ESem: To Harden Process Synchronization for Servers

Zhanbo Wang[1,2], Jiaxin Zhan[1,3], Xuhua Ding[4], Fengwei Zhang[3,1,†], Ning Hu[2]
[1]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, China*
[2]*Peng Cheng Laboratory, China*
[3]*Department of Computer Science and Engineering, Southern University of Science and Technology, China*
[4]*Singapore Management University*
{12131105,zhanjx}@mail.sustech.edu.cn,xhding@smu.edu.sg
zhangfw@sustech.edu.cn,hun@pcl.ac.cn

ABSTRACT

Process synchronization primitives lubricate server computing involving a group of processes as they ensure those processes to properly coordinate their executions for a common purpose such as provisioning a web service. A malfunctioned synchronization due to attacks causes friction among processes and leads to unexpected, and often hard-to-detect, application transaction errors. Unfortunately, synchronization primitives are not naturally protected by existing hardware-assisted isolation techniques e.g., SGX, because their process-oriented isolation conflicts with the primitive's demand for cross-process operations.

This paper introduces the *Enclave-Semaphore* service (ESem) which shelters application semaphores and their operations against kernel-privileged attacks. ESem encapsulates all semaphores in the platform with a dedicated SGX enclave and polices accesses from users processes, thus ensuring a consistent view of the data and resources shared among collaborative processes. Although ESem provides secure semaphores only, it supports all kinds of synchronization needs, owning to the expressiveness of semaphores.

We have built a prototype of ESem and conducted rigorous evaluation with micro-benchmarks, macro benchmark and real-world applications including Redis and Apache HTTP Server. ESem incurs only a modest performance overhead (around 2%) to the legacy systems. We also run a case study to demonstrate attacks against the synchronization in an SGX-hardened file server and how ESem neutralizes the attacks successfully with only one function call change to the applications. All these experiments show that ESem is lightweight yet effective solution to the security hole left open by existing isolation schemes.

CCS CONCEPTS

• **Security and privacy → Systems security**.

KEYWORDS

Secure synchronization, Kernel semaphore, SGX enclave

ASIA CCS '24, July 1–5, 2024, Singapore, Singapore

ACM ISBN 979-8-4007-0482-6/24/07
https://doi.org/10.1145/3634737.3657025

ACM Reference Format:
Zhanbo Wang[1,2], Jiaxin Zhan[1,3], Xuhua Ding[4], Fengwei Zhang[3,1,†], Ning Hu[2]. 2024. ESem: To Harden Process Synchronization for Servers. In *ACM Asia Conference on Computer and Communications Security (ASIA CCS '24), July 1–5, 2024, Singapore, Singapore.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3634737.3657025

1 INTRODUCTION

Many servers, such as web, database, and multimedia, are composed of a collection of collaborative processes or threads which undertake different computation tasks with intensive data and resource sharing [8, 29, 32]. As these component processes run concurrently for the sake of high performance, it is imperative for them to keep synchronized when accessing shared data and resources. A disrupted synchronization lead to processes to have inconsistent views and thus fail to coordinate their accesses in an orderly fashion. Hence, an attack upon process synchronization could result in not only undesired resource utilization, but also application logic errors which are hard to detect in real-time. For instance, consider a typical producer and consumer scenario where a lock is used to synchronize their accesses to a shared memory buffer. When the adversary (presumably with the kernel privilege) presents inconsistent lock states to the producer and the consumer, the data in the buffer will not be processed with atomicity. Moreover, if the applications rely on synchronization to fend off race condition attacks, a kernel adversary can easily scheme and the attack regardless whether an individual process is hardened or not.

Despite its critical role for server computing, the security of process synchronization has not been systematically investigated, in contrast to various user-space isolation techniques [2, 13, 19, 35, 42, 47]. Although Patel et. al. proposed adversarial synchronization [28], they used synchronization only as the vehicle to deliver their attacks, rather than safeguarding it. To the best of our knowledge, our work is the first that identifies the threat against process synchronization and proposes a countermeasure.

We observe that, under the threat of a malicious kernel, secure synchronization plugs the hole which is left open between isolated user-space executions. With intro-process and inter-process protection, sensitive server computing receives a fully-covered shield against attacks from a compromised kernel. Clearly, to protect synchronization without intro-process isolation does not attain much security assurance. Thus, the main challenge of securing process

[†]Fengwei Zhang is the corresponding author.

synchronization stems from the conflict between intro-process isolation and inter-process sharing. No existing hardware-assisted isolation technique has the built-in support for both needs.

In this paper, we propose to harden process synchronization using Intel SGX [20], a hardware feature available on Intel processors for server and cloud computing though being withdrawn from client platforms [23]. Out of various synchronization methods, we choose *semaphore* as our focus since it is the most expressive primitive supporting all types of synchronization paradigms. Specifically, we propose ESem, an enclave-assisted semaphore service provided by the kernel for collaborative processes to use. A system-wide enclave is dedicated to host semaphores used by all processes in the platform. To overcome the aforementioned challenge, we introduce the notion of "enclave roaming" in the sense that the semaphore enclave is not confined to a single process's address space but made accessible to all processes upon the kernel's approval. As the semaphore enclave is shared across multiple processes, ESem enforces access controls within the enclave to prevent unwanted or illegal accesses to semaphore variables inside.

ESem ensures that collaborative processes perceive the same state and order of synchronization even at the presence of a kernel-privileged adversary. Note that, it neither deals with a rogue process which abuses the semaphore, nor protects intro-process computation. A security-savvy developer needs to apply both intro-process isolation (e.g., using an enclave), secure communication channel and ESem to get a full coverage protection.

Caveat. Our hardened semaphore service still relies on the kernel to manage and enforce its access control policies against processes. This is in the same vein as SGX design in that the kernel is tasked to create and managed enclaves, though it is not in SGX's trusted computing base.

As ESem targets server applications, we design it with performance in mind in order to minimize the impact to user processes especially to their concurrency. For instance, ESem provides two ways to bind semaphores with enclave threads to meet the concurrency demands due to different synchronization profiles. Although entering and exiting from enclaves cost a few more microseconds than regular memory accesses, the incurred overhead is not big enough to downgrade the server's overall performance. Applications with intensive use of semaphores can opt for Intel's "switchless" enclave invocation to streamline their semaphore accesses. As we navigate through the security and performance challenges, we also aim to minimize kernel changes and maintain compatibility with the POSIX standard.

We have built a prototype of ESem on an Intel `i7 9700k` machine with Linux Ubuntu `22.04` installed and rigorously evaluated its performance and security. In our case study with File Vault [1], we first demonstrate two attacks against its synchronization: *file tree race* and *service thread blocking* and then harden it with ESem. The outcomes show that ESem effectively safeguards against kernel-level attacks with slight change of the application. We conduct thorough performance assessments with micro- and macro-benchmarks as well as real-world application. The experiments report modest performance overhead incurred by ESem.

The key contributions of this paper are summarized as follows:

- We design ESem that preserves synchronization semantics against kernel-privileged attacks, so that a group of collaborative processes always maintain a consistent view of their shared data and resources.
- ESem maintains the same degree of parallelism as Linux's semaphore service and is compatible with existing POSIX APIs.
- We implement a prototype of ESem and measure its performance. We also conduct a case study to show ESem's benefits and usage.

Organization. The next section briefly explains the background of *POSIX semaphore*, *Multi-thread in SGX enclaves*, and *SGX switchless function call*. Section 3 overviews the design of ESem. The two different design methods for semaphores in enclaves are presented in Section 4. We describe our prototype implementation in Section 5. We conduct a comprehensive case study in Section 6, followed by the evaluation of ESem performance in Section 7. We present related work in Section 8. Section 9 further discusses ESem. Finally, Section 10 concludes the paper.

2 BACKGROUND

2.1 POSIX semaphore

The POSIX semaphore implementation in glibc is an interface for inter-process synchronization that adheres to the POSIX standard. Under the hood, when a process invokes POSIX semaphore operations through GlibC, the library translates these requests into appropriate system calls to the Linux kernel.

Storage. POSIX semaphores are usually defined using the `sem_t` data structure in a shared memory region mapped to user space. This structure contains the current value of the semaphore, which indicates the number of resources available or the number of threads that can pass without blocking.

System Call Invocation. Contrary to some beliefs, operations on POSIX semaphores don't always result in direct system calls. The glibc implementation attempts to handle some operations quickly in user space without invoking a system call. For instance, if a `sem_post()` operation (which increments the semaphore value) is called and no threads are waiting on the semaphore, glibc can optimize this by simply updating the semaphore's value in user space.

FUTEX. The "fast userspace mutex", or FUTEX, is a tool in kernel for efficient thread synchronization. The main idea is that they allow many locking operations to be handled in user space, without system calls. Only when contention occurs (i.e., a resource is not available, and a thread needs to be put to sleep or woken up) does a system call become necessary. When the glibc's semaphore implementation detects contention, it resorts to futex system calls to manage the blocking and waking up of threads.

2.2 Multi-thread in SGX Enclaves

SGX introduces a novel approach to secure computation, providing hardware-enforced trusted execution environments called enclaves. At the heart of SGX's security model is the enclave, a protected memory region (EPC) within an application's address space that's shielded against external access, including privileged software like

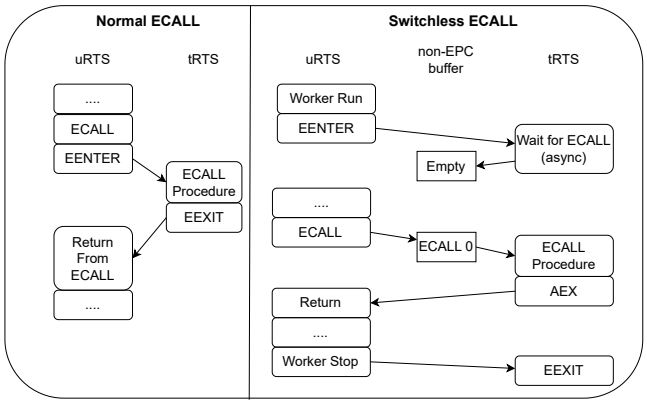

Figure 1: Enclave for One User App

the operating system or the hypervisor. When it comes to threading, SGX supports multi-threaded enclaves. Threads inside an enclave are analogous to regular application threads, but with the added assurance of running within the enclave's secure boundaries. It's crucial to understand that while the enclave protects against external snooping and tampering, it does not inherently provide inter-thread synchronization.

Each enclave thread is associated with a Thread Control Structure (TCS). The TCS is a data structure used to manage and control threads entering and exiting the enclave. Each TCS is stored in an EPC page whose content is not directly accessible, with an EPCM entry type of PT_TCS [16]. When an application wants to enter an enclave, it does so using a specific TCS. Each TCS is tied to a single enclave thread, meaning it can only be used by one logical processor at a time.

As traditional applications can utilize Thread Local Storage (TLS) to store thread-specific data, SGX enclaves can also maintain their TLS. The enclave's TLS is protected within the enclave's memory boundary, When a thread enters an enclave via the `ecall` interface, it does so with a reference to a specific TCS. The TCS, in turn, contains an offset pointing to the start of the TLS for that thread. As a result, each enclave thread can have its own secure, private storage space.

The state of each enclave thread, including its registers and call stack, is saved outside of the enclave in a special unprotected area when the thread exits the enclave, either due to an `ocall` (calling out of the enclave) or other reasons like context switches. This state is encrypted by SGX to ensure its confidentiality. When the thread re-enters the enclave, the saved state is restored.

2.3 SGX Switchless Function Call

Switchless calls in SGX are designed to handle short-duration tasks more efficiently by avoiding the overhead associated with context switches (Figure 1). Moreover, the performance of Switchless can be greatly improved after optimization of algorithm and configuration [39, 46]. The basic premise involves utilizing dedicated worker threads that busy-wait, or spin, on specific memory locations to detect and process tasks.

Shared Buffers. For switchless calls, both the enclave and the untrusted application pre-allocate shared buffers in memory. These buffers are used to pass information and tasks between the enclave and the untrusted application.

Worker Threads. Both the enclave and the untrusted application create dedicated worker threads. These threads are responsible for handling the tasks in the switchless model.

Busy-Waiting. The busy-waiting threads spin on the shared buffers, continually checking (or polling) for new tasks. Specifically, they're looking for flags or indicators in these buffers that signal the presence of a new task. When a task is detected, the appropriate worker thread processes the task immediately. For example, when the untrusted part of the application has a small task for the enclave, it places the task in the shared buffer and sets an indicator. The enclave's busy-waiting worker thread detects this and processes the task without the typical overhead of an `ecall`. The reverse is true for `ocall`, where an enclave task for the outside application is placed in the buffer, and an untrusted worker thread picks it up.

Trade-offs. While the busy-waiting approach in switchless calls minimizes the latency associated with enclave entry and exit, it is at the expense of increased CPU utilization. Continuously spinning threads can consume significant CPU cycles, potentially leading to higher power consumption and reduced availability of CPU cycles for other tasks.

3 DESIGN

3.1 Synopsis

Attack Model. We consider a software adversary which resides in a server platform and attempts to manipulate data objects (e.g., spin-locks, mutexes and semaphores) for process synchronization in the same platform in order to induce erroneous data processing and undesired resource usage to victim applications' executions. The adversary can either be a user-space malware who breaks the kernel's process isolation to access other processes' semaphores or one which has gained the kernel privilege by exploiting the kernel's vulnerabilities.

The adversary has many ways to disrupt process synchronization. For instance, it can alter a lock to an open state while one process has acquired it. It can present two different semaphore values to a produce and a consumer. Nonetheless, we do not consider denial-of-service attacks that aims at resource availability only, such as denying a process from accessing its semaphore or a critical section, because a kernel-privileged adversary can always achieve it as long as the kernel manages resources in the platform and the attack does not lead to incorrect process state. It is also outside of our design scope to protect an application's internal code and data as it can be addressed by using existing techniques. The adversary also cannot intercept the secure communication channel between applications and ESem.

Goals. Our goal is to design a hardened synchronization mechanism that resists attacks from the aforementioned adversary, namely to preserve the synchronization semantics so that participating processes *always* maintain a consistent view of shared data and resources. Besides security assurance, we also aim to keep the degree of parallelism provided by the original synchronization

mechanisms, and remain compatible with the existing POSIX APIs exposed to application developers, so that legacy applications can benefit from the new scheme without code modification.

Straw-man Solutions. To protect process synchronization against kernel-privileged attacks is not as straightforward as it appears. One may suggest to follow the approach of Occlum [35] which imports multiple distrusted applications into one enclave. However, it is unrealistic to place multiple heave-engineered server processes into one enclave due to address space conflicts, not to mention significant performance drop due to intensive I/O and system calls. Another tentative solution is to leverage encrypted virtual machine techniques such as Intel TDX [21], Arm CCA [10] or AMD SEV [7] so that all collaborative processes run in one "trusted" domain. We argue that this approach does not *truly* solve the problem because it only counters attacks from outside of the domain, not against any from the kernel servicing the processes in concern. Moreover, for servers hosted by local machines instead of the cloud, adding a layer of virtualization and memory encryption incur a significant performance overhead without the reward.

Our Approach. We design a novel synchronization mechanism named as the *Enclave-Semaphore* (or ESem) which provides a hardened semaphore service for applications to use. We create an enclave named as the *semaphore enclave* (or *s-enclave* for short) to host all secure semaphores as well as their operations. The s-enclave is dynamically mapped by the kernel to all processes requesting ESem service. As shown in Figure 2, it holds three semaphores x, y and z for five processes. Among them process $P1$, $P2$ and $P3$ have their own enclaves protecting sensitive data and code related to shared resources in the critical section, while $P4$ and $P5$ do not have. A process uses `ecall` to enter the enclave and operate its semaphores through predefined P and V functions inside the enclave.

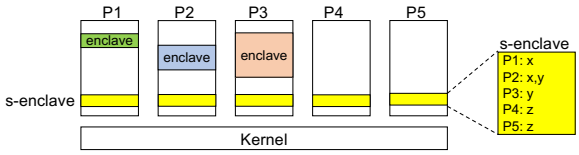

Figure 2: Illustration of the design approach of ESem

The s-enclave has an internal access control mechanism to authenticate the process that intends to enter the enclave and access a semaphore. For processes with their own enclaves, their semaphores can only be accessed after a key based authentication, while for other processes, the s-enclave relies on the kernel's process management.

Caveat. We reiterate that ESem is proposed as a hardened semaphore service for secure process synchronization and it is beyond the scope of ESem to protect sensitive code and data within an application. As such, ESem is geared for both types of processes as shown in Figure 2.

3.2 Design Overview

The architecture of ESem consists of the s-enclave, the *ESem manager* in the kernel and the *ESem glue code* in Glibc, as depicted in Figure 3. The ESem manager initiates the s-enclave after kernel launch and handles ESem related requests from userspace applications at runtime such as creating a semaphore and opening a semaphore. Note that the kernel still undertakes the duty of process management for semaphores accesses based on `uid`, `pid` and others.

The ESem glue code shields the details of ESem operations from applications so that they continue to follow POSIX APIs. The changes on the C library is necessary as invoking a function in the enclave is more complicated than ordinary function calls.

The internals of the s-enclave are also shown in Figure 3, including the memory section dedicated to store semaphore objects. We call them *e-semaphores* to avoid ambiguity with the conventional semaphore objects managed by the kernel if the context is not clear enough to differentiate them. The s-enclave is configured to be multi-threaded in order to support concurrent executions. Its *TCS allocator* manages a pool of SGX Thread Control Structures (TCS)es and determines how a user process uses one enclave thread to access its semaphore. The *authenticator* module in the s-enclave guards the entry to the enclave. It checks whether the process that issues the `ecall` to operate a semaphore is legitimate or not. After passing the authentication, the corresponding thread uses the proper *operator* function to operate the semaphores.

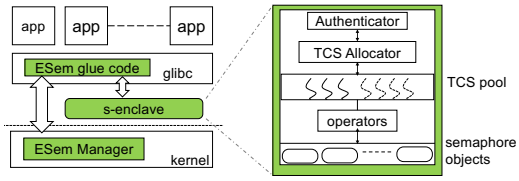

Figure 3: The architecture overview of ESem

Next, we describe how the kernel manages the s-enclave and then present the enclave's details in the next section.

3.3 Semaphore Enclave Management

After system bootstrap, the s-enclave is created and initialized by the ESem manager in the kernel using a dummy process (denoted by P_0) which has no other application code pages. The ESem manager loads the enclave at one 1-GB aligned virtual address such that the ensuing one gigabytes region are *not* used by most applications. The selection of the base address is to avoid VA collisions between the enclave and applications.

The core task of semaphore enclave management is to support *enclave roaming*, namely to make the s-enclave accessible to a process which the kernel grants the permissions to operate a semaphore in the s-enclave. Specifically, when process P is granted by the kernel to create or open a semaphore x, the ESem manager copies the entry in the Page-Directory-Page-Table (PDPT) page that maps the s-enclave in the dummy process to the corresponding entry in the PDPT page used for process P. Figure 4 below illustrates this step with the dashed arrow indicating the added mapping to P's paging hierarchy. As a result, all processes accessing ESem's semaphore share the same enclave mappings. When a process closes its semaphore, the ESem manager removes the mapping from its PDPT page entry.

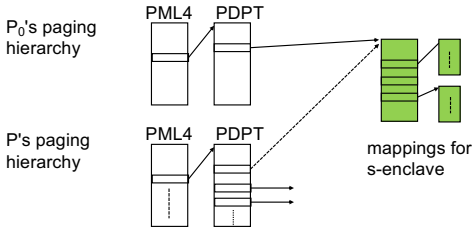

Figure 4: Enclave Roaming: to dynamically merge mappings of the s-enclave to process P's virtual memory

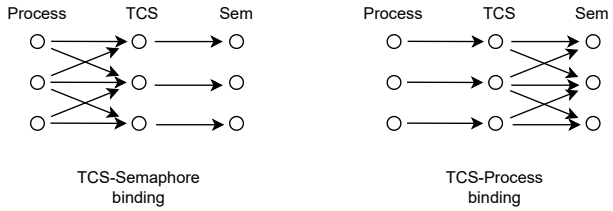

Figure 5: Two Mapping Schemes for Enclave Thread.

Note that adding the enclave to different user processes do not interfere with enclave execution since the hardware uses the same mappings to access EPC pages in the enclave.

4 THE SEMAPHORE ENCLAVE INTERNALS

In our design, different enclave threads undertake different tasks and interface with different untrusted address spaces. Confronting this complexity, our design handles the allocation of different enclave threads and enforces allocation within the enclave, providing additional security properties. This specificity in thread management and allocation is leveraged to facilitate two semaphore access paradigms.

4.1 Semaphore Access Control

Since the s-enclave is shared across multiple processes, a rogue process may invoke the enclave to operate semaphores not allocated to it. Note that under the current system, a prerequisite of such a misuse is to breach process isolation enforced by the kernel since the victim semaphore is not mapped to the rogue process. In contrast, it is much easier to launch the attack as the s-enclave is accessible to any process with an e-semaphore.

ESem relies on the Authenticator inside the s-enclave (as shown in Figure 3) to check whether a process's request to access an e-semaphore should be allowed or denied. The checking is performed differently according to whether the target semaphore is associated with an application enclave. As shown in Table 1, the s-enclave maintains a semaphore metadata table which keeps the data used for authentication.

Table 1: Semaphore metadata table with an entry example

Semaphore	PID	Key	TCS
x	30000	0xFE...23	...
y	30001	-	...

During e-semaphore creation, the owner process may request for a key binding if it has an application enclave. Through a local attestation, the owner process's enclave and the s-enclave share a 128-bit secret key k which is attached to the created e-semaphore. On subsequent operations upon the e-semaphore, the process appends an HMAC of its operation argument, including the e-semaphore name and operation type. When receiving a request for an e-semaphore with a binding key, the Authenticator uses the key to validate the integrity of the `ecall` arguments and only allows it to proceed if the HMAC is verified true.

When receiving a request for an e-semaphore without a key binding, the Authenticator only relies on the `pid` within the `ecall` arguments. Note that the security strength of the process id based authentication is reduced to limited local context. As it is predictable and reusable, thus vulnerable to `pid` spoofing.

Sem Access Manager. The Sem Access Manager along with Authenticator deals with privilege and identity in ESem. It manages access rights between processes and semaphores. The design of this component revolves around a robust and secure access control mechanism, ensuring that access privileges are regulated. Process identification is established through a combination of `pid`, bolstered by optional secret key authentication mechanism. These keys, residing within both the application enclave and the global enclave, form a secure and authenticated channel. This secure channel serves as the linchpin for all subsequent communications, safeguarding the interactions between processes and semaphores.

Internal TCS and Semaphore Checks. This internal checks against the metadata to ensures that each semaphore access attempt aligns with the authorized TCS. Should there be an attempt to access a semaphore using an unauthorized or mismatched TCS, ESem identifies this discrepancy and leads to an invalidation of the operation.

4.2 Thread Management

To accommodate semaphores within enclave, we propose two design alternatives leveraging SGX ability. Both proposed designs fundamentally employ distinct mapping binding mechanisms of enclave threads. Prior to delving into the particulars of these designs, we introduce an intermediary mapping solution that underpins them in Figure 5. Within the context of this work, two mapping levels are depicted, both of which are managed and enforced inside the enclave.

Since applications are required to first access a specific enclave thread and subsequently delegate the semaphore operation within said enclave thread, a twofold mapping becomes viable and efficient. The first mapping is from the applications to the enclave thread, ensuring secure and authenticated access. The second mapping is from the enclave thread to the semaphores, safeguarding the interactions with semaphore operations. From the two mappings, next we present two binding schemes.

4.2.1 Binding Enclave Thread with Semaphore. This mode exclusively binds one enclave thread with a single semaphore. In essence,

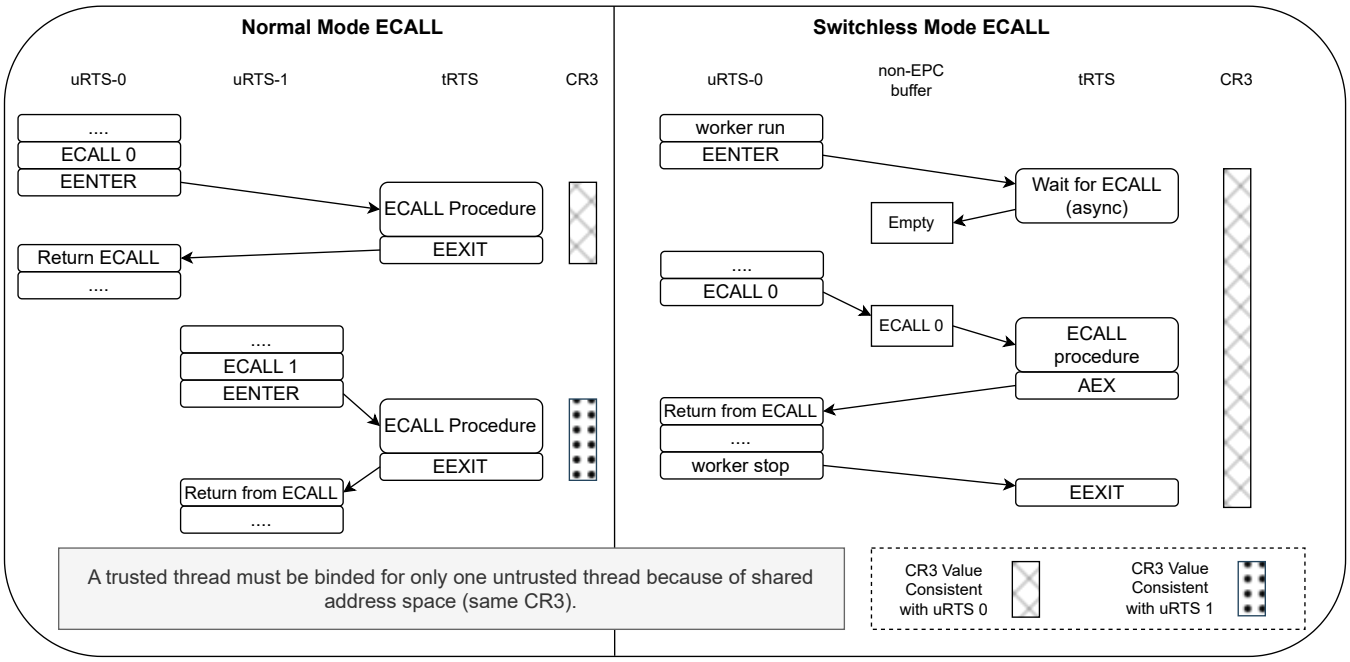

Figure 6: Enclave for Multiple User App

when an application initiates a new semaphore, the s-enclave selects an available enclave thread and establishes a mapping between the two entities. Thereafter, any processes that seek to access this particular semaphore are compelled to use the designated enclave thread, establishing a singular, secure pathway for interaction.

The design insights of this paradigm is to utilize the enclave thread as a lock mechanism, facilitating exclusive access. Leveraging the properties of the multi-thread enclave, the semaphore can only be accessed by a single application at any given moment, therefore naturally enforces mutual exclusive access control.

4.2.2 Binding Enclave Thread with Process. Another design alternative uses a model where the user process is directly bounden with a enclave thread, ensuring that one enclave thread serves exclusively for a singular user process. In order to explicate this design, it is imperative to understand how an enclave runtime accesses untrusted data.

Given that SGX is architected to operate aside the untrusted program, possessing the capability to access data outside its Enclave Page Cache (EPC), the enclave runtime utilizes the page table of the untrusted program. Pertaining to the EENTER instruction, wherein transition from untrusted to trusted space occurs, no privilege alteration happens, implying that the Control Register 3 (CR3) for the enclave remains unaltered from where the EENTER was executed. This property ensures that the enclave invariably accesses its 'corresponding' untrusted address space. Thus, in this design, the intimate and exclusive binding between a user process and a enclave thread ensures that the specific untrusted address space is available for access.

In our design, where the s-enclave roams across various address spaces, binding the process with an enclave thread essentially translates to binding the Control Register 3 (CR3) with a Thread Control Structure (TCS) as show in Figure 6. This design selection is particularly optimal for facilitating switchless mode. Using this mode effectively mitigates the substantial overhead associated with EENTER and EEXIT instructions, which are markedly more resource-intensive than a standard system call [48].

4.3 Semaphore Access Pattern

ESem introduces a mixed approach to semaphore access by employing two distinct access patterns, each corresponding to a types of thread binding model. These patterns reflect the underlying design principles and operational requirements associated with each thread type. In this section, we'll provide detail on associated with these semaphore access patterns.

Lockless Access. The fundamental characteristic of the Thread-Semaphore binding is its reliance on lockless access. When utilizing this pattern, there are direct read or write operations on the semaphore's value, without the imposition of any traditional locking mechanism. The inherent benefits of this approach are:

- Speed: By bypassing locking protocols, the system can achieve faster read/write operations, reducing latency.
- Simplicity: The lockless access mechanism is relatively straightforward, offering a clean and uncomplicated approach to semaphore access.

Atomic Exchange with Spin Lock. For the Thread-Process binding mode, the process is more involved. Given the async nature of different enclave threads, it's exclusive to a process but might collide with other enclave threads. the semaphore values operation necessitates an atomic exchange operator, coupled with a spin lock. The key aspects of this approach are:

- Atomicity: The atomic exchange ensures that the write operation to the semaphore value is completed entirely without interruption, maintaining the integrity of the value.
- Spin Locking: If another enclave thread attempts access while a current thread is engaged, the accessing thread will enter a spin-wait state. This spin lock mechanism ensures that threads in contention are effectively managed without causing a deadlock. The spinning thread will continually check for the lock's availability and will proceed once the lock is released.

ESem's bifurcated semaphore access pattern design, corresponding to the enclave thread binding mechanisms, reflects a deliberate and strategic approach to managing different operational scenarios. By offering lockless access for straightforward use cases and introducing atomic exchanges with spin locking for more complex interactions, ESem delivers a balanced combination of speed, reliability, and security in semaphore operations.

5 IMPLEMENTATION

The ESem prototype is implemented on an Intel platform for the purpose of functionality and performance evaluation and testing under different conditions. The platform is equipped with an Intel `i7-9700k` processor and `16` Gigabytes RAM and installed with Linux Ubuntu `22.04`, revised Intel SGX SDK `2.18` [22] and GLibC `2.36` [18]. Our prototype consists of `2820` lines of C code whose breakdown is shown in Table 2.

Table 2: ESem Component Codebase Sizes

Module	Line of Code
Enclave	1350
Lib-C	452
Kernel Module	1020

The s-enclave image is about `290` KB, occupying `71` pages. The base address for the s-enclave is set as `0x00007F4A10000000`. As a default, the TCSNum, TCSMaxNum and TCSMinPool are equally set to `200`. Apart from one TCS reserved for semaphore management, `100` TCSes are allocated for semaphore-binding and the rest are for process-binding.

During runtime, we measure the total page count ranges from `880` to `912` pages, as the available thread is set from `10` to `50`. The overhead from ESem's EPC usage does not significantly strain the entire system. In a production environment, especially with SGX EDMM support [44], the memory constraints are not particularly restrictive.

In the next subsections, we present two implementation specifics on enclave handling and POSIX compatibility.

5.1 Enclave Handling

In ESem, the handling of the enclave is a critical aspect. The life cycle of the enclave begins with its creation during the kernel's bootstrapping of ESem. At this initial stage, the dummy process is established to serve as the bootstrapper for the s-enclave. Once the initialization is successful, the virtual address of the EPC is recorded.

During the initialization phase, the dummy process also takes on the responsibility of documenting all available TCS. The allocation and registration of these TCS within the enclave mark them as available. Significantly, the first TCS that entered is designated as the manager thread, tasked with the semaphore management. With the completion of the total initialization process, the kernel then transitions to a readiness state.

For the first instance of s-enclave utilization, the ESem enclave is conveyed as a shared memory object. This is materialized through a function that yields a file descriptor, which can be subsequently mapped and utilized by the processes. Diverging from typical shared memory behavior, if the object is not found, the kernel opts not to create a new file but instead returns an error. This approach is a deliberate departure from the norm as we do not support creating s-enclave on-the-fly.

The subsequent step maps the file descriptor into the process's address space. In this operation, the mapping is not left to random system selection; it strictly adheres to the Virtual Address (VA) that the kernel recorded, based on the decisions made during the dummy process phase. The use of flags such as `MAP_ANONYMOUS` in this context would result in an error, aligning with the stringent mapping protocol.

To further fortify the system against potential issues, ESem's also eschews the use of lazy loading in roaming. This decision is a safeguard, ensuring stability and reliability in the enclave's operation within the ESem framework.

5.2 POSIX Compatibility

The second implementation complexity roots from adherence to the POSIX standard, a crucial aspect for ensuring seamless integration and minimal disruption to users accustomed to POSIX semaphores. ESem achieves this by incorporating an additional flag `USE_ESEM` within semaphore calls. This flag is instrumental in distinguishing whether the call is intended for a POSIX semaphore or an ESem semaphore. This approach ensures that users do not perceive any significant differences between traditional POSIX semaphore operations and those performed by ESem.

The complexity introduced by ESem is predominantly encapsulated within the GNU C Library (GLibC) [18]. This added complexity is categorized into two parts. Firstly, it surfaces during the execution of semaphore management operations. Secondly, it becomes evident in the handling of routine semaphore operations.

For the initialization of semaphore-using applications, LibC plays a pivotal role by triggering the enclave roaming process, which is executed only once. During this phase, the kernel also returns the address of the management TCS to user space. Subsequently, LibC takes over to complete the rest of the `ECALL` process. Then, the TCS allocated to access the semaphore is also recorded by LibC. This approach differs from alternative designs that simply substitute the semaphore shared memory region with an enclave-hosted file system [14]. By leveraging LibC, ESem effectively obscures the complexities associated with enclave-related checks from the end user.

In standard semaphore operations, LibC continues to play a critical role. It records the binding of each semaphore, ensuring accurate forwarding to the appropriate ECALL. This process is a building block for maintaining operational integrity, especially in scenarios where an attacker in user space might attempt to bypass LibC to initiate their own semaphore ECALL using an incorrect TCS. Such potential threats are mitigated within the enclave, demonstrating ESem's robust defense mechanisms against unauthorized access.

6 CASE STUDY

In this section, we provide a case study to demonstrate the effectiveness of ESem and show how an app uses it. This case study examines the robustness of File Vault, an application that leverages SGX to secure a file system. The application's primary function is to seal and unseal files with its enclave, providing these services over a secure socket. It supports up to five threads to concurrently manage file transactions.

6.1 File Vault Background

File Vault is an open source project [38] and is enclosed in the Open Enclave samples [1]. We have modified and deployed the prototype in the same hardware environment as Section 5. The application is designed for the secure encryption and decryption of data through network socket interface, utilizing SGX for enhanced security. The architecture of File Vault is shown in Figure 7.

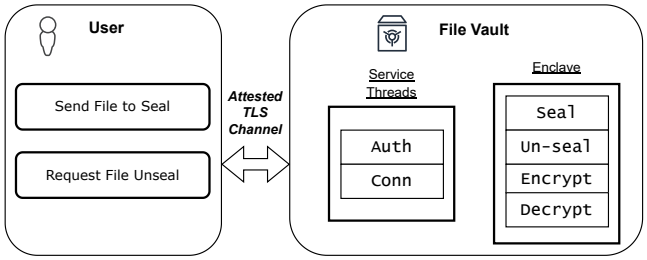

Figure 7: The architecture of File Vault.

File Vault is chosen for the study for two reasons. Firstly, its role in safeguarding confidential file data with SGX makes it an ideal case to exemplify the demand for the strongest-possible security protection. Secondly its unprotected synchronization among concurrent operations epitomizes the motivation of ESem. Moreover, File Vault is simple and easily to explain as compared with servers like Apache.

6.1.1 Service Threads. Prior to any encryption or decryption operations, the File Vault application requires user authentication in service threads. This process ensures that only authorized users can access the application's functions. Upon initiation, the application establishes a TLS connection to the user. The application, upon receiving a response, grants or denies user access based on the verification of the user's credentials. Then one available service thread begin to process a user request. A total of 5 service threads are deployed in our prototype. A semaphore `sem_service` is used at the entrance of each service thread to enter enclave, whose initial value is 5 and is decreased by 1 upon one thread activation.

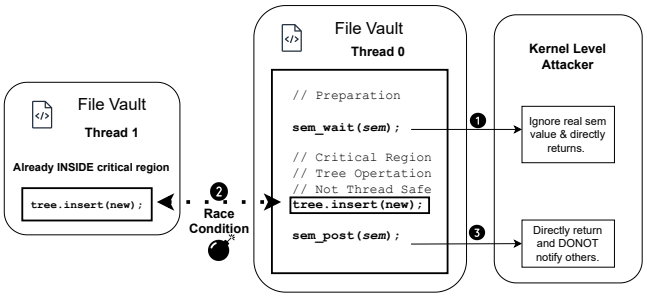

Figure 8: Attack simulation on File Vault semaphore.

6.1.2 File Vault Enclave. For encryption, the application takes user input plaintext file and a sealed encryption key. The file, along with the sealed key, is then passed to the SGX enclave, which performs encryption and returns the ciphertext for storage in an untrusted location identified by the file tree. The file tree is protected by `sem_filetree` with the initial value set as 1. Each enclave thread successfully returning from `sem_wait()` can then maintain the sealed file tree. Similarly, the decryption function accepts the sealed key and a ciphertext file. The contents of the file, along with the sealed key, are passed to the SGX enclave. Then the enclave obtain the semaphore before accessing the sealed file tree. Next, the enclave decrypts the data and returns the plaintext.

6.2 Attacks on Synchronization

We have designed and experimented two attacks both targeting the aforementioned semaphore mechanisms but with different consequences. Figure 8 shows the compromise process.

6.2.1 File tree race. The process unfolds as a race condition due to the compromise of the semaphore mechanism that coordinates access to the file tree. The attack specifically targets the process by which two threads attempt to add nodes to the file tree, leading to an unexpected and erroneous outcome.

Initially, the first thread calls `sem_wait`, the attacker bypasses semaphore check and directly returns. Then it successfully enters the critical section where accesses the file tree. It locates a next-to-add pointer, identifying where in the tree structure its new node should be added. At this point, the thread begins preparing the content for the new tree node, but before it can complete the operation and properly link the node into the tree, the execution context switches to the second thread.

The second thread, due to the compromised semaphore, erroneously gains access to the same critical section, finding the same free pointer in the tree as identified by the first thread. It proceeds to mount its node onto the tree, effectively doing so before the first thread has the chance to complete its operation. This premature action by the second thread disrupts the intended sequence of operations.

As both threads eventually exit their respective operations, they return a status of success. However, the final state of the file tree is compromised: only the first thread's content is written to the tree,

while the node allocated by the second thread becomes inaccessible and effectively lost. This mismanagement in the node addition process leads to a corrupted file tree structure.

6.2.2 Service thread blocking. The second attacker behavior is the same as the first one, but attacking on different critical region. A service thread uses semaphore to keep too many process from entering the enclave at the same time, since the enclave threads are pre-determined and can not scale up in the runtime. A kernel level attacker ignores the real semaphore value and allows the 'wait' thread to proceed without checking the semaphore status. Then more than expected service thread enters enclave, the outflows fails to enter since no TCS available.

6.3 Attack Outcome

The outcome of these attacks are severe. The first attack causes corruption of the file tree structure. The ramifications of this attack manifest later when an attempt is made to unseal a file. The file content is found to be incorrect due to the second file node pointer being overwritten in the earlier race condition. This scenario vividly illustrates how a breach in semaphore synchronization can lead to subtle yet significant data integrity issues in concurrent processing environments, particularly within the secure confines of an SGX enclave where such inconsistencies can have far-reaching consequences.

The second attack causes the semaphore that manages service threads to appear perpetually busy, effectively hogging the system, leading to reduced performance. These types of attacks are particularly nefarious as they specifically target the synchronization mechanism, causing subtle yet detrimental effects that only manifest during concurrent operations, making them difficult to detect and audit.

6.4 ESem Hardened Version

To harden File Vault with ESem, two semaphore function call is changed in code: An extra marco `USE_ESEM` is added to function `sem_open(sem_service)`, `sem_close(sem_service)`, along with `sem_open(sem_filetree)` and `sem_close(sem_filetree)`. The enhanced `sem_service` is normal and its already discussed, while `sem_filetree` uses a different paradigm.

Authenticate enclaves. Because `sem_filetree` is inside an enclave, we harden it with extra mutual-enclave authentication. First a secure channel is build with mTLS. Both enclaves agree on a predefined secret key P_s. For simplicity, we consider the key dispatch are done securely in bootstrap. When party File Vault enclave sends a semaphore request Req to ESem, it computes an HMAC $M_{auth} = HMAC(P_s, Req)$ and appends M_{auth} to the message. Upon receiving request with M_{auth}, ESem independently computes the HMAC using the same secret key and the received message. ESem then compares the computed HMAC with the one received from File Vault. If the two HMACs match, it confirms that the request and perform the operation. To mutual authentication, ESem then send a return message back to File Vault, also with an HMAC appended. Next, File Vault performs the same verification process upon receipt to ensure that the response is authentic and normal operations protocol is executed.

The adoption of ESem within the File Vault application improves its defense against these attacks. Each semaphore access is authenticated and accounted since both party is enclave. This case study underscores the importance of enclave-based semaphore mechanisms for enclave applications dealing with sensitive data. ESem not only enhances the security of the File Vault 's operations but also ensures the consistency and reliability of its service in the face of synchronization attacks.

6.5 Summary

The case study of the File Vault application with ESem demonstrates the enhanced security that SGX enclaves provide for semaphore-based synchronization. By securely encapsulating semaphore operations within the enclave, ESem protects against even sophisticated kernel-level attacks, ensuring that semaphore values cannot be manipulated maliciously. This protection is vital for maintaining service availability, making ESem an ideal hardening component in the synchronization security of applications that handle sensitive data.

7 EVALUATION

For the evaluation of the ESem prototype, the analysis includes *micro*, *macro*, and *real-world application*. In this section, we aim to provide a understanding of ESem's performance in diverse workloads and real world scenarios.

Micro Evaluation. Assessing the performance in single operations, particularly on no-contention scenarios. These operations are fundamental to ESem, as they are essential for semaphore state management within the enclave.

Marco Evaluation. Evaluating the performance of ESem when integrated into benchmark routines, thereby assessing the semaphore module's collective performance. We integrated ESem into widely used LMBench [26] and PTS [37]. This macro-level evaluation aims to assess how ESem behaves when it is arranged as workflows.

Real-World Application Workloads. We substituted the traditional POSIX semaphores with ESem in several applications and conducted performance benchmarks to evaluate its impact.

7.1 Micro Evaluation

In the micro evaluation, we examined the performance breakdown of enclave operations integral to the ESem. This analysis was aimed at understanding the time efficiency of operations. To provide a detailed insight, the performance metrics of these operations are presented in the accompanying Table 3, with most operations being completed within `15` μs.

Each operation within these categories are executed `50` times to ensure reliability of results. The average time taken for each metric reported. These findings are critical in validating the efficacy of ESem in real-world applications where workload can vary significantly. The largest relative difference is the Post and Wait in Row `5` and `6`. The result is expected since the fastest route of legacy POSIX semaphore only contains a compact user-space operation. This extreme performance can not be attained in many real-world cases.

It's important to highlight that in our prototype, we employed a basic linear search algorithm from the standard library to establish

a baseline for performance evaluation. This choice was made to demonstrate the fundamental efficiency of ESem without the enhancements that could be provided by more sophisticated search algorithms. While there is room for further fine-tuning and optimization of these operations – which could potentially reduce the time taken for multi-lookup tasks – we leave that to future work to discuss. This approach ensures that our evaluation focuses on the inherent capabilities of the ESem framework in its fundamental form, providing a clear understanding of its baseline performance breakdown.

Table 3: Semaphore Operation Time for Legacy and ESem. Each test is taken 50 times and the average is presented. For legacy post and wait operation, syscall is included in the timing. For legacy post- and wait-switchless, the fastest no-contention userspace route is taken.

Operation	ESem (μs)	Legacy (μs)	Difference (μs)
Open	17.02	9.87	7.15
Close	12.45	10.73	1.72
Post	7.02	5.3	1.72
Wait	7.39	6.2	1.19
Post-lockless	6.78	0.014	6.77
Wait-lockless	6.23	0.012	6.22
Unlink	10.05	9.03	1.02

7.2 Marco Evaluation

Table 4: PTS-NG Semaphore Benchmark [37] on ESem and Legacy. This test reports the largest semaphore operation available in a stress test, indicating best effort of the available system resource to be filled with a tiny piece of operation. 'Op/s' indicates operations accomplished per second.

Metric	Legacy	ESem	ESem-Switchless
PTS-NG [37] (Op/s)	11,488,876	9,533,037	9,125,001
Difference	–	-1,955,839	-2,363,875
(Op/s, %)	–	-17.02%	-20.58%

In the macro evaluation phase of our study, we employed two kernel benchmark suites, LMBench [26] and PTS [37], to assess the overall performance of workflows utilizing the ESem. LMBench [26] is particularly well-suited for this purpose as it offers a specialized latency test suite for semaphores, allowing us to tailor the testing environment. This adaptability includes the ability to predefine the number of applications and semaphores used within the system, which is crucial for accurately representing a workflow that relies heavily on semaphore synchronization across multiple processes.

One of the key advantages of using LMBench [26] in our evaluation is its capacity to simulate the contention among applications in a controlled manner. Additionally, we introduced a modification to the LMBench [26] tests to further examine ESem's handling of semaphore restarting mechanisms. This involved periodically closing and reopening a semaphore after a certain number of normal

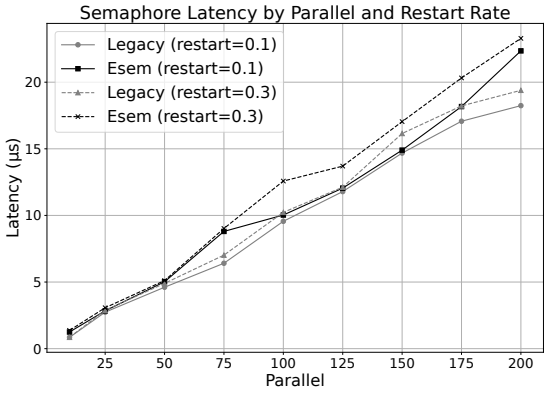

Figure 9: LMBench [26] Semaphore Latency for Legacy and ESem. This test is performed to measure semaphore in two typical restart ratio 0.1 and 0.3.

operations, a process designed to mimic typical application behavior in a dynamic computing environment. This adjustment to the benchmark process evaluates how well ESem manages semaphore lifecycle events, including creation, usage, and termination, which are critical aspects of semaphore-based synchronization in practical applications. Restart rate is shown in the table indicates the ratio between close-then-reopen operation and normal PV-operation.

The evaluation results, as detailed in the accompanying Figure 9 and Table 4, demonstrate that the ESem framework exhibits a marginally slower performance compared to legacy semaphore systems in most tested configurations. This performance discrepancy is particularly noticeable in scenarios categorized under the middle parallel group, where ESem's operations take slightly longer than traditional semaphore mechanisms. However, an interesting trend emerges as we move towards configurations with higher levels of parallelism. In these scenarios, the relative time difference between ESem and legacy semaphores slightly diminishes. This phenomenon can be attributed to the fact that in this settings, a considerable portion of the processing time is consumed in jumping across different processes, rather than in the semaphore operations themselves. Consequently, in such contexts, the additional overhead introduced by ESem becomes less impactful on the overall workflow. This observation suggests that while ESem introduces delay in semaphore operations, it does not result in a substantial slowdown in the entire workflow, particularly in environments where high parallelism is a dominant factor.

In our evaluation of the ESem framework, we also explored a variant "switchless mode ESem," which yielded some intriguing results, as depicted in Figure 10. This mode is specifically designed to optimize performance by minimizing the overhead associated with switching to enclave threads.

In scenarios categorized under 'High frequency' - a special mode within switchless ESem - the framework demonstrates reduction in performance overhead. In such environments, the additional burden introduced by extra enclave threads is minimized, allowing the switchless mode to operate more efficiently.

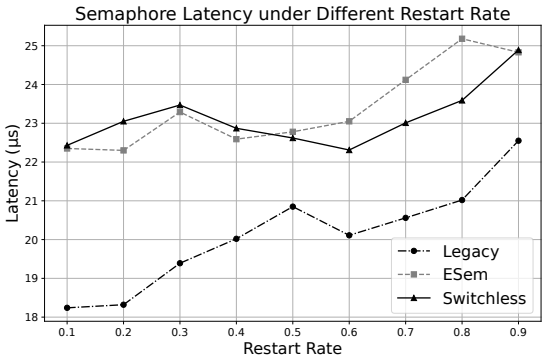

Figure 10: LMBench [26] Semaphore Latency under High Frequency Mode.

The data presented in Figure 10 illustrates that the switchless mode ESem not only mitigates the performance impact typically associated with standard ESem operations but can actually surpass it in terms of efficiency. This enhanced performance in low parallelism scenarios highlights the potential of switchless ESem in specific operational contexts.

Switchless mode ESem's ability to outperform the normal ESem in certain conditions underscores the importance of tailoring the enclave mechanism to the specific needs of the application environment. It suggests that for applications characterized by high-frequency semaphore operations, switchless mode ESem could offer a more optimal solution, balancing the needs for security and performance effectively.

7.3 Real World Application Workload

In the real-world application workload evaluation of the ESem framework, we focused on benchmarking its performance in widely-used applications: Redis [32], Apache HTTP Server (httpd) [8], and PostgreSQL [29]. These applications were chosen due to their prevalence in industry and their reliance on semaphore mechanisms for process synchronization and resource management. Redis [32], an in-memory data structure store, often used as a No-SQL database and cache, is known for its high-speed operations. Apache HTTP Server [8], a widely-used web server software, requires efficient synchronization for handling multiple concurrent connections. PostgreSQL [29], a sophisticated object-relational database system, demands robust semaphore mechanisms for transaction and connection handling.

For each of these applications, ESem replaced the existing POSIX semaphores to evaluate its impact on overall performance. The benchmarks and reported metrics are provided by the project. Focused on key performance indicators relevant to each application, such as transaction throughput for PostgreSQL [29], request handling capacity for Apache HTTP Server [8], and read-write benchmark in Redis [32]. The results provided insight into how ESem behaves under the load of real-world application scenarios, especially in environments where efficient synchronization matters. All applications are designed to use different semaphores and provides an extra layer of abstraction. We made use of this characteristic and ported ESem before experiment.

Table 5: Real World Application Performance. The tested benchmarks are all native to the application. PostgreSQL [29] benchmark tests the largest transaction intake per second. Redis [32] benchmarks the highest request per second. In httpd [8], it tests time in microseconds per request. For the first two benchmarks, the higher the better. For the third one, the lower.

Application	Legacy	ESem	Difference
PostgreSQL [29] (tps)	1,088.49	1,112.79	+24.30 (+2.23%)
Redis [32] (rps)	62,847.97	61,097.82	-1,750.15 (-2.79%)
Apache [8] (tpr)	9,208.09	9,427.83	+219.74 (+2.39%)

The results from our real-world application benchmarks, as detailed in Table 5, indicate that ESem framework does not cause a significant slowdown when compared to traditional legacy semaphore mechanisms. All relative differences are below *3%* compared to legacy.

Despite the added security features and the complexities associated with managing semaphore operations within the secure confines of an SGX enclave, ESem maintains a performance level that is close to legacy semaphore systems.

The minimal performance impact observed with ESem can be explained for two major reasons. The first is the critical region size. These applications well follows the design guideline to use system semaphores as a protection for system allocated resource. The critical region code takes considerable amount of time compared to previous tests. This effectively reduced the performance slowdown brought in by a slower sync mechanism. The second reason is that sync primitive usage is not heavily. These applications are well designed and concurrency control is cautiously used. This lifts the performance bottle neck from the sync mechanism.

In all, ESem balances the dual demands of security and performance. The ability to provide this enhanced security without compromising on overall performance makes it an attractive option for systems where semaphore operations are critical like hosting a service on cloud platforms.

8 RELATED WORK

8.1 SGX Secured Systems

SGX plays a pivotal role in fortifying the security of various system components. SGX-Shield introduces an innovative ASLR scheme for SGX, enhancing security through finely-grained randomization [34]. Glamdring presents a groundbreaking source-level partitioning framework with SGX, providing rich OS abstractions to enclaved code [25]. PANOPLY prioritizes TCB reduction over performance, employing a delegate-rather-than-emulate design philosophy [36]. Graphene-SGX, a comprehensive library OS, facilitates the deployment of unmodified applications in SGX enclaves [41]. SGX-LKL securely runs Linux binaries within SGX enclaves, offering a minimal, protected, and oblivious host interface [30]. SGXIO introduces a versatile trusted path for secure execution in untrusted

OS environments [43]. SGXLog guarantees the integrity and confidentiality of log data [24]. Custos provides a practical framework for tamper detection in system logs, featuring a tamper-evident logging layer and decentralized auditing protocol [27]. Obliviate presents a data-oblivious file system [3]. SGXKernel overcomes enclave transition limitations through a switchless design and innovative cross-enclave communication [40]. In distributed systems, SGX protects through system components or algorithms. Haven ports the Windows Library OS to SGX, enabling secure execution of applications in an untrusted kernel environment [12]. SCONE fortifies container processes, providing a secure C standard library interface with seamless I/O data encryption [11]. Opaque, optimized for SGX-protected Spark layers, enhances data security in distributed analytics [49]. VC3 emphasizes data privacy in cloud-based analytics, while SEED pioneers workflow scheduling algorithms for public cloud environments using Intel SGX [4, 33]. EnclaveDB utilizes SGX to establish a secure database engine with a trusted kernel [31].

8.2 Process Synchronization

Patel et al. have shown that synchronization attacks are a threat to kernel synchronization in Linux containers [28]. These attacks, manipulable by unauthorized users, extend to framing attacks, persistently impacting performance by expanding data structures. Akkan et al. conducted a study on the x86-specific instructions MONITOR and MWAIT, aiming to develop a more efficient locking and synchronization mechanism adaptable to high-concurrency scenarios [5]. SyncProf addresses performance issues in concurrent programs by employing a graph-based relationship representation and multiple program executions [45]. It effectively identifies and optimizes synchronization bottlenecks, offering optimization strategies for developers [45]. SyncPerf, designed for multi-threaded programs, tackles performance problems associated with synchronization primitives like locks and semaphores [6]. The Malthusian lock provides a method to resolve performance degradation resulting from over-threading in a multi-threaded environment [17].

9 DISCUSSION

9.1 Authorized Synchronization

By introducing authorization mechanisms, ESem will inherently exposes itself less to vulnerabilities. A simple authorization layer,like `pid`, could inadvertently grant undue access and privileges, potentially leading to confused deputy attacks.

Several extensions can be easily proposed to bolster the security posture of ESem. One enhancement is the establishment of a handshake secure communication channel to the application side. By doing so, not only can we restrict and monitor access based on verified credentials, but we also reduce the risk of man-in-the-middle attacks, eavesdropping, and data tampering.

Furthermore, the idea of integrating a remote trusted party introduces an added layer of verification and trust. With a remote entity overseeing or validating operations, we can provide an external checkpoint against anomalous or malicious behavior. Similar a system can further be strengthened by considering scenarios with a group of enclaves [15].

9.2 Shared Resource Protection

Semaphore usually protects some critical shared resources, ensuring that these resources are accessed in a controlled manner. One promising direction of ESem is to port memory-based resources directly inside the enclave. Doing so serves a dual purpose. Firstly, it provides an additional layer of security by encapsulating these resources within the enclave's protective boundaries, shielding them from external attacks. The encapsulation can deter direct tampering, unauthorized access, and other malicious intents that target these shared resources. Secondly, by housing these resources within the enclave, the synchronization mechanisms themselves can operate with increased efficiency and precision, given the reduced overhead of inter-component communication.

However, transferring more components and resources into the enclave invariably enlarges the Trusted Computing Base (TCB). A larger TCB implies a broader surface area for potential vulnerabilities, as there are more components that need to be verified, monitored, and maintained. The more expansive the TCB, the higher the complexity and, correspondingly, the risk. This added complexity might inadvertently introduce new vulnerabilities or make it challenging to ensure the integrity and security of all components comprehensively.

9.3 Contrast with other TEE-based solutions

Two current TEE paradigms apart from ESem can also address similar syncronization problem. The first type is trusted Library OS, like [9, 41]. ESem exactly suits the need to port a credible sync mechanism for LibOS. Slightly shift the ECALL entrance in libC can fit into this paradigm. The second type is Confidential VMs (CVM), like TDX[21] and SEV[7]. But there is still significant difference from ESem. Since CVMs inherently preserves all guest kernel, the synchronization scheme is automatically protected. The expensive ESem operations can be reduced to normal cost. But the kernel inside CVM is still vulnerable to attacks from other surfaces. In all, the performance under large number of writers/reader should be better than ESem, but only partial protection can be covered.

10 CONCLUSION

We study the security of process synchronization against kernel-privileged attacks and propose ESem, an enclave-based semaphore service as the countermeasure. ESem combines enclave roaming and in-enclave access control to provide a balanced performance and security guarantee. We have implemented and evaluated ESem along with a demonstrative case study. The results show that ESem fulfills its design goals with a modest overhead for real-world applications.

11 ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers and COMPASS members for their insightful comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218, Shenzhen Science and Technology Program under Grant No.SGDX202011030 95408029, and Peng Cheng Laboratory Grant PCL2022A03-01.

REFERENCES

[1] 2023. Open Enclave SDK. https://openenclave.io/sdk/. Build Trusted Execution Environment based application.

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

[3] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. 2018. OBLIVIATE: A Data Oblivious Filesystem for Intel SGX. In *Proceedings of the Network and Distributed System Security Symposium*.

[4] Ishtiaq Ahmed, Saeid Mofrad, Shiyong Lu, Changxin Bai, Fengwei Zhang, and Dunren Che. 2020. SEED: Confidential big data workflow scheduling with Intel SGX under deadline constraints. In *Proceedings of the IEEE International Conference on Services Computing*. IEEE, 108–115.

[5] Hakan Akkan, Michael Lang, and Latchesar Ionkov. 2013. HPC runtime support for fast and power efficient locking and synchronization. In *Proceedings of the IEEE International Conference on Cluster Computing*. IEEE, 1–7.

[6] Mohammad Mejbah Ul Alam, Tongping Liu, Guangming Zeng, and Abdullah Muzahid. 2017. Syncperf: Categorizing, detecting, and diagnosing synchronization performance bugs. In *Proceedings of the 25th European Conference on Computer Systems*. 298–313.

[7] AMD. 2023. AMD Secure Encrypted Virtualization. https://www.amd.com/en/developer/sev.html.

[8] Apache Software Foundation. 2023. Apache HTTP Server. https://httpd.apache.org/

[9] Apache Software Foundation. 2023. Apache TEACLAVE. https://teaclave.apache.org/

[10] ARM. 2021. Arm CCA Security Model 1.0. https://developer.arm.com/documentation/DEN0096/latest.

[11] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, et al. 2016. SCONE: Secure Linux containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 689–703.

[12] Andrew Baumann, Marcus Peinado, and Galen Hunt. 2015. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems* 33, 3 (2015), 1–26.

[13] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 335–348.

[14] Dorian Burihabwa, Pascal Felber, Hugues Mercier, and Valerio Schiavoni. 2018. SGX-FS: hardening a file system in user-space with Intel SGX. In *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science*. IEEE, 67–72.

[15] Guoxing Chen and Yinqian Zhang. 2022. MAGE: Mutual Attestation for a Group of Enclaves without Trusted Third Parties. In *Proceedings of the USENIX Security Symposium*. 4095–4110.

[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[17] Dave Dice. 2017. Malthusian locks. In *Proceedings of the Twelfth European Conference on Computer Systems*. 314–327.

[18] Free Software Foundation. 2023. GNU C Library. https://www.gnu.org/software/libc/

[19] Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. A Hardware-Software Co-design for Efficient Intra-Enclave Isolation. In *Proceedings of the 31st USENIX Security Symposium*. 3129–3145.

[20] Intel. 2023. 64 and IA-32 Architectures Software Developer's Manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[21] Intel. 2023. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/tools/trust-domain-extensions/documentation.html.

[22] Intel Corporation. 2023. Intel Software Guard Extensions (SGX) for Linux. https://github.com/intel/linux-sgx.

[23] JamesAndersonJr et al. 2020. NEWS: Intel plans to drop SGX support from its 11th Gen Desktop Processors in favor of TME/MKTME. Proceedings of the CyberLink Community Forum. https://forum.cyberlink.com/forum/posts/list/83604.page

[24] Vishal Karande, Erick Bauman, Zhiqiang Lin, and Latifur Khan. 2017. SGX-Log: Securing system logs with SGX. In *Proceedings of the ACM on Asia Conference on Computer and Communications Security*. 19–30.

[25] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. 2017. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference*. 285–298.

[26] LMBench. 2023. LMBench. https://lmbench.sourceforge.net/. A System Performance Measurement Tool.

[27] Riccardo Paccagnella, Pubali Datta, Wajih Ul Hassan, Adam Bates, Christopher Fletcher, Andrew Miller, and Dave Tian. 2020. Custos: Practical tamper-evident auditing of operating systems using trusted execution. In *Proceedings of the Network and Distributed System Security Symposium*.

[28] Yuvraj Patel, Chenhao Ye, Akshat Sinha, Abigail Matthews, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, and Michael M Swift. 2022. Using Tratr to tame Adversarial Synchronization. In *Proceedings of the USENIX Security Symposium*. 3897–3916.

[29] PostgreSQL. 2023. PostgreSQL. https://www.postgresql.org/. An open source database.

[30] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. 2019. SGX-LKL: Securing the host OS interface for trusted execution. *arXiv preprint arXiv:1908.11143* (2019).

[31] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. EnclaveDB: A secure database using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy*. IEEE, 264–278.

[32] Redis. 2023. Redis. https://redis.io/. An in-memory database that persists on disk.

[33] Felix Schuster, Manuel Costa, Cedric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceeding of the IEEE Symposium on Security and Privacy*. IEEE, 38–54.

[34] Jaebaek Seo, Byoungyoung Lee, Seong Min Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. 2017. SGX-Shield: Enabling Address Space Layout Randomization for SGX Programs.. In *Proceedings of the Network and Distributed System Security Symposium*.

[35] Youren Shen, Hongliang Tian, Yu Chen, Kang Chen, Runji Wang, Yi Xu, Yubin Xia, and Shoumeng Yan. 2020. Occlum: Secure and efficient multitasking inside a single enclave of intel sgx. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 955–970.

[36] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. 2017. Panoply: Low-TCB Linux Applications With SGX Enclaves.. In *Proceedings of the Network and Distributed System Security Symposium*.

[37] Phoronix Test Suite. 2023. Phoronix Test Suite. https://github.com/phoronix-test-suite/phoronix-test-suite. A benchmark software.

[38] Siddharth Syal. 2023. File Encryption Using Intel SGX. https://github.com/siddharthsyal/File-Encryption-Using-Intel-SGX.git.

[39] Hongliang Tian, Qiong Zhang, Shoumeng Yan, Alex Rudnitsky, Liron Shacham, Ron Yariv, and Noam Milshten. 2018. Switchless calls made practical in Intel SGX. In *Proceedings of the 3rd Workshop on System Software for Trusted Execution*. 22–27.

[40] Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: A library operating system optimized for Intel SGX. In *Proceedings of the Computing Frontiers Conference*. 35–44.

[41] Chia-Che Tsai, Donald E Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *Proceedings of the USENIX Annual Technical Conference*. 645–658.

[42] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. 2020. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *Proceedings of the 41st IEEE Symposium on Security and Privacy*. IEEE, 592–607.

[43] Samuel Weiser and Mario Werner. 2017. Sgxio: Generic trusted i/o path for Intel SGX. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*. 261–268.

[44] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel software guard extensions (Intel SGX) software support for dynamic memory allocation inside an enclave. In *Proceedings of the International Workshop on Hardware and Architectural Support for Security and Privacy*. 1–9.

[45] Tingting Yu and Michael Pradel. 2016. Syncprof: Detecting, localizing, and optimizing synchronization bottlenecks. In *Proceedings of the International Symposium on Software Testing and Analysis*. 389–400.

[46] Peterson Yuhala, Michael Paper, Timothée Zerbib, Pascal Felber, Valerio Schiavoni, and Alain Tchana. 2023. SGX Switchless Calls Made Configless. *arXiv preprint arXiv:2305.00763* (2023).

[47] Yiming Zhang, Yuxin Hu, Zhenyu Ning, Fengwei Zhang, Xiapu Luo, Haoyang Huang, Shoumeng Yan, and Zhengyu He. 2023. SHELTER: Extending Arm CCA with Isolation in User Space. In *Proceedings of the USENIX Security Symposium*.

[48] ChongChong Zhao, Daniyaer Saifuding, Hongliang Tian, Yong Zhang, and ChunXiao Xing. 2016. On the performance of Intel SGX. In *Proceedings of the Web Information Systems and Applications Conference*. IEEE, 184–187.

[49] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation*. 283–298.