as a proxy to communicate the operating system. To interact with the operating system, a bi-enclave has to establish a secure shared memory channel to a monitor enclave. For communication with other enclaves, the same secure shared memory channel is supported. As shown in Figure 5 (b), with the monitor enclave attached to a bi-enclave, it delegates system call to OS. A monitor enclave verifies system calls based on a given profile and validates the return values of system calls to prevent known Iago attacks. In addition, the monitor enclave can track system call usage records in a mutually trusted way.

Application model: In STOCKADE model, a service consists of one or more mutually distrustful modules. Each module is enclosed in a bi-enclave, and mutually distrustful modules do not reside in the same bi-enclave. With the protection boundary, both bi-enclave and monitor can have multiple threads.

Hardware changes: The SGX security features are mostly implemented in microcode which incurs much less implementation overheads than CPU circuitry [45] Therefore, the majority of modifications for STOCKADE on data structures, instructions, and access control, can be done via minor microcode changes. Table 2 shows required hardware changes

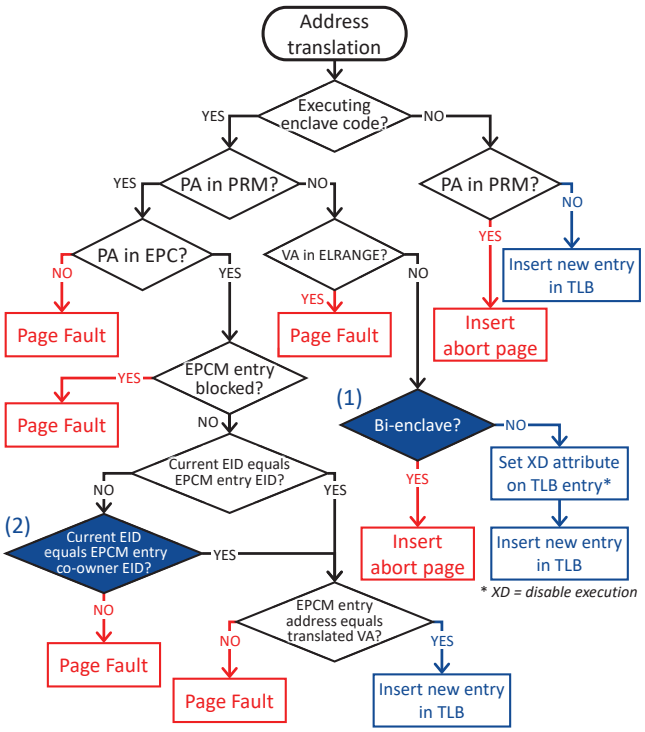

Figure 6: Access control flow for STOCKADE. Modification are painted blue on the original SGX's flow [45]

for STOCKADE and related subsections. First, to support new enclave type, bi-enclave, SECS and EPCM entries are modified. Second, new instructions *ESADD* and *ESACCEPT* for secure channel are added, while existing instructions (*EINIT*, and *EEXIT*) are modified. Finally, the TLB miss handler has been changed to support different permission checks of STOCKADE with SGX. Note that the access validation is done only when TLB miss occurs. STOCKADE does not change any other components in processors and cache hierarchy.

4.2. Memory Protection for Stockade

Access Validation: STOCKADE leverages SGX memory protection features to enable bi-directional memory protection. Based on the SGX original memory isolation, STOCKADE provides additional memory protection in the opposite direction. STOCKADE protects the non-enclave memory context by preventing memory translation from a bi-enclave. Figure 6 is the hardware flowchart for STOCKADE's address translation. **(1)** in the figure indicates the additional memory protection added for STOCKADE. When an enclave is not in bi-enclave mode, memory access to a non-ELRANGE virtual address is allowed. Thus, STOCKADE inserts a new entry to the TLB in the same way as the original SGX. However, when the enclave is in bi-enclave mode, the sandboxed code must not be allowed to translate to the outside memory. STOCKADE inserts an abort page to cause a failure in resolving the non-ELRANGE virtual address to a physical address. As shown in the figure, the extra access control for bi-enclave does not require any significant hardware changes; STOCKADE needs a minor extra condition check while handling a TLB miss.

Control transition: By calling *ocall*, an enclave performs a control transfer from the enclave to the non-enclave context. To perform an ocall, a normal enclave saves its state in the protected memory, cleanses all internal CPU states to prevent security leaks, and switches its mode into the non-enclave mode with *EEXIT*. Unlike normal enclaves, STOCKADE isolates a bi-enclave by disabling *EEXIT* instruction. The hardware modification for *EEXIT* is minor since CPU only checks whether the current context is in bi-enclave mode or not by reading flag in the SECS structure. However, Asynchronous Enclave Exit (AEX) is still allowed even for the bi-enclave because operating systems must handle exceptions such as page faults or interrupts. When AEX occurs, all execution contexts are securely saved in the enclave memory, and STOCKADE switches its execution mode to handle the exit events. The event is handled by designated hardware exception handlers in processors. During AEX, it erases any context (secrets) that may exist in the execution state [45]. Therefore, the software in a bi-enclave cannot exploit AEX for escaping the sandbox. The saved context will be restored during the next *EENTER* or *ERESUME*. STOCKADE does not modify the flow of AEX from the original SGX.

Advantages: Compared to the prior SW and HW+OS confinement approaches [28, 50, 57, 96], STOCKADE can provide hardware-supported strong isolation efficiently. STOCKADE does not require any extra SW layers or compiler-based validations for the confinement, unlike the prior SW approaches. Furthermore, STOCKADE is more robust against Spectre-like attacks which attempt to bypass protection boundary checks of the SW approaches because an unauthorized speculative access will incur a TLB miss and the address translation fails. Compared to the HW+OS approaches which keep domain IDs in vulnerable page tables, STOCKADE keeps the critical meta-data in the secure memory region (PRM). Therefore, the meta-data is protected from OS and from DRAM attacks including rowhammer because any bitflips in PRM are detected by the integrity validation of the hardware engine.

4.3. Pairwise Secure Shared Memory

For efficient communication between enclaves, STOCKADE introduces a new secure channel using the protected shared memory. The communication channel is a small piece of SGX memory exclusively shared with two enclave parties. To share an EPC page between two enclaves, the enclave pair has to agree on the memory sharing. Once the channel is established, the software-based encryption is not necessary for inter-enclave communication. In STOCKADE, establishing the shared channel is done by hardware. Moreover, the channel can avoid even the encryption by hardware, if the contents fit in the CPU caches for efficient data exchange.

EPCM in SGX contains the EPC mapping information to validate translation. To support the shared memory, the EPCM

entry should be modified to include the sharer information. In addition to the owner enclave's SECS address, an shared EPCM entry includes a single co-owner enclave's SECS. The SECS address is represented physical page number related to start address of EPC [45]. STOCKADE extends EPCM in the same way for the co-owner. In STOCKADE, a single EPC page can be shared only between two enclaves. Figure 6 shows the hardware extension for the ownership checking during a TLB miss at EPC. In **(2)** of the figure, when a memory access occurs to EPC, STOCKADE checks the corresponding EPCM entry and verifies the owner enclave. The EPCM entry has at most two enclaves as its owner and co-owner, and thus only two different enclave contexts can access the EPC page.

Sharing EPC memory: STOCKADE provides a new user level instruction, *ESADD*, to share a EPC page with a co-owner enclave. The instruction takes the EPC page address of owner enclave and the ID of co-owner enclave as inputs. When *ESADD* is invoked, the SGX hardware zeros the page and blocks any access to the page until the corresponding co-owner invokes *ESACCEPT*. *ESACCEPT* performs TLB synchronization to remove old mappings in TLB, and write co-owner down on corresponding EPCM. This step is similar to the dynamic EPC expansion instructions in SGX2 [67].

Once a shared memory is established, two enclaves initiate the local attestation step. Each enclave's digest (*MRENCLAVE*) is passed through the newly established shared memory. The digest uniquely identifies each enclave because it records all the enclave contents (page contents, related position, security flags) [45]. If both enclaves verify each other successfully, then they finalize the channel establishment. Otherwise, STOCKADE destroys the channel.

Communication via APIs: The communication API is similar to ecall and ocall, but all the arguments are secured. When an enclave module invokes the API, the module performs sanitizing and marshalling all the parameters to the structure allocated in shared EPC. After that, STOCKADE copies the parameters into callee's private memory to prevent possible TOCTOU(time-of-check-time-of-use) attacks [34] and performs de-marshalling to execute a callee's function. Because caller and callee belong to different enclaves, CPU state flush is not necessary.

4.4. Sandbox Monitor

A monitor enclave executes a software reference monitor which verifies system calls and returns values. For legitimate system calls, the monitor enclave executes those on behalf of bi-enclaves. The monitor can execute system calls to the kernel or can leverage Intel-provided C standard libraries [20].

Policy Loading: The monitor enclave reads a policy definition file for the bi-enclave which is mutually agreed and shared in advance by the application module provider and the cloud provider. To verify the policy file is correctly loaded, both a bi-enclave and an OS can query the monitor enclave to obtain the digest of the file. Bi-enclave can request the query with the key made during local attestation. Bi-enclave checks the digest value matches with its own policy digest to make sure the file is not manipulated by the OS. The monitor enclave opens an upcall interface only for serving the digest query from the OS. This is similar approach to Intel's attestation to verify an enclave's identity.

```
SYS_NUM ACTION
0       0     // read  ALLOW
1       2     // write NOTIFY
2       1     // open  LOG
42      5     // connect KILL
43      3     // accept TRAP

BLACKLIST 0 "/path/to/top/secret*"
WHITELIST 2 "/path/to/no/secret/[a-z_\-\s0-9\.]"
BLACKLIST 43 "112.233.0.0/16"
```

Listing 1: Example policy file

Monitor as mediator: Listing 1 shows an example policy definition file supported in our system. Similar to seccomp-bpf [44], the monitor enclave filters each system call with system call ID and its arguments based on fine-grained privileges to system resources through a blacklist and a whitelist. To speed up the syscall filtering, the sandbox monitor can adopt an action-based policy. If an action specifies KILL, the monitor enclave sends a request to the kernel to terminate the bi-enclave. On NOTIFY, the monitor makes a notification to the kernel and continue the execution. When the action is LOG, it writes encrypted logs of the system call to a file. On TRAP, the monitor runs a customized logic (e.g. sends a message to the module provider).

When a system call returns, the monitor enclave verifies the return value from the kernel to prevent Iago attacks. For example, Since most system calls return boolean or integer type [85], the monitor can check whether the return values belong to a proper range of values. STOCKADE checks whether *futex*, *locks*, and *semaphore* are not shared between bi-enclave and untrusted world. For a system call that returns a descriptor or reference (e.g. *open*, *socket*), the monitor enclave keeps it in its memory so that the returned descriptor is not substituted and reused. In addition, STOCKADE is resilient to pointer misuses since the reference is not accessible by a bi-enclave.

Trusted accounting: As discusses, the monitor enclave builds mutual trust. The monitor enclave is isolated from both the user-provided bi-enclave and the host operating system, so it works in the neutral area where the bi-enclave and kernel can trust. This model makes new functionalities deployed in the monitoring enable other than system call monitoring. One use case is a trusted resource accounting system for function-as-a-service. The cloud resource usage accounting needs to be verified by both users and provider [58]. For example, the monitor enclave can record tamper-proof evidence (e.g., log file) for network and file usages because all accesses (system calls) to the resource must pass through the monitor enclave. Therefore, the monitor enclave can log resource requests from

Benchmark	SGX enabled Lib	# of allowed interfaces	Modified LOC
NBench	None	0	8
SSL Server	OpenSSL [25]	18	8
File I/O bench	Protected FS [22]	19	12
YCSB (SQL)	SQLite [24]	12	56
ML benchmark	LibSVM [39]	7	8
FTPS Server	OpenSSL & Protected FS	37	20

Table 3: Benchmarks for evaluation.

Type	Attacker	Target	Section
Read / Write / Execute	OS	Bi-enclave, Monitor	§4.2
Read / Write / Execute	Bi-enclave	Other Bi-enclave, Monitor	§4.2
Read / Write / Execute	Monitor	Bi-enclave	§4.2
Read / Write / Execute	Bi-enclave	Outside sandbox	§4.2
Transfer control	Bi-enclave	Other Bi-enclave, Monitor	§4.2
Transfer control	Bi-enclave	Outside sandbox	§4.2
Establish a connection	OS	Bi-enclave, Monitor	§4.3
Evasdrop / Modify	OS	Shared Channel	§4.3
Known Iago attacks	OS	Bi-enclave, Monitor	§4.4

Table 4: Summary of security analysis of STOCKADE

each bi-enclave, and neither a bi-enclave nor the host OS cannot modify the log contents.

5. Discussion

5.1. Development in STOCKADE

Like other SGX-based sandboxing [57], applications using STOCKADE are compartmentalized based on protection domains. STOCKADE executes each protection domain in separate bi-enclaves. Module providers should specify a policy for their modules as discussed in Section 4.4. To communicate among bi-enclaves, STOCKADE provides APIs in the SGX SDK. Application developers simply replace the existing communication APIs (e.g. ecall/ocall) with STOCKADE's APIs so that the module providers do not need to care about new interfaces. Table 3 lists benchmark with allowed interfaces and modified LOC. When the application is already ported into SGX, porting to STOCKADE only requires few lines for initial setup. Most of porting would be done in Makefile, which would be provided by cloud provider. If the application involves mutually untrusted modules written by multiple parties, developer has to map each untrusted module to a bi-enclave.

A case requiring developers' porting efforts is when an application's SGX module is written to communicate with non-enclave code (e.g., ocall to untrusted libraries in non-enclave mode). Because STOCKADE does not allow such communications, developers must port the non-enclave code to run inside another bi-enclave. In addition, each bi-enclave must not access over its sandbox limit to avoid abort page.

5.2. Security Analysis

Table 4 summarizes the security analysis of STOCKADE. An attacker tries to break isolations of bi-enclave by compromising or launching a bi-enclave. However, the attacker cannot run or access untrusted context even with *ret*, *jmp* and *EEXIT* as described in §4.2. The compromised bi-enclave cannot transfer its control to the monitor enclave or another bi-enclave because they are separate enclaves. Also, the compromised bi-enclave cannot create a shared memory with an arbitrary bi-enclave because every shared memory establishment is verified with the attestation. On the other hands, malicious system software cannot eavesdrop or hijack communication to mount the man-in-the-middle attacks [85]. STOCKADE allows the enclave-to-enclave communication channel only via the SGX-protected memory, so the OS or hypervisor is not able to access the communication channel unlike what the original SGX does. The possible attack surfaces are syscall interfaces for the monitor enclave. We implemented known Iago attack protections by checking the file descriptor from syscall [41,56] and POSIX semaphore invocations [62].

Limitation: An attacker may subvert the entire application by gathering code-reuse gadgets one by one across multiple modules and exploiting vulnerable APIs between them. As we mentioned in section 3.5 we leave this our limitation.

	Hardware mode	Simulation mode
SGX-NBench(geomean) [23]	6.0	6.4
SGX ecall / ocall (switchless)	2283.8 / 3748.5	3110.3 / 3783.7
STOCKADE inter-enclave call	-	4930.5

Table 5: Hardware and simulation mode performace comparison (1000 iterations/sec)

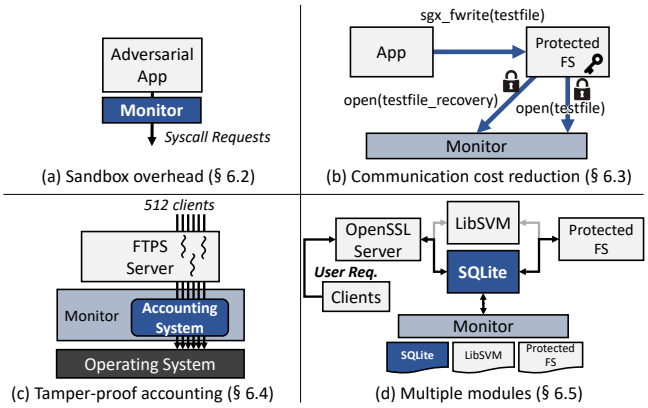

Figure 7: Evaluation scenarios

6. Evaluation

6.1. Methodology

Environment: We evaluate STOCKADE in servers consisting of Intel CPU i7-7700, 64GB DDR4 DRAM, and Ubuntu 16.04 with Linux kernel 4.13.0. To add new hardware features, we use the simulation mode in Intel SGX driver and SDK version 2.2. The simulation mode supports SGX APIs, trusted libraries, and emulation for SGX instructions [5]. Table 5 shows performance comparisons between the hardware mode and the simulation mode. To capture the effect of TLB shootdown, STOCKADE sends ioctl to SGX driver. The driver runs *mov cr3, cr3*, which flushes TLB of the process.

STOCKADE features: STOCKADE hardware features are implemented mostly in SDK and Driver. Disabling ocall from

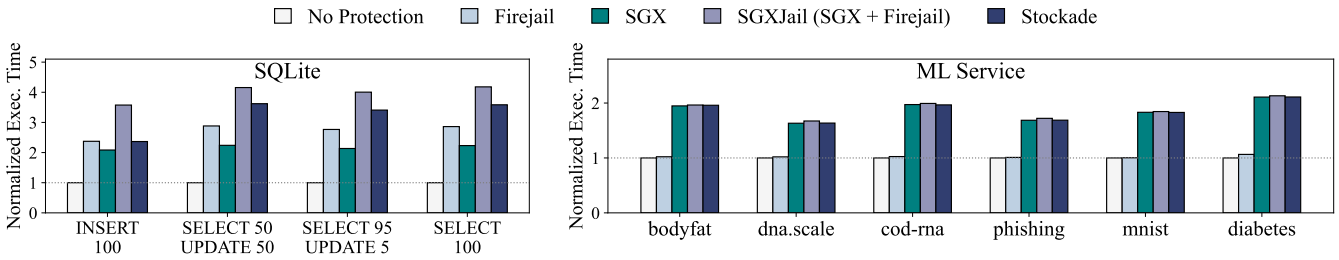

Figure 8: Comparison of execution times between applications run on several secure systems including STOCKADE

bi-enclave is done by modifying emulated *EEXIT* instruction as described in §4.2. We modified *Edger8r* in SDK to generates APIs from Enclave Defined Language (EDL) format. Based on the format, the *Edger8r* per-API data structure for type and boundary checking. For example, pointer arrays require the number of elements to be passed for marshaling. The generated APIs are linked to enclave modules at compile time.

6.2. Sandbox Overhead

In this section, we measure sandboxing overhead of STOCKADE compared to software-based approaches that use process isolation and binary instrumentation. Figure 7 (a) shows the evaluation scenario.

Comparison to process-based filtering: SGXJail is a sandbox that isolates an enclave instance in a separated process confined by seccomp filters [44]. Because SGXJail is not open-sourced, we emualte SGXJail in our platform. To reproduce SGXJail's performance, we use Firejail [4] which leverages Linux namespace and seccomp to provide system call interposition. We compared STOCKADE with four control groups: no protections from SGX nor sandbox (`No Protection`), software-based sandbox (`Firejail`), SGX enclave (`SGX`), and SGX enclave with the software sandbox (`SGXJail`).

Figure 8 shows normalized execution time running SQLite and ML Service. We evaluate SQLite as an I/O-intensive benchmark. SQLite runs a set of queries generated by YCSB [43]. Each set contains 10,000 queries of INSERT, SELECT, and UPDATE according to the uniform key distribution in different ratios. A higher ratio in SELECT queries degrades the performance over INSERT/UPDATE since SELECT generates more syscalls than others to traverse a database. `SGXJail` shows the slowest performance: 1.83× slower on average compared to `SGX` due to frequent system call monitoring by `Firejail`. Meanwhile, `Stockade` is only 1.49× slower on average than `SGX` as `Stockade` passes syscall requests without costly IPC. As a result, `Stockade` shows 18.6% better performance over `SGXJail` while providing stronger hardware-based isolation. In ML services, `SGX`, `SGXJail`, and `Stockade` are similar in speed, less than 3%, as ML inference is CPU-intensive and seldom invokes syscalls.

Comparison to binary instrumentation: We compare STOCKADE with Chancel [28], a software-based binary instrumentation approach for bi-directional isolation like STOCKADE. We run `NBench` which consists of ten benchmarks ex-

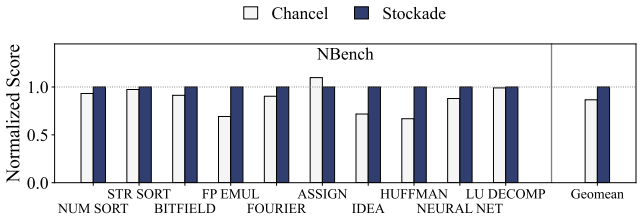

Figure 9: Normalized performance of NBench.

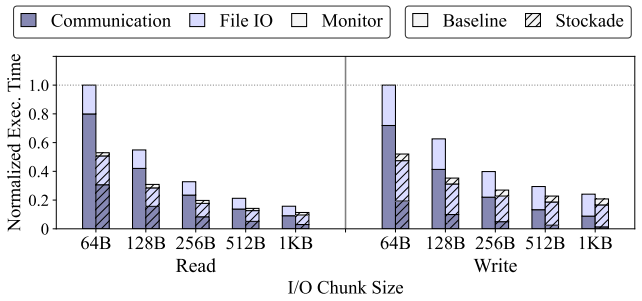

Figure 10: Execution time breakdown in file I/O scenario

posing CPU, FPU, and memory capabilities [19]. For fair comparision, we run STOCKADE with clang-4.0 with -O0 option as the same configure of Chancel. Figure 9 shows normalized performance degradations. We normalize performance by non-confined baseline. In `NBench`, Chancel shows 12.3% performance degradations on average over its baseline. The overhead is from Chancel's binary instrumentation which adds additional instructions (+23.5%). However, STOCKADE runs `NBench` on hardware confined areas and doesn't degrade performance compared to the baseline because `NBench` doesn't communicate to other modules, thereby it does not incur IPC and system call monitoring overhead in the monitor enclave.

6.3. Communication cost reduction with STOCKADE

In this section, we compare the communication cost of STOCKADE (HW) to SW-based approach. Figure 7 (b) represents the scenario. We develop Protected FS which provides integrity and confidentiality protection of files. A module runs the Protected FS, and a communicating module uses the Protected FS to secure its file I/O. The communication is done via the STOCKADE's secure channel, so the content of the file and messages are secured. For the same guarantee, We implement `Baseline` which uses software encryption for secure communication, but it does not confine modules.

Figure 10 presents normalized execution time in various

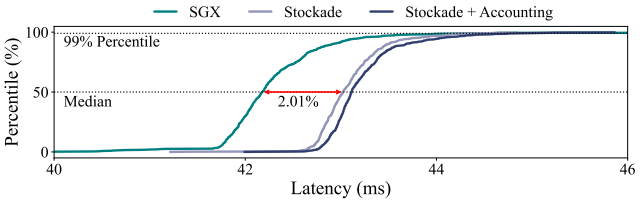

Figure 11: Latency distribution of FTPS requests

chunk sizes. The execution time is normalized to when the chunk size is 64B of `Baseline`. To observe communication cost clearly, we perform file I/O on mounted tmpfs (DRAM backend). We breakdown its performance by three factors. `File I/O` indicates the time taken for file APIs, and `Communication` shows execution time for all communications between modules including message serialization and encryption. `Monitor` is the overhead of STOCKADE monitor. STOCKADE's HW-based communication effectively saves the communication cost than SW-based approaches. As consequence, `Stockade` shows up to 1.38× faster in `read`, 1.16× faster in write comparing to `Baseline` when the chunk size is 1KB. This speedup is caused by the elimination of costly software encryption and decryption. When the chunk size is 64B, `Stockade` shows up to 1.89× faster in read, 1.92× faster in write comparing to `Baseline`. Fine-grained file I/O operations causes more frequent communication overhead between modules, thus it stresses the communication cost.

6.4. Tamper-proof Accounting System

In a cloud system, multiple tenants compete to use the resource from limited hardware. Trusted tamper-proof accounting systems provide useful feature to securely manage resources across the tenants [50, 58, 88]. STOCKADE can provide such a system using a monitor enclave as shown in figure Figure 7 (c). Because the monitor enclave intervenes all the system calls from bi-enclaves, it can account every stat of system calls (e.g. file I/O access, network request, memory consumption). STOCKADE provides a trustworthy accounting reports that the service provider can verify through attestation.

To demonstrate the scenario, we implement the following accounting system in the monitor enclave. We spawn 512 clients; each of them sends a request for a 1MB file to secure FTP server. The server takes the request and sends back corresponding files to clients. Handling the requests, the monitor enclave logs per-request resource consumption in its secure memory. Figure 11 shows the latency distribution of the requests with three different systems: `SGX`, `Stockade`, `Stockade + Accounting`. The median latency of `Stockade` is only 2% slower than `SGX` and the `Stockade + Accounting` shows negligible overhead in median and tail latencies. This implies that mutually trustful secure accounting can be achieved without large overhead via STOCKADE.

6.5. Secure Services with Multiple Modules

To evaluate combined benefits of STOCKADE, we build a secure query server containing DB and ML services. Figure 7 (d)

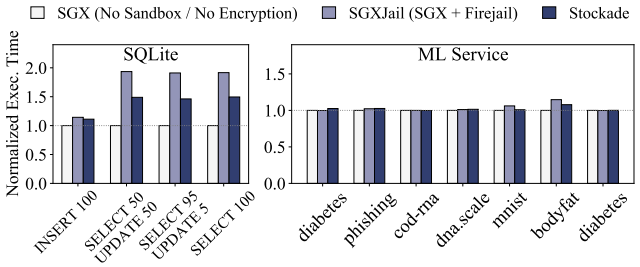

Figure 12: Normalized execution time of distributed query server scenarios

describes the system; each service consists of multiple modules and they are isolated to each bi-enclave. The modules attests to each other and establishes secure communication channels at first. The monitor enclave reads a system call policy that specifies the least privilege of each module. For example, SQLite is not allowed to use network-related system calls such as *connect* or *send*. Each service uses confined third-party modules for secure network communication `(SSL Server)` and safe file management `(Protected FS)`. Whenever a client sends security-sensitive data to DB service via `SSL Server`, SQLite module asks the protected file system to store or load the data. `Protected FS` has its own encryption key which is not accessible from `SSL Server` or SQLite module. In addition, the LibSVM module handles a prediction with the inference model stored in the local file system encrypted by `Protected FS`. Even though `SSL Server` is compromised, the inference model cannot be stolen due to STOCKADE's isolation.

Figure 12 shows the performance of the two services. `SGX` indicates each module runs in an enclave and communicates with each other via unprotected channel without encryption. Each enclave is not confined, so it can perform any system calls. However, in `SGXJail` and `Stockade`, every system call has verified. The sandbox overhead of `Stockade` incurs a slowdown as shown in §6.2, but the efficient hardware-based encryption amortizes the performance degradation. As a result, for I/O-intensive SQLite, `Stockade` shows 38.9% overhead compared to `SGX` and 19.5% better to `SGXJail` on average. Despite communication overhead from adding monitor enclave, STOCKADE outperformed `SGXJail` by leveraging hardware encryption. For ML service, the performance among the three models are similar, but some overhead is added in `SGXJail` and `Stockade` due to secured communication.

7. Conclusion

This paper explores a new extension model, STOCKADE, for SGX to support distributed sandboxing. With a minor change in SGX, STOCKADE provides strong sandboxing. In addition, it allows the mutually trusted monitor enclave between the user bi-enclave and the operating system, by filtering system call requests and validating return values. The performance results show the viability of STOCKADE. In multi-module,

STOCKADE shows an average 19.5% speedup for SQLite, and 1.4% speedup for ML service over the SW sandbox approach.

8. Acknowledgements

This work was supported by Institute for Information & communications Technology Promotion (IITP2017-0-00466). The grant is funded by the Ministry of Science and ICT, Korea. This work was also partly supported by Samsung Electronics Co., Ltd. (IO201209-07864-01).

References

[1] AWS Lambda Service. https://aws.amazon.com/lambda/.

[2] Choosing an App Engine environment. https://cloud.google.com/appengine/docs/the-appengine-environments.

[3] Comodo Antivirus versions up to 12.0.0.6810 are vulnerable to Local Privilege Escalation due to CmdAgent's handling of COM clients. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-3969. Jan, 2019.

[4] firejail. https://firejail.wordpress.com/.

[5] How to Run Intel Software Guard Extensions' Simulation Mode. https://software.intel.com/content/www/us/en/develop/blogs/usage-of-simulation-mode-in-sgx-enhanced-application.html.

[6] Instructions for RVS Sandbox Environment. https://developer.amazon.com/docs/in-app-purchasing/iap-rvs-setup-sandbox.html.

[7] Introduction to Intel® Memory Protection Extensions. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-memory-protection-extensions.html.

[8] Mozilla Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox.

[9] musl-libc. http://musl.libc.org/.

[10] Papal Sandbox Notification. https://developer.paypal.com/developer/notifications/.

[11] Possible seccomp bypass due to seccomp policies that allow the use of ptrace. https://nvd.nist.gov/vuln/detail/CVE-2019-2054. May, 2019.

[12] Process-injection: Ptrace System Calls. https://attack.mitre.org/techniques/T1055/008/. Jan, 2020.

[13] Setting the environment occurs across a privilege boundary from Bash execution, aka "ShellShock". https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-6271. Sep, 2014.

[14] Windows Kernel Local Elevation of Privilege Vulnerability. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-17087. August, 2020.

[15] ARM Developer Suite Developer Guide. https://developer.arm.com/documentation/dui0056/d/caches-and-tightly-coupled-memories/memory-management-units/memory-access-permissions-and-domains, 2009.

[16] Papal Sandbox user guide. https://web.archive.org/web/20070128140333/http://developer.paypal.com/en_US/pdf/PP_Sandbox_UserGuide.pdf, 2012.

[17] Welcome to Developer Sandbox. https://sandbox.ebay.com/, 2012.

[18] Building Your Dev and Test Sandbox with Windows Azure Infrastructure Services. https://azure.microsoft.com/ko-kr/resources/videos/build2013-dev-test-sandbox-with-windows-azure-infrastructure-services/, 2013.

[19] NBench. https://github.com/petabridge/NBench, 2015.

[20] Intel(R) Software Guard Extensions SDK Developer Reference for Linux* OS, 2016.

[21] Sandbox Types and Templates. https://help.salesforce.com/articleView?id=create_test_instance.htm&type=5, 2016.

[22] Overview of Intel Protected File System Library Using Software Guard Extensions. https://software.intel.com/content/www/us/en/develop/articles/overview-of-intel-protected-file-system-library-using-software-guard-extensions.html, 2017.

[23] SGX enabled NBench. https://github.com/utds3lab/sgx-nbench, 2017.

[24] SGX enabled SQLite. https://github.com/yerzhan7/SGX_SQLite, 2017.

[25] sgx-openssl. https://github.com/sparkly9399/SGX-OpenSSL, 2017.

[26] Affected Processors: Transient Execution Attacks & Related Security Issues by CPU. https://software.intel.com/security-software-guidance/processors-affected-transient-execution-attack-mitigation-product-cpu-model, 2021.

[27] Adil Ahmad, Byunggill Joe, Yuan Xiao, Yinqian Zhang, Insik Shin, and Byoungyoung Lee. Obfuscuro: A Commodity Obfuscation Engine on Intel SGX. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[28] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. CHANCEL: Efficient Multi-client Isolation Under Adversarial Programs. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

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

[30] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure Linux Containers with Intel SGX. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[31] Adam Barth, Charles Reis, Collin Jackson, and Google Inc. Google Chrome Team.

[32] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding Applications from an Untrusted Cloud with Haven. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[33] Andrea Biondo, Mauro Conti, Lucas Davi, Tommaso Frassetto, and Ahmad-Reza Sadeghi. The Guard's Dilemma: Efficient Code-Reuse Attacks Against Intel SGX. In *USENIX Security Symposium (USENIX Security)*, 2018.

[34] Matt Bishop, Michael Dilger, et al. Checking for Race Conditions in File Accesses. *Computer Systems*, 1996.

[35] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[36] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium (USENIX Security)*, 2018.

[37] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *USENIX Security Symposium (USENIX Security)*, 2019.

[38] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast Byte-Granularity Software Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[39] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A Library for Support Vector Machines. *ACM Transactions on Intelligent Systems and Technology*, 2011.

[40] Stephen Checkoway and Hovav Shacham. Iago Attacks: Why the System Call API is a Bad Untrusted RPC Interface. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[41] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A Virtualization-based Approach to Retrofitting Protection in Commodity Operating Systems. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2008.

[42] Yueqiang Cheng, Zhi Zhang, and S. Nepal. Still Hammerable and Exploitable: on the Effectiveness of Software-only Physical Kernel Isolation. *ArXiv*, 2018.

[43] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking Cloud Serving Systems with YCSB. In *ACM Symposium on Cloud Computing (SoCC)*, 2010.

[44] Jonathan Corbet. Seccomp and sandboxing. *LWN. net, May*, 25, 2009.

[45] Victor Costan and Srinivas Devadas. Intel SGX Explained. In *IACR Cryptology ePrint Archive*, 2016.

[46] Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz. Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity. In *Network and Distributed System Security Symposium (NDSS)*, 2015.

[47] Rongzhen Cui, Lianying Zhao, and David Lie. Emilia: Catching iago in legacy code. 2021.

[48] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege Escalation Attacks on Android. In *International Conference on Information Security (ICS)*, 2010.

[49] Tal Garfinkel, Ben Pfaff, Mendel Rosenblum, et al. Ostia: A Delegating Architecture for Secure System Call Interposition. In *Network and Distributed System Security Symposium (NDSS)*, 2004.

[50] David Goltzsche, Manuel Nieke, Thomas Knauth, and Rüdiger Kapitza. AccTEE: A WebAssembly-Based Two-Way Sandbox for Trusted Resource Accounting. In *International Middleware Conference (Middleware)*, 2019.

[51] Antonios Gouglidis, Ioannis Mavridis, and Vincent C Hu. Security policy verification for multi-domains in cloud systems. *International Journal of Information Security*, 2014.

[52] Ben Gras, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *USENIX Security Symposium (USENIX Security)*, 2018.

[53] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016.

[54] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the Web up to Speed with WebAssembly. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2017.

[55] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with OpenLambda. In *USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2016.

[56] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett Witchel. InkTag: Secure Applications on an Untrusted Operating System. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[57] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[58] Seongwook Jin, Jinho Seol, Jaehyuk Huh, and Seungryoul Maeng. Hardware-Assisted Secure Resource Accounting under a Vulnerable Hypervisor. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2015.

[59] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. COIN Attacks: On Insecurity of Enclave Untrusted Interfaces in SGX. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[60] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre Attacks: Exploiting Speculative Execution. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[61] Esmaeil Mohammadian Koruyeh, Khaled N Khasawneh, Chengyu Song, and Nael Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2018.

[62] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive Trusted Metadata for Efficiently Verified Untrusted System Services. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[63] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In *USENIX Security Symposium (USENIX Security)*, 2017.

[64] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX security symposium (USENIX Security)*, 2017.

[65] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A Two-Way Sandbox for x86 Native Code. In *USENIX Annual Technical Conference (ATC)*, 2014.

[66] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading Kernel Memory from User Space. In *USENIX Security Symposium (USENIX Security)*, 2018.

[67] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave. In *Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[68] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments. *ArXiv*, 2019.

[69] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. Swivel: Hardening WebAssembly against Spectre. *arXiv preprint arXiv:2102.12730*, 2021.

[70] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel MPX Explained: A Cross-layer Analysis of the intel MPX System Stack. *ACM on Measurement and Analysis of Computing Systems*, 2018.

[71] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *International Symposium on Computer Architecture (ISCA)*, 2020.

[72] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel MPK). In *USENIX Annual Technical Conference (ATC)*, 2019.

[73] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. Rethinking the Library OS from the Top Down. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2011.

[74] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. SGX-LKL: Securing the Host OS Interface for Trusted Execution. *arXiv preprint arXiv:1908.11143*, 2019.

[75] Octavian Purdila, Lucian Adrian Grijincu, and Nicolae Tapus. LKL: The Linux kernel library. In *RoEduNet IEEE International Conference*, 2010.

[76] Ashay Rane, Calvin Lin, and Mohit Tiwari. Raccoon: Closing Digital Side-Channels through Obfuscated Execution. In *USENIX Security Symposium (USENIX Security)*, 2015.

[77] Mark Seaborn and Thomas Dullien. Exploiting the DRAM rowhammer bug to gain kernel privileges. https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html, 2015.

[78] David Sehr, Robert Muth, Cliff L Biffle, Victor Khimenko, Egor Pasko, Bennet Yee, Karl Schimpf, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *USENIX Security Symposium (USENIX Security)*, 2010.

[79] Jaebaek Seo, Daehyeok Kim, Donghyun Cho, Insik Shin, and Taesoo Kim. FLEXDROID: Enforcing In-App Privilege Separation in Android. In *Network and Distributed System Security Symposium (NDSS)*, 2016.

[80] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *NDSS*, 2017.

[81] Youren Shen, Yu Chen, Kang Chen, Hongliang Tian, and Shoumeng Yan. To Isolate, or to Share? That is a Question for Intel SGX. In *Asia-Pacific Workshop on Systems (APSys)*, 2018.

[82] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. Occlum: Secure and Efficient Multitasking Inside a Single Enclave of Intel SGX. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[83] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[84] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing your faults from telling your secrets. In *ACM on Asia Conference on Computer and Communications Security (Asia CCS)*, 2016.

[85] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. In *Network and Distributed System Security Symposium (NDSS)*, 2017.

[86] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W Fletcher. Microscope: Enabling microarchitectural replay attacks. In *International Symposium on Computer Architecture (ISCA)*, 2019.

[87] Jakub Szefer. Survey of Microarchitectural Side and Covert Channels, Attacks, and Defenses. *Journal of Hardware and Systems Security*, 2019.

[88] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. T-Lease: A Trusted Lease Primitive for Distributed Systems. In *ACM Symposium on Cloud Computing (SoCC)*, 2020.

[89] Bohdan Trach, Oleksii Oleksenko, Franz Gregor, Pramod Bhatotia, and Christof Fetzer. Clemmys: Towards secure remote execution in faas. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, 2019.

[90] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *European Conference on Computer Systems (EuroSys)*, 2014.

[91] Chia-Che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *USENIX Annual Technical Conference (ATC)*, 2017.

[92] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *USENIX Security Symposium (USENIX Security)*, 2017.

[93] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic Rowhammer Attacks on Mobile Platforms. In *ACM Conference on Computer and Communications Security (CCS)*, 2016.

[94] Mario Villamizar, Oscar Garces, Lina Ochoa, Harold Castro, Lorena Salamanca, Mauricio Verano, Rubby Casallas, Santiago Gil, Carlos Valencia, Angee Zambrano, et al. Infrastructure cost comparison of running web applications in the cloud using AWS lambda and monolithic and microservice architectures. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, 2016.

[95] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient Software-Based Fault Isolation. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1993.

[96] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. SGXJail: Defeating Enclave Malware via Confinement. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2019.

[97] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.

[98] Yuan Xiao, Xiaokuan Zhang, Yinqian Zhang, and Radu Teodorescu. One Bit Flips, One Cloud Flops: Cross-VM Row Hammer Attacks and Privilege Escalation. In *USENIX Security Symposium (USENIX Security)*, 2016.

[99] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[100] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2009.

[101] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *ACM Conference on Computer and Communications Security (CCS)*, 2014.