A Comparative Analysis of Rust-Based SGX Frameworks: Implications for building SGX applications*

Heekyung Shin[1], Jiwon Ock[1], Hyeon No[1], and Seongmin Kim[1]**

Sungshin Women's University, Seoul, Korea[1]
{220224009, 220224011, 220224007, sm.kim}@sungshin.ac.kr

Abstract. The widespread adoption of Intel Software Guard Extensions (SGX) technology has garnered significant attention, primarily owing to its robust hardware-based data-in-use protection. To alleviate the complexities of SGX application development, an approach involving the incorporation of a Library Operating System (LibOS) within an enclave has gained prominence. This strategy enables SGX utilization without necessitating extensive modifications to legacy code. However, this approach increases the potential attack surface and may be susceptible to memory corruption vulnerabilities. To address this challenge, the trend of leveraging Rust programming language offering memory safety guarantees for implementing system components has prompted the development of Rust-based SGX frameworks. But still, a gap exists in providing guidelines or systematic analyses to aid developers in selecting a suitable Rust-based SGX framework, considering factors like implementation cost and runtime overhead. This study undertakes a comprehensive comparative analysis of three representative SGX frameworks implemented with Rust: Rust SGX SDK, Occlum, and Fortanix EDP. Our analysis encompasses an exploration of their internal implementations, focusing on their impact on both performance and security. Additionally, we quantify the engineering effort required for migrating legacy Rust applications and evaluate the supplementary overhead incurred when subjecting these frameworks to CPU and memory-intensive workloads. By conducting this analysis, we aim to provide valuable guidance to developers seeking to choose a Rust-based SGX framework that aligns with their application's specific purpose and workload characteristics.

Keywords: Trusted Execution Environment · Intel SGX · Rust

1 Introduction

The commercialization of Intel Software Guard Extensions (SGX) technology [12] has garnered substantial industrial and academic attention. In particular, In-

* This work was supported by the Sungshin Women's University Research Grant of H20210012.

** Corresponding author: Department of Future Convergence Technology Engineering, Sungshin Women's University, Seoul, Korea, Tel: +82-2-920-7449

tel SGX technology plays a pivotal role in evolving the confidential computing paradigm [28]. This interest is primarily driven by its robust hardware-based data-in-use protection and its inherent practicality, notably its compatibility with the x86 architecture ensuring native speed [6]. By leveraging SGX to legacy applications, it is possible to guarantee the confidentiality and integrity of cloud-based TEE service. In fact, leading cloud service providers (CSPs) have begun offering public cloud instances supporting SGX functionalities. These ground-breaking solutions, known as confidential VMs, include commercial products like Amazon Nitro Enclaves [3] and Azure Confidential Computing [26]. Such innovation has expedited the widespread adoption of confidential computing across diverse domains, such as safeguarding AI/ML models [13, 21], protecting digital assets [22], and securing key management services [10, 35].

Basically, there are two primary approaches for implementing the SGX program: 1) porting an application based on SGX SDK [1] and 2) running unmodified applications on top of frameworks that support SGX compatibility [6]. In particular, the adoption of a Library Operating System (LibOS) within the enclave has emerged as a viable strategy to facilitate the utilization of SGX without necessitating modifications to legacy code [5,6,27,33]. The LibOS-based strategy offers distinct advantages when porting legacy applications into the SGX environment. Developers are relieved from the complexities of segregating security-sensitive components from the original code-base and re-implementing system call wrappers for enclave transitions. However, it is important to note that this design choice expands the potential attack surface, given that the entire LibOS codebase is loaded and executed within an SGX enclave. SGX does not guarantee the memory safety of the enclave, which means that memory corruption vulnerabilities inherent in traditional code written in languages like C or C++ (e.g., Heartbleed [7]) can still be effective even when executed within the security boundary provided by SGX CPU [20, 29]. Therefore, an additional instrumentation or protection mechanism is required to achieve robustness over memory vulnerabilities.

Simultaneously, the rise of the Rust programming language has equipped developers with a potent instrument for constructing robust and secure applications. Rust delegates memory safety checking (e.g., rust pointer always references valid memory) to the Rust compiler. In contrast to low-level codes implemented in C or C++ that are prone to subtle memory bugs, Rust guarantees memory safety by rejecting the compilation of them by introducing features, such as ownership and lifetime elision rules [24]. Furthermore, Rust is fast and memory-efficient as its runtime does not require a garbage collector to reclaim memory space, making it well-suited for the development of performance-critical services. This appeal leads to the adoption of Rust in state-of-the-art system software, including container runtimes [2], microkernels [19], and storage systems [17].

Such a trend has also spurred the development of the SGX framework tailored for Rust utilization. The state-of-the-art LibOS-based SGX frameworks have extended support for the execution of Rust applications [27, 33]. Besides, several studies [8, 31, 34] utilize Rust programming language [24] as the foun-

dation for building SGX frameworks. Such design choice enables developers to reduce runtime overhead (e.g., garbage collection), thereby drawing attention to the potential of leveraging Rust in SGX framework development. Nevertheless, a notable gap persists in the absence of comprehensive guidelines or systemic analyses that can aid developers in selecting the most suitable Rust-based SGX framework for their applications. Such guidelines would encompass considerations related to implementation cost and runtime overhead, crucial factors when deciding to execute existing applications or develop new Rust applications in the SGX environment.

This study conducts a comparative study on existing Rust-based SGX frameworks to provide implications for newly implementing or porting legacy security-sensitive Rust applications. For this, we conduct an in-depth analysis between three cutting-edge Rust-based SGX frameworks: Rust SGX SDK, Occlum, and Fortanix EDP. First, we explore the internal implementation details of each framework relevant to the application performance and security. Then, we quantify the engineering effort required to deploy legacy Rust applications atop these frameworks, providing insights into the ease of transition. Finally, we evaluate the additional overhead incurred by each framework, subjecting them to CPU-intensive and memory-intensive workloads to gauge their performance implications. We believe our analysis provides guidance for developers to select an appropriate Rust-based SGX framework when implementing an SGX application according to its purpose and workload characteristics.

2 Background

2.1 Intel SGX and LibOS-based SGX Framework

Intel SGX is a secure processor architecture to ensure trustworthiness of application to protect sensitive and valuable information. It offers an isolated protection domain in memory called an *enclave*, which is only decrypted within the CPU package when executing it as an enclave mode. This ensures that even system administrators or other software running on the host cannot access the sensitive data in the enclave. To help developers implement SGX applications, Intel provides the SGX Software Development Kit (SDK). The SDK offers essential libraries and toolchains for tasks such as enclave signing and debugging [25]. It simplifies the process of creating secure enclaves and managing their execution. For building an SGX application using SDK, a developer needs to separate an application codebase into two parts, an enclave region and an untrusted region. In addition, the transition interface between them must be defined by a developer in the Enclave Definition Language (EDL). This interface specifies the secure functions `ECALLs` for entering an enclave mode and functions `OCALLs` that can be invoked to switch execution to the untrusted region. Additionally, EDLs detail how data should be transferred in and out of the enclave, specifying data structures and communication mechanisms. Note that `OCALLs` are typically used for handling system calls, as SGX does not allow executing `syscall` instructions in an enclave mode.

LibOS-based SGX focuses on using a Library OS that provides operating system functionality in the form of a library to act as an interface between applications and hardware. It runs entirely within an enclave, and to port an application into an enclave, the application binary needs to be loaded and executed along with the libraries it relies on. One of the key advantages of LibOS-based SGX is the simplification of the enclave interface. This minimizes the number of system calls that occur within the enclave, ensuring that the code running within the enclave does not require system calls that involve crossing between user and kernel domains. LibOS also plays a crucial role in implementing and managing necessary operating system functionalities within the enclave when executed in user space. This allows enclaves to handle privileged operations that would typically require execution in processor supervisor mode, maintaining security isolation while performing necessary tasks. Operations represented as system calls, particularly those related to file system operations, can be straightforwardly implemented within LibOS by modifying data structures related to the file system implementation. These system calls do not impact the security of other application programs and do not require execution by privileged system software [30]. Frameworks such as Grammine [33], SGX-LKL [27], and Haven [6], which implement LibOS-based SGX, offer the advantage of enhancing portability by freeing applications from dependence on a specific operating system.

2.2 Rust Programming Language

Rust is a newly introduced programming language developed by Mozilla Research that guarantees safety on the memory side with cost-free abstraction [24]. Rust delegates memory safety checking (e.g., rust pointer always references valid memory) to the Rust compiler. In contrast to low-level codes implemented in C or C++ prone to subtle memory bugs, Rust guarantees memory safety by rejecting their compilation by introducing features, such as ownership and lifetime elision rules [24]. Such design choice enables developers to minimize a runtime overhead (e.g., garbage collection), which in turn introduces the attention to utilizing Rust for implementing system software [24]. Rust introduces a unique *ownership* system central to its memory safety guarantees [16]. The ownership system enforces strict rules about how memory is allocated and deallocated, ensuring that memory is managed safely without the risk of common bugs like null pointer dereferences, data races, and memory leaks. Rust also incorporates *lifetime*, which are annotations that specify the scope or duration for which references are valid [16]. It prevents references from outliving the data they point to or being used after the data has been deallocated.

3 Characteristics Analysis of Frameworks

To take advantage of Rust mentioned above (e.g., guaranteeing in-enclave memory safety), recent studies utilize Rust when implementing an SGX framework itself and enable developers to execute Rust applications on SGX environment [8,

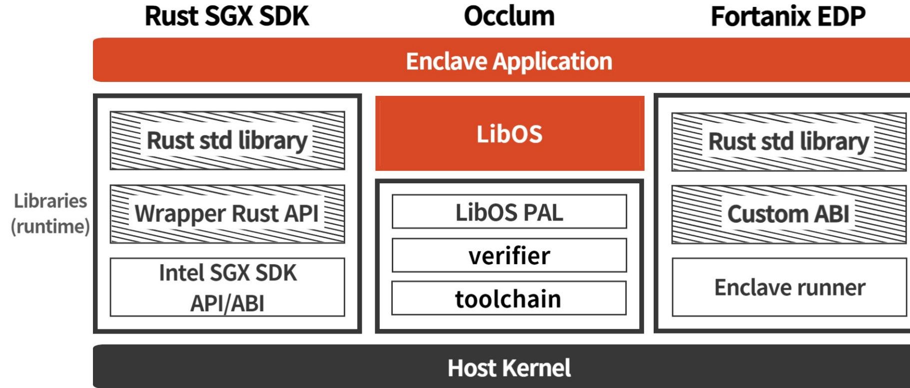


Fig. 1: Rust-based SGX Framework Overview. (The red boxes indicate regions that are isolated and protected by the enclave application, while the black dashed boxes are regions that are written in Rust.)

31, 34]. In particular, we provide an overview of three existing frameworks that facilitate the development of SGX applications in Rust: *Rust SGX SDK, Occlum,* and *Fortanix EDP*. As depicted in Figure 1, these frameworks each exhibit a distinct system architecture. It is worth noting that Occlum exclusively employs a LibOS-based approach, while both Rust SGX SDK and Fortanix EDP offer a custom interface to interact with the host OS for system operations.

3.1 Fortanix EDP

Enclave development platform (EDP) [8], developed by Fortanix, offers a distinct advantage in generating and running enclave from scratch with Rust code, eliminating the dependency on the Intel SGX SDK [8]. Notably, Fortanix EDP introduces its own unique API and ABI while ensuring binary-level compatibility for Rust applications. Specifically, EDP's `usercall` interface is designed not to expose existing enclave interface attack surfaces. It achieves this by incorporating elements that handle memory allocation in user space and data copying from user memory within the context of a Rust-type system. This approach effectively safeguards against direct memory access, preemptively mitigating time-of-check time-of-use (TOCTOU) attacks. It's worth noting that the `usercall` interface establishes a connection to the syscall interface through an enclave. Within the untrusted region, an enclave runner takes on the responsibility of managing enclave loading and serves as an intermediary layer bridging the gap between `usercall` requests originating from the enclave and the syscall interface required for external interactions. While EDP enables the utilization of much of Rust's standard library for application implementation, it intentionally imposes restrictions on specific functionalities, such as multi-processing support and file system operations, for security reasons.

3.2 Occlum

Occlum is a memory-safe multi-process LibOS for Intel SGX to enable execution of legacy applications without modifying the source code [31]. Occlum proposes multi-domain software fault isolation (MMDSFI) by leveraging Intel Memory Protection Extensions (MPX) technology [14] to preserve isolation between processes that share a single address space. To support this, the Occlum framework has newly implemented SGX LibOS, the Occlum toolchain, and the Occlum verifier. Untrusted C/C++ code can generate executable binaries through the Occlum toolchain and be verified by the Occlum verifier, ensuring the integrity of MMDSFI. Consequently, the verified MMDSFI enables the secure construction of the LibOS within the enclave.

LibOS based on Intel SGX SDK and Rust SGX SDK is predominantly implemented in Rust, accounting for approximately 90% of the codebase, with the remainder implemented in C. This supports the execution of enclaves in both C and Rust, providing protection for enclave programs against potential memory vulnerabilities. Furthermore, to protect LibOS from unsafe entities, a shim layer called `occlum-PAL` is provided to the application, offering APIs. This isolation mechanism is crucial for security as it prevents one process from interfering with or accessing the memory of another with strict boundary checking. By securely sharing the enclave's single address space with Occlum's SFI-isolated processes (SIPs) which is a unit of application domain, it supports multi-tasking efficiently. For example, compared to other SGX frameworks that utilize LibOS with supporting multi-tasking [5, 6, 33], startup time is 1000 times faster and IPC (inter-process communication) is up to 3 times faster [31].

3.3 Rust SGX SDK (Teaclave SGX SDK)

The Rust SGX SDK, developed by Baidu, offers a secure platform for executing Rust-based applications within SGX environments [34]. This SDK introduces a wrapper Rust API that layers Rust functionalities on top of the SGX SDKs, originally implemented in C and C++. Through this layered approach, it establishes a secure connection between the Intel SGX SDK code and the trusted application. Notably, as a dependency on the Intel SGX SDK, it places trust exclusively in the software operating within an enclave while maintaining untrusted towards the rest of the system. The SDK doesn't provide its own Application Binary Interface (ABI) but instead adheres to the same ABI as the vanilla Intel SGX SDK. This strategic choice ensures seamless compatibility between the Rust SGX SDK and the Intel SGX environment. Consequently, any updates or alterations within the SGX ecosystem can be swiftly accommodated without the risk of breaking compatibility.

4 Qualitative aspects affecting application performance

In this section, we conduct in-depth analysis by systemically exploring the internal design of each framework and categorize three key indicators related to

application performance: Memory boundary check, Enclave transition, and additional runtime overhead. Table 1 summarizes our analysis result.

	Memory boundary check	Enclave Transition	Runtime Overhead	Memory Safety
Occlum	MMDSFI	PAL API	Enclave SIP	Enclave SIP
Incubator Teaclave SGX SDK	Runtime (Enclave-runner)	Legacy ECALL/OCALL	Rust Wrapper API	Rust Wrapper API
Fortanix EDP	Sanitizable function	Usercall (Custom)	Own ABI	Own API and ABI

Table 1: Estimating framework performance impact overhead based on framework analysis

4.1 Memory boundary check

To avoid overhead caused by unnecessary bound checking, Rust SGX SDK provides a `Sanitizable` function to check the raw byte array and verify that memory represents a valid object when binding an application. For the case of Fortanix EDP, the `enclave-runner` runtime checks before entering an enclave to ensure processor state sanitation, similar to Rust SGX SDK. Finally, Occlum utilizes SFI (Software Fault Isolation), a software instrumentation technique that sandboxes untrusted domains within a single address space to reduce the enclave size in a multi-tasking environment. However, Occlum performs boundary checking for every memory access to ensure that it does not deviate from the domain boundary, which becomes a runtime overhead.

4.2 Enclave transition (ECALL/OCALL)

Rust SGX SDK follows the design choice made by Intel SGX SDK for implementing enclave transition wrapper, `ECALL` (enclave call) and `OCALL` (out-call)[1]. To make legacy `ECALLs` and `OCALLs` implemented in C compatible with Rust application code, Rust SGX SDK provides wrapper routines by leveraging Rust's unsafe keyword, which explicitly translates the boundary between C code and Rust code for foreign function interface (FFI). During the conversion, sanity checking is performed, resulting in runtime overhead. Fortanix EDP, on the other hand, defines the `usercall` interface written in Rust, instead of writing `ECALL` and `OCALL` for enclave transition. Because they use their own call process, which is not optimized for SGX, each interaction related to the enclave would generate transition overhead using the `usercall` interface [34]. Similarly, Occlum inserts a trampoline code with a byte that identifies the domain ID in MMDSFI

[1] Note that `ECALLs` are used for to enter the enclave and `OCALLs` are used to switch an execution flow to untrusted region, respectively.

to securely implement untrusted binaries generated by the toolchain in LibOS. In other words, entry into the LibOS within the Enclave can only occur using this trampoline code. Furthermore, to exit outside the LibOS, one must verify the predefined domain ID once again before being allowed to escape. Therefore, from the user's perspective in Occlum, there is no need to write an EDL file. Instead, users can utilize the pre-defined `occlum build` command to build the enclave image and the `occlum run` command to use the enclave entry point. Within the Occlum framework, the run command is passed to the PAL API Layer to enter the enclave. The process of passing through the PAL Layer to enter the enclave can involve transition overhead [18].

4.3 Runtime overhead (Miscellaneous)

The Rust SGX SDK raises an additional overhead due to the dependency on Intel SGX SDK by calling a different directory SGX instruction with the Rust layer, rather than directly executing the assembly code. On the other hand, Fortanix EDP uses its own ABI, called `fortanix-sgx-abi` [9], implemented with a pure rust abstraction layer, so it is relatively overhead-free [15]. When assuming multi-tasking scenario, Occlum has an advantage compared to other frameworks, as it handles multiple process domains(SIPs) within a single enclave region. Such a design also saves the cost of inter-process communication (IPC) overhead between processes.

4.4 Memory safety guranteed by each framework

Both the Rust SGX SDK and Occlum have dependencies on the C language Intel SGX SDK layers, with the Rust SGX SDK utilizing a wrapper API implemented in Rust, and Occlum having 90%of its LibOS code written in Rust. When these frameworks have dependencies on the Intel SGX SDK, they remain susceptible to various vulnerabilities, including DoS attacks and side-channel attacks. In other words, Occlum and Rust SGX SDK may share similar security threats at the library level. However, Occlum can leverage enclave SIP to defend the enclave against attacks such as code injection and ROP attacks by providing isolation between processes that protect SIP from other SIPs and between processes that protect LibOS itself from any SIP and LibOS.

In contrast, Fortanix EDP distinguishes itself by defining its own API and ABI based on the Rust language, thereby enhancing security against vulnerabilities like side-channel attacks that are inherent in the Intel SGX SDK. Additionally, Fortanix EDP is designed in a way that similar to how a LibOS operates, does not expose the enclave interface surface to the user. Additionally, by limiting the number of usercall interfaces to fewer than 20, it reduces the attack surface. Furthermore, it allocates memory in user space and utilizes elements like `fortanix_sgx::usercalls::alloc` to prevent direct memory access, thereby proactively mitigating Time-of-Check-to-Time-of-use (TOCTOU) attack.

Rust SGX SDK introduces an extra layer of wrappers, which can lead to performance degradation. This may manifest as slower enclave execution and a

higher demand for system resources. While Occlum provides isolation between SIPs, there can be overhead in terms of communication and data sharing between processes due to this isolation. Fortanix EDP makes changes to memory allocation and access methods to defend against TOCTOU attacks. However, these changes can result in additional overhead for memory management and internal enclave operations. Additionally, limiting the number of user call interfaces for security purposes can restrict the functionality and flexibility of enclaves. All three frameworks may require extra security and compliance checks during enclave execution and communication, which can slow down the overall execution speed.

5 Performance evaluation

In this section, we describe our experimental setup and present the results of our experimental evaluations of application workloads on each framework. Based on the analysis Section 4, specified the following evaluation metrics: 1)Execution time measurement to evaluate the performance of the application according to the characteristics, 2) Enclave size measurement result to evaluate the enclave hardening and security. The results of the two performance evaluations are summarized in Table 2 and Table 3.

Experimental Setup. Our evaluation was assessed on Ubuntu 20.04. The SGX SDK for developing SGX applications utilized 2.18v. For the Rust language, we used rustc 1.66.0-nightly, which is compatible with all frameworks. Additionally, Occlum used glibc 2.31, as there are glibc versions compatible with running musl-based applications.

Application Benchmark. *Ring* is a library that exposes a Rust API, primarily utilized for performing CPU-intensive workloads related to encryption. It emphasizes the implementation, testing, and optimization of a core set of cryptographic operations exposed through an API that is both easy to use and resistant to misuse. Considering the computationally intensive nature of encryption and decryption processes, we intend to leverage this code to evaluate the CPU computational load of each framework.

HashMap in Rust is utilized for mapping and storing keys and values, offering swift search and insertion operations. However, this process entails the need for basic object implementations, an array of hash tables, and individual objects for each hash item, resulting in a memory-intensive workload with substantial RAM consumption. Moreover, this hash map not only provides a default hash function but also allows users to specify hash functions for custom data types. It permits custom hash behavior for specific data, enabling the implementation of optimal hashing strategies. Chaining is primarily employed for collision handling, and the size dynamically adjusts to automatically optimize memory usage when adding or removing data. We intend to employ this *HashMap*s to assess the memory computational load of each framework.

5.1 Performance Overhead

We evaluated the execution times of Ring, and Hashmap core logic within an Enclave, using a local environment as a baseline, without employing SGX Enclave.

Occlum performs processes by excluding the Occlum toolchain and Occlum verifier from the LibOS, instead delivering only verified MMDSFI to the LibOS. Accordingly, the necessary code (LibOS) is loaded inside the Enclave, minimizing time delays associated with context switching and exhibiting execution times similar to baseline environment. On the other hand, Fortanix EDP, which employs an intermediate Shim layer called `enclave-runner` to load the Enclave and handle logic processing, resulted in significantly higher program execution times. When a user invokes the enclave, the Enclave-runner inspects and sanitizes the code using the Enclave entry ABI, then loads and enters the enclave. Once inside the Enclave, after performing the logic between the `enclave-runner` and the Enclave, the enclave exit ABI is called to terminate the thread. Therefore, including these processes, Fortanix EDP had the longest execution times for application workloads.

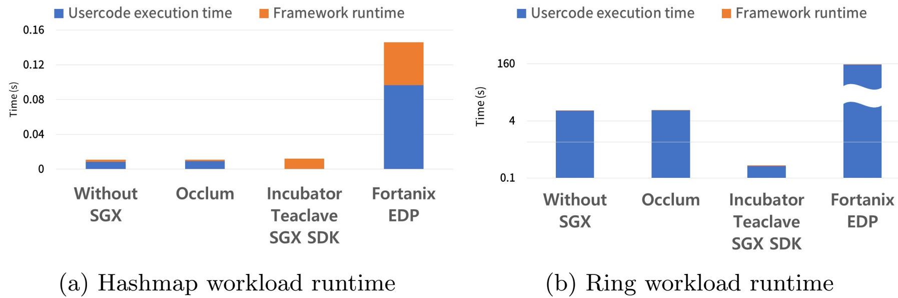
Incubator Teaclave SGX SDK demonstrated the fastest execution times in the Hashmap and Ring workloads. This can be attributed to the use of a Rust wrapper optimized for the Intel SGX API, enabling faster execution even within the SGX environment, including Without SGX execution. Notably, the `sgx_tcrypto` used in the Ring workload called the crypto module implemented in C through unsafe calls, resulting in faster execution times. However, it did not guarantee Rust's memory safety. Therefore, Incubator Teaclave SGX SDK implements functions such as Rust's Lifetimes to ensure memory safety by automatically invoking `drop` functions when the lifespan of objects within `sgx_tcrypto` expires, securely releasing internal references to data in the C/C++ heap, without relying on unsafe calls.

In summary, the performance overhead shows that Incubator Teaclave SGX SDK, which uses SGX-optimized APIs, is the fastest, while Fortanix EDP, which utilizes the intermediate layer of enclave-runner, incurs the most significant performance overhead.

5.2 Enclave Size

Our goal is to evaluate the confidentiality of each framework by measuring the size of the TCB(Trusted Computing Base) that must be safeguarded within the enclave.

In the case of Occlum, we determine the enclave's size by assessing the size of the generated binary. For the Rust SGX SDK, the enclave size can be determined by examining the Enclave.so file generated during the compilation process. In the case of Fortanix EDP, the process involves converting binary files generated using Cargo into SGXS (SGX Stream) files, which adhere to the SGX enclave format. The measurement of enclave size in Fortanix EDP is based on the resulting SGXS file.



(a) Hashmap workload runtime

(b) Ring workload runtime

Fig. 2: Breakdown of benchmark execution time. (Figure 2a and Figure 2b represent charts illustrating the overall runtime of the frameworks and the runtime within the SGX Enclave, respectively. In particular, in the Hashmap workload, the runtime attributed to memory access increases, rendering the framework runtime itself negligible in the representation.)

The usercall API of Fortanix EDP is included within the enclave, yet it allows for the creation of the smallest possible enclave size. This is attributed to the intentional design choice of keeping the usercall API minimal, which is considered to be the reason for this outcome. The Rust SGX SDK follows the enclave design of the Intel SGX SDK but necessitates the inclusion of various Rust wrapper libraries depending on the nature of the workload. As a result, it can be observed that Fortanix EDP generates a relatively larger enclave size compared to the Rust SGX SDK.

As a result, Occlum's Enclave size is assessed as the largest among the frameworks. Occlum incorporates the entire LibOS within a single Enclave. Within the LibOS, there are components such as a binary loader for verifying whether the binary files are signed by the Occlum verifier or Occlum's encrypted file system to securely protect files, contributing to the larger Enclave size evaluation.

	Without SGX (baseline)	**Occlum**	**Incubator Teaclave SGX SDK**	**Fortanix EDP**
Framework runtime	0.011s	0.011s	0.012s	0.146s
Usercode Execution time	0.0084s	0.0090s	0.0004s	0.0965s
Enclave size	N/A	4.4MB	1.4MB	1.18MB

Table 2: Hashmap workload results for each framework

6 Qunatifying engineering effort

To assess the qualitative effort in development, we describe the engineering effort according to the characteristics of the framework and analyze the results for Lines of Code as a factor to evaluate.

	Without SGX (baseline)	Occlum	Incubator Teaclave SGX SDK	Fortanix EDP
Framework runtime	7.661s	7.863s	0.225s	149.037s
Usercode Execution time	7.6584s	7.8610s	0.2130s	148.9848s
Enclave size	N/A	4.5MB	1.6MB	1.19MB

Table 3: Ring(sha2) workload results for each framework

Basically, Rust SGX SDK and Fortanix EDP support utilizing the Rust standard library, and Occlum utilizes the C standard library(`musl_libc` and `glibc`). However, Rust SGX SDK and Fortanix EDP have limitations of several functionalities (e.g., environment variable, timing, networking) due to security concerns. Therefore, development costs are incurred in that developers have to implement these functions themselves to use. In contrast, Occlum not only utilizes using easy-of-use command-line tools unique to Occlum but also provides several built-in toolchains and libraries to facilitate developer porting or development tasks. Then, developers have the disadvantage of having to spend a lot of time learning about SGX SDK APIs, programming models, and systems. In addition, Fortanix EDP can implement the ability to handle memory isolation, `usercalls`, and SGX instruction sets by adding only `std::os::fortanix_sgx` proprietary modules compared to general Rust standard libraries, and relatively reduce programmer development costs. Fortanix EDP also has the advantage of not requiring much experience from developers because it does not require SGX background knowledge and does not require EDL files to separate trust areas.

		Rust Code	EDL File (ECALL/OCALL def)	Cargo.toml	Configuration File
Without SGX (baseline)		12	N/A	10	N/A
Incubator Teaclave SGX SDK	**modified**	2	N/A	8	N/A
	add	81	10	34	N/A
Occlum	**add**	0	N/A	0	17
Fortanix EDP	**add**	0	N/A	3	N/A

Table 4: Hashmap Workload Lines of Code

This evaluation is based on a Hashmap workload in a local environment without utilizing the SGX enclave as a reference. The results of the additional Lines of Code are summarized in Table 4 as follows. Rust's Cargo serves as a package manager for building and managing Rust applications. To build packages using Cargo, the creation of a Cargo.toml configuration file is required. Additionally, SGX also requires the Enclave.edl file with the context switch. This file defines ECALLs for entering the reserved Enclave and OCALLs for returning from the Enclave to the user space.

Rust SGX SDK provides a Rust wrapper for the Intel SGX SDK, originally written in C/C++. It uniquely distinguishes between the app and Enclave areas, necessitating the definition of the Enclave.edl file. As a result, in the main logic of the app layer, instead of using the pure Rust standard libraries, the developer employed the provided `sgx_types` and `sgx_urts`. It also, involved writing code for creating the Enclave, making function calls to enter the Enclave, executing code within the Enclave, and retrieving the results. Within the Enclave, the developer performed the Hashmap workload. Ultimately, this resulted in 2 lines being modified and an additional 81 lines of source code being written.

Occlum offers a user-friendly Occlum-cargo command to execute Rust applications, and it provides shell scripts and yaml files for this purpose. As a result, there was no need to modify or add significant code to the core logic of the Hashmap workload or the Cargo.toml file. However, there was a requirement to write 17 lines of source code for the shell scripts and yaml file.

In Fortanix EDP, a pure Rust language approach was utilized, along with a custom ABI/API, to ensure security by not exposing the Enclave interface to developers. This design choice allowed for the avoidance of writing an Enclave.edl file. The core logic of the Hashmap workload was leveraged without any modifications, thanks to the support of the Rust standard library. Instead of using a custom ABI/API, the Cargo.toml file was configured with a build target of `x86_64-fortanix-unknown-sgx` for building. As a result, only three lines of source code were added to the Cargo.toml file.

To minimize the developer's effort, it is evaluated as most suitable to utilize Fortanix EDP, which allows the development of applications using only the Rust language without requiring background knowledge of the SGX architecture.

7 Related Work

Gramine [18], previously known as Graphene, is a lightweight library operating system designed for Linux multi-process applications. This unique library OS facilitates the execution of existing applications within SGX enclaves without necessitating any modifications, except for the inclusion of an enclave manifest specifying security settings and configurations. Gramine uses this manifest to perform authenticity and integrity verification and subsequently leverages it to load the application along with its requisite dependencies.

SCONE [4] is a software platform designed for securely running container-based applications using SGX within Docker containers. It offers a secure C standard library interface that automatically encrypts and decrypts input/output (I/O) data, thereby minimizing the performance impact of thread synchronization and system calls during the enclave transition. In addition, SCONE supports user-level threading and asynchronous system calls to improve performance.

PANOPLY [32] represents a system designed to bridge the gap between the standard OS abstraction and the specific requirements of SGX for commercial Linux applications. Inspired by the principles of micro-kernels, PANOPLY has completely rethought the logic of the OS without trying to emulate it. It achieves

this by intercepting calls to the glibc API, which allows the glibc library to reside outside the enclave's TCB. Consequently, even if the underlying OS encounters issues or malfunctions, PANOPLY ensures the application's integrity attributes remain intact, ensuring its continued proper functioning.

Among them, SCONE and PANOPLY employ thin "shim" layers that encapsulate API layers like system call tables. This architectural strategy serves the purpose of minimizing the code required within the enclave, thereby reducing both the interface's size and the potential attack surface between the enclave and the untrusted OS. Gramine, SCONE, and Panoply all represent solutions for enhancing the security of applications in container environments. They share the common characteristic of being developed in the C programming language, which means that they may not exhibit the same level of robust memory safety as the Rust-based SGX frameworks examined in this paper.

Several studies have aimed to streamline the engineering effort required for deploying applications in SGX environments, simplifying the process for developers. Glamdring [23] proposes automating the code partitioning process to utilize SGX. Once developers annotate security-sensitive data of the target application, Glamdring automatically splits the application into two sections: one for the trusted enclave and the other for the untrusted, non-enclave part. Through efficient code relocation, including the creation of SDK interface specifications and the relocation of resource-intensive features outside the enclave via runtime profiling, Glamdring minimizes the engineering effort involved.

Hasan et al. [11] conduct the comparison of the comparison between 'Port' and 'Shim' approaches for implementing SGX applications. The porting approach entails rewriting or modifying the application's code to align with the SGX environment. While it may be more complex, it typically offers superior performance. Conversely, the shimming approach involves the creation of an intermediary layer that acts as an adapter between the application and the new SGX environment. This approach requires fewer code changes due to the presence of SGX libraries but may introduce some performance overhead. The choice between 'Port' and 'Shim' hinges on various factors, including time constraints, available resources, and performance requirements, providing developers with flexibility in their approach.

Existing research on SGX-related studies for enhancing application security in container environments commonly share the characteristic of being developed in the C programming language. However, it is essential to note that, compared to the Rust-based frameworks analyzed in this paper, these solutions may not be as robust in terms of memory safety, owing to their development in C/C++. In contrast to the aforementioned studies, our studies focus on analyzing SGX frameworks that utilize the Rust programming language to enhance the security of user code and data from a memory safety perspective. Furthermore, we assess the performance of these three frameworks, each with distinct methods of supporting SGX, from the standpoint of developers. This assessment aims to provide guidelines that can promote the adoption of SGX.

8 Conclusion

This paper analyzes the implementation cost when developing Rust applications with existing Rust-based SGX frameworks. Through the comparative analysis over three frameworks, we confirm that Occlum has strength in performance, while developing Rust applications using Fortanix EDP is effective from the implementation cost perspective.

References

1. Intel Software Guard Extensions (Intel SGX) SDK. https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html (accessed on Jun. 2021)
2. Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., Popa, D.M.: Firecracker: Lightweight virtualization for serverless applications. In: 17th USENIX symposium on networked systems design and implementation (NSDI 20). pp. 419–434 (2020)
3. Amazon: AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/
4. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: SCONE: Secure linux containers with intel SGX. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). pp. 689–703. USENIX Association, Savannah, GA (Nov 2016), https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov
5. Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'Keeffe, D., Stillwell, M.L., Goltzsche, D., Eyers, D., Kapitza, R., Pietzuch, P., Fetzer, C.: Scone: Secure linux containers with intel sgx. In: SCONE: Secure Linux Containers with Intel SGX. p. 689–703. OSDI'16, USENIX Association, USA (2016)
6. Baumann, A., Peinado, M., Hunt, G.: Shielding applications from an untrusted cloud with haven. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 267–283. USENIX Association, Broomfield, CO (Oct 2014), https://www.usenix.org/conference/osdi14/technical-sessions/presentation/baumann
7. Durumeric, Z., Kasten, J., Adrian, D., Halderman, A., Bailey, M., Li, F., Weaver, N., Amann, J., Beekman, J., Payer, M., Paxson, V.: The Matter of Heartbleed. In: Proc. IMC. ACM (2014)
8. Fortanix: Fortanix EDP. https://edp.fortanix.com/
9. Fortanix: Fortanix sgx abi. https://edp.fortanix.com/docs/api/fortanix_sgx_abi/
10. Han, J., Yun, I., Kim, S., Kim, T., Son, S., Han, D.: Scalable and secure virtualization of hsm with scaletrust. IEEE/ACM Transactions on Networking (2022)
11. Hasan, A., Riley, R., Ponomarev, D.: Port or shim? stress testing application performance on intel sgx. In: 2020 IEEE International Symposium on Workload Characterization (IISWC). pp. 123–133 (2020). https://doi.org/10.1109/IISWC50251.2020.00021

12. Hoekstra, M., Lal, R., Pappachan, P., Phegade, V., Del Cuvillo, J.: Using innovative instructions to create trustworthy software solutions. HASP@ ISCA **11**(10.1145), 2487726–2488370 (2013)
13. Hunt, T., Song, C., Shokri, R., Shmatikov, V., Witchel, E.: Chiron: Privacy-preserving machine learning as a service. arXiv preprint arXiv:1803.05961 (2018)
14. Intel: Intel MPX. https://intel-mpx.github.io/
15. Jo Van Bulck, Fritz Alder, F.P.: A case for unified abi shielding in intel sgx runtimes. In: A Case for Unified ABI Shielding in Intel SGX Runtimes. systex '22 (2022)
16. Jung, R., Jourdan, J.H., Krebbers, R., Dreyer, D.: Rustbelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang. **2**(POPL) (dec 2017). https://doi.org/10.1145/3158154, https://doi.org/10.1145/3158154
17. Kulkarni, C., Moore, S., Naqvi, M., Zhang, T., Ricci, R., Stutsman, R.: Splinter:{Bare-Metal} extensions for {Multi-Tenant}{Low-Latency} storage. In: 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). pp. 627–643 (2018)
18. Kuvaiskii, D., Kumar, G., Vij, M.: Computation offloading to hardware accelerators in intel sgx and gramine library os (2022)
19. Lankes, S., Breitbart, J., Pickartz, S.: Exploring rust for unikernel development. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. pp. 8–15 (2019)
20. Lee, J., Jang, J., Jang, Y., Kwak, N., Choi, Y., Choi, C., Kim, T., Peinado, M., Kang, B.B.: Hacking in darkness: Return-oriented programming against secure enclaves. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 523–539 (2017)
21. Lee, T., Lin, Z., Pushp, S., Li, C., Liu, Y., Lee, Y., Xu, F., Xu, C., Zhang, L., Song, J.: Occlumency: Privacy-preserving remote deep-learning inference using sgx. In: The 25th Annual International Conference on Mobile Computing and Networking. pp. 1–17 (2019)
22. Liang, X., Shetty, S., Zhang, L., Kamhoua, C., Kwiat, K.: Man in the cloud (mitc) defender: Sgx-based user credential protection for synchronization applications in cloud computing platform. In: 2017 IEEE 10th International Conference on Cloud Computing (CLOUD). pp. 302–309. IEEE (2017)
23. Lind, J., Priebe, C., Muthukumaran, D., O'Keeffe, D., Aublin, P.L., Kelbert, F., Reiher, T., Goltzsche, D., Eyers, D., Kapitza, R., Fetzer, C., Pietzuch, P.: Glamdring: Automatic application partitioning for intel SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 285–298. USENIX Association, Santa Clara, CA (Jul 2017), https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind
24. Matsakis, N.D., Klock, F.S.: The rust language. In: The Rust Language. p. 103–104. HILT '14, Association for Computing Machinery, New York, NY, USA (2014). https://doi.org/10.1145/2663171.2663188, https://doi.org/10.1145/2663171.2663188
25. McKeen, F.X., Alexandrovich, I., Berenzon, A., Rozas, C.V., Shafi, H., Shanbhogue, V., Savagaonkar, U.R.: Innovative instructions and software model for isolated execution. In: HASP '13 (2013)
26. Microsoft: Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/
27. Priebe, C., Muthukumaran, D., Lind, J., Zhu, H., Cui, S., Sartakov, V.A., Pietzuch, P.R.: SGX-LKL: securing the host OS interface for trusted execution. CoRR **abs/1908.11143** (2019), http://arxiv.org/abs/1908.11143

28. Rashid, F.Y.: The rise of confidential computing: Big tech companies are adopting a new security model to protect data while it's in use-[news]. IEEE Spectrum **57**(6), 8–9 (2020)
29. Schwarz, M., Weiser, S., Gruss, D.: Practical enclave malware with intel sgx. In: Detection of Intrusions and Malware, and Vulnerability Assessment: 16th International Conference, DIMVA 2019, Gothenburg, Sweden, June 19–20, 2019, Proceedings 16. pp. 177–196. Springer (2019)
30. Shanker, K., Joseph, A., Ganapathy, V.: An evaluation of methods to port legacy code to sgx enclaves. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1077–1088 (2020)
31. Shen, Y., Tian, H., Chen, Y., Chen, K., Wang, R., Xu, Y., Xia, Y., Yan, S.: 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. ACM (mar 2020). https://doi.org/10.1145/3373376.3378469, `https://doi.org/10.1145%2F3373376.3378469`
32. Shinde, S., Le, D., Tople, S., Saxena, P.: Panoply: Low-tcb linux applications with sgx enclaves (01 2017). https://doi.org/10.14722/ndss.2017.23500
33. che Tsai, C., Porter, D.E., Vij, M.: Graphene-SGX: A practical library OS for unmodified applications on SGX. In: 2017 USENIX Annual Technical Conference (USENIX ATC 17). pp. 645–658. USENIX Association, Santa Clara, CA (Jul 2017), `https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai`
34. Wang, H., Wang, P., Ding, Y., Sun, M., Jing, Y., Duan, R., Li, L., Zhang, Y., Wei, T., Lin, Z.: Towards memory safe enclave programming with rust-sgx. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security. p. 2333–2350. CCS '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3319535.3354241, `https://doi.org/10.1145/3319535.3354241`
35. Wang, J., Wang, J., Fan, C., Yan, F., Cheng, Y., Zhang, Y., Zhang, W., Yang, M., Hu, H.: Svtpm: Sgx-based virtual trusted platform modules for cloud computing. IEEE Transactions on Cloud Computing (2023)