applied sciences

MDPI

Article

An Optimization Methodology for Adapting Legacy SGX Applications to Use Switchless Calls

Seongmin Kim

Department of Convergence Security Engineering, Sungshin Women's University, Seoul 02844, Korea; sm.kim@sungshin.ac.kr; Tel.: +82-2-920-7449

Abstract: A recent innovation in the trusted execution environment (TEE) technologies enables the delegation of privacy-preserving computation to the cloud system. In particular, Intel SGX, an extension of x86 instruction set architecture (ISA), accelerates this trend by offering hardware-protected isolation with near-native performance. However, SGX inherently suffers from performance degradation depending on the workload characteristics due to the hardware restriction and design decisions that primarily concern the security guarantee. The system-level optimizations on SGX runtime and kernel module have been proposed to resolve this, but they cannot effectively reflect application-specific characteristics that largely impact the performance of legacy SGX applications. This work presents an optimization strategy to achieve application-level optimization by utilizing asynchronous switchless calls to reduce enclave transition, one of the dominant overheads of using SGX. Based on the systematic analysis, our methodology examines the performance benefit for each enclave transition wrapper and selectively applies switchless calls without modifying the legacy codebases. The evaluation shows that our optimization strategy successfully improves the end-to-end performance of our showcasing application, an SGX-enabled network middlebox.

Keywords: intel SGX; enclave switches; benchmarking

Citation: Kim, S. An Optimization Methodology for Adapting Legacy SGX Applications to Use Switchless Calls. *Appl. Sci.* **2021**, *11*, 8379. https://doi.org/10.3390/app11188379

Academic Editors: Konstantinos Rantos, Konstantinos Demertzis and George Drosatos

Received: 16 August 2021
Accepted: 7 September 2021
Published: 9 September 2021

1. Introduction

Cloud computing allows enterprises to migrate their services from traditional physical servers to the virtualized environments. In particular, cloud computing has been widely adopted for video streaming [1], data analytics [2,3], and networking [4,5] to utilize powerful computation and storage resources. However, cloud-based services suffer from the lack of a mechanism to protect their software from the privileged software controlled by the cloud platform provider, which means that they have to trust the cloud platform not to leak, corrupt, or misuse their secrets. CPU vendors have released powerful hardware-based protection mechanisms called trusted execution environments (TEEs) [6–8] to address this. TEE ensures the integrity and confidentiality of the encrypted memory region and provides a trusted computing base (TCB) to launch a secure execution environment from the untrusted part of the system. These advantages overcome the limitations of previous trusted platform modules (TPMs), suffering from limited functionality that only supports cryptographic operations [6].

The recent innovation in cloud computing and trusted execution environment (TEE) technology introduces a new paradigm, confidential computing [9]. Confidential computing enables isolated execution of privacy-sensitive services in a hardware-protected secure container—a CPU enclave. In particular, Intel software guard extension (SGX) [10] accelerates the adoption of confidential computing, providing near to the native speed of a processor and compatibility with x86 architecture. Such advantages inspire researchers to leverage Intel SGX to various cloud-native applications for enhancing security and privacy [11–15]. However, even if SGX guarantees isolated execution of applications running on the cloud, it suffers from performance degradation depending on the workload characteristics.

The hardware restriction and design decisions in the software counterpart of SGX to preserve security guarantee introduces performance overhead [16–19]. Specifically, the state-of-the-art SGX technology only supports an encrypted memory space up to 128 MB [18], which incurs enclave paging overhead when the memory footprint of the target application is much larger than the physical memory available. In addition, invoking enclave entry and exit to handle several operations (e.g., system calls) in a secure manner leads to redundant CPU mode switches (e.g., secure mode vs. non-secure mode). Even a tiny improvement in throughput or latency dramatically impacts the business, especially for cloud-based services. In fact, a report from Amazon showed that only a second of additional delay while loading a Web page will cost $1.6 billion in sales each year [20]. Therefore, performance optimization to minimize the systematic overhead caused by applying a commoditized TEE is crucial for cloud-native applications.

Recent studies have focused on improving the performance of SGX applications based on system-level optimization [16,21,22], while application-level optimization is underexplored. However, system-level optimization might be insufficient; design and implementation choices made by the developer and application-specific characteristics (e.g., I/O intensive) would largely impact the performance of applications. Moreover, it requires significant engineering effort when conducting manual tuning to achieve application-level performance optimization, such as redesigning an application's execution flow or modifying its threading model [19].

This paper explores the optimization strategy to efficiently utilize SGX switchless calls, a technique updated in a recent version of Linux SGX software development kit (SDK) [23]. The switchless call operates enclave entries and exits asynchronously with worker threads to reduce enclave transition, similar to asynchronous I/O implementation in operating systems. However, the switchless call is essentially designed for asynchronous threading models, which makes applying it to legacy codes not always efficient. To this end, we propose an optimization strategy of leveraging switchless calls on legacy SGX-ported applications. Specifically, we develop a heuristic method to derive a metric, switchless efficiency, based on comprehensive analysis over switchless calls. To demonstrate the proposed strategy satisfies our design goal, we perform a case study to optimize a performance-critical network application, an SGX-enabled network middlebox [11]. The evaluation shows that our optimization successfully improves the performance of the SGX-enabled middlebox while a naive adoption of switchless calls degrades its performance.

The remainder of this paper is organized as follows. Related studies are reviewed in Sections 2 and 3 explains the technical background of our paper. In Section 4, we first describe the motivation, challenges, and the problem scope of our work. We then demonstrate how we build our optimization strategy. Section 5 presents our evaluation with micro-benchmarks and macro-benchmarks and Section 6 includes further discussion, respectively. Finally, we conclude this chapter in Section 7.

2. Related Work

2.1. Optimizing SGX Performance

Several studies have been reported to reduce the overhead of using SGX. In particular, they have focused on improving the SGX performance based on system-level optimization by modifying SGX runtime or kernel module [16,21,22,24]. However, the system-level optimization cannot precisely reflect the application's characteristics (e.g., threading model), so it does not always work perfectly for legacy SGX applications. Our proposed methodology achieves application-level optimization by measuring the efficiency of leveraging switchless calls affected by the application-specific characteristics without modifying the legacy codebase.

SCONE [24] suggests the container-based SGX framework that enables the execution of legacy applications without modification by providing its library. Based on user-level threading and asynchronous system call mechanism, SCONE avoids thread blocks due to synchronization.

Tian et al. [22] design SGXKernel that leverages library OS to execute unmodified binaries to make it more practical, similar to the SCONE framework. The proposed system includes asynchronous communication primitives between enclaves, called delegated calls. With delegated calls, an SGX enclave does not trigger any enclave switches when communicating with other enclaves.

Similar to switchless calls that we further explain in Section 3.2, Eleos [21] propose exitless paging mechanism and exitless system calls to reduce both main factors of SGX overhead, EPC paging, and enclave transition. In particular, they implement an exitless system call service routine in a Remote Procedure Call (RPC) fashion, and the RPC-based implementation can be transparently integrated with a vanilla SGX SDK.

Weisse et al. [16] explore the performance implications of running real-world applications on SGX hardware to achieve performance optimization. Based on the preliminary evaluations with various micro-benchmarks, they build a new application-enclave interaction framework called HotCalls. The core idea of HotCalls is utilizing shared variables located in the untrusted part of the SGX application and spin-lock synchronously. The evaluation shows that HotCalls dramatically improves both latency and throughput of popular macro-benchmarks, such as memcached, openVPN, and lighthttpd.

Meanwhile, Aublin et al. [17] extend existing transport layer security (TLS) libraries to protect sensitive data, such as session contexts and private keys. While our methodology optimizes the performance of legacy codebases, it ports and redesigns an existing TLS library to reduce enclave transition overhead. To make it become SGX-friendly, they modify the implementation of enclave entry-calls and out-calls. For this, it uses an in-enclave memory pool and thread locks implementation to handle operations that raise enclave transitions internally within an enclave.

2.2. SGX Performance Profiling

Similar to our proposal that conducts comprehensive performance profiling to identify metrics and values associated with the performance of SGX-ported applications, there also have been attempts to benchmark SGX applications and the SGX runtime [18,25,26]. Shanker et al. [25] quantify engineering effort to port legacy code of real-world applications to SGX and compare the performance of ported applications running atop various SGX frameworks [14,15]. Dinh et al. [18] figure out the actual available EPC memory space for an SGX application is around 93MB in practice due to the reserved memory for metadata. Weichbrodt et al. [26] provide a set of toolchains to precisely identify the critical factors related to SGX during the performance analysis of SGX-ported workloads. The above studies have focused on analyzing the intrinsic performance penalty of using SGX or the cost of manual tuning for improving performance, while our approach enables performance optimization with minimal engineering effort.

3. Background

3.1. Intel SGX Overview

The goal of Intel SGX technology is to provide a trusted execution environment (TEE) for cloud-based systems. For SGX functionalities, Intel introduces new 24 instructions (18 for revision 1 and 6 for revision 2, respectively) by extending Intel x86 Instruction Set Architecture (ISA). The hardware support for SGX becomes available starting from Intel Skylake CPUs. The purpose of SGX technology is to provide a secure container, called an *enclave*, for cloud-native applications. An SGX enclave is protected against access from privileged software (e.g., OS and hypervisor) to leak sensitive data or manipulate control flow, which can be potentially malicious due to the untrustworthy nature of the cloud environment [27–29]. Essentially, service providers who run their applications in the cloud platform cannot access or control hardware components and the underlying privileged software components. Moreover, they cannot even be unaware of whether data leakage incidents [30] from the cloud happens unless a cloud platform provider notifies the leakage. This means that service providers have to trust the cloud platform provider not to be

malicious or curious [31]. Accordingly, SGX's threat model assumes a powerful adversary who can control hardware components except for CPU package (e.g., Memory and I/O bus) and privileged software.

SGX memory protection: The trustworthy component of SGX consists of an SGX processor and an enclave. An enclave is part of the virtual address space of an SGX application, and it is a hardware-protected region mapped to a reserved physical memory specialized for SGX memory protection, Enclave Page Cache (EPC). While launching an SGX application, an `EADD` instruction copies each page to be protected into an enclave region with the measurement using `EEXTEND` instructions. Once an enclave region is finalized with `EINIT`, the memory content in the EPC region is encrypted with Memory Encryption Engine (MEE) in the processor with a hardware-specific key.

To execute an enclave code, the processor calls an `EENTER` instruction to enter a new CPU mode, called *enclave mode*. The plain-text of memory content stored in the region can only be accessed when the processor switches to an enclave mode with an `EENTER`. When the processor successfully passes the hardware checks (e.g., verifying the integrity of the enclave measurement), MEE decrypts the content stored in the EPC region for executing enclave code or accessing data. Again, the content is re-encrypted when leaving the CPU package (switches to a normal mode with `EEXIT`) and stored in the EPC region. This hardware-based protection occurs on every single memory access when entering/leaving an enclave, such that memory access to an enclave region from privileged software or other enclaves is prohibited.

Intel SGX SDK: To help developers implement an SGX program, Intel released SGX SDK for both Windows and Linux environments. The SDK consists of trustworthy libraries (e.g., cryptographic and standard libraries) that are securely ported for SGX functionalities, toolchains, and sample projects. Note that toolchains include signature tools for signing an enclave with the developer's key and debugging tools. The procedure of implementing an SGX program using SGX SDK is as follows. First, it requires specifying an enclave configuration, such as the enclave heap and stack size, as an XML file. Then, a developer generates an asymmetric key pair used for signing an enclave. Note that `EINIT` verifies the signature before an enclave launch.

While implementing an SGX program with Intel SGX SDK, the program is divided into an app region and an enclave region, where the app part resides in the untrusted region. This means that a developer needs to separate privacy-sensitive operations to be protected and put them into an enclave region when porting legacy code to SGX. After the design choices, a developer writes the source code corresponding to the app and the enclave, compiles it to acquire an SGX-compatible binary.

3.2. Enclave Transition Interface

ECALL/OCALL interface: Intel SGX SDK provides secure ECALL (enclave call) and OCALL (outside call) interfaces to enter and leave an enclave region, respectively.

ECALL is a wrapper routine for `EENTER` that transfers control flow into the enclave code. In contrast, OCALL is a wrapper code that invokes `EEXIT` instruction to leave an enclave. In particular, the OCALL interface is frequently used for handling system calls, which can not be executed in the enclave mode. Because an enclave mode runs in an unprivileged user-level mode(e.g., ring 3), it cannot operate instructions that require kernel privilege. For this, an enclave should rely on the untrusted code and switch back to the enclave via `EEXIT` and `ERESUME` instructions.

Both ECALLs and OCALLs can be defined by an SGX developer in the Enclave Definition Language (EDL) file, an SGX-specific syntax provided by SGX SDK. Note that when parameters of ECALLs and OCALLs contain a pointer variable, the corresponding size of the pointer and their attributes (e.g., whether it is input argument or output argument) should be necessarily specified together in the EDL for sanity checking against the parameter marshaling. During the compilation, an edger8r tool of the SGX SDK parses the

EDL file and automatically generates a glue code for ECALL and OCALL declared by an SGX developer.

Switchless calls: A recent patch in Linux SGX SDK contains switchless calls [32], which reduces an enclave mode switch overhead during the enclave transition. The goal of switchless calls is to eliminate enclave switches from SGX applications by making ECALL and OCALL themselves switchless, which are functions used for entering/leaving SGX enclaves. For this, an SGX run-time library executes two worker threads, one in the application (untrusted) memory region and the other in the enclave (trusted) memory region. The application worker thread handles ECALL, while the enclave worker thread handles OCALL, respectively. There are two thread pools to handle switchless ECALL and OCALL, and worker threads are executed asynchronously. For asynchronous execution, switchless calls utilize two shared queues: a request queue and a response queue.

Figure 1 illustrates the workflow of a switchless OCALL. The implementation of switchless calls adopts a sleep-wake approach for efficiency. When a caller thread inside an enclave invokes an OCALL, it first updates the request queue. Then, one of the worker threads from the thread pool in the untrusted region is assigned and handles the OCALL. Finally, the worker thread updates the response queue. Note that the current version of Linux SDK reflects switchless SGX implementation for common operations inside an enclave, such as threading, file I/O, and system clock, to eliminate OCALLs [23].

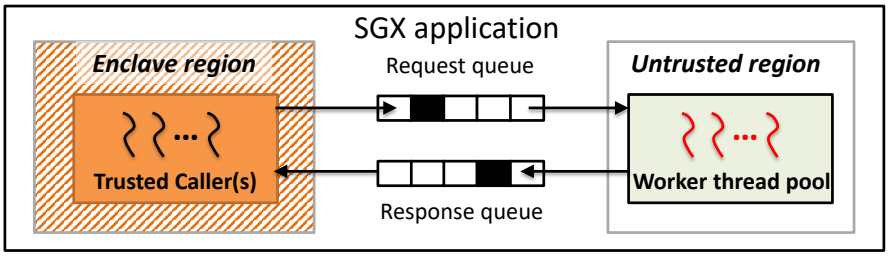

Figure 1. Work flow of switchless OCALL.

SGX developers can quickly adopt switchless SGX by linking libraries and specifying which OCALLs and ECALLs to apply switchless SGX. Specifically, the source code for switchless SGX is located in the '/sdk/switchless' directory, and SGX developers can get two libraries after the SDK compilation, which are libsgx_tswitchless.a and libsgx_uswitchless.so. The libsgx_tswitchless.a is used for the trusted region and the libsgx_uswitchless.so for the untrusted region, respectively. When building an SGX binary, developers should link the two libraries to the compiled object files. Finally, developers specify which OCALLs and ECALLs use switchless SGX by adding the 'transition_using_threads' configuration option in the EDL file. According to the Linux SGX SDK, switchless SGX currently does not work for simulation mode and 32-bit CPUs, and the requirement for utilizing switchless SGX is an installation of Linux SGX SDK version 2.2.

4. Approach

This section describes the problem scope and key challenges of utilizing switchless calls to existing SGX-ported applications. Then, we present our key insight and propose an optimization strategy to optimize the performance of an existing SGX application by selectively adopting switchless calls depending on ECALL/OCALL characteristics.

4.1. Motivation, Challenges, and Problem Scope

Even if SGX guarantees isolated but secure execution of applications running on the cloud, it suffers from performance degradation depending on application characteristics. The main performance overhead of utilizing SGX can be categorized into EPC paging and enclave transition. First, when an SGX enclave program demands a large enclave heap space, secure paging occurs to reclaim EPC pages as SGX only provides 128 MB of the EPC region due to the hardware restriction [18,33] . During the secure paging, an evicted

EPC page should be encrypted before copying it to regular main memory and vice versa, leading to an SGX application's slowdown. In addition, an enclave transition—or called enclave mode switch, incurs significant overhead when it frequently happens. The enclave mode switch is expensive because it is accompanied by (1) saving and restoring the CPU state and registers and (2) TLB flush.

Among the above two obstacles that hinder the performance of SGX applications, we mainly focus on addressing enclave transition overhead for performance optimization. SGX applications that heavily depend on ECALL/OCALL may suffer from performance degradation caused by enclave transition. Batching system calls and I/O operations to amortize the context switch cost is not a perfect solution, as it cannot reduce the inherent enclave transition overhead. We believe switchless calls can be an alternative solution to address this problem.

The switchless call aims to enhance the performance of SGX applications that use the asynchronous threading model. However, building a proper optimization strategy for adopting switchless calls to legacy applications that are implemented synchronously is not trivial. Due to the lack of metrics to evaluate the switchless efficiency, SGX developers cannot explicitly determine which ECALLs and OCALLs achieve the gain of adopting switchless calls. In other words, a naive adoption of switchless SGX might be ineffective to synchronous threading model, because it introduces pending application threads due to the scheduling of switchless worker threads. Furthermore, modifying the internal structure of software or conducting manual tuning to make it switchless-friendly requires lots of sunk costs in terms of engineering effort.

Based on a comprehensive analysis of the SGX switchless call, we establish an optimization strategy and apply it to our prior work called SGX-Box [11], an SGX-enabled network middlebox. Recent studies [11,12,34] leverage SGX to protect network middleboxes running on the cloud-based network function virtualization (NFV) architecture. In particular, SGX protects deep packet inspection and a ruleset used for pattern matching in the security-purpose in-network functions, such as Web firewalls and intrusion detection systems (IDS). Note that SGX-Box processes each packet by a run-to-completion model on a per-flow basis to achieve high performance, which makes threads of SGX-Box running more synchronously. Moreover, network middleboxes, including SGX-Box, frequently incur enclave mode switch to handle network I/O operations. In summary, our goal is to establish an optimization strategy to improve the performance of SGX-Box by leveraging switchless calls.

4.2. Building an Optimization Strategy

Our key insight to address this problem is that an application developer can selectively determine whether utilize switchless calls for ECALL and OCALL. Rather than naively adopting switchless calls to entire ECALLs and OCALLs, our approach selectively applies switchless calls by examining the performance benefit for each ECALL and OCALL. To achieve this, we explore factors that affect the efficiency of switchless SGX when adopting it to legacy SGX applications. For simplicity, we assume that a caller thread and a worker thread are allocated per core. A switchless call is efficient for an ECALL or OCALL when it satisfies the below condition. For a unit of synchronous execution, the gain of using switchless call is defined as:

$$\begin{aligned} \text{Gain of using switchless} &= \frac{\text{The CPU time saved during ECALL or OCALL}}{\text{The pended CPU time due to worker threads}} \\ &= \frac{N_{et} * T_{et}}{\text{The pended CPU time due to worker threads}} \\ &> 1 \end{aligned} \tag{1}$$

where N_{et} is the number of enclave transitions occurred and T_{et} is an enclave transition time.

However, calculating the pended CPU time due to worker threads is not trivial. Therefore, we examine which factors of ECALL and OCALL affect the pended CPU time.

Based on the analysis, we learn that two major factors are related to pending caller threads: a frequency of ECALL/OCALL and a completion time of ECALL/OCALL.

Frequency of ECALL/OCALL: The pended CPU time raised by the worker thread is affected how ECALL/OCALL is frequently called. For convenience, we will explain the case of ECALL. We first take a look at the implementation of switchless calls to understand the execution flow. The execution flow of switchless worker threads is divided into three steps:

1. If there are sleeping workers, an SGX application wakes them up when an ECALL is invoked.
2. Once a worker thread is scheduled, it looks up the request queue and processes an enclave function.
3. If the request queue is empty, it retries until max_retries (default = 20,000) and falls asleep.

As the Figure 2 shows, the workflow of worker threads depends on the emptiness of the request queue. If an ECALL is rarely invoked, the corresponding worker thread accesses the request queue until max_retries (Step 3) and waking up worker threads (Step 1) frequently happens, which wastes the CPU time. Therefore, pended CPU time due to worker threads is proportional to the frequency of enclave transition.

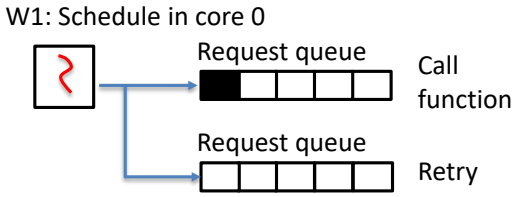

Figure 2. Execution flow of switchless worker threads depending on the queue's emptiness.

Completion time: The other factor that affects the pending time is the completion time of ECALL/OCALL. We bring the execution flow of the SGX-Box thread with switchless calls as an example for better understanding. Figure 3 demonstrates workflows of the SGX-Box caller thread and the corresponding ECALL conducted by the switchless worker thread. When an ECALL (`ecall_start_tls_process` in Figure 3) is invoked by an SGX-Box thread, it updates the request queue. Then, worker threads execute the corresponding ECALL function once the worker is scheduled. While executing the ECALL, context switches might occur if the completion time of the ECALL is long enough to make the worker thread scheduling out. It leads to pending of the SGX-Box caller thread because it works synchronously, which means that it cannot perform the remaining tasks until the worker thread completes the execution of the ECALL. Therefore, a completion time of ECALL/OCALL affects the number of context switching, which also involves the pending time due to the switchless worker thread.

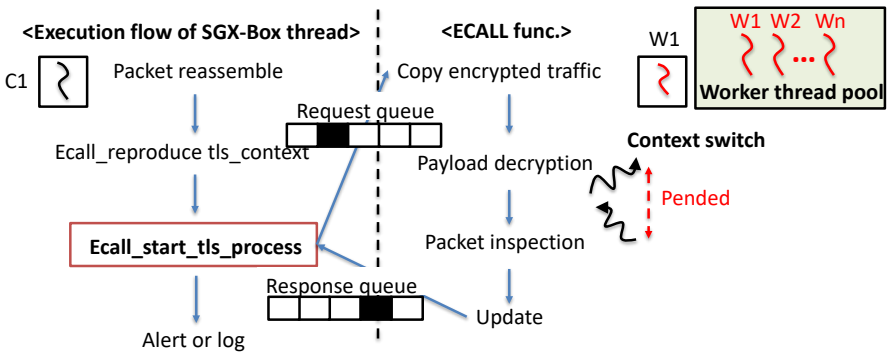

Figure 3. An example of Switchless ECALL in SGX-Box application. Context switches while executing ECALL function introduce pending of the SGX-Box caller thread.

In summary, we learned that

$$\text{Pended CPU time due to worker threads} \propto T_w, \quad \propto \frac{1}{N_{et}} \tag{2}$$

where T_w is a completion time of ECALL/OCALL.

Based on analysis, We formulate the gain of using switchless as:

$$\begin{aligned} \text{Gain of using switchless} &= \frac{\text{CPU time saved during ECALL or OCALL}}{\text{Pended CPU time due to worker threads}} \\ &= \frac{N_{et} * T_{et}}{\gamma * (\alpha / N_{et}) + (1-\gamma) * \beta * T_w} \\ &= \frac{{N_{et}}^2 * T_{et}}{\gamma * \alpha + (1-\gamma) * \beta * T_w * N_{et}} \end{aligned} \tag{3}$$

where α and β are constant, and γ is a ratio that worker faces an empty request queue.

If the above value is larger than one, we regard an ECALL or OCALL as switchless-friendly. With the above formula, we define a metric to decide a switchless efficiency and empirically determine the threshold as follows:

$$\text{A Switchless call is efficient when} \quad N_{et} * T_{et} / T_w > 0.09. \tag{4}$$

4.3. Determining Adaptiveness of Switchless Calls

Based on our metric to determine switchless efficiency, we perform profiling to establish a proper optimization strategy on our target application, SGX-Box. Note that we adopt switchless calls to ECALLs used by SGX-Box only due to its internal implementation. SGX-Box threads rarely call OCALLs because the mOS framework [35] part in the untrusted region handles the packet I/O and packet reassembly procedures by design. Such design makes an enclave of SGX-Box threads not invoking system calls (and OCALLs) to handle I/O operations (e.g., read and write). Instead, we measure the number of enclave transitions and the completion time of seven ECALLs used by SGX-Box. Then, we calculate the switchless efficiency and compare it with the threshold in (4). For estimation, we use Quad-core Intel i7-6700 3.4 GHz CPU machines with Linux 3.19.0 and Linux SGX SDK version 2.2 that supports switchless call functionality [23]. We use four worker threads for ECALLs and enable hyperthreading. Finally, we configure max_retries as 1000, and the enclave transition latency is 2.67 µs in our environment.

Based on our profiling, we adaptively utilize a switchless call for the ECALL and make other ECALLs use the traditional ECALL interface for optimization. Figure 4 shows the calculated switchless efficiency of ECALLs defined in SGX-Box. As the figure shows, a single ECALL, `ecall_reproduce_ssl_context`, satisfies the condition among seven ECALLs under our evaluation environment. Note that `ecall_reproduce_ssl_context` reproduces SSL context within an enclave when a packet arrives in the untrusted region. Therefore, it is less effective when applying switchless calls to other ECALLs, except for `ecall_reproduce_ssl_context`.

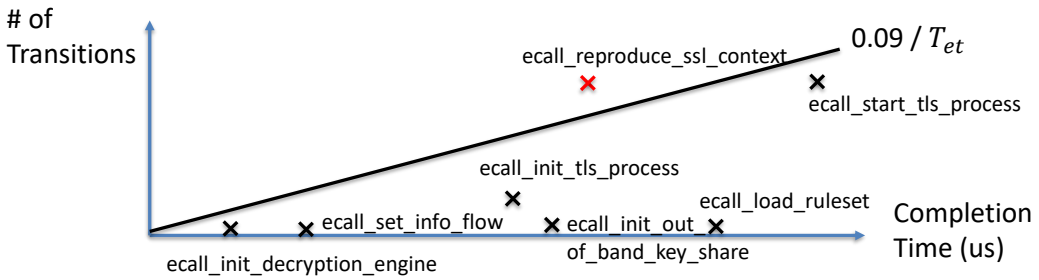

Figure 4. Switchless efficiency of each ECALL used by SGX-Box.

5. Evaluation

To validate our optimization on our target application, we evaluate the performance of SGX-enabled network middleboxes with micro-benchmarks and end-to-end performance analysis. We use the same machine specification used for determining the adaptiveness of switchless calls in Section 4.3. Also, we use 10 Gbps link in a lab environment for connecting servers, clients, and an SGX-enabled middlebox to avoid network bandwidth becoming a bottleneck. Finally, we use TLS v1.2 encryption protocol and select AES256-GCM-SHA384 as a cipher suite for packet encryption and decryption.

5.1. Micro-Bench Evaluation

Breakdown of CPU time: For the ECALL that satisfies the switchless efficiency condition, we measure the breakdown of CPU time to compare the traditional ECALL interface and switchless ECALL. For evaluation, we set clients to send 32 long-running flows, and each flow sends 1 KB-sized random packets. Then, we measure the elapsed time of `ecall_reproduce_ssl_context` for a single packet, starting from an SGX-Box thread creation. Note that the elapsed time includes the packet I/O and payload reassembly procedures. Figure 5 shows the result. For the case of a traditional ECALL, the elapsed time consists of time consumed in an untrusted region, time consumed in an enclave region, and an enclave transition, while switchless ECALL does not include the time consumed for enclave transition, respectively. Our result shows that adopting a switchless call to `ecall_reproduce_ssl_context` delivers 10% reduced elapsed time compared to traditional ECALL. This improvement comes from the result that pended time by worker threads (2.6 µs) is smaller than the total enclave transition latency (5.3 µs).

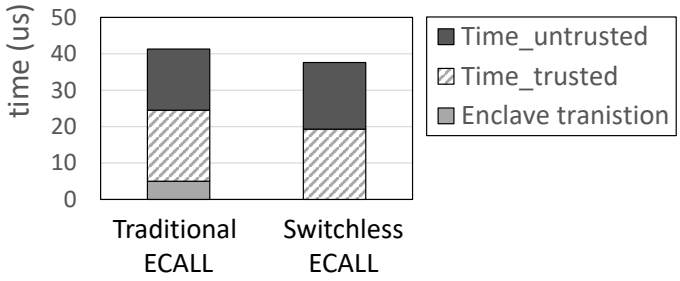

Figure 5. The comparison of CPU time breakdown between an ECALL and a switchless ECALL.

TLS Decryption Throughput: We measure the TLS decryption throughput by increasing the TLS record size from 256 Bytes from 4 KB. As the Figure 6 shows, a naive adoption of switchless call to every ECALL used by SGX-Box degrades the performance by 33% on average. This result supports our claim that it requires an appropriate adoption of switchless calls for SGX applications, which execute synchronously (e.g., run-to-completion model). In contrast, SGX-Box that utilizes switchless SGX based on our strategy gives 5% better throughput than the original SGX-Box with 256 Byte-sized TLS records, while it delivers 2.7% of improvement for 4 KB record size, respectively. The benefit of adopting switchless calls is reduced for larger record sizes because an execution time for record decryption dominates the time consumed for enclave transition.

5.2. End-to-End Performance Evaluation

We evaluate the end-to-end performance of SGX-Box, including TLS decryption and pattern matching, when applying switchless calls by varying the number of patterns in the ruleset. We use the DFC [36] string pattern matching algorithm and a custom regular expression matching engine with a commercial ET-Pro ruleset [37] for pattern matching. We measure the throughput for 32 long-running flows where each flow sends 1 KB-sized random packets. Figure 7 shows the end-to-end throughput of (1) using traditional ECALL, (2) naively adopting switchless calls for every ECALL, and (3) adopting switchless calls based on our optimization strategy. As the result shows, naive adoption of switchless

calls significantly degrades the performance of SGX-Box by 80% on average compared to utilizing the traditional ECALL interface. In contrast, our optimization delivers 1% improved performance on average compared to the non-switchless version of SGX-Box. Note that there is no enclave transition during the packet inspection, including exact string matching and regular expression matching, and most of the CPU time is consumed by packet inspection. In other words, it reduces the proportion of overhead that comes from enclave transition, which makes switchless optimization less effective. Therefore, the end-to-end throughput is saturated with the throughput of packet inspection. However, for the case of naive adoption, redundant scheduling of worker threads hinders SGX-Box threads from proceeding with packet inspection, which leads to the pending of the SGX-Box thread.

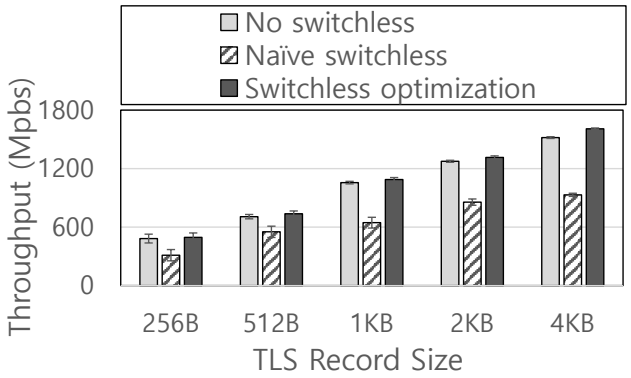

Figure 6. Decryption throughput while varying the TLS record size.

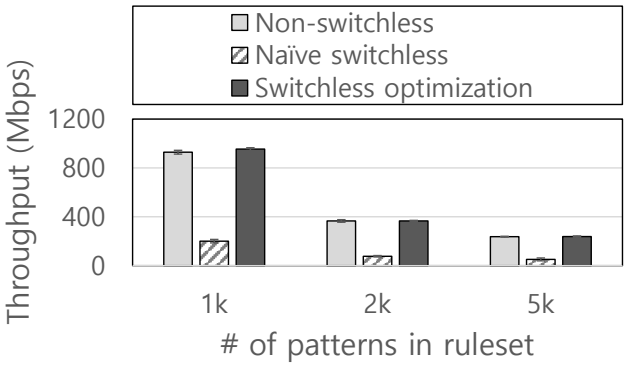

Figure 7. End-to-end pattern matching throughput while varying the number of ruleset patterns.

6. Discussion

In this section, We further investigate potential factors that affect switchless efficiency. Inspired by the performance-critical elements in the traditional multi-core processing architectures [38–42], we explore the impact of thread-to-core affinity and scheduling policy on the performance of our proposed methodology.

Core Affinity: One factor affecting the switchless efficiency is the proper core affinity methodology [38,39] when switchless worker threads and application threads run simultaneously. Note that a race condition between the threads might occur in the multi-core environment if the total number of threads of an SGX application exceeds the number of available cores. Let us assume that we configure two threads as switchless worker threads, and an SGX application generates two threads containing an OCALL function, where the SGX CPU has two available cores. In this scenario, there are two possible options to set core pinning: (1) pins switchless worker threads together to a single core and pins the other for application threads—we call it a grouping strategy; and (2) pairs a switchless worker thread and an application thread and assign them for each core—we call it a pairing strategy.

To evaluate which method benefits the performance of switchless SGX, we first measure the time to operate a million empty ECALLs and OCALLs and calculates the context

switch latency of single switchless ECALL and OCALL. We utilize two CPU cores during the estimation and creates four threads in total, two for ECALL or OCALL worker threads and two for application threads. While evaluating ECALL transition latency, we disabled the usage of OCALL worker threads and vice versa. As the Table 1 shows, the grouping strategy delivers lower enclave transition latency for both empty ECALL and OCALL functions, compared to the pairing strategy.

Table 1. Comparison between two core affinity strategies for switchless ECALL and OCALL.

Operation Type	Methodology	ECALL	OCALL
Empty Call	Assigns CPU core for each thread group	0.902 µs (13.8%↑)	0.602 µs (27.7%↑)
	Pairs worker and application thread	1.05 µs	0.833 µs
OCALL with I/O (1 KB Read)	Assigns CPU core for each thread group	-	8.90 µs
	Pairs worker and application thread	-	4.91 µs (44.9%↑)

We also perform the same estimation with an OCALL that contains file I/O operations. The OCALL function reads 1 KB of data using `read()` system call. In contrast to an empty OCALL, the evaluation result shows that the pairing strategy delivers lower latency. For the case of the grouping strategy, worker threads are pended until the I/O operation is finished, which leads to the overuse of the CPU core. Therefore, separating worker threads into different CPU cores is a better option for OCALLs with a long completion time (e.g., handling I/O). In summary, we learn that pairing strategy is better for network applications when the ECALL/OCALL takes a long time, while grouping strategy is appropriate for short-term ECALLs/OCALLs, respectively.

Saving CPU time consumed by workers: As we explained in the technical background (Section 3.2) and workflow of switchless calls (Section 4.2), worker threads retry until max_retries, set 20,000 as default, and fall asleep if the request queue is empty. It might lead to a waste of CPU time when an SGX application is implemented synchronously. We believe that it is possible to save the wasted CPU time by leveraging dependency-aware scheduling [40–42]. It enables scheduling other tasks that can be independently pre-executed, regardless of the completion of ECALL or OCALL. For example, when a worker thread is scheduled and occupies the CPU core, it executes the corresponding ECALL function if caller threads fill the request queue. Otherwise, it pre-executes other enclave functions.

7. Conclusions and Further Work

This paper proposes an application-level optimization methodology by adaptively leveraging switchless calls to reduce SGX overhead. Based on a systematic analysis, we define a metric to measure the efficiency of leveraging switchless calls for each wrapper function that raises enclave transitions. Compared with the previous optimization schemes, our approach reflects the characteristics of legacy SGX applications without introducing a significant engineering effort. Our evaluation shows that the adoption of our optimization methodology improves the pattern matching throughput of SGX-enabled middleboxes, one of the performance-critical cloud applications, while a naive adoption dramatically degrades the performance.

Our scheme uses a heuristic to estimate the efficiency of utilizing switchless calls based on a systematic study. To prove its validity or improve optimization efficiency, leveraging machine learning techniques would be effective to precisely infer the threshold of efficiency. The data acquisition for applying such schemes and accurate performance profiling for cloud-native applications are also important and challenging issues to be addressed. In future work, we will elaborate our methodology to find optimal or near-optimal parameters with machine learning techniques based on the practical implications that we have identified from this study.

Funding: This work was supported by the Sungshin Women's University Research Grant of H20200128.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Wang, X.; Kwon, T.; Choi, Y.; Wang, H.; Liu, J. Cloud-assisted adaptive video streaming and social-aware video prefetching for mobile users. *IEEE Wirel. Commun.* **2013**, *20*, 72–79. [CrossRef]
2. Simmhan, Y.; Aman, S.; Kumbhare, A.; Liu, R.; Stevens, S.; Zhou, Q.; Prasanna, V. Cloud-based software platform for big data analytics in smart grids. *Comput. Sci. Eng.* **2013**, *15*, 38–47. [CrossRef]
3. Talia, D. Clouds for scalable big data analytics. *Computer* **2013**, *46*, 98–101. [CrossRef]
4. Barona Lopez, L.I.; Valdivieso Caraguay, Á.L.; Sotelo Monge, M.A.; García Villalba, L.J. Key technologies in the context of future networks: Operational and management requirements. *Future Internet* **2017**, *9*, 1. [CrossRef]
5. Lv, Z.; Xiu, W. Interaction of edge-cloud computing based on SDN and NFV for next generation IoT. *IEEE Internet Things J.* **2019**, *7*, 5706–5712. [CrossRef]
6. Valadares, D.C.G.; Will, N.C.; Caminha, J.; Perkusich, M.B.; Perkusich, A.; Gorgônio, K.C. Systematic Literature Review on the Use of Trusted Execution Environments to Protect Cloud/Fog-based Internet of Things Applications. *IEEE Access* **2021**, *9*, 80953–80969. [CrossRef]
7. Dai, W.; Jin, H.; Zou, D.; Xu, S.; Zheng, W.; Shi, L.; Yang, L.T. TEE: A virtual DRTM based execution environment for secure cloud-end computing. *Future Gener. Comput. Syst.* **2015**, *49*, 47–57. [CrossRef]
8. Sun, H.; Lei, H. A design and verification methodology for a trustzone trusted execution environment. *IEEE Access* **2020**, *8*, 33870–33883. [CrossRef]
9. Confidential Computing Consortium. Available online: https://confidentialcomputing.io/ (accessed on 15 August 2021).
10. Hoekstra, M.; Lal, R.; Pappachan, P.; Phegade, V.; Del Cuvillo, J. Using innovative instructions to create trustworthy software solutions. *HASP@ ISCA* **2013**, *11*, 2487726–2488370.
11. Han, J.; Kim, S.; Ha, J.; Han, D. SGX-Box: Enabling Visibility on Encrypted Traffic using a Secure Middlebox Module. In Proceedings of the First Asia-Pacific Workshop on Networking, Hong Kong, China, 3–4 August 2017; pp. 99–105.
12. Wang, J.; Yu, Y.; Li, Y.; Fan, C.; Hao, S. Design and Implementation of Virtual Security Function Based on Multiple Enclaves. *Future Internet* **2021**, *13*, 12. [CrossRef]
13. Yoon, H.; Moon, S.; Kim, Y.; Hahn, C.; Lee, W.; Hur, J. SPEKS: Forward Private SGX-Based Public Key Encryption with Keyword Search. *Appl. Sci.* **2020**, *10*, 7842. [CrossRef]
14. Tsai, C.C.; Porter, D.E.; Vij, M. Graphene-sgx: A practical library OS for unmodified applications on SGX. In Proceedings of the 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17), Santa Clara, CA, USA, 12–14 July 2017; pp. 645–658.
15. Shinde, S.; Le Tien, D.; Tople, S.; Saxena, P. Panoply: Low-TCB Linux Applications with SGX Enclaves. In Proceedings of the NDSS, San Diego, CA, USA, 26 February–1 March 2017.
16. Weisse, O.; Bertacco, V.; Austin, T. Regaining lost cycles with HotCalls: A fast interface for SGX secure enclaves. *ACM Sigarch Comput. Archit. News* **2017**, *45*, 81–93. [CrossRef]
17. Aublin, P.L.; Kelbert, F.; O'keeffe, D.; Muthukumaran, D.; Priebe, C.; Lind, J.; Krahn, R.; Fetzer, C.; Eyers, D.; Pietzuch, P. TaLoS: Secure and transparent TLS termination inside SGX enclaves. *Imp. Coll. Lond. Tech. Rep.* **2017**, *5* , 1–4. [CrossRef]
18. Dinh Ngoc, T.; Bui, B.; Bitchebe, S.; Tchana, A.; Schiavoni, V.; Felber, P.; Hagimont, D. Everything you should know about Intel SGX performance on virtualized systems. In Proceedings of the ACM on Measurement and Analysis of Computing Systems, Phoenix, AZ, USA, 24–28 June 2019 ; Volume 3, pp. 1–21.
19. Mazzeo, G.; Arnautov, S.; Fetzer, C.; Romano, L. SGXTuner: Performance Enhancement of Intel SGX Applications via Stochastic Optimization. *IEEE Trans. Depend. Secur. Comput.* **2021**. [CrossRef]
20. How One Second Could Cost Amazon $1.6 Billion In Sales. Available online: http://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales (accessed on 15 August 2021).
21. Orenbach, M.; Lifshits, P.; Minkin, M.; Silberstein, M. Eleos: ExitLess OS services for SGX enclaves. In Proceedings of the Twelfth European Conference on Computer Systems, Belgrade, Serbia, 23–26 April 2017; pp. 238–253.
22. Tian, H.; Zhang, Y.; Xing, C.; Yan, S. Sgxkernel: A library operating system optimized for intel SGX. In Proceedings of the Computing Frontiers Conference, Siena, Italy, 15–17 May 2017; pp. 35–44.
23. Intel Software Guard Extensions (Intel SGX) SDK. Available online: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions/sdk.html (accessed on 15 August 2021).
24. 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 Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), Savannah, GA, USA, 2–4 November 2016; pp. 689–703.

25. 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, Virtual Event, USA, 8–13 November 2020; pp. 1077–1088.
26. Weichbrodt, N.; Aublin, P.L.; Kapitza, R. sgx-perf: A performance analysis tool for intel sgx enclaves. In Proceedings of the 19th International Middleware Conference, Rennes, France, 10–14 December 2018; pp. 201–213.
27. Baumann, A.; Peinado, M.; Hunt, G. Shielding applications from an untrusted cloud with haven. *ACM Trans. Comput. Syst. TOCS* **2015**, *33*, 1–26. [CrossRef]
28. Coppolino, L.; D'Antonio, S.; Formicola, V.; Mazzeo, G.; Romano, L. VISE: Combining Intel SGX and Homomorphic Encryption for Cloud Industrial Control Systems. *IEEE Trans. Comput.* **2020**, *70*, 711–724. [CrossRef]
29. Sun, H.; He, R.; Zhang, Y.; Wang, R.; Ip, W.H.; Yung, K.L. eTPM: A trusted cloud platform enclave TPM scheme based on intel SGX technology. *Sensors* **2018**, *18*, 3807. [CrossRef]
30. Jiang, J.; Han, G.; Shu, L.; Chan, S.; Wang, K. A trust model based on cloud theory in underwater acoustic sensor networks. *IEEE Trans. Ind. Inform.* **2015**, *13*, 342–350. [CrossRef]
31. Ning, J.; Huang, X.; Susilo, W.; Liang, K.; Liu, X.; Zhang, Y. Dual access control for cloud-based data storage and sharing. *IEEE Trans. Depend. Secur. Comput.* **2020**. [CrossRef]
32. Tian, H.; Zhang, Q.; Yan, S.; Rudnitsky, A.; Shacham, L.; Yariv, R.; Milshten, N. Switchless Calls Made Practical in Intel SGX. In Proceedings of the 3rd Workshop on System Software for Trusted Execution, Toronto, ON, Canada, 15 October 2018; pp. 22–27.
33. Kim, S.; Han, J.; Ha, J.; Kim, T.; Han, D. Sgx-tor: A secure and practical tor anonymity network with sgx enclaves. *IEEE/ACM Trans. Netw.* **2018**, *26*, 2174–2187. [CrossRef]
34. Han, J.; Kim, S.; Cho, D.; Choi, B.; Ha, J.; Han, D. A secure middlebox framework for enabling visibility over multiple encryption protocols. *IEEE/ACM Trans. Netw.* **2020**, *28*, 2727–2740. [CrossRef]
35. Jamshed, M.A.; Moon, Y.; Kim, D.; Han, D.; Park, K. mos: A reusable networking stack for flow monitoring middleboxes. In Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), Boston, MA, USA, 27–29 March 2017; pp. 113–129.
36. Choi, B.; Chae, J.; Jamshed, M.; Park, K.; Han, D. DFC: Accelerating String Pattern Matching for Network Applications. In Proceedings of the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16), Santa Clara, CA, USA, 16–18 March 2016; pp. 551–565.
37. ET Pro Ruleset. Available online: https://www.proofpoint.com/us/threat-insight/et-pro-ruleset (accessed on 15 August 2021).
38. Paznikov, A.; Shichkina, Y. Algorithms for optimization of processor and memory affinity for Remote Core Locking synchronization in multithreaded applications. *Information* **2018**, *9*, 21. [CrossRef]
39. Abbasi, S.I.; Kamal, S.; Gochoo, M.; Jalal, A.; Kim, K. Affinity-Based Task Scheduling on Heterogeneous Multicore Systems Using CBS and QBICTM. *Appl. Sci.* **2021**, *11*, 5740. [CrossRef]
40. Grandl, R.; Kandula, S.; Rao, S.; Akella, A.; Kulkarni, J. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), USENIX Association, Savannah, GA, USA, 2–4 November 2016; pp. 81–97.
41. Liu, Y.; Wang, S.; Zhao, Q.; Du, S.; Zhou, A.; Ma, X.; Yang, F. Dependency-aware task scheduling in vehicular edge computing. *IEEE Internet Things J.* **2020**, *7*, 4961–4971. [CrossRef]
42. Lee, J.; Ko, H.; Kim, J.; Pack, S. DATA: Dependency-aware task allocation scheme in distributed edge clouds. *IEEE Trans. Ind. Inform.* **2020**, *16*, 7782–7790. [CrossRef]