저작자표시-비영리-변경금지 2.0 대한민국

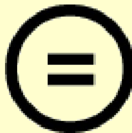
이용자는 아래의 조건을 따르는 경우에 한하여 자유롭게

- 이 저작물을 복제, 배포, 전송, 전시, 공연 및 방송할 수 있습니다.

다음과 같은 조건을 따라야 합니다:

저작자표시. 귀하는 원저작자를 표시하여야 합니다.

비영리. 귀하는 이 저작물을 영리 목적으로 이용할 수 없습니다.



변경금지. 귀하는 이 저작물을 개작, 변형 또는 가공할 수 없습니다.

- 귀하는, 이 저작물의 재이용이나 배포의 경우, 이 저작물에 적용된 이용허락조건을 명확하게 나타내어야 합니다.
- 저작권자로부터 별도의 허가를 받으면 이러한 조건들은 적용되지 않습니다.

저작권법에 따른 이용자의 권리는 위의 내용에 의하여 영향을 받지 않습니다.

이것은 이용허락규약(Legal Code)을 이해하기 쉽게 요약한 것입니다.

Disclaimer

dCollection

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

Heekyung Shin

Department of Future Convergence
Technology Engineering
The Graduate School of Sungshin Women's
University

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

A Master's Thesis
Submitted to the
Graduate School of Sungshin Women's University

in partial fulfillment of the requirements
for the degree of
Master of Future Convergence Technology Engineering

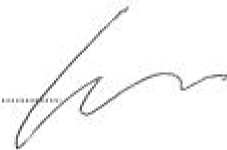
Heekyung Shin

November, 2023

This is to certify that we have examined the
Master's Thesis of
Heekyung Shin
Submitted to Department of Future Convergence
Technology Engineering

Thesis Advisor 김 성 민

Committee Chairman 이 일 구

Committee Member 임 연 섭 

The Graduate School of Sungshin University

Abstract

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

Heekyung Shin
Department of Future Convergence Technology engineering
Graduate School of
Sungshin Women's University

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. Furthermore, quantifying the engineering effort for migrating legacy Rust applications and comparing the supplementary overhead incurred when subjecting these frameworks to CPU and memory-intensive workloads are essential aspects of our analysis. Through this examination, the goal is to offer valuable guidance to developers who are in the process of selecting a Rust-based SGX framework that aligns with their application's specific purpose and workload characteristics.

Contents

Abstract

Ⅰ. Introduction ······ 1

Ⅱ. Background ······ 5

1. Intel SGX and LibOS-based SGX Framework ······ 5
2. Rust Programming Language ······ 7

Ⅲ. Characteristics Analysis of Frameworks ······ 9

1. Occlum ······ 10
2. Fortanix EDP ······ 11
3 Rust SGX SDK (Incubator Teaclave SGX SDK) ······ 12

Ⅳ. Qualitative Aspects Affecting Application Performance ······ 14

1. Memory Boundary Check ······ 15
2. Enclave Transition (ECALL/OCALL) ······ 15
3. Runtime Overhead (Miscellaneous) ······ 16
4. Memory Safety Guaranteed by Each Framework(Miscellaneous) ······ 17

Ⅴ. Comparison of Framework Performance Using Application Benchmarks ······ 19

1. Performance Overhead ······ 20

2. Enclave Size ········ 22

Ⅵ. Quantifying engineering effort ········ 27

Ⅶ. Related Work ········ 32

Ⅷ. Future Work ········ 35
1. Design Goal and Challenges ········ 38
2. A Proof-of-concept System Design ········ 42

Ⅸ. Conclusion ········ 44

References
논문개요
ACKNOWLEDGEMENTS

Table Contents

TABLE 1. Estimating framework performance impact overhead based on framework analysis 14
TABLE 2. Hashmap workload results for each framework 26
TABLE 3. Ring(sha2) workload results for each framework 27
TABLE 4. Hashmap Workload Lines of Code 31

Figure Contents

FIGURE 1. Rust-based SGX Framework Overview ····················· 10
FIGURE 2. Breakdown of benchmark execution time: Hashmap workload runtime ····· 25
FIGURE 3. Breakdown of benchmark execution time: Ring workload runtime ····· 25
FIGURE 4. System overview of SGX-enabled 5G VNFs in action ··· 39

I. Introduction

The commercialization of Intel Software Guard Extensions (SGX) technology[1] has garnered substantial industrial and academic attention. In particular, Intel SGX technology plays a pivotal role in evolving the confidential computing paradigm[2]. 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[3]. 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 groundbreaking solutions, known as confidential VMs, include commercial products like Amazon Nitro Enclaves[4] and Azure Confidential Computing[5]. Such innovation has expedited the widespread adoption of confidential computing across diverse domains, such as safeguarding AI/ML models[6][7], protecting digital assets[8], and securing key management services[9][10].

Basically, there are two primary approaches for implementing the SGX program: 1) porting an application based on SGX SDK[11] and 2) running unmodified applications on top of frameworks that support SGX compatibility[3]. 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[3][12][13][14].

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 [15]) can still be effective even when executed within the security boundary provided by SGX CPU[16][17]. 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[18]. 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[19], microkernels[20], and storage systems[21].

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[13][14]. Besides, several studies[22][23][24] utilize Rust programming language[18] as the foundation 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 analysis of existing Rust-based SGX frameworks to provide implications for newly implementing or porting of legacy security-sensitive Rust applications. The analysis involves an in-depth examination between three cutting-edge Rust-based SGX frameworks: Rust SGX SDK, Occlum, and Fortanix EDP. First, exploration is undertaken into the internal implementation details of each framework, specifically focusing on aspects relevant to application performance and security. Then, the engineering effort required to deploy

legacy Rust applications on these frameworks is quantified, providing insights into the ease of transition. Finally, an comparison is performed on the additional overhead incurred by each framework, as they are subjected to CPU-intensive and memory-intensive workloads, with the aim of gauging their performance implications. This comprehensive analysis is intended to guide developers in selecting an appropriate Rust-based SGX framework for implementing an SGX application, considering its purpose and workload characteristics.

Ⅱ. Background and Related Work

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. Before creating an enclave, the User should identify the assets that need to be protected, the data structures that contain the assets, and the code sets that work with them. This step is a required step because when the enclave is initialized and code is loaded into memory, untrusted components cannot read or change that code. After identifying them, they are placed in a separate trusted library. 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. To help developers implement an SGX application, Intel released an SDK that provides libraries and toolchains for signing and debugging an enclave. an API, library, documentation, sample source codes, and toolchains for signing and debugging an enclave. Developers

can use the Intel SGX SDK after removing code modules that access secrets and then moving them to separate libraries/packages and modifying applications.

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. After defining the trusted and untrusted (Host) components of the Intel SGX program, the interface between the two must be defined as a reliability criterion. 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. Enclave should expose APIs ECALLs that can be invoked by untrusted code and express whether features provided by untrusted code are required OCALLs. 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 [26]. Frameworks such as Gramine[14], SGX-LKL[13], and Haven[3], which implement LibOS-based SGX, offer the advantage of enhancing portability by freeing applications from dependence on a specific operating system.

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[18]. 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[18]. 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 [18]. Rust introduces a unique ownership system central to its memory safety guarantees[27]. 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[27]. It prevents references from outliving the data they point to or being used after the data has been deallocated.

Ⅲ. 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[22][23][24]. In particular, this study provides 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.

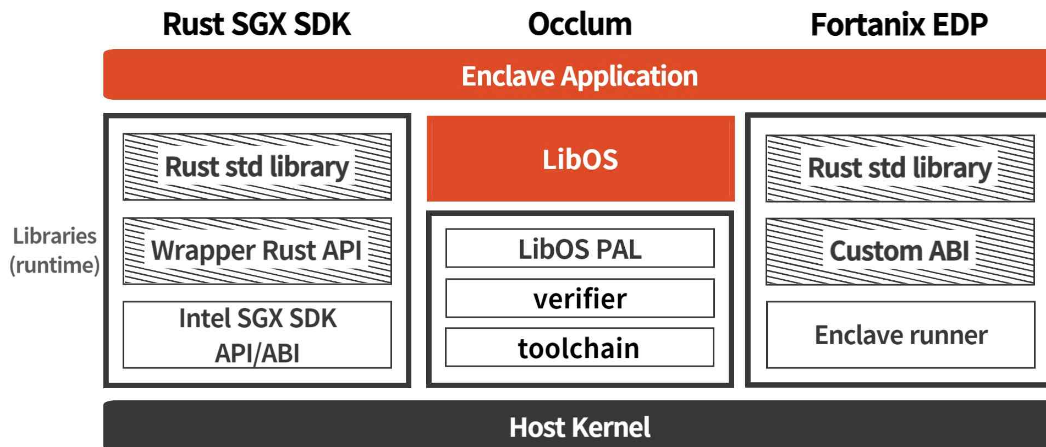


FIGURE 1. Rust-based SGX Framework Overview. (The red boxes indicate regions that are isolated and protected by the enclave application, while the black dashed boxes are regions that are written in Rust.

1. Occlum

Occlum is a memory-safe multi-process LibOS for Intel SGX to enable execution of legacy applications without modifying the source code[23]. Occlum proposes multi-domain software fault isolation (MMDSFI) by leveraging Intel Memory Protection Extensions (MPX) technology [28] 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[3][12][14] startup time is 1000 times faster and IPC (inter-process communication) is up to 3 times faster[23].

2. Fortanix EDP

Enclave development platform (EDP)[22], 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[22]. 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. Rust SGX SDK (Incubator Teaclave SGX SDK)

The Rust SGX SDK, developed by Baidu, offers a secure platform for executing Rust-based applications within SGX environment[24]. 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.

Ⅳ. Qualitative Aspects Affecting Application Performance

In this section, an in-depth analysis is conducted by systematically exploring the internal design of each framework and categorizing three key indicators related to application performance: memory boundary check, Enclave transition, and additional runtime overhead. Table 1 summarizes the analysis result.

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

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

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.

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). 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[24]. Similarly, Occlum eliminates the need for writing EDL files, allowing users to build enclave images for enclave invocation using pre-defined Occlum build commands. Users can then use the Occlum run command as an entry point to the enclave. Within the Occlum framework, the run command is passed to the PAL API Layer, enabling entry into the enclave. Therefore, the process of passing through the PAL Layer to enter the enclave can introduce transition overhead[29].

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*[30], implemented with a pure rust abstraction layer, so it is relatively overhead-free[31]. 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. Memory safety guaranteed 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.

V. Comparison of Framework Performance Using Application Benchmarks

In this section, the experimental setup is described, and the results of the experimental comparisons of application workloads on each framework are presented. 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, by performing

10,000 iterations of the operation, aiming to assess the CPU workload 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. Utilizing a Hashmap function, 1 million iterations were conducted to incrementally increase memory consumption, with the intention of evaluating the memory workload of each framework in terms of memory operations.

1. Performance Overhead

The execution times of Ring and Hashmap core logic within an enclave were compared using a local environment as a baseline, without employing SGX Enclave. Occlum, by loading all necessary code (LibOS) inside the enclave, exhibited execution times similar to baseline execution, as it did not introduce time delays for region switching. 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

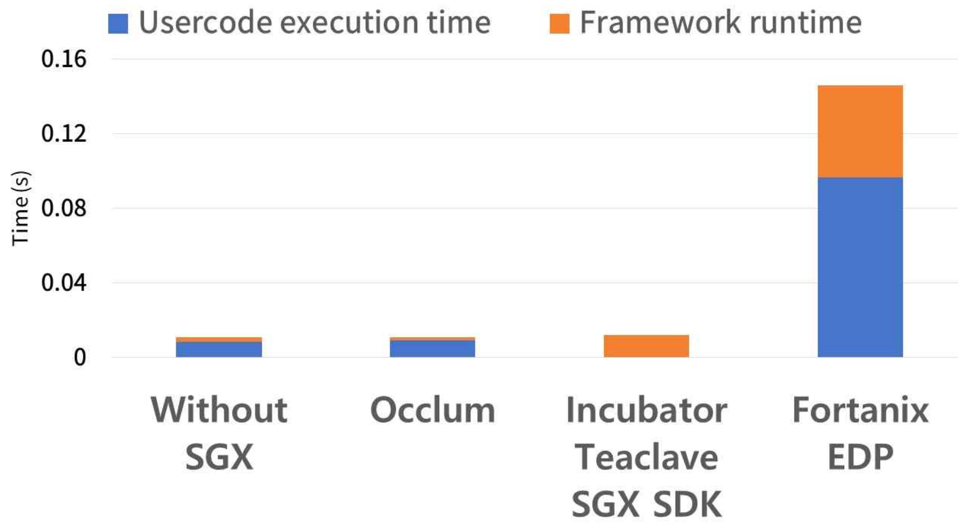
C/C++ heap, without relying on unsafe calls.

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.

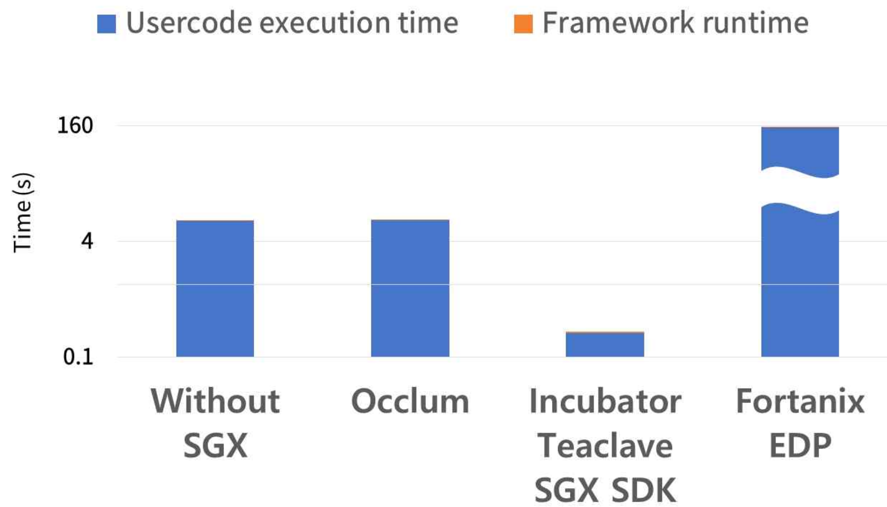
2. Enclave Size

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, the enclave's size is determined by assessing the size of the generated binary, as all the necessary code is enclosed within the enclave. 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. 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.



(a) Hashmap workload runtime



(b) Ring workload runtime

FIGURE 2: Breakdown of benchmark execution time. ((a) and (b) 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.)

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

TABLE 2: Hashmap workload results for each framework

	Without SGX(baseline)	Occlum	Incubator Teaclave SGX SDK	Fortanix EDP
Framework runtime (s)	7.661	7.863	0.225	149.037
Usercode Execution time (s)	7.6584	7.8610	0.2130	149.9848
Enclave Size (MB)	N/A	4.5	1.6	1.19

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

Ⅵ. Quantifying engineering effort

To assess the qualitative effort in development, the engineering effort is described according to the characteristics of the framework, and the results for Lines of Code are analyzed as a factor to evaluate.

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 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.

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.

		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

Ⅶ. Related Work

Gramine[29], 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[32] 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[33] 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[34] 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. conduct a 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, the performance of these three frameworks, each with distinct methods of supporting SGX, is assessed from the standpoint of developers. This assessment aims to provide guidelines that can promote the adoption of SGX.

Ⅷ. Future Work

When Rust-based SGX frameworks are reliably utilized across industries, it is anticipated that these frameworks will enable the development, deployment, and commercialization of secure 5G network function virtualization (NFV) components within a secure environment. Therefore, a system design that utilizes Intel SGX to protect VNFs (Virtual Network Functions) within 5G cores is proposed. This simulation demonstrates the potential advantages of SGX in terms of security and privacy protection within existing 5G core systems. Originating from industrial efforts to integrate hardware-based security features into existing 4G networks, SGX is viewed as a pivotal technology for successfully integrating 5G cores with SDN/NFV (Software-Defined Networking/Network Function Virtualization).

Recent advances in software-defined networking (SDN) and network function virtualization (NFV) technologies accelerate the evolution of telecommunication networks towards next-generation mobile applications. In particular, mobile packet core in 5G is one of the examples that perfectly aligned with the SDN and NFV paradigms, taking advantage of a cloud-native approach such as flexibility, agility, and extensibility.

According to the 3GPP 5G specification release 15[36], the 5G core has standardized as a service-based architecture (SBA) that consists of modularized but interconnected network functions (NFs), unlike evolved

packet core (EPC) system of 4G (LTE). Aligned with the industrial efforts, SDN and NFV research activities for developing 5G core pave the way to meet the quality of service (QoS) requirements of emerging mobile services such as virtual/augmented reality and autonomous driving.

However, the trend of softwarization in the 5G network enabled by SDN and NFV technology introduces new security and privacy challenges. Due to the untrusted nature of the cloud, VM (or container) instances cannot be protected from honest but curious cloud platform providers and inside intruders. Such inherent limitation allows the adversaries to directly monitor virtualized network functions (VNFs) that compose the 5G mobile packet core and observe the communication channel between VNFs. For example, a malicious intruder can observe the communication interface between authentication server function(AUSF) and unified data management (UDM) to stealthily obtain subscriber information and a charging policy. Even worse, it is almost impossible to detect the potential threats using the cloud privilege, which makes 5G service providers unaware of the attack occurrence.

This section tackles the security and privacy problems in the state-of-the-art 5G mobile packet core NFs based on SBA. To this end, we present a proof-of-concept design for a 5G core network that leverages recent hardware-based trusted execution environment (TEE) technology, Intel software guard extension (SGX)[1], to provide secure containers for the 5G VNFs. Using the isolated execution provided by

Intel SGX, even powerful adversaries who can control the entire software stack of the cloud platform cannot compromise the core operations of the VNFs, such as subscriber registration, authentication, session management, and et cetera. In addition, our design employs remote attestation functionality to enable integrity verification of NFs running on the cloud, allowing each VNF (or 5G service provider) can figure out whether the target VNF is benign or not before interaction. To demonstrate its viability, a case study is provided when deploying SGX-enabled unified data management (UDM) alongside the VNFs in the control plane. The envisioned system design aims to enhance the security and privacy issues raised in a recent innovation of the 5G core.

1. Design Goal and Challenges

Security is one of the main challenges for deploying SDN/NFV to a mobile packet core. It is challenging to provide isolation between the packet core entities, such as mobile management entity (MME), home subscriber server (HSS), and policy and charging rules function (PCRF). In addition, additional authentication and authorization against virtualized entities are required before their intercommunications because each packet core entity is running on the untrusted cloud. Note that the SDN deployment also leads to the increment of the participating entities due to the separation of control/data plane, increasing the total requests/responses during the authentication procedure. To address this, Open Networking Foundation (OMEC) pioneered an effort to utilize recent hardware-based TEE technology[37]. With the collaboration with Intel, they employ SGX for a secure offline charging service (OFCS) of EPC in the LTE system. Likewise, the upcoming 5G core networks should satisfy two design goals: 1) protecting NFs not to reveal user's data that resides in the application layer (ring-3) from the untrusted system software (e.g., OS and hypervisor) and 2) providing a secure interface between modularized NFs.

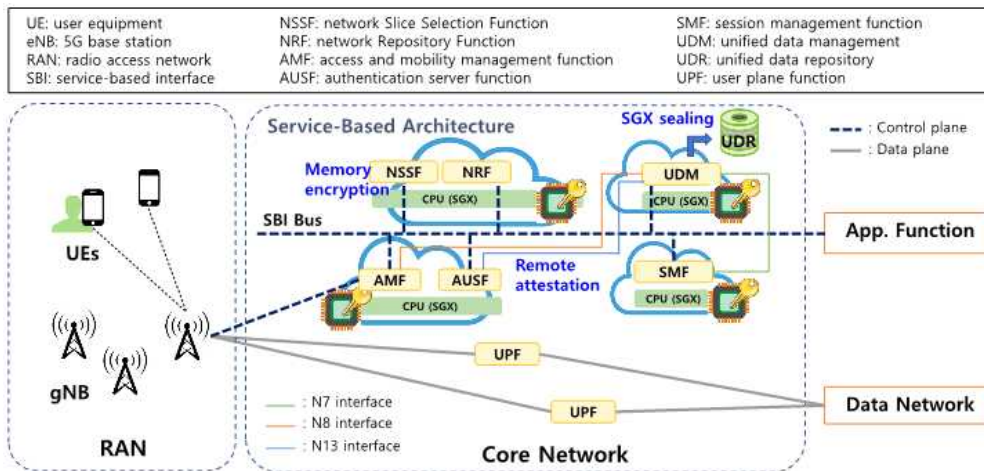


FIGURE 4. System overview of SGX-enabled 5G VNFs in action

2. A Proof-of-concept System Design

Threat Model and Assumptions. In this paper, the SDN/NFV based service-based architecture defined by 3GPP [36] is considered. This architecture separates the control plane and the data plane functionality, and each entity to serve the 5G mobile packet core has a form of a virtualized network function. Among the 5G deployment options following the NSA (Non-stand-alone) and SA (Stand-alone) specifications [36], the focus is on leveraging SGX to SA standard without considering LTE network entities (e.g., Home Subscriber Server). Also, SGX threat model stated by Intel [38] is also assumed, where the software and hardware components are considered untrusted, except for CPU packages. By leveraging SGX technology, 5G service providers can protect the code and data of network function within an enclave—a CPU-hardened secure container provided by SGX—under our assumption that SGX enclaves, including platform service enclaves (PSE) for managing SGX services (e.g., enclave launch and remote attestation), are trustworthy. In other words, a powerful adversary who can control over system software of the cloud platform cannot compromise the code and data of network functions (or some parts) within an SGX enclave. Note that side-channel against SGX and memory vulnerabilities within an enclave are outside of the scope.

System Design. To achieve the above design goals, A new 5G core system is proposed, consisting of VNF instances running on the cloud equipped with SGX. Inspired by previous studies that leverage SGX in

NFV[39, 40, 41], core SGX functionalities are used to provide enhanced security and privacy for the 5G core. This includes isolated execution and sealing/unsealing properties to protect privacy-sensitive data and corresponding operations, as well as remote attestation to verify the genuineness of a target NF before the communication establishment, respectively.

Figure 1 shows the overall system architecture. As shown in the figure, 5G VNF instances located in the control plane are executed on top of SGX hardware. This means that cryptographic operations (e.g., key generation) and authentication processes between NFs related to 5G core service are securely protected, and sensitive information such as subscription data never leaves SGX enclaves. SGX also provides sealing/unsealing APIs to securely export to persistent storage by encrypting enclave secrets with a sealing key derived from a hardware-specific root sealing key. Therefore, the enclave that previously encrypts the sealed data can only decrypt it. Finally, to avoid performance overhead introduced by SGX (e.g., SGX mode switching and enclave paging), one possible option is to leave operations that do not process sensitive data and are already protected by cryptographic protocols (e.g., IPSec and TLS) in the untrusted region. Another core functionality to protect 5G VNFs is secure channel establishment based on SGX remote attestation. It offers integrity verification against an SGX enclave that is running on the remote platform. This allows 5G service providers to figure out which VNFs are modified and not protected by SGX hardware (e.g., SGX emulation).

For example, the network repository function (NRF) performs remote attestation to verify the integrity of a target VNF before its registration. Therefore, potentially malicious VNFs controlled by adversaries are explicitly excluded from the 5G packet processing and signal processing routines. In addition, it is possible to establish a secure channel between VNFs(specifically, enclave region) by combining the TLS handshake procedure with remote attestation. It is also noted that the key design strategy can apply to the existing 4G EPC NFs that reside on the control plane, as discussed in [37].

Protecting subscription/authentication data. Now illustrating a scenario to protect subscription and authentication data from the untrusted cloud involves deploying SGX-enabled unified data management (UDM) service alongside the VNFs in the control plane. Similar to HSS in EPC, the main functions of UDM are user identification handling, access authorization, and subscription data management. By leveraging SGX, UDM generates authentication credentials that hold the information in the enclave memory. Then, it provides trustworthy authentication credentials to the access and mobility management function (AMF) and the session management function (SMF), so they can securely retrieve subscriber data and context. Finally, UDM can use SGX sealing when exporting data to the external Unified Data Repository (UDR) to retrieve the information for further use.

This section presents a system design for protecting VNFs of the 5G

core by leveraging a recent TEE technology, Intel SGX. Our showcasing example shows how SGX benefits the existing 5G core system regarding security and privacy. Starting from an industrial effort to integrate hardware-based security features into the existing 4G network, it believes that SGX is a key enabling technology for the 5G core to integrate with SDN/NFV successfully. We plan to implement SGX-enabled VNFs and evaluate their practicality with real SGX hardware by porting open-source 5G projects[42, 43] to the Occlum[23].

Ⅸ. Conclusion

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

References

[1] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. HASP@ ISCA, 11(10.1145):2487726–2488370, 2013.

[2] Fahmida Y Rashid. 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.

[3] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14), pages 267–283, Broomfield, CO, October 2014. USENIX Association.

[4] Amazon. AWS Nitro Enclaves. https://aws.amazon.com/ec2/nitro/nitro-enclaves/.

[5] Microsoft. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/.

[6] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In The 25th Annual International Conference on Mobile Computing and Networking, pages 1–17,

2019.

[7] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. arXiv preprint arXiv:1803.05961, 2018.

[8] Xueping Liang, Sachin Shetty, Lingchen Zhang, Charles Kamhoua, and Kevin Kwiat. 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), pages 302 - 309. IEEE, 2017.

[9] Juan Wang, Jie Wang, Chengyang Fan, Fei Yan, Yueqiang Cheng, Yinqian Zhang, Wenhui Zhang, Mengda Yang, and Hongxin Hu. Svtpm: Sgx-based virtual trusted platform modules for cloud computing. IEEE Transactions on Cloud Computing, 2023.

[10] Juhyeng Han, Insu Yun, Seongmin Kim, Taesoo Kim, Sooel Son, and Dongsu Han. Scalable and secure virtualization of hsm with scaletrust. IEEE/ACM Transactions on Networking, 2022.

[11] 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).

[12] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, R ¨udiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In SCONE: Secure Linux Containers with Intel SGX, OSDI'16, page 689 - 703, USA, 2016. USENIX

Association.

[13] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A. Sartakov, and Peter R. Pietzuch. SGX-LKL: securing the host OS interface for trusted execution. CoRR, abs/1908.11143, 2019.

[14] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-SGX: A practical library OS for unmodified applications on SGX. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 645 – 658, Santa Clara, CA, July 2017. USENIX Association.

[15] Zakir Durumeric, James Kasten, David Adrian, Alex Halderman, Michael Bailey, Frank Li, Nicolas Weaver, Johanna Amann, Jethro Beekman, Mathias Payer, and Vern Paxson. The Matter of Heartbleed. In Proc. IMC. ACM, 2014.

[16] Michael Schwarz, Samuel Weiser, and Daniel Gruss. 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, pages 177 – 196. Springer, 2019.

[17] Jaehyuk Lee, Jinsoo Jang, Yeongjin Jang, Nohyun Kwak, Yeseul Choi, Changho Choi, Taesoo Kim, Marcus Peinado, and Brent ByungHoon Kang. Hacking in darkness: Return-oriented programming against secure enclaves. In 26th USENIX Security Symposium (USENIX Security 17), pages 523 – 539, 2017.

[18] Nicholas D. Matsakis and Felix S. Klock. The rust language. In The Rust Language, HILT '14, page 103 – 104, New York, NY, USA,

2014. Association for Computing Machinery.

[19] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In 17th USENIX symposium on networked systems design and implementation (NSDI 20), pages 419 – 434, 2020.

[20] Stefan Lankes, Jens Breitbart, and Simon Pickartz. Exploring rust for unikernel development. In Proceedings of the 10th Workshop on Programming Languages and Operating Systems, pages 8 – 15, 2019.

[21] Chinmay Kulkarni, Sara Moore, Mazhar Naqvi, Tian Zhang, Robert Ricci, and Ryan Stutsman. Splinter:{Bare-Metal} extensions for {Multi-Tenant}{Low-Latency} storage. In 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18), pages 627 – 643, 2018.

[22] Fortanix. Fortanix EDP. https://edp.fortanix.com/.

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

[24] Huibo Wang, Pei Wang, Yu Ding, Mingshen Sun, Yiming Jing, Ran Duan, Long Li, Yulong Zhang, Tao Wei, and Zhiqiang Lin. Towards memory safe enclave programming with rust-sgx. In Proceedings of the 2019 ACM SIGSAC Conference on Computer and

Communications Security, CCS ’19, page 2333 – 2350, New York, NY, USA, 2019. Association for Computing Machinery.

[25] Francis X. McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In HASP ’13, 2013.

[26] Kripa Shanker, Arun Joseph, and Vinod Ganapathy. 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, pages 1077 – 1088, 2020.

[27] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: Securing the foundations of the rust programming language. Proc. ACM Program. Lang., 2(POPL), dec 2017.

[28] Intel. Intel MPX. https://intel-mpx.github.io/.

[29] Dmitrii Kuvaiskii, Gaurav Kumar, and Mona Vij. Computation offloading to hardware accelerators in intel sgx and gramine library os, 2022.

[30] Fortanix. Fortanix sgx abi. https://edp.fortanix.com/docs/api/fortanix_sgx_abi/.

[31] Frank Piessens Jo Van Bulck, Fritz Alder. A case for unified abi shielding in intel sgx runtimes. In A Case for Unified ABI Shielding in Intel SGX Runtimes, systex ’22, 2022.

[32] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth,

Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Still-well, David Goltzsche, Dave Eyers, R ¨udiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pages 689 - 703, Savannah, GA, November 2016. USENIX Association.

[33] Shweta Shinde, Dat Le, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. 01 2017.

[34] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, R¨udiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In 2017 USENIX Annual Technical Conference (USENIX ATC 17), pages 285 - 298, Santa Clara, CA, July 2017. USENIX Association.

[35] Aisha Hasan, Ryan Riley, and Dmitry Ponomarev. Port or shim? stress testing application performance on intel sgx. In 2020 IEEE International Symposium on Workload Characterization (IISWC), pages 123 - 133, 2020.

[36] 3GPP. 5G specification Release 15. https://www.3gpp.org/release-15.

[37] Chakrabarti, Somnath. Protecting Telecom Core with Intel SGX. https://www.intel.com/content/dam/www/public/us/en/documents/research/somnath-chakrabarti-sgx-day-telco-security.pdf.

[38] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. HASP@ ISCA,

11(10.1145):2487726 – 2488370, 2013.

[39] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu Han. A first step towards leveraging commodity trusted execution environments for network applications. In Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets), pages 1 – 7, 2015.

[40] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. S-NFV: Securing NFV states by using SGX. In Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, pages 45 – 48, 2016.

[41] Juhyeng Han, Seongmin Kim, Daeyang Cho, Byungkwon Choi, Jaehyeong Ha, and Dongsu Han. A secure middlebox framework for enabling visibility over multiple encryption protocols. IEEE/ACM Transactions on Networking, 28(6):2727 – 2740, 2020.

[42] NextEPC. Open5GS: Open source project of 5GC and EPC. https://open5gs.org/.

[43] ONF. OMEC: Open Mobile Evolved Core. https://opennetworking.org/omec/.

논문 개요

Rust 언어 기반 SGX 프레임워크의 비교분석: SGX 응용프로그램 구현을 위한 시사점

신희경
미래융합기술공학과
성신여자대학교 대학원

견고한 하드웨어를 기반으로 사용 중인 데이터에 대해 보호 기능을 제공하는 Intel SGX (Software Guard Extensions, SGX) 기술이 널리 채택되면서 개발자들의 주목을 받고 있다. 특히, 레거시 코드를 수정하지 않고 라이브러리 운영 체제 (LibOS)를 Enclave 내에 이식하는 방법을 통해 SGX 응용프로그램 개발의 복잡성을 줄일 수 있다. 그러나, LibOS 기반 SGX는 잠재적인 공격 표면을 증가시키며 메모리 손상 취약성에 노출될 수 있다. 이 한계를 극복하고자, 시스템 구성 요소를 안전하게 구현하여 메모리 안전성을 보장하는 Rust 프로그래밍 언어를 활용하는 추세이다. 이에 따라 Rust 기반 SGX 프레임워크를 활용한 응용프로그램 개발 방법론이 대두되었다. 하지만, Rust 기반 SGX 프레임워크를 선택하는 것에 있어 구현 비용 및 런타임 오버헤드와 같은 요소를 고려하는 개발자들을 위한 지침이나 체계적인 분석이 미비하다.
본 연구에서는 Rust로 구현된 세 가지 대표적인 SGX 프레임워크인 Rust SGX SDK, Occlum, Fortanix EDP에 대해 전반적인 성능 비교 분석을 수행한다. 이 분석은 성능 및 보안에 미치는 영향에 중점을 두어, 각 프레임워크의 내부 구현을 분석한다. 더욱이, Rust 프로그래밍 언어로 작성된 응용프로그램을 마이그레이션 하는 데 필요한 개발 비용을 양적으로 평가하며, 프레임워크를 CPU 및 메모리 집약적인 워크로드에 적용했을 때 발생하는 추가적인 오버헤드를 평가한다. 이 분석을 통해 개발자들에게 응용프로그램 및 워크로드 특성에 적합한 Rust 기반 SGX 프레임워크를 선택하도록 하는 지침을 제공하고자 한다.

ACKNOWLEDGEMENTS

본 논문을 지도해주신 김성민 교수님과 공저자로서 도움을 주신 노현, 옥지원 학생께 감사드립니다.