SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX

Yuan Chen, Jiaqi Li, Guorui Xu, and Yajin Zhou, *Zhejiang University;*
Zhi Wang, *Florida State University;* Cong Wang, *City University of Hong Kong;*
Kui Ren, *Zhejiang University*

https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yuan

This paper is included in the Proceedings of the
31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the
31st USENIX Security Symposium is
sponsored by USENIX.

SGXLock: Towards Efficiently Establishing Mutual Distrust Between Host Application and Enclave for SGX

Yuan Chen
Zhejiang University

Jiaqi Li
Zhejiang University

Guorui Xu
Zhejiang University

Yajin Zhou*
Zhejiang University

Zhi Wang
Florida State University

Cong Wang
City University of Hong Kong

Kui Ren
Zhejiang University

Abstract

Since its debut, SGX has been used to secure various types of applications. However, existing systems usually assume a trusted enclave and ignore the security issues caused by an *untrusted* enclave. For instance, a vulnerable (or even malicious) third-party enclave can be exploited to attack the host application and the rest of the system. In this paper, we propose an efficient mechanism to confine an untrusted enclave's behaviors. In particular, the threats of an untrusted enclave come from the enclave-host asymmetries, which can be *abused* to access arbitrary memory regions of its host application, jump to any code location after leaving the enclave and forge the stack register to manipulate the saved context. Our solution breaks such asymmetries and establishes mutual distrust between the host application and the enclave. Specifically, it leverages Intel MPK for efficient memory isolation and the x86 single-step debugging mechanism to capture the exiting event of the enclave. Then it performs the integrity check of the jump target and the stack pointer. We have implemented a prototype system and solved two practical challenges. The evaluation with multiple micro-benchmarks and representative real-world applications demonstrated the effectiveness and the efficiency of our system, with less than 4% performance overhead.

1 Introduction

Intel Software Guard eXtension (SGX) provides a hardware Trusted Execution Environment (TEE). With SGX, the sensitive code and data can be put into a protected memory region, i.e., the *enclave*, which is hardware-isolated from the rest of the system. Even system software, e.g., the OS and the hypervisor, is unable to access the content of the enclave.

The strong security and privacy guarantees provided by SGX make it attractive to build the confidential cloud computing infrastructure [10, 12, 15, 25, 26, 29, 38]. In these systems, the major concern is that the underlying computation platform is out of the tenant's control and untrusted. SGX addresses this by removing the cloud provider out of the trusted computing base (TCB).

Problem statement However, existing systems usually assume a trusted enclave. The security issues caused by an *untrusted* enclave have been neglected. Such an assumption is problematic from the perspective of an enclave's host application. One representative scenario is the adoption of third-party enclaves. With the popularity of SGX, service providers tend to deploy their services (e.g., trained machine learning models) inside the enclave to protect their intellectual property. This greatly simplifies the interactions between the service providers and consumers, thus creating valuable business benefits. However, embedding the service provider's enclave into an application brings huge security risks to users, considering the fact that the service enclave comes from an (untrusted) third-party and could be vulnerable and even malicious . Even worse, the isolation provided by SGX makes it much harder to inspect the third-party enclave. One recent research [27] has demonstrated the hazard of an enclave malware to hijack its host application's control flow *stealthily*. Furthermore, potential software bugs in an enclave also makes it untrusted to the host application. For instance, Van Bulck et al. [33] vetted six popular SGX runtimes and found security issues in all of them. A buggy enclave can be exploited by the attackers to compromise its host application and even the system software further.

Current solutions and their limitations Because of the isolation provided by SGX, an enclave cannot be inspected at runtime from the host application [14]. Besides, the code in an enclave may be deployed as cipher-text and only decrypted inside the enclave at runtime to protect the code secrecy [9]. This further impedes introducing a full vetting process on the enclave code.

One proposal is to detect the malicious actions of an enclave by monitoring its I/O behaviors [14]. However, how to (automatically) recover the enclave's semantics from underlying I/O operations is unclear. Moreover, Costan et al. dis-

*Corresponding author (yajin_zhou@zju.edu.cn)

cussed embedding a standardized static analysis framework into the enclave [14]. This brings two concerns, i.e., the source credibility of the static analysis framework and the increased TCB introduced by the framework. As the first implemented defense system, SGXJail [36] leverages the process isolation to confine an untrusted enclave. However, it requires creating a dedicated sandbox process for each enclave, which is not scalable for multiple enclaves and multi-threading scenarios. Weiser et al. proposed a hardware-based defense mechanism named HSGXJail [36]. It requires hardware modification, thus cannot be applied to existing platforms.

Our solution In this paper, we propose an efficient defense mechanism to confine an untrusted enclave's behaviors. Threats caused by an untrusted enclave are due to the blind trust of the host application to the enclave [36]. Such blind trust causes the enclave-host asymmetries, i.e., *the data access asymmetry and the control flow asymmetry*.

Specifically, with the data access asymmetry, an enclave can read and write arbitrary memory regions of its host application, while the memory region of an enclave is hardware-protected by SGX. With the assistance of Intel *Transactional Synchronization Extensions* (TSX), an enclave can even probe the whole address space of its host application *stealthily* without triggering any exceptions [27]. Besides, with the control flow asymmetry, an enclave can jump to *any code location of its host application* after leaving the enclave and forge the stack register to manipulate the saved context.

Our solution intends to break such asymmetries and establishes mutual distrust between the host application and the enclave. To this end, it leverages two x86 hardware features, i.e., *Intel memory protection key (MPK) and x86 single-step mode*, to break the data access and control flow asymmetries, respectively. Specifically, it leverages MPK for efficient memory isolation. MPK enables an efficient partition of a process's address space into disjoint protection keys and provides the permission control over these protection keys from a per-thread view. Our solution assigns different protection keys for the host application and the enclave, so that the enclave cannot access arbitrary memory regions of its host application. It leverages a shared buffer for the data sharing crossing the enclave boundary with the same protection key. Moreover, it leverages the x86 single-step debugging mechanism to capture the event when an enclave is exiting. It then performs the integrity check for the jump target and the stack pointer. Note that, applying these two hardware features to confine the enclave's behavior is non-trivial since we need to ensure that the enforcement of our solution cannot be bypassed (Section 4.3).

Because our solution is based on in-process confinement, it is more scalable compared to the previous system [36], which relies on the inter-process isolation. Our solution does not introduce any performance overhead for the code execution inside the enclave. The overhead only occurs when the execution crosses the boundary of the host application and the enclave.

We have implemented a prototype system named SGXLock based on Intel SGX SDK for Linux. Note that our design is coupled with neither Intel's SGX SDK nor Linux. It is applicable to other SGX runtimes and OSs as long as MPK and x86 single-step mode are supported.

We use multiple micro-benchmarks and real-world applications to evaluate the performance overhead of our system. In particular, we use three representative applications, i.e., the privacy-preserving machine learning service, the relational database, and a HTTPS web server in the evaluation. The machine learning service usually requires the large size parameter passing, e.g., for model weights or inputs. The database and a web server require lots of system calls, which represents the scenarios of high-frequent context switches between the host application and the enclave. The evaluation result shows that our prototype is efficient. It only introduces an average performance overhead of 0.84% for the machine learning service, 1.26% for the database and 3.98% for the HTTPS web server.

In summary, this paper makes the following main contributions.

- We summarize two types of the enclave-host asymmetries and work towards establishing mutual distrust between them. In particular, we leverage two x86 hardware features, i.e., MPK and the single-step mode, to efficiently break the asymmetries (Section 4.1 and Section 4.2).
- We have solved two practical challenges, i.e., blocking PKRU update inside the enclave and host stack pointer manipulation by the enclave (Section 4.3), and implemented a prototype system named SGXLock (Section 5).
- The evaluation with multiple micro-benchmarks and representative applications shows the efficiency of our system, with less than 4% performance overhead (Section 6).

To engage the community, we will release the source code of our system [1].

2 Background

2.1 Intel SGX

Intel Software Guard eXtension (SGX) allows an application to create a so-called *enclave*, which contains sensitive information (code and data). The confidentiality and integrity of an enclave are hardware-guaranteed, even the system software (e.g., OS and hypervisor) cannot access the content of the enclave. Architecturally, an enclave is a protected memory region residing in the host application's address space.

Host-enclave interaction The programming model of the Intel SGX SDK [3] allows developers to specify two kinds of host interfaces for the enclave. First, an invocation into the enclave is referred as an `ECALL`, which is used by the host application to invoke a specific pre-defined function inside

[1] https://github.com/blocksecteam/sgxlock

the enclave. Second, OCALLs are used by the enclave to invoke the functions outside the enclave, which are usually aimed to request OS services (e.g., system calls). Besides, the SDK performs proper sanitizing and marshaling for the ECALL/OCALLs' parameters according to the argument attributes specified by the enclave developer. For instance, for a data pointer argument of an ECALL with the [in,size=10] attribute, the SDK will allocate a 10-byte memory buffer inside the enclave and copy the corresponding data from the host application into the allocated buffer inside the enclave.

The ECALL/OCALL interface functions are defined via a so-called Enclave Definition Language (EDL). The Edger8r tool, shipping as part of the SDK, can generate (dispatch/receiving) edge routines for ECALLs/OCALLs according to developer-defined EDL files. These edge routines reside in both the host application and the enclave to route ECALL/OCALL requests.

Besides, the Trusted Runtime System (tRTS) and Untrusted Runtime System (uRTS) are provided to embed into an enclave and its host application to perform enclave management and ECALL/OCALL requests routing. The dispatch (receiving) edge routines reside in both sides will send (accept) the ECALL/OCALL requests to (from) the tRTS/uRTS and redirect them from (to) the user code. The combination of the tRTS/uRTS and the edge routines facilitate the (host application/enclave) developers to invoke an ECALL/OCALL as a function call.

The implementation of ECALL/OCALL interfaces is based on two *user-mode* instructions, EENTER and EEXIT, respectively. Specifically, the EENTER instruction is used by the host application to transfer the control to a predefined address inside the enclave. The EEXIT instruction makes the execution leave from the enclave. As an operand of the EEXIT instruction, the RBX register specifies the jump target outside the enclave, whose value will be filled into the RIP register after the execution leaves from the enclave. Therefore, *the jump target of the* EEXIT *instruction is enclave-manipulable.*

Moreover, SGX leaves most of the execution states (e.g. registers) switching and sanitizing to the software when the execution crosses the boundary of the host application and the enclave via the EENTER and EEXIT instructions. For instance, when the execution leaves the enclave via the EEXIT instruction, it is the responsibility of the enclave code to refill the host stack pointers, i.e., RSP and RBP. Thus, *the enclave can refill a fake stack pointer inside the host application to manipulate the saved context.*

Furthermore, the enclave execution can be aborted asynchronously when an exception (e.g., a page fault or a hardware interrupt) occurs. This is referred as *asynchronous enclave exit* (AEX). On an AEX event, the processor saves the enclave's current execution context, such as general purpose registers (GPRs) and processor extended states, into the enclave's *state save area* (SSA) frame. Then the processor sets the RIP register to the value of the Asynchronous Exit Pointer (AEP, normally points to the ERESUME instruction) and exits the enclave. After that, the processor delivers the AEX event as an normal exception to the system software. Note that the AEP is specified during enclave entry (as the operand of the EENTER/ERESUME instruction) and cannot be modified during the enclave execution. After handling the AEX event, the host application can reenter the enclave and resume the previously saved execution context of the enclave via the ERESUME instruction.

SGX supports multi-threading with a special enclave data structure, named *Thread Control Structure* (TCS). A TCS contains information of a thread executing inside an enclave, such as the address of the enclave entry and the relative address of SSA. The content of a TCS is specified by the enclave (developers) and is *not accessible* by the software (including the enclave) after being initialized. Whenever the execution wants to enter the enclave (via EENTER/ERESUME), a free TCS needs to be specified in the instruction operand. By doing this, SGX allows concurrent execution inside the enclave.

2.2 Intel MPK

Memory Protection Key (MPK) is a new hardware feature introduced in recent Intel processors to provide permission control over the page groups from a per-thread view. MPK exploits four previously-unused bits of the page table entry to serve as the page's protection key. Thus, MPK partitions a process's address space into 16 disjoint protection domains. To enforce the per-thread permission control, a per-core 32-bit *protection key rights register* (PKRU) is introduced. Every two bits in PKRU determine the access permission to one specific protection key. Specifically, PKRU[$2i$] represents the access-disable bit while PKRU[$2i+1$] represents the write-disable bit for protection key i. Based on access-disable and write-disable bits, three permission policies are enforced: read/write, read-only and no-access.

To access PKRU, MPK introduces two new *user-mode* instructions, RDPKRU (for reading) and WRPKRU (for writing). Updating the value of the PKRU register with the WRPKRU instruction has negligible overhead, which takes less than 20 cycles [24]. This makes MPK a highly efficient user-space memory permission control primitive. Note that the execution permission is *not* impacted by MPK.

Interact with SGX During the enclave execution, the value of PKRU can be stored into and restored from memory as part of the *processor's extended states*, as long as the bit 9 in the enclave's XFRM attribute field is set. Inside the enclave, the processor's extended states can be stored into and restored from memory in two scenarios. First, an enclave can save and restore them via XSAVE and XRSTOR instructions. Second, the processor's extended states are saved into the enclave's *state save area* (SSA) frame on an AEX event and restored when reentering the enclave with the ERESUME instruction.

2.3 The x86 Single-Step Mode

The x86 architecture supports the single-step mode for debugging. It allows the processor to generate a trap after executing each instruction. The single-step mode is activated by setting a bit, named Trap Flag (TF), within the processor's `FLAGS` register. If the TF bit is set by an application using a `POPF`, `POPFD`, or `POPFQ` instruction, the CPU will execute one instruction and then stop. At the same time, a single-step debug exception is generated. Accordingly, the single-step mode can be disabled by clearing the TF bit.

Interact with SGX For the opt-out enclave (i.e. enclave with the debugging feature disabled), if the TF bit is set at the time of the enclave entry, a single-step debug exception would be pending immediately after exiting the enclave via the `EEXIT` instruction. In other words, the processor in the single-step mode treats the whole life-cycle of the enclave execution (from enclave entry to enclave exiting) as a single instruction. Besides, the enclave is *not allowed* to manipulate the value of the TF bit. The processor guarantees this with two properties. First, at the time of the enclave entry, the processor stores the TF bit into a software *invisible* register and then clears it, while the value of the TF bit is restored after exiting the enclave. Second, the processor ensures that the TF bit always remains cleared inside the enclave.

3 Threat Model

Our threat model considers the mutual distrust between the enclave and its host application. Therefore, the threat model consists of the following two perspectives.

- **From the enclave's view** It assumes the same attack model as other systems that leverage SGX for protection. Specifically, the trust anchors for an enclave (developer) are the SGX-enabled CPU and the code inside the enclave. The rest of the software stack, including the host application, OS, and hypervisor, is considered as untrusted. The goal of an enclave is to prevent the leakage of its sensitive information (data or/and code) both at runtime and at rest. Thus, code secrecy mechanism [9] may be used by an enclave to protect its private code.
- **From the host application's view** It considers the enclave as *untrusted* [27, 36]. As mentioned previously, such scenarios include third-party enclaves and potentially buggy enclaves. The goal of the host application is to leverage the functionalities provided by the enclave, while at the same time confining the enclave's behaviors.

Out of scope The main purpose of our work is to restrict the host-enclave interactions to specified interfaces. It lays the basis for *the establishment of the mutual distrust between the enclave and its host application*. The high-level attacks, e.g., the Iago attack [13], that exploits the specified interfaces, are out of the scope. Besides, side-channel and denial-of-service (DoS) attacks are not considered. Defending against such attacks are orthogonal to our work.

4 System Design

From the enclave's view, the security mechanisms provided by SGX can be used to protect the sensitive data and code inside the enclave. However, from the host application's view, there exist two types of asymmetries [27, 36] between the enclave and its host application.

- **Data access asymmetry** An enclave can read and write arbitrary memory regions of its host application, while the memory region of an enclave is hardware-protected by SGX. With the assistance of Intel Transactional Synchronization Extensions (TSX), an enclave can even probe the whole address space of its host application *stealthily* without triggering any exceptions [27].
- **Control flow asymmetry** An enclave can jump to any code location of its host application by specifying the target address inside the `RBX` register, which will be used by the `EEXIT` instruction. It can also forge the stack register when leaving from the enclave to manipulate the saved context (Section 2.1). However, the host application is not allowed to specify the entry point when entering the enclave via the `EENTER/ERESUME` instruction.

Our work is to confine the *untrusted* enclave for its host application, so that the host-enclave interaction can be restricted to the specified interfaces. This helps to establish the mutual distrust between them.

To achieve this goal, SGXLock leverages two x86 hardware features to efficiently eliminate the enclave-host asymmetries. First, for data access asymmetry, SGXLock relies on Intel MPK [24] to efficiently confine the enclave access to limited regions of the host memory (Section 4.1). Only specific regions in the host memory's address space are accessible by the enclave. These regions serve as parameter passing buffers for the host-enclave interaction. Second, for control flow asymmetry, SGXLock leverages the x86 single-step mode to ensure that the execution continues from the predefined location after leaving the enclave (Section 4.2).

It is non-trivial to enforce such confinements. There are two challenges, i.e., how to block the update of the PKRU register inside the enclave and how to protect the host application's saved context when leaving the enclave. We will illustrate our solutions in Section 4.3.1 and Section 4.3.2, respectively. Figure 1 shows the overall system architecture.

4.1 Data Access Asymmetry Elimination

To eliminate the data access asymmetry, MPK is used to restrict the data access of an enclave to the specified regions in its host application's address space. For simplicity, we first assume there is only one enclave i in the host application and then discuss the scenario of multiple enclaves.

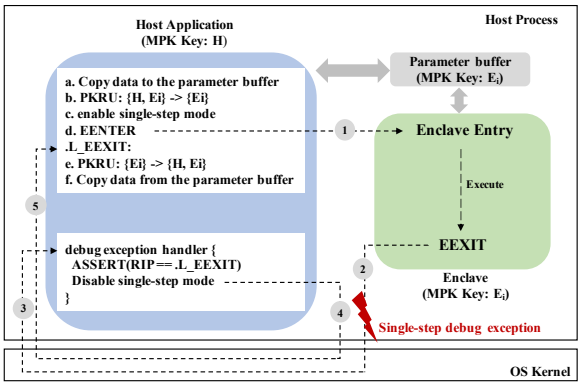

Figure 1: The overall system architecture.

The host memory regions are assigned with the protection key *H*, which is exactly the default protection key in the OS (e.g., key 0 in Linux). SGXLock then assigns the memory region of enclave *i* with a unique protection key *Ei* at the time of the enclave creation. To support parameters passing for the host-enclave interaction, SGXLock couples each TCS of the enclave *i* with a piece of the host memory region, called the *parameter buffer*. A parameter buffer is organized as a stack to enable the *nested* host-enclave interactions. Because a parameter buffer is coupled with each TCS, SGXLock can support concurrency inside the enclave without having the *scalability* issue. Accordingly, All the parameter buffers of the enclave *i* belong to the MPK protection key *Ei*.

First, the host application has the access permission to protection keys {*H*, *Ei*}. This does no harm to the security of the enclave since the content of the enclave is hardware-protected by SGX. Second, the enclave can only access the protection key *Ei*. Before the execution is transferred to the enclave *i*, the access permission of the current thread is changed from {*H*, *Ei*} to *Ei* via updating the PKRU register (step b in Figure 1). SGXLock guarantees that the PKRU register is not changed while the execution remains inside the enclave. By doing so, the enclave cannot access the host memory regions other than the enclave's parameter buffer. After exiting from the enclave, the execution resumes the host application's original access permission (step e in Figure 1). In this way, SGXLock eliminates enclave-host data access asymmetry *efficiently*, since no performance overhead is introduced inside the enclave.

Multiple enclaves The support of multiple enclaves is straightforward. First, a unique protection key is allocated for each enclave. The memory region and the parameter buffer of an enclave are then assigned with the enclave's protection key during the process of the enclave creation. Second, the host application has the access permission to all the enclaves' protection keys, whereas each enclave can only access its own protection key. If the number of enclaves exceeds the maximum number of available protection keys, the virtualization of protection keys could be leveraged [24].

About the AEX event AEX events cannot be abused by an enclave to bypass the MPK-based data access asymmetry elimination. First, SGXLock guarantees that the PKRU register is not part of the processor's extended states (See Section 4.3.1), thus it is not saved into the enclave's SSA frame when an AEX event happens. Second, an enclave should be forbidden to specify the control flow of the AEX handler outside the enclave, considering this will compromise the security guarantees from both the enclave and its host application's view. [2] Third, the PKRU register value is maintained as part of the host application's thread context during the AEX event handling outside the enclave. Therefore, the execution can reenter the enclave via the ERESUME instruction with the original PKRU register value set.

4.2 Control Flow Asymmetry Elimination

In Section 4.1, we show how SGXLock utilizes MPK to eliminate the data access asymmetry. However, the design goal of MPK is intended to provide permission control for data access. Instruction fetching is not constrained by MPK. Hence, MPK cannot be used to eliminate the control flow asymmetry. Thus, the code inside the enclave can jump to arbitrary executable locations inside the host application's memory space after executing the `EEXIT` instruction.

The root cause of the control flow asymmetry is that the jump target of the `EEXIT` instruction is enclave-manipulable. To solve this problem, SGXLock leverages the x86 single-step mode feature to detect whether the jump target of the `EEXIT` instruction matches the pre-defined location, i.e., the next instruction after the `EENTER` instruction (label .L in Figure 1).

Specifically, whenever the execution gets into the enclave, the TF bit within the `FLAGS` register is set (step c in Figure 1) to ensure the pending of a single-step debug exception follows the `EEXIT` instruction (② in Figure 1). The corresponding exception handler then performs the check for the jump target (③ in Figure 1). If the check passes, the execution is resumed and continues from the pre-defined location in the host application's code region (⑤ in Figure 1). Otherwise, the potential abuse (or exploit) of the control flow asymmetry is detected. The execution is aborted.

Same with the MPK-based data access asymmetry elimination, eliminating control flow asymmetry with the assistance of the single-step mode introduces no performance overhead for the execution *inside the enclave*.

About the AEX event On an AEX event, the processor resumes the value of the TF flag from the software invisible register and exits the enclave. Then, same as the PKRU register, the TF flag is treated as part of the host application's thread

[2] A proper mechanism to allow the enclave to get involved into the exception handling caused by itself is that the handler is put inside the enclave and is invoked by the host application via the EENTER instruction when handling the AEX event. This mechanism is exactly the one used by Intel SGX SDK and is compatible with SGXLock.

context and remains set when the execution is resumed from the AEP. Since the AEP points to the ERESUME instruction, the TF flag is kept as set at the enclave entry.

4.3 Challenges and Solutions

We have described the main idea of adopting two x86 hardware features to eliminate the enclave-host asymmetries. However, there are still some challenges to ensure that the security enforcement of our system cannot be bypassed. In the following, we illustrate these challenges and our solutions.

4.3.1 Challenge I: Block PKRU Update Inside the Enclave

The PKRU register can be updated with two *user-mode* instructions, `WRPKRU` and `XRSTOR`. An untrusted enclave may use these two instructions to extend its access permission to the protection keys other than itself. To prevent this, SGXLock needs to keep the PKRU register unchanged during the enclave execution. A similar challenge is also addressed by the ERIM system [32]. However, the solution proposed by the ERIM cannot be directly applied to our system. The reasons are listed in Appendix A. To solve this challenge, SGXLock adopts the following strategies for the `XRSTOR` and the `WRPKRU` instruction, respectively.

XRSTOR instruction The usage of the `XSAVE/XRSTOR` instruction is intended to save/restore the current execution's extended states *efficiently*. Thus they are unavoidable inside the enclave. However, in our design, an enclave is not allowed to maintain the PKRU register since SGXLock requires the PKRU register be unchanged during the enclave execution. Besides, there is no need for an enclave to maintain its own PKRU register value. That is because *MPK cannot be used by an enclave to provide access permission control for itself*. MPK's permission control is based on two components: the per-core PKRU register and the 4-bit protection key field in the page table entry. Apparently, the latter one is out of the enclave's control. Thus, if an enclave leverages MPK to enforce the permission control, the untrusted host application or operating system can easily corrupt it.

We observed that the PKRU register is treated as part of the enclave execution's extended states *only when the bit 9 within an enclave's XFRM attribute field is set.* Therefore, our solution is straightforward. SGXLock guarantees that the bit 9 within an enclave's XFRM attribute field is never set. Specifically, during the enclave creation, an additional check is performed by SGXLock. If it is set, SGXLock refuses to create the enclave. Since the XFRM attribute field cannot be modified after the enclave creation, the `XRSTOR` instruction cannot affect the PKRU register even though it occurs inside the enclave. Note that with this method, SGXLock also guarantees that the PKRU register would not be affected by the AEX event.

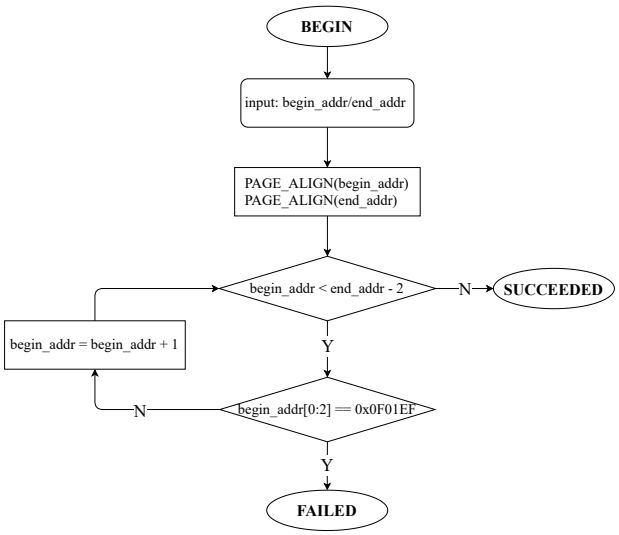

Figure 2: The flow chart of the embedded inspection code.

WRPKRU instruction SGXLock leverages binary inspection to prevent the occurrence of the `WRPKRU` instruction inside the enclave. In some scenarios, an enclave may load or generate code at runtime for the purpose of code secrecy [9]. Our system leverages the binary inspection mechanism to deal with both the plain enclave code and the dynamic loaded or generated code (abbreviated as *DLGC*), respectively. For the plain enclave code, SGXLock can directly check whether there exists `WRPKRU` instructions while loading the code into the enclave during the enclave creation. For DLGC, since its content is inaccessible outside the enclave, SGXLock chooses to *embed* a small piece of inspection code into the enclave at the enclave compilation stage. Figure 2 shows the logic of our embedded inspection code: *scanning the corresponding code region linearly to detect the occurrence of a specific byte sequence (i.e., `WRPKRU`'s machine code: `0x0F01EF`).* At runtime, by enforcing the $W \oplus X$ attribute for the enclave's page table entries, the host application can detect the execution of DLGC inside an enclave. Then the inspection code will be invoked by the *host application* before giving the execution permission to the DLGC region.

As shown in Figure 2, our embedded inspection code only includes a simple loop and can be easily implemented with less than one hundred lines of assembly code. In fact, the implementation of our embedded inspection code can be integrated into the SGX software development kit (SDK). Thus no development burden is imposed to enclave developers. To ensure the usability and security of the embedded inspection code, SGXLock made the following design choices. First, an *assembly* template, that implements exactly the logic of our embedded inspection code (See Figure 2), is provided by SGXLock. The template only uses the registers to store its data (including input/output) and can be instantiated by replacing the template parameter with the specific registers.

Second, the template instance is integrated into the *plain* in-enclave runtime and its input/output should always remain inside the registers during the data transfer (from the enclave entry to the enclave exit point). Thus the template instance needs to be placed to the location that is close to the enclave entry/exiting. For example, a conditional branch instruction is inserted into the enclave entry and one branch targets for the template instance, while an unconditional branch instruction targeting the enclave exiting is inserted into the end of the template instance. Third, the related information of the template instance is provided with the enclave file to assist the verification of the template instance.

Based on the previous design, during the enclave creation, SGXLock can verify the existence of the template instance (i.e. inspection code). SGXLock also confirms that the template instance is always reachable from the enclave entry when invoked by the host application and the (input/output/immediate) data never leaves the registers during the invocation process. Always keeping data inside the registers prevents other concurrent enclave execution from corrupting the inspection code's execution state. Besides, obfuscation should not be applied to the template instance. Doing so fails the verification of SGXLock and the execution of DLGC would not be allowed.

Note that, page alignment operation is performed at the beginning of our inspection code. This ensures that the potentially malicious host application can only snoop whether there is a specific three-byte sequence within the inspected region at the page granularity. Such information is rather limited to leak the enclave's sensitive information.

We have described that SGXLock can enforce no occurrences of the `WRPKRU` instruction in the DLGC of an enclave with two components: $W \oplus X$ for the detection of DLGC and the embedded code for inspection. In the following, we illustrate the overall *workflow* of the binary inspection mechanism that targets both the plain enclave code and DLGC.

Static binary inspection It happens at the time of the enclave creation. First, SGXLock scans the plain enclave code while loading it into the enclave to inspect the occurrence of the WRPKRU instruction. If the inspection passes, the corresponding enclave code page will be assigned with *RX* permission. Besides, for the enclave page with *WX* permission set, SGXLock removes its *X* permission and records its information. Meanwhile, SGXLock checks whether the inspection code is embedded inside the enclave *properly*. If the checking passes, an enclave-invisible flag *dlgc_allowed* is set to indicate the created enclave is allowed to execute DLGC. Note that the embedded inspection code is part of the plain enclave code and keeps *RX* during the enclave's life-cycle. With the static binary inspection, SGXLock achieves three guarantees. First, there is no `WRPKRU` instruction inside the enclave code after the enclave initialization. Second, the execution of the DLGC inside the enclave will be detected by SGXLock (since the executable permission has been removed.) Third, for an enclave using DLGC, the inspection code has been properly embedded.

Dynamic binary inspection This is triggered by the violation of $W \oplus X$ during the enclave execution. As an AEX event, the violation of $W \oplus X$ makes the execution exit the enclave and allow the host application to get control. First, the host application checks whether the *dlgc_allowed* flag is set and the enclave page's original permission set has the *X* permission. If both of them are satisfied, this means a DLGC code region is executed and the dynamic binary inspection needs to be invoked.

Specifically, the host application changes the permission of the enclave page that needs to be inspected to *read-only*. [3] This prevents the concurrent enclave executions from modifying the enclave page that is to be inspected. Note that the system call is not allowed inside the enclave, thus the enclave cannot directly manipulate the enclave's page table entries. Before the host application specifies a free TCS and gets into the enclave via the `EENTER` instruction to invoke the embedded inspection code, it changes the protection key of the SSA frame of the TCS to the host application's protection key (i.e. *H*). Accordingly, the host application keeps the access permission of the current execution to the protection keys of both the host application and the enclave (i.e., $\{H, Ei\}$) before entering the enclave. This aims to prevent other concurrent enclave execution from modifying the register values of the inspection code, which are stored into the SSA frame on an AEX event during the execution process. Besides, as we mentioned before, the embedded inspection code only uses registers to store its data. Therefore, there is no window for other concurrent enclaves to corrupt the current execution context of the embedded inspection code.

Then, the host application gets into the enclave to invoke the embedded inspection code. The inspection code will scan the corresponding page(s) according to the address information provided by SGXLock. One bit is returned to indicate whether the inspection passes. If the inspection fails, the host application forbids the execution of the DLGC inside the enclave. Otherwise, the host application assigns the corresponding page(s) with *RX* permission and the original enclave execution is resumed.

We can see that before invoking the inspection code, the host application changes the permission of the enclave page that needs to be inspected to *RO*. After the inspection passes, the inspected enclave page will be assigned with *RX* permission by the host application. There is no time window that the enclave page to be inspected is writable. This prevents other concurrent enclave execution from modifying the page to be inspected during the whole process. Thus the possible Time-of-check to Time-of-use (TOCTOU) attack is blocked.

[3] In Linux, it is implemented via `pkey_mprotect`, which would guarantee the TLB coherence of the permission update.

4.3.2 Challenge II: Block Host Stack Pointer Manipulation

Even though the jump target of the EEXIT instruction is constrained with the help of x86 single-step mode, an untrusted enclave is still capable of controlling the value of the host application's stack pointer (i.e., RSP and RBP) while the execution leaves the enclave. Even worse, considering the parameter buffer of an enclave is accessible to both the enclave and the host application, a fake host stack can be forged by an untrusted enclave [27]. As a result, the host application would continue execution with a forged stack context after the execution leaves the enclave.

Our solution SGXLock borrows the idea from keyed-hash message authentication code (HMAC) to guarantee the integrity of the host stack pointers. Specifically, SGXLock maintains a global 64-bit secret value, named *g_sp_key*, for the host application. Whenever the execution gets into the enclave via the EENTER instruction, SGXLock performs bit-wise exclusive or (XOR) operations on the host stack pointers, i.e., RSP and RBP, with *g_sp_key*, respectively. The results, *(RSP⊕g_sp_key)* and *(RBP⊕g_sp_key)*, are then stored into the host stack. Note that this step takes place before trimming the current execution's access permission and the host stack would be inaccessible while the execution is inside the enclave.

Since the jump target of the EEXIT instruction is constrained with the help of the x86 single-step mode, SGXLock is able to perform the integrity check for the host stack pointer when the enclave execution finishes. In detail, SGXLock uses the current (*potentially corrupted*) RSP register for addressing to retrieve the previously stored *(RSP⊕g_sp_key)* and *(RBP⊕g_sp_key)* from the host stack. Then, SGXLock performs XOR operation on them with *g_sp_key* and compares the result with the current RSP/RSP register. If their values match, SGXLock confirms that the integrity of the host stack pointer is preserved. Otherwise, potentially malicious behavior is detected.

Besides, the execution could exit the enclave on an AEX event and the exception handling may need the user-space involvement (e.g., a signal handler in Linux). For this situation, the integrity check could be performed in the corresponding user-space handler, too.

In summary, SGXLock ensures the integrity of the host stack pointers during the enclave execution. First, *(RSP⊕g_sp_key)*, *(RBP⊕g_sp_key)* and *g_sp_key* are all stored in the enclave invisible memory region. This is enforced by SGXLock with MPK. Thus, an untrusted enclave is unable to steal or tamper with these values directly. Second, the only information known to the enclave is the plain host stack pointers. Therefore, the untrusted enclave cannot forge a fake stack since the value (i.e., *(RSP⊕g_sp_key)*, *(RBP⊕g_sp_key)*) to fill into the stack is unknown to the enclave.

5 Implementation Details

We have implemented a prototype of SGXLock based on Intel SGX SDK (v2.9.1) for Linux. The goal of our prototype is making SGXLock transparent to enclave developers and keeping backward-compatibility with legacy enclaves at source code level as much as possible. A developer can port a legacy enclave into our prototype with minor or even no modification. In this section, we illustrate some implementation details.

5.1 Data Access Asymmetry Elimination

Our modification mainly targets for the SDK to support ECALL/OCALL's parameter passing through SGXLock's param_buffer. Besides, we also make minor modifications (no more than 20 lines) to the signal handling of the Linux kernel (See Appendix C for details).

In the Intel SGX SDK, each ECALL/OCALL definition has a specific marshaling data structure, generated by the Edger8r tool, to interpret its arguments. The tRTS/uRTS only uses a void pointer for an ECALL/OCALL's parameter passing. When routing an ECALL/OCALL requests, the marshaling data structure is used by the edge routines of both sides to interpret the arguments by converting the void pointer to the marshaling data structure pointer. This design assumes an enclave can access its host application's memory regions arbitrarily. This assumption does not hold after applying our system. Therefore, we modified the Edger8r tool and the tRTS/uRTS to support parameter passing of ECALL/OCALL through param_buffer.

Edger8r For the Edger8r tool, we add a wrapper data structure, ms_param_meta_t (Listing 2 in Appendix D), to enable the uRTS to interpret an ECALL/OCALL's marshaling data structure. Then instead of passing a void pointer, the dispatch edge routine passes the ms_param_meta_t pointer to the tRTS/uRTS. The passed ms_param_meta_t pointer is finally used by the uRTS to copy data to/from the param_buffer.

tRTS The sgx_ocalloc function of the tRTS is used by the dispatch edge routine of the OCALL to allocate memory regions outside the enclave for parameter passing. We modify the implementation of sgx_ocalloc to allocate memory from the param_buffer instead of the host stack.

uRTS When receiving an ECALL request, the uRTS chooses a free TCS and copies the parameters to the corresponding param_buffer by interpreting the ms_param_meta_t pointer received from the dispatch edge routine of the ECALL. Besides, an instance of the marshaling data structure for the ECALL is created in the param_buffer, to interpret the parameters residing in the param_buffer. Then the uRTS routes the ECALL request to the tRTS with the address of the newly created marshaling data structure instance as a void pointer.

When receiving an OCALL request from the tRTS, the uRTS also receives a ms_param_meta_t pointer that points to some area within the param_buffer. Then the uRTS copies the parameters from the param_buffer to a private memory region

based on the ms_param_meta_t pointer. Besides, an instance of the marshaling data structure for the OCALL is created in the private memory region to interpret the copied parameters. Note that different from a param_buffer, the private memory region belongs to the host application's protection key and thus cannot be accessed by the enclave. The uRTS maintains a private memory for each TCS of an enclave to prevent the possible TOCTOU attack on the OCALL parameters.

Compatibility Our modification to the SDK will disable the support of the user_check attribute. Specifically, in Intel SGX SDK, the pointer argument of an ECALL/OCALL can be marked with the user_check attribute. For such a pointer, the receiving edge routine will neither verify the pointer nor copy the pointed buffer. The design of the user_check attribute leaves the duty of data validation to the developer and is prone to the TOCTOU attack. SGXLock only supports parameter passing via the developer-invisible param_buffers. Thus, the user_check attribute is not supported in our prototype.

5.2 Control Flow Asymmetry Elimination

The TF bit within the FLAGS register is set via the POPFQ instruction before the EENTER instruction. After the enclave execution exits via the EEXIT instruction, a single-step debug exception is triggered immediately. In our implementation, a single handler is registered to catch the SIGTRAP signal. In the signal handler, the jump target of the EEXIT instruction and the integrity of the host stack pointer are checked. If the check fails, the signal handler will report this behavior and abort the execution. Otherwise, the TF bit in the FLAGS register of the original context is cleared and the original execution is resumed.

5.3 Binary Inspection

To support the code secrecy, Intel SGX SDK provides *Protected Code Loader* (PCL) mechanism. With the PCL mechanism, the user enclave code is encrypted at build time and decrypted at runtime. To support this mechanism, a static library libsgx_pcl.a is included in the enclave and invoked by the uRTS to decrypt user code immediately after the enclave creation. Our prototype provides the support for the PCL mechanism and implements a three-stage binary inspection mechanism.

Static binary inspection We modify the uRTS to inspect the *plain* enclave code during the enclave creation. The uRTS also checks whether our inspection code is properly integrated into the enclave. If so, an enclave-invisible flag *dlgc_allowed* is set. The uRTS is also modified to enforce the $W \oplus X$ property for the enclave's page table entries by trimming *W* from the *WX* permission during the enclave creation.

PCL binary inspection Since our prototype enforces $W \oplus X$ for the enclave pages, the encrypted user enclave code does not have the *X* permission after the enclave creation. Our modified uRTS will invoke the embedded inspection code to inspect the newly decrypted user enclave code if it exists. Note that at this time, there is no concurrent enclave execution. Thus our system does not need to change the corresponding enclave page(s) to read-only or modify the protection key of the corresponding SSA frame to the host protection key. If the inspection passes, the modified uRTS will change the permission of the enclave page(s) to *RX* via the pkey_mprotect system call.

Dynamic binary inspection Since $W \oplus X$ is enforced for the enclave's pages, executing newly loaded or generated code inside the enclave will cause a $W \oplus X$ violation and trigger a page fault exception. A signal handler is registered to catch the SIGSEGV signal. If the signal code is SEGV_ACCERR and the fault page's original permission is *RWX*, the runtime binary inspection is triggered. First, the fault enclave page is marked as read-only via the pkey_mprotect system call. The embedded inspection code is then invoked by the host application to inspect the fault page. If the inspection passes, the permission of the fault page is changed to *RX* by the host application. Finally, the original execution is resumed and reenters the enclave via the ERESUME instruction.

6 Evaluation

6.1 Security Analysis

The goal of SGXLock is to eliminate the enclave-host asymmetries (i.e., data access and control flow asymmetry), so that the host-enclave interaction can be confined to the specified interfaces. Our security analysis focuses on the threats from an untrusted enclave that aims to bypass SGXLock's asymmetry elimination enforcement.

Bypass data access elimination There are three methods for an enclave to change the PKRU register value, i.e., using the XRSTOR instruction, leveraging the AEX event and using the WRPKRU instruction. SGXLock requires that the bit 9 of an enclave's XFRM attribute field is clear and thus the PKRU register would not be treated as part of the processor's extended states during the enclave execution. This blocks the first two methods since both of them manipulate the PKRU register through the processor's extended states. Since SGXLock ensures there is no WRPKRU instruction existed during the enclave's life-cycle via the binary inspection mechanism, the third attack is also blocked. Moreover, after handling the AEX event, SGXLock guarantees that the enclave's original PKRU register is resumed when reentering the enclave's via the ERESUME instruction.

Bypass control flow elimination SGXLock relies on the single-step mode to check the target of the EEXIT instruction. The TF flag is saved in a software-invisible register during the enclave execution and resumed after exiting the enclave. Besides, for enclave exiting due to an AEX event, the control

flow of the AEX event handling is out of the enclave's control and the TF flag would remain set when the execution reenters the enclave via the `ERESUME` instruction. Thus, the enclave is unable to clear the TF flag. Based on the single-step mode, SGXLock can further verify the integrity of the host stack pointer by adopting the HMAC-like strategy.

Case study We used three attack primitives leveraged by SGX-ROP [27], including host memory read, host memory write, and host stack pointer manipulation, to demonstrate the effectiveness of SGXLock. Since there is no TSX support in our experiment platform, we suppose the malicious enclave has the prior knowledge to the host memory layout. As a result, the host memory read/write primitive are detected by our MPK-based data access asymmetry elimination mechanism (Section 4.1), while the host stack pointer manipulation attack primitive is blocked by the combination of the single-step based control flow asymmetry elimination (Section 4.2) and the host stack pointer integrity checking (Section 4.3.2) mechanism.

Specifically, for host memory read/write manipulation, when the enclave tries to read/write host memory regions other than its *parameter buffers*, a `SIGSEGV` signal is captured by our signal handler due to the violation of MPK's permission restriction, and the execution is aborted. For host stack pointer manipulation, when the enclave execution exits, a single-step debug exception is issued and the control flow transfers to the corresponding signal handler. Since the host stack pointer is corrupted by the enclave, the corresponding integrity check in the signal handler fails and the execution is aborted.

6.2 Performance Evaluation

In this section, we focus on the performance evaluation of our prototype to demonstrate its effectiveness. Our evaluation was performed on a platform with the Intel i7-10700F CPU (2.90GHz), which supports both SGX and MPK, and the 16GB physical memory. The system software running on the platform is Ubuntu 18.04.4 (Kernel v5.4.28) with SGX driver v2.6 installed.

6.2.1 Evaluation Methodology

First, we use *micro-benchmarks* to measure the introduced host-enclave interaction latency by our prototype, which is the main cause of the performance overhead. Specifically, we evaluate the *raw latency overhead*. This is introduced by the additional interaction code, e.g., PKRU register update, single-step mode enabling/disabling, and single-step debug exception handling. We also evaluate the performance overhead for the *parameter passing*, i.e., copying to/from the parameter buffer (*param_buffer*).

Second, we choose three representative scenarios, i.e., the privacy-preserving machine learning service, the relational database, and the web server to evaluate the overhead of real-world applications. The choice is based on three reasons. First, they are popular scenarios for SGX that have been evaluated in previous systems. Second, both the database system and the web server require high-frequency system calls, which represent the real scenarios of high-frequent `OCALL`s. Third, as a data-intensive service, the machine learning service usually passes large parameters, including model weights or inputs. The benchmark programs used in the evaluation have been released in the link [7].

Last, we evaluate the overhead introduced by the code inspection mechanism.

Table 1: The evaluation result of the raw ECALL/OCALL latency.

	Original	SGXLock	SGXLock *[2]
ECALL	7,636	11,662 (52.7%)	9,288 (21.6%)
OCALL	5,908	9,588 (62.3%)	7,303 (23.6%)

[1] Time is measured in CPU cycles.
[2] SGXLock * represents our prototype with single-step mode disabled.

6.2.2 Micro-Benchmarks

Raw latency overhead We implement an empty `ECALL` routine with no arguments [4]. We then invoke the empty `ECALL` routine from the host application and record the execution time to represent the raw `ECALL` latency. To measure the raw `OCALL` latency, we implement an empty `OCALL` routine with no arguments and invoke it within the empty `ECALL`. Then we record the execution time of invoking the `ECALL` routine again and subtract the raw `ECALL` latency. Last, to evaluate the latency introduced by the data access and the control flow asymmetry elimination respectively, we disable the single-step mode and perform the above evaluation again.

Table 1 shows the overall result. Our prototype introduces 52.7% overhead for the raw `ECALL` latency and 62.3% for the raw `OCALL` latency. The control flow asymmetry elimination contributes 31.1% and 38.7% of the result, respectively.

Note that, the latency shown in Table 1 represents the *upper bound* of the performance overhead. That's because in real-world applications, there are usually complex workloads inside the worker function of the `ECALL/OCALL`.

To further show the raw latency overhead in practice, we use different `ECALL` worker functions that have different execution time inside the enclave. We also use various numbers of invoked `OCALL`s during this process. This can represent different types of applications. For each test, the function inside the enclave invokes an empty `OCALL` without any arguments at different frequencies. We repeat the measurement 20 times

[4] The *empty* routine means that the worker function of an ECALL/OCALL returns immediately when it is invoked.

Table 2: The overhead under different execution time inside the enclave and the OCALL frequency.

Frequency	Execution Time						
	1ms	5ms	10ms	50ms	100ms	500ms	1000ms
1	0.2%	0.1%	0.1%	0.04%	0.05%	0.03%	0.03%
10	1.4%	0.4%	0.5%	0.06%	0.02%	0.0%	0.0%
100	10.8%	2.4%	1.2%	0.3%	0.1%	0.05%	0.03%
1,000	42.6%	18.0%	10.3%	2.4%	1.2%	0.3%	0.1%
10,000	61.1%	51.6%	41.5%	17.6%	10.5%	2.4%	1.2%
50,000	63.7%	61.2%	56.7%	41.6%	31.6%	10.5%	**5.6%**
100,000	63.9%	62.8%	59.8%	50.2%	42.5%	18.0%	10.3%

and report the arithmetic mean values. Table 2 shows the result. The execution time inside the enclave ranges from *1ms* to *1s*.

The overhead is negatively correlated with the execution time inside the enclave, while positively correlated with the `OCALL` frequency. Note that in our evaluation, the `OCALL` is empty. Thus the overhead represents the corresponding upper bounds in practice. Generally speaking, the worker function inside the enclave tends to invoke the `OCALLs` at low-frequency, since the main purpose for an `OCALL` is to perform the system calls (e.g. file operations). According to Gruss et al. [16], Netflix had studied the system call rates of their cloud services and found the highest rate is around 50,000 system calls per second per CPU. As shown in Table 2, under the setting that the execution time is 1s and the OCALL frequency is 50,000, the performance overhead is 5.6%. It demonstrates the effectiveness of our system for real-world scenarios.

Parameter Passing Overhead We use the empty `ECALL` and `OCALL` routines, with a pointer argument `buf` whose size is 64MB. We use a large buffer size to make the time of parameter passing dominate the whole application's execution, thus omitting the raw latency overhead previously discussed.

After that, we specify the direction attribute of the `buf` as `[in]`, `[out]`, or `[in, out]` and then record the time cost for each specification, respectively. The `[in]`, `[out]` and `[in, out]` for an `ECALL` means the parameters will be copied into the enclave, copied from the enclave and copied in both directions. The meanings of the attributes for an `OCALL` are similar.

Table 3 shows the result. The worst case is for the pointer argument of an `OCALL` with the `[in]` attribute, whose overhead achieves 26.4%. Besides, the additional parameter passing time under the `[in,out]` setting is obviously longer than that under the `[in]` or `[out]` setting. This is because it requires two additional data copy operations, i.e., copy to and from the *param_buffer*, while the others only require one operation, i.e. copy to or from the *param_buffer*. Again, this evaluation only considers the raw parameters passing overhead, which shows the upper bound of the overhead.

One interesting finding is that a pointer argument with the `[out]` attribute requires much longer parameter passing time compared to others. Accordingly, the overhead is much lower

Table 3: The evaluation result of parameter passing. Time is measured in milliseconds.

	ECALL			OCALL		
	`[in]`	`[out]`	`[in,out]`	`[in]`	`[out]`	`[in,out]`
Original	28.0	145.5	45.6	17.9	157.0	45.7
SGXLock	32.9	150.9	55.7	22.7	161.1	54.4
Additional Time	4.9	5.4	10.1	4.8	3.1	8.7
Overhead	17.6%	3.7%	22.3%	26.4%	1.9%	18.9%

Table 4: The execution time (milliseconds) and overhead of the machine learning service using our system.

	Original	SGXLock	Overhead
Enclave creation	283.80	283.91	0.04%
Model loading	43.59	54.00	23.90%
Inference	1,962.10	1,970.85	0.45%
Total	2,289.49	2,308.76	0.84%

(3.7% and 1.9%, respectively). That is because the edge routines of `ECALL/OCALL` in the enclave side would call `memset` to clear the buffer that is newly allocated for the pointer argument with the `[out]` attribute. The `memset` function is specified as `__attribute__((optimize("O0")))`, thus the compiled `memset` function is not optimized and involves lots of memory access operations.

6.2.3 Real-world Applications

Machine learning as a service We run a trained neural network model (with around 34MB weights data) inside an enclave. The ML model is for predicting the responses of the cancer treatment [8]. SGX is used to protect both the model structure/parameters and input/output. All of them are encrypted outside the enclave and only decrypted inside the enclave at runtime. The implementation of the ML model has been highly optimized for SGX scenario to shorten its inference time [7]. Thus, using it for evaluation is representative to demonstrate the effectiveness of our system. In our experiment, we perform the model inference on 392 inputs (total size is 19MB) and record the time cost of each stage. Note that each stage is implemented as an `ECALL` but may perform multiple `OCALLs`. The measurement is repeated 30 times and the arithmetic median is reported.

As shown in Table 4, our prototype introduces negligible overhead (0.84% on average). In our prototype, extra data copies are required to transfer the `ECALL` parameters to the *param_buffer*. As mentioned above, the size of the model weights is about 34MB. Therefore, the overhead of the model loading stage is relatively high, which achieves 23.9%. For a real-world machine learning inference service, the model loading is performed only once to serve thousands of inputs or even more. Thus the overall overhead is acceptable, just as shown in our experiment (0.84% for 392 inputs).

Table 5: The execution time (milliseconds) and overhead of database operations.

	Original	SGXLock	Overhead
INSERT	1,856,643	1,867,392	0.58%
SELECT	8,428	8,629	2.38%
UPDATE	183,271	186,740	1.89%
DELETE	182,417	182,763	0.19%

Table 6: The evaluation result of a HTTP web server.

	Transfer Rate (kb/s)		Overhead	# of OCALLs
	Original	SGXLock		
1	5.94	5.69	4.21%	~742,000
2	5.93	5.7	3.88%	~742,270
4	5.9	5.7	3.39%	~742,810
8	5.93	5.67	4.38%	~743,890
16	5.92	5.68	4.05%	~744,960

Database We use a ported SGX version of the SQLite [6] database in our evaluation. As a lightweight database engine, SQLite is widely deployed in real-world applications. Besides, the execution of SQLite involves many file operations, thus causing many `OCALLs`. In the evaluation, we record the time cost of four common database operations, i.e., *INSERT, SELECT, UPDATE* and *DELETE*, respectively. Each operation is performed for 10,000 times and the SQL statements are generated with SQLite's performance testing script (`tools/speedtest.tcl`). We repeat our measurements 15 times and report the arithmetic mean.

As shown in Table 5, the overhead for the four operations rangers from 0.19% to 2.38%, with an average value of 1.26%. The highest overhead comes from the INSERT operation, which is 2.38%. We find that there are 11,0338 `OCALLs` and 10,000 `ECALLs` involved for the INSERT operations. In other words, the frequency for the INSERT operation under the original SDK achieves 13,092 `OCALLs` and 1,186 `ECALLs` per second (and 12,787 `OCALLs` and 1,159 `ECALLs` per second under our prototype). Under this high frequent switching between the host application and the enclave, the performance overhead is (only) 2.38%.

A HTTP web server The mbedtls-SGX [4] is a ported version of the `mbedtls`, which can run inside the SGX and provide an SGX-enabled TLS suite for developers. We evaluate the performance of a simple single-thread HTTP server, which is included in the mbedtls-SGX. The HTTP server serves a static web page whose size is 108 bytes. We use the Apache HTTP benchmark tool [1] to perform the evaluation. In each test, the server will serve 1,000 requests. The concurrency level, which represents the number of multiple requests to serve at a time, is increased from 1 to 16 with different intervals. The transfer rate, number of `OCALLs`, and the overhead are collected or computed and shown in Table 6. The introduced overhead is ranging from 3.39% to 4.38%.

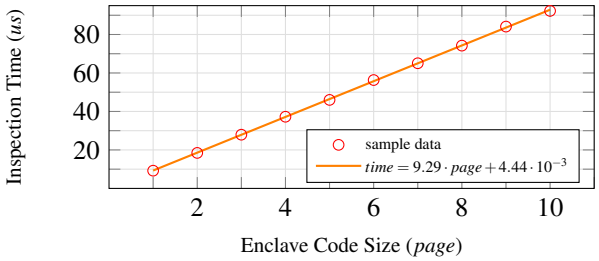

Figure 3: The dynamic inspection time of enclaves with different code size.

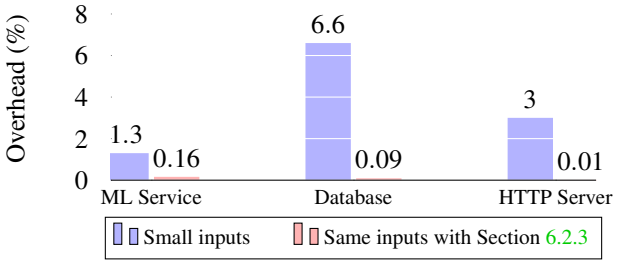

Figure 4: The result of three macro-benchmark applications for the dynamic binary inspection.

6.2.4 Dynamic Binary Inspection

The binary inspection mechanism consists of the following three ones: static binary inspection, PCL binary inspection and dynamic binary inspection. Static and PCL binary inspection only introduce the one-time overhead when creating an enclave. We report the result in Appendix B. In the following, we present the overhead of the dynamic binary inspection.

Raw inspection overhead Specifically, we first increase the size of the enclave code that needs to be dynamically inspected and record the corresponding inspection time. The enclave code size ranges from 4K to 40K. We repeat the inspection 100,000 times for each test and then calculate the average value. Figure 3 shows that the overhead introduced by the dynamic binary inspection is positively correlated with the enclave code size to be inspected. Specifically, it takes about 9.29 us (i.e. ~26,941 cycles for the CPU used in the evaluation) for dynamically inspecting one enclave code page.

Macro-benchmark applications We further port three macro-benchmark applications to emulate the code secrecy protection scenario by leveraging Intel SGX SDK's PCL mechanism. Moreover, PCL binary inspection is disabled. Thus the first execution of the enclave code at runtime will trigger the dynamic binary inspection. First, we feed each application with a small number of inputs (thus a short execution time inside the enclave) and records their *runtime* performance (i.e. the enclave creation time is excluded). Specifically, 4 samples for the ML service, 20 *SELECT* operations for the database service and 1 request for the web server are

used. We compare the execution time of the ported version application with the original one.

As shown in Figure 4, dynamic binary inspection introduces acceptable overhead for three macro-benchmark applications *even* with a fraction of inputs. Note that the overhead present here is obviously higher than real-world scenarios, sine more inputs will be served. To this end, we also evaluate the overhead with the same inputs used in Section 6.2.3, the performance overhead is negligible.

7 Discussion

High-level attacks SGXLock focuses on eliminating the enclave-host asymmetries so that the host-enclave interaction can be restricted to the specified interfaces. High-level attacks [13, 22] based on the specified interaction interfaces (e.g., `ECALL/OCALL` in Intel SGX SDK [3]) are not considered. Nevertheless, SGXLock lays a foundation for the defense mechanism to mitigate such high-level attacks. We regard exploring the defense mechanisms for high-level attacks based on SGXLock as our future work.

SGX2 support Intel released the 2nd generation SGX in its IceLake platform, called SGX2. Compared with SGX1, SGX2 supports the dynamic enclave memory management. It provides three new run-time primitives for an enclave, including enclave page allocation/deallocation, enclave page permission (EPCM) restriction/extension, and dynamic thread (TCS) creation and destruction. Currently, our prototype implementation is based on SGX1. However, the design of SGXLock itself is compatible with SGX2. First, the page allocation and the dynamic threading management in SGX2 require the involvement of the OS. So the host application can be aware of the corresponding operations. Second, SGX2 enables an enclave to modify an enclave page's permission in the page's corresponding EPCM entry. It is orthogonal to SGXLock's access control mechanism, which relies on page tables and the PKRU register to perform the access permission control.

Legacy enclaves support The legacy enclaves can be supported at the source code and the binary level, respectively. If the source code is available, our prototype can keep backward-compatibility with minor or even no modification. The only possible required modification is to remove the `user_check` attribute from the enclave code. Our experience shows that it only requires a minor engineering effort (e.g. changing about 30 lines of code when porting the ML inference service). Besides, the use of `user_check` is not recommended. Because it is prone to the TOCTOU attack at design level. Similar discussions or opinions are also expressed in other work [36].

Legacy binary enclaves can be transparently supported if the `user_check` attribute or DLGC is not used by the enclave. Otherwise, embedding our inspection code into the enclave binary to support the DLGC auditing and removing the `user_check` attribute are needed. They can be implemented using the binary rewriting technique, but only by enclave developers. That is because the binary needs to be re-signed to satisfy the remote attestation. Note that while embedding inspection code can be easily implemented via enclave entry hooking (as described in Section 4.3.1), removing the `user_check` attribute *automatically* via binary rewriting is kind of challenging considering it depends on the enclave-specific semantic. Exploring it is regarded as our future work.

Impact to the debugging functionality Leveraging the single-step mode to eliminate control flow asymmetry affects the normal debugging function for neither the enclave nor its host application. As a security enhancement mechanism, SGXLock is used in the production environment, where debugging with the single-step mechanism is no need. At the development stage, since SGXLock is transparent to developers, it could be temporarily disabled (e.g., using unmodified uRTS) if then developers want to leverage the single-step mode to debug the program.

8 Related Work

Untrusted enclave confinement Several mechanisms have been proposed to confine the *untrusted* enclave. One proposal is to detect the malicious actions of an enclave by monitoring its I/O behaviors [14], which is not practical yet. Costan et al. also proposed a solution to perform the static analysis on the enclave code [14]. To deal with runtime generated code, it requires all enclaves to include a standardized static analysis framework into themselves. This brings two security concerns, i.e., the source credibility of the static analysis framework and the increased TCB introduced by the framework. SGXJail [36] confines the enclave's behaviors by residing each enclave in a separated sandbox process. It suffers from the scalability issue, especially for multiple enclaves and multi-threading scenarios. SGXLock does not have such an issue because of its in-process confinement design. Some systems [18, 29] apply the software fault isolation (SFI) technique into the enclave to confine user code inside the enclave. However, they do not support dynamically generated code inside the enclave. Our system does not have this limitation. Moreover, Zhang et al. [39] proposes a black-box detection framework, SGX-Bouncer, for SGX enclave malware. However, enclave malware could hide its attack behavior while it is aware of SGX-Bouncer to evade detection.

Usage of Intel MPK and SGX Intel MPK provides a hardware primitive to implement efficient intra-process isolation [17, 31, 32]. ERIM [32] leverages MPK to provide intra-process isolation for normal applications with low performance overhead. However, ERIM is not applicable to SGX scenarios due to the hardware isolation of SGX. The libmpk [24] system virtualizes MPK protection keys in software so that an unlimited number of protection keys are available. The libmpk system is orthogonal to our work and its

design about protection key virtualization can be borrowed by our system to implement the support of an unlimited number of enclaves in the future.

SGX has been leveraged to protect user code and data in different scenarios. For instance, VC3 [26] uses SGX to implement trustworthy data analytics in the cloud. LightBox [15] leverages SGX to provide a highly efficient solution for middleboxes in the cloud. To facilitate the usage of SGX, different software development kits (SDK) [2, 3, 5, 34] have been provided to the developers. Moreover, several library operating systems (libOS) [10, 12, 29] have been developed to minimize the effort of legacy code refactoring for SGX. However, all of these SDKs and libOSes have not considered the issue of untrusted enclaves. SGXLock can be integrated into them to address this issue.

Attacks towards SGX and security enhancements for SGX Researchers have proposed different classes of attacks towards SGX. Most of them are side-channel based attacks. A representative type of attack is controlled-channel attacks [11, 35, 37]. Another class of attacks is based on the software vulnerabilities inside the enclave [19, 21]. For instance, Khandaker et al. [19] systematically summarized the attack surface of the host-enclave interaction interfaces, from the host application's view.

Accordingly, security enhancement mechanisms [23, 30] have been introduced to enhance the security guarantees. To defend against controlled-channel attacks, T-SGX [30] relies on the TSX hardware feature to suppress the occurrence of page faults inside the enclave. Besides, some defense mechanisms [20, 28, 40] are proposed to address the vulnerabilities inside the enclave code. These security enhancement mechanisms can collaborate with SGXLock to strengthen the mutual distrust relationship between an enclave and its host application.

9 Conclusion

In this paper, we propose an efficient and scalable defense mechanism to confine an untrusted enclave's behaviors. The threats of an untrusted enclave come from the enclave-host asymmetries, i.e., the data access asymmetry and the control flow asymmetry. Our solution leverages two x86 hardware features, i.e., *Intel MPK and x86 single-step mode*, to break the data access and control flow asymmetries, respectively. We have implemented a prototype system. The evaluation shows the efficiency and effectiveness of our system.

Acknowledgement

We would like to thank the anonymous reviewers for their comments that greatly helped improve the presentation of this paper. We also want to thank Prof. Fengwei Zhang for shepherding our paper. Additionally, the first author of this paper would like to thank Xiaoxuan Lou personally for his help with the author's study. This work is partially supported by the National Natural Science Foundation of China under Grant 61872438, Leading Innovative and Entrepreneur Team Introduction Program of Zhejiang (2018R01005), the Fundamental Research Funds for the Central Universities (K20200019), Research Grants Council of Hong Kong under Grant R6021-20F and RFS2122-1S04. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of funding agencies.

References

[1] Apache http server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, Referenced September, 2020.

[2] Google asylo. https://github.com/google/asylo, Referenced August, 2020.

[3] Intel sgx sdk for linux. https://github.com/intel/linux-sgx, Referenced August, 2020.

[4] mbedtls-sgx. https://github.com/bl4ck5un/mbedtls-SGX, Referenced July, 2020.

[5] Microsoft open enclave sdk. https://github.com/openenclave/openenclave, Referenced August, 2020.

[6] Sgx-sqlite. https://github.com/yerzhan7/SGX_SQLite, Referenced July, 2020.

[7] Anonymous, 2019. https://github.com/anonymouspaper101/anonymouspaper101.

[8] Anonymous. Privacy-preserving machine learning as a service on sgx, 2019. http://www.humangenomeprivacy.org/2019/competition-tasks.html.

[9] Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. Sgxelide: Enabling enclave code secrecy via self-modification. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, 2018.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[11] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your secrets without page faults: Stealthy page table-based attacks on

enclaved execution. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[12] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *Proceedings of the 2017 USENIX Annual Technical Conference*, 2017.

[13] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2013.

[14] Victor Costan and Srinivas Devadas. Intel sgx explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[15] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. Lightbox: Full-stack protected stateful middlebox at lightning speed. In *Proceedings of the 26th ACM Conference on Computer and Communications*, 2019.

[16] Daniel Gruss, Dave Hansen, and Brendan Gregg. Kernel isolation: From an academic idea to an efficient patch for every computer. 2018.

[17] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *Proceedings of the 2019 USENIX Annual Technical Conference*, 2019.

[18] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[19] Mustakimur Rahman Khandaker, Yueqiang Cheng, Zhi Wang, and Tao Wei. Coin attacks: On insecurity of enclave untrusted interfaces in sgx. In *Proceedings of the 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[20] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the 12th European Conference on Computer Systems*, 2017.

[21] 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 *Proceedings of the 26th USENIX Security Symposium*, 2017.

[22] Marion Marschalek. The wolf in sgx clothing. https://www.troopers.de/troopers18/agenda/dldub8/, Referenced July, 2020.

[23] Meni Orenbach, Andrew Baumann, and Mark Silberstein. Autarky: Closing controlled channels with self-paging enclaves. In *Proceedings of the 15th European Conference on Computer Systems*, 2020.

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

[25] Christian Priebe, Kapil Vaswani, and Manuel Costa. Enclavedb: A secure database using sgx. In *Proceedings of the 2018 IEEE Symposium on Security and Privacy*, 2018.

[26] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. Vc3: Trustworthy data analytics in the cloud using sgx. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[27] Michael Schwarz, Samuel Weiser, and Daniel Gruss. Practical enclave malware with intel SGX. 2019.

[28] Jaebaek Seo, Byoungyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. 01 2017.

[29] 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 25th International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[30] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Network and Distributed System Security Symposium 2017*, 2017.

[31] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. Intra-unikernel isolation with intel memory protection keys. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2020.

[32] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Proceedings of the 28th USENIX Security Symposium*, 2019.

[33] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[34] 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*, 2019.

[35] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in sgx. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017.

[36] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. Sgxjail: Defeating enclave malware via confinement. In *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses*, 2019.

[37] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, 2015.

[38] Bingshen Zhang, Yuan Chen, Jiaqi Li, Yajin Zhou, Phuc Thai, Hong-Sheng Zhou, and Kui Ren. Succinct scriptable nizk via trusted hardware. In *Proceedings of the 26th European Symposium on Research in Computer Security*, 2021.

[39] Zeyu Zhang, Xiaoli Zhang, Qi Li, Kun Sun, Yinqian Zhang, Songsong Liu, Yukun Liu, and Xiaoning Li. See through walls: Detecting malware in sgx enclaves with sgx-bouncer. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, 2021.

[40] Wenjia Zhao, Kangjie Lu, Yong Qi, and Saiyu Qi. Mptee: Bringing flexible and efficient memory protection to intel sgx. In *Proceedings of the 15th European Conference on Computer Systems*, 2020.

A Challenges of adopting ERIM's design to block PKRU update inside the enclave

The solution proposed by the ERIM cannot be directly applied to our system for the following reasons.

```
1  /******** wrapper of WRPKRU ********/
2  xor  %ecx, %ecx
3  xor  %edx, %edx
4  mov  $PKRU_DISALLOW_TRUSTED, %eax
5  WRPKRU              // %PKRU <= %eax
6  //the inserted code is as follow
7  cmp  $PKRU_DISALLOW_TRUSTED, %eax
8  je   .Lcontinue
9  syscall  exit
10 .Lcontinue:
11 // execution continues...
12
13 /******** wrapper of XRSTOR ********/
14 XRSTOR
15 // the inserted code is as follow
16 bt  %eax, $0x9
17 jnc .Lcontinue
18 syscall exit
19 .Lcontinue:
20 // execution continues...
```

Listing 1: The wrappers of WRPKRU/XRSTOR in ERIM

CI-1) *The register value could be changed after WRPKRU/XRSTOR instructions.* ERIM leverages the wrappers for `WRPKRU/XRSTOR` instructions for a safe PKRU update. As shown in Listing 1, a few instructions are inserted after `WRPKRU/XRSTOR` instructions to check whether the PKRU register is updated as expected. The assumption is that the value of the `EAX` register between the `WRPKRU` instruction (line 5) and the check of the `EAX` value (line 7) cannot be changed. The same assumption applies to the `XRSTOR` (line 14) and the following check routine (line 16). However, this assumption does not hold in our scenario. When the enclave execution is interrupted on an AEX event and is going to exit the enclave, the program states of the execution would be stored into the enclave's SSA frame by hardware. The SSA frame resides inside the enclave and is software visible. Other enclave executions, if existing, then can corrupt the program states of the interrupted execution by modifying the content of the corresponding SSA frame.

CI-2) *The dynamically loaded code inside the enclave could be encrypted.* Besides, the dynamically loaded and runtime-generated code make the problem even harder. Some enclaves may load or generate code at runtime for the purpose of code secrecy [9]. For instance, an enclave developer may want to protect its proprietary code. So he/she chooses to deploy the proprietary code in the ciphertext. A small piece of the code loader is included in the enclave to load and decrypt the proprietary code after the enclave creation and the remote attestation. The hardware isolation of SGX prevents the host application's inspection mechanism, such as the interception mechanism proposed in ERIM [32], from inspecting the newly loaded or generated code.

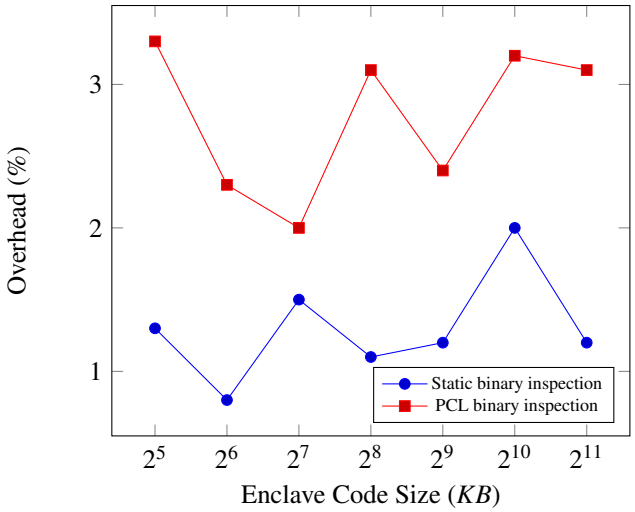

Figure 5: The performance overhead of static and PCL binary inspection with different enclave code size.

CI-3) *The occurrence of the XRSTOR instruction is unavoidable inside the enclave.* One intuitive thought is to forbid the occurrence of `WRPKRU/XRSTOR` instructions inside the enclave. However, SGX leaves the switching of the execution's program states to the software (i.e., enclave code) when the execution crosses the enclave boundary via the `EENTER/EEXIT` instruction. Thus, `XSAVE/XRSTOR` instructions are usually used by enclave to efficiently save/restore the extended states of the execution. The occurrence of the `XRSTOR` instruction is unavoidable inside the enclave.

B The overhead of static and PCL binary inspection

The static and PCL binary inspection mechanisms are only invoked when creating an enclave. To evaluate the performance overhead, we use Intel SGX SDK's *SampleEnclave* and *SampleEnclavePCL* in our evaluation, respectively. Specifically, we increase the enclave code size and record the corresponding enclave *creation* time (with or without static/PCL binary inspection). Figure B shows the result. We can see the static and PCL binary inspection mechanisms introduce no more than 2% and 4% performance overhead for the *enclave creation*, respectively.

C Kernel Modifications

Issues of the signal handler Whenever invoking a signal handler, the Linux kernel will override the access permission of the current thread to its initial setting (only access to the protection key 0). This indicates that the kernel assumes the (userspace) signal handler stack belongs to the protection key

```
1   // ECALL definition
2   public void ecall_pointer_in_size(
3       [in, size=len] void *ptr, size_t len);
4
5   // generated by Edger8r
6   // marshaling data structure for arguments
7   typedef struct ms_ecall_pointer_in_size_t {
8     void* ms_ptr;
9     size_t ms_len;
10  } ms_ecall_pointer_in_size_t;
11
12  // generated by our modified Edger8r
13  // wrapper data structure of ECALL/OCALL's
14  // marshaling data structure
15  typedef struct ms_buf_meta_t {
16      size_t offset;
17      size_t size;
18      int in_out;
19  } ms_buf_meta_t;
20  typedef struct ms_param_meta_t {
21      void*   ms;
22      size_t size;
23      ms_buf_meta_t* arr;
24      size_t arr_size;
25      size_t ret_offset;
26      size_t ret_size;
27  } param_meta_t;
```

Listing 2: The `ms_ecall_pointer_in_size_t` is the marshaling data structure generated by `Edger8r` according to the argument attributes of the `ECALL` definition (Line 3). The `ms_buf_meta_t` and `ms_param_meta_t` are wrapper data structures of `ECALL/OCALL`'s marshaling data structure to enable the uRTS to copy the parameters from/to the `param_buf`. Specifically, `ms_buf_meta_t` is used to describe the pointer member of the marshaling structure (e.g., Line 9). `ms_param_meta_t` provides the overall description of the marshaling structure.

0. However, during invoking a signal handler, the kernel will access the signal handler's stack (e.g., for frame setup) with the thread's original access permission (i.e. the PKRU register value). This brings up the functional issue (i.e. permission violation) when the thread has no permission to the protection key zero.

In our prototype, this can happen in two cases. First, the enclave execution exits on an AEX event, and a signal handler is then invoked (e.g., for the dynamic binary inspection). Second, the execution leaves the enclave via the `EEXIT` instruction and a signal-step debug exception is immediately triggered. The corresponding signal handler is invoked to check the jump target of the `EEXIT` instruction.

Solution We make minor modifications (no more 20 lines of code) to the Linux kernel. When accessing the signal handler stack, the kernel will be assigned with the access permission to the protection key 0 *temporarily*, no matter what its current permission is. Specifically, we mainly modify two functions.

- `handle_signal` Before it calls `setup_rt_frame` to set up the stack frame for the signal handler, the PKRU register is updated to enable the access to the protection key 0. The function `setup_rt_frame` is responsible for saving a copy of the current thread's context (including the PKRU register) into the signal handler's stack. Therefore, after returning from the `setup_rt_frame`, the function `handler_signal` updates the saved PKRU register in the copy with its actual value.
- `SYSCALL_DEFINE0(rt_sigreturn)` This is the system call to return from the signal handler and restore the original context. It calls the function `restore_altstack` after restoring the original context. Therefore, the PKRU register is updated to enable the access to the protection key 0 before invoking the `restore_altstack` function.

D Wrapper data structure for the Edger8r tool

The code snippet in Listing 2 shows the wrapper data structure for the `Edger8r` tool (Section 5).