A Hardware-Software Co-design for Efficient Intra-Enclave Isolation

Jinyu Gu, Bojun Zhu, Mingyu Li, Wentai Li, Yubin Xia, Haibo Chen
Engineering Research Center for Domain-specific Operating Systems, Ministry of Education, China
Institute of Parallel and Distributed Systems (IPADS), SEIEE, Shanghai Jiao Tong University

Abstract

The monolithic programming model has been favored for high compatibility and easing the programming for SGX enclaves, i.e., running the secure code with all dependent libraries or even library OSes (LibOSes). Yet, it inevitably bloats the trusted computing base (TCB) and thus deviates from the goal of high security. Introducing fine-grained isolation can effectively mitigate TCB bloating while existing solutions face performance issues. We observe that the off-the-shelf Intel MPK is a perfect match for efficient intra-enclave isolation. Nonetheless, the trust models between MPK and SGX are incompatible by design. We hence propose LIGHT-ENCLAVE, which embraces non-intrusive extensions on existing SGX hardware to incorporate MPK securely and allows multiple light-enclaves isolated within one enclave. Experiments show that LIGHTENCLAVE incurs up to 4% overhead when separating secret SSL keys for server applications and can significantly improve the performance of Graphene-SGX and Occlum by reducing the communication and runtime overhead, respectively.

1 Introduction

Trusted execution environment (TEE) has been a hot topic for both the architecture and security community over the past decade [3,9,14,16,26,30,37,42]. Intel SGX enclave, a widely-deployed TEE, can enhance both confidentiality and integrity for user-level code/data against untrusted software, including the privileged operating system and hypervisor, and becomes promising protection for secret data processing on the public clouds [17,19,31,32,39,44].

To enable SGX-based secret processing for developers, the official Intel SGX SDK [33] requires all libraries to be explicitly statically-linked within an enclave image. All the code and the secret data share the same address space, forming a large TCB. Another favored programming model is SGX-oriented LibOS [17,19,31,49,53,55] which loads unmodified binaries into the enclave for ease of development.

LibOSes also provide in-enclave system services (e.g., user-level scheduling [17], in-memory filesystem [31]). However, any vulnerabilities (for example, HeartBleed [25]) may tamper with the control flow of the enclave program and can even lead to secret breaches. The root cause is that Intel SGX enclave adopts a monolithic model which bloats the TCB.

To mitigate this problem, prior efforts have deconstructed a program into different independent enclaves [29, 31, 54]. Unfortunately, this choice introduces significant overhead due to secret transfer across enclave boundaries. Hence, intra-enclave isolation is explored by [15, 47, 51, 53]. However, these approaches encounter issues on either (in)flexibility or (in)efficiency. Some [15, 51, 53] instrument enclave memory access instructions to enforce bound checks and thus establish isolated domains within an enclave. Instrumentations cause non-negligible runtime overhead and enlarge the code size that can harm the locality. Moreover, their isolated domains must be contiguous in memory to make the bound checks feasible. New hardware proposals such as Nested Enclave [47] refrains from the above issues by modifying hardware to support inner and outer enclaves. Nevertheless, the overhead across inner-outer boundaries is still expensive.

This work aims at achieving intra-enclave isolation with both efficiency and security. We find that Intel MPK, an off-the-shelf hardware feature for partitioning an address space into multiple memory domains, can be a natural fit to facilitate intra-enclave isolation.

Challenges. Although applying MPK for SGX is a promising approach to efficiency because MPK incurs nearly zero overhead for memory access validation and both of them work in the user-level, it poses several security challenges. On the one hand, MPK requires trusting the underlying OS to faithfully configure the page table with correct domain IDs, which involves a conflict trust model against SGX. An untrusted OS can easily modify the domain-IDs or disable the MPK check to violate the isolation. On the other hand, MPK provisions the user-level instruction *WRPKRU* for changing the domain access permission. A compromised entity within the enclave may exploit this to bypass the intra-enclave isolation.

Thereby, the core technical challenge is how to securely use MPK within an SGX enclave for intra-enclave isolation while facing the untrusted OS (manipulating the page table) and malicious/compromised entities in the enclave (manipulating the domain access permission).

Our proposal. We propose a hardware-software co-design, LIGHTENCLAVE, for secure and efficient intra-enclave isolation. It provides a lightweight and flexible abstraction of *light-enclave* and allows constructing multiple isolated *light-enclave*s within one SGX enclave. It includes non-intrusive hardware extensions to solve the trust model conflict between SGX and MPK, and provides a friendly programming model compatible with existing development processes. The OS retains its capability of configuring MPK domain-ID in the page table while being deprived of the ability to arbitrarily modify the MPK configuration since the extended SGX hardware will validate the domain-ID during both initialization and runtime. With LIGHTENCLAVE, enclave developers can assign different memory domains to different light-enclaves and enforce the privilege separation. To prevent a light-enclave from escalating its own privilege (e.g., by abusing *WRPKRU*), LIGHTENCLAVE combines binary inspection and carefully-designed *light-enclave-gates*.

LIGHTENCLAVE has the following advantages:

- Compared with traditional multi-enclave isolation [29, 31, 54]: LIGHTENCLAVE provides more efficient communication. Control flow transfers are through lightweight *light-enclave-gates* instead of exiting/reentering hardware enclaves. Data sharing leverages secure shared domain rather than data re-encryption through unprotected memory.

- Compared with the instrumentation-based approaches [15, 51, 53]: LIGHTENCLAVE utilizes hardware-enforced domain permission check instead of bounds checking, which incurs nearly zero overhead for memory isolation and imposes no requirement of continuous domain region.

- Compared with the pure-hardware approach [47]: LIGHTENCLAVE is more flexible owing to hardware-software co-design. The underlying hardware offers the mechanism of partitioning enclave domains while software manages a flexible abstraction of light-enclave. Light-enclaves can be either mutually-distrusted or hierarchical.

As SGX is not open source, we validate the hardware extensions of LIGHTENCLAVE on the official SGX emulator (Intel SGX SDK simulation-mode) [33] except one added check on enclave memory access since the emulator does not emulate it (one limitation of our work). We implement the software designs of LIGHTENCLAVE and apply them to two state-of-the-art SGX LibOSes, Graphene-SGX [55] and Occlum [53]. Similar to [39, 47], we conducted the performance evaluation on the real SGX machine by adding the estimated performance overhead of our hardware proposal since the emulator shows much better performance than the real SGX due to no emulation of the memory encryption engine. For Graphene-SGX, LIGHTENCLAVE achieves $10.5\times$ speedup for CPU-intensive workloads and $46.5\times$ for multi-tasking intensive workloads; For Occlum, the speedups are $1.49\times$ and $1.28\times$, respectively. Besides accelerating SGX LibOSes, LIGHTENCLAVE can provide higher security for server applications by deconstructing them into components for isolating untrusted ones, which incurs less than 4% overhead. For privacy-preserving serverless applications, LIGHTENCLAVE decreases the latency by 50% to 77% for on-demand function startups.

Contributions. (1) A proposal of hardware extensions for how to securely use MPK in an SGX enclave. (2) An easy-to-use abstraction named light-enclave for intra-enclave isolation. (3) A preliminary evaluation to show LIGHTENCLAVE's performance benefit in different cases.

2 Background

2.1 Intel SGX

SGX protects user-level code/data by providing hardware-enforced trusted execution environments dubbed *enclaves*.

Secure memory. SGX reserves some DRAM as secure memory called Enclave Page Cache (EPC) for storing enclave memory pages. CPU tracks the metadata of each EPC page through Enclave Page Cache Map (EPCM). EPCM also resides in the secure memory, holds one entry for each EPC page, and each entry contains the read, write, execute permission, the mapped virtual address, the owner enclave, etc. For enclave memory accesses, memory management unit (MMU) will check not only the page table information but also the EPCM information (e.g., an EPC page can only be accessed by its owner enclave).

Enclave creation. Enclave creation instructions are privileged and thus are executed by the OS. Since SGX does not trust OS, CPU records a measurement during an enclave creation. *ECREATE* instruction creates an enclave's SECS which contains the enclave metadata and is located in EPC. *EADD* instruction adds one EPC page to the enclave. When executing EADD, an argument structure named PAGEINFO is needed for provisioning information like the page content and permission. *EINIT* finishes the creation. A remote user can ask for the enclave measurement and use the attestation service to examine the enclave construction.

Enclave execution. A thread executes unprivileged *EENTER* or *EEXIT* instructions for entering or exiting from the enclave, respectively. When entering the enclave, a thread exclusively occupies an enclave TCS (one EPC page) which designates one fixed entry point and the State Save Area (SSA) for the execution. The enclave execution can be interrupted by exceptions or interrupts. If so, CPU performs Asynchronous Enclave Exit (AEX) which saves the execution context into the SSA within the enclave, scrubs the context, etc. The enclave execution can be restored by *ERESUME* instruction. In

SGXv2, dynamic enclave memory management is allowed. An EPC page can be added to a running enclave through the cooperation of the enclave and the OS: the OS executes *EAUG* instruction for adding one page, and then the enclave executes *EACCEPT* or *EACCEPTCOPY* instruction for finishing the adding procedure.

2.2 Intel MPK

MPK and memory domains. MPK allows an application to partition its virtual address space into 16 different memory domains. Each memory page can be associated with a 4-bit domain-ID by storing it in four previously reserved bits of the corresponding page table entry.

MPK register and instructions. There is a per-core register named PKRU which specifies the access permission (read-only, read-write, none) to different domains for the CPU core. The register has 32 bits, and every two bits represent the access permission to one domain. Two unprivileged instructions, *WRPKRU* and *RDPKRU*, can be used to modify and read PKRU, and both usually take less than 30 CPU cycles [28, 48, 56]. MMU transparently enforces the MPK checks that incur almost zero runtime overhead. Besides, MPK has no effects on the execution permission of memory pages, even if PKRU forbids the read permission.

MPK interaction with SGX. MMU supports applying MPK domain permission checks to enclave memory accesses in addition to original SGX memory permission checks. The PKRU register can also be automatically saved during AEX and restored by *ERESUME*.

3 Overview

SGX provides trusted execution environments called enclaves in an application's address space to protect security-sensitive code/data. To minimize the code-refactoring efforts and avoid the performance penalties caused by the decomposition of applications, there is a popular programming trend that running the whole application with the third-party libraries [17, 33, 54] and even a library OS [19, 31, 55] inside a single SGX hardware enclave, which, however, bloats the TCB and endangers sensitive code/data. For example, once a third-party library containing vulnerabilities is imported into an enclave, attackers may leverage the vulnerabilities to tamper with the integrity and even confidentiality of the enclave.

We propose LIGHTENCLAVE, which brings MPK-based intra-enclave isolation to an SGX enclave, to isolate secure-sensitive code/data from untrusted ones. Specifically, it supports building multiple *light-enclaves* within one hardware enclave, as depicted in Figure 1.

Light-enclaves. LIGHTENCLAVE can partition the enclave memory into different memory domains by marking the (MPK) domain-IDs in the page table entries of enclave pages. A light-enclave can exclusively occupy one memory domain,
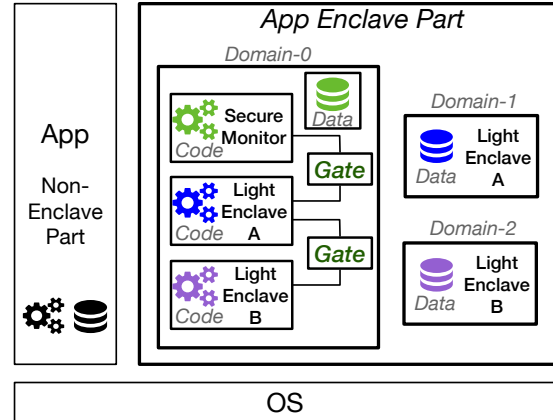

Figure 1. LIGHTENCLAVE supports constructing multiple mutual-distrusted light-enclaves within an SGX enclave.

named *private domain*, to store its private data, e.g., light-enclave-A and light-enclave-B take domain-1 and domain-2, respectively. By default, a light-enclave only has the access permission of its private domain, which means different light-enclaves are mutually distrusted. Nonetheless, LIGHT-ENCLAVE also allows one light-enclave to have a higher privilege than another, i.e., one light-enclave can access the other one's private domain but not vice versa. LIGHTENCLAVE accommodates a light-enclave's data, stack, and heap in its private domain while placing its code in domain-0. A light-enclave can never acquire the access (read/write) permission of domain-0 and thus cannot modify the code. However, it can execute the code in domain-0 normally since the MPK domain isolation enforces no restriction on the execution permission. Therefore, the code is *execute-only* for each light-enclave.

The domain access permission of a light-enclave is its unique identity. LIGHTENCLAVE ensures that the PKRU register always stores its identity during the light-enclave's execution and thus prevents it from accessing other domains. Note that LIGHTENCLAVE also deprives light-enclaves of their ability to illegally modify PKRU. § 5.2 explains how LIGHTENCLAVE achieves this.

Secure monitor. Each hardware enclave contains a secure monitor. As its name indicates, the secure monitor is considered trustworthy and can access all memory domains. Its code and data are both in domain-0 and thus inaccessible to light-enclaves. It works as the manager of the hardware enclave and has responsibilities including creating new light-enclaves at runtime and dynamic enclave memory management.

Usage model. The abstraction of light-enclave enables programmers to apply the *principle of least privilege* in an enclave. For example, the secure-sensitive code can run in one light-enclave while the other libraries are located in another light-enclave, each light-enclave only having the necessary permission. By extending the official Intel SGX SDK, LIGHTENCLAVE allows programmers to separate code/data into

different light-enclaves and declare the interfaces between each other, similar to existing SGX programming. It *automatically* partitions the enclave memory, merges the light-enclaves into one hardware enclave, and generates *light-enclave-gates* for their interaction. A light-enclave-gate can efficiently transfer the control flow between two light-enclaves (switching the execution contexts) as well as switch the identity (domain access permission), detailed in § 5.2. Besides static construction, both light-enclaves and light-enclave-gates can also be built during runtime by the secure monitor. § 5.1 describes more about the programming model.

Incompatible trust model between SGX and MPK. To use MPK in an SGX enclave for memory isolation, the enclave pages' page table entries should be tagged with different domain-IDs. Since the page table is controlled by the OS, LIGHTENCLAVE has to ask the OS to set the desired domain-IDs. The implicit assumption of MPK is that the OS is trusted and will faithfully configure the domain-IDs in the page table. However, the OS is usually considered untrusted for SGX enclaves. Thereby, using existing MPK-based memory isolation in an SGX enclave is unreliable. Specifically, a compromised OS can collude with some malicious/compromised light-enclave to break the isolation boundary between different light-enclaves. To solve this security challenge, we propose non-intrusive SGX hardware extensions on validating the domain-IDs in the page table. § 4 gives concrete attack examples and the corresponding secure extensions. It will also illustrate the security challenges and solutions related to dynamic enclave memory management.

Threat Model. LIGHTENCLAVE inherits the threat model of SGX and thus assumes an adversary can take full control of all the software (including the OS) except SGX enclaves. Besides, it does not assume all the code inside an enclave is trusted. From the perspective of one light-enclave, it needs to trust the SGX hardware, the secure monitor, and the light-enclaves with higher privilege than it (if existed); It does not need to trust any other software, including other light-enclaves in the same SGX enclave; Yet, it still needs to ensure no secret leakage to potentially malicious dependencies during the interaction because LIGHTENCLAVE assumes each light-enclave does not expose its secrets and takes no step forward to eliminate such software bugs. LIGHTENCLAVE also does not consider hardware bugs [22, 34, 36, 45] or side-channel attacks [20, 27, 38].

4 Hardware Extensions

LIGHTENCLAVE is a hardware-software co-design for efficient intra-enclave isolation. This section introduces the proposed hardware extensions. We first analyze the possible attacks when naively combining MPK and SGX in the current hardware design, which further motivates us to improve it with minimal hardware modifications.

4.1 Attacks due to Incompatible Trust Model

Suppose one enclave contains two mutually distrusted light-enclaves, light-enclave-A and light-enclave-B, and the former contains a secret key in its private domain (domain-1), while the latter runs a library which contains a vulnerability and might be compromised. Although light-enclave-B runs in the same enclave address space as light-enclave-A, Naive MPK-based LIGHTENCLAVE ensures that light-enclave-B has no access permission of other domains (i.e., domain-1) and thus cannot retrieve light-enclave-A's secret key. Besides the insider attacker (light-enclave-B), the untrusted OS, as an outsider attacker, also cannot access the secret key since the key is protected in the SGX enclave.

However, the outsider and insider attackers can *collude* to steal the secret key. Specifically, light-enclave-B first attempts to read the secret key and thus triggers a page fault because such memory access violates the domain access permission. Then, in the page fault handler, the untrusted OS can modify the domain-ID of the faulting page to the private domain of light-enclave-B (e.g., domain-2) in the page table. Afterward, the execution of light-enclave-B can be resumed, and now it can successfully read the secret key. Later, the untrusted OS can also restore the domain-ID to make the victim light-enclave-A unaware of the attack.

In this way, even if a malicious light-enclave cannot acquire the access permission of other light-enclaves' private domains, the untrusted OS can modify the domain-IDs in the page table and thus help the malicious light-enclave to access any sensitive data. Besides, the untrusted OS can also directly disabling the MPK feature before the colluding light-enclave executes, which can also break the isolation between different light-enclaves. On a real machine with SGX and MPK, we have successfully launched attacks in both the above ways.

4.2 Secure Domain Access

The key reason why the above attacks can succeed is that the trust models of MPK and SGX conflict. The effectiveness of MPK relies on the OS to correctly set the domain-IDs in the page table entries, whereas the OS is untrusted in SGX. Hence, LIGHTENCLAVE proposes hardware extensions to solve this security conflict.

The high-level idea is to preserve the OS's ability to configure the domain-IDs in the page table while validating the settings of domain-IDs at both initialization time and runtime. The behavior of maliciously modifying domain-IDs of the enclave pages will be detected by MMU during address translation and then stop the hardware enclave execution. The hardware extensions should achieve the following three security properties.

- *Security-property-1:* Upon creation time, the domain-ID of each enclave page should be included into the enclave's measurement for attestation.
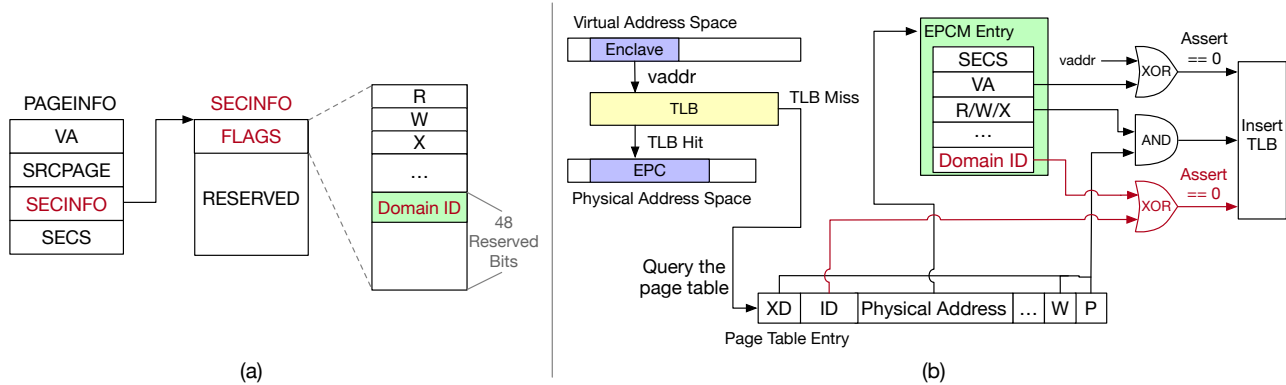
Figure 2. (a) Add the MPK domain-ID in the reserved bits of SECINFO. (b) Validate the MPK domain-ID during enclave memory address translation.

- *Security-property-2:* During runtime, the OS cannot change the domain-ID of an enclave page.
- *Security-property-3:* the MPK feature cannot be disabled during the enclave execution if requested during creation time.

Including the domain-ID into an enclave page's secure metadata. During the enclave creation, the OS executes *EADD* instruction to add an EPC page to the enclave. *EADD* takes a PAGEINFO structure as an argument. As shown in Figure 2(a), PAGEINFO contains four fields that specify both the metadata and the data for the enclave page to add. The third field points to an SECINFO structure that contains the enclave page information such as the read, write, execution permission. *EADD* not only records such metadata in the EPC page's EPCM entry, as depicted in Figure 2(b), but also leverages the metadata to update the enclave measurement. To meet *Security-property-1*, the MPK domain-ID should also be treated as the secure metadata of an enclave page.

We notice that the SECINFO structure has enough reserved space for adding the domain-ID information. Specifically, its FLAGS field has 64 bits while the last 48 bits are unused. So, the domain-ID, which only consists of several bits (e.g., 4 bits can represent 16 domains), can be added here. Besides SECINFO, the EPCM entry of an enclave page should also add an extra field for storing the domain-ID. Two changes are made to *EADD*: it will also update the enclave measurement according to the domain-ID; it will also record the domain-ID in the EPCM entry. Therefore, after an enclave is built, a remote user can attest whether the domain-ID of each enclave page is correctly set (i.e., meets *Security-property-1*).

Validating the domain-ID during address translation according to EPCM. When an enclave thread accesses some enclave address (*vaddr*), MMU translates *vaddr* into the physical address by first searching the corresponding TLB entry and then querying the page table (upon TLB misses). Since the page table is controlled by the untrusted OS, MMU will further check against the EPCM for the security of translation, as shown in Figure 2(b). Specifically, after retrieving the physical address from the page table entry, MMU locates the corresponding EPCM entry indexed by the physical address and validates the legality of the enclave memory access. A legal access requires: the running enclave matches SECS field in the EPCM, *vaddr* matches VA field in the EPCM, etc. Note that the domain-ID of an enclave page has already been recorded in the EPCM during the execution of *EADD*. To meet *Security-property-2*, MMU will further check the domain-ID retrieved from the page table entry against the domain-ID stored in the EPCM. The equality of the two values is a new necessity for a legal enclave memory access. For legal accesses, MMU will cache the address translation in the TLB as before. No modifications on the structures of TLB or page table are required.

By adding the validation, although the OS can still arbitrarily alter domain-ID of enclave pages in the page table, this dishonesty will be detected upon EPC page accessing. Thereby, the OS can no longer conduct the above-mentioned collusion attack by changing the domain-ID.

Saving and restoring domain-ID during EPC page swapping. SGX introduces specialized instructions for the OS to swap EPC pages. The *EWB* instruction is responsible for encrypting an EPC page, dumping the encrypted content to the normal memory, and generating PCMD that contains the Message Authentication Code (MAC) of the swapped-out EPC page. Because the EPCM entry of the swapped-out page will be cleared and reused, *EWB* should also save the domain-ID in PCMD just like saving other secure metadata and take it as an extra input for generating the MAC. When the OS executes swap-in instructions like *ELDU* to load back enclave pages, the instructions can ensure the domain-ID remains unchanged during EPC page swapping by verifying the MAC.

By checking the newly added domain-ID in EPCM during enclave address translation and keeping the domain-ID intact during enclave page swapping, the OS cannot change

the domain-ID of enclave pages without being detected. So, *Security-property-2* is met.

Checking whether MPK is enabled during enclave transition. Whether MMU enables MPK check refers to the PKE bit of the control register, i.e., CR4.PKE. The untrusted OS can clear CR4.PKE and thus disable the MPK check for the enclave (rendering intra-enclave isolation useless), without being noticed by the enclave. To solve this security problem, the hardware should check whether MPK is enabled as required *just* before the execution of an enclave. Specifically, one new bit (PKE) is added in SECS for specifying whether the enclave desires MPK domain permission check. *EENTER*, the instruction for starting the enclave execution, additionally checks whether CR4.PKE is set when PKE in SECS is set. If not, it refuses to let the enclave run. The same check is also needed in *ERESUME* that is for resuming the enclave execution. The OS has no way to clear CR4.PKE when an enclave executes since the CPU runs in enclave mode (the OS is not running), so adding the check at enclave entry points is enough for meeting *Security-property-3*.

4.3 Dynamic Enclave Page Management

Since SGXv2 supports dynamic enclave memory management, LIGHTENCLAVE further introduces more hardware extensions to be compatible with this flexible feature.

With SGXv2, the OS can execute *EAUG* instruction to add an enclave page to a running enclave. The page cannot be used until the enclave issues *EACCEPT* or *EACCEPTCOPY* (we name them *accept instructions* for short) to accept it. Similar to *EADD*, *EAUG* also takes *PAGEINFO* (contains *SECINFO*) as one parameter. Thereby, when executing it, the domain-ID of the page should also be specified in *SECINFO*. Then, an enclave can leverage *accept instructions* to check whether the OS specifies its desired domain-ID. By extending these three instructions, an enclave can still rely on the untrusted OS to add pages on the fly without breaking MPK-based intra-enclave isolation.

We also consider modifying the domain-ID of an enclave page dynamically from the perspective of flexibility. SGXv2 already provides *EMODPE* for an enclave to proactively extend the access permission of pages. Since *EMODPE* also takes *SECINFO* as one parameter and updates EPCM accordingly, it can be reused to update the domain-ID of some enclave pages in the corresponding EPCM. Besides executing *EMODPE*, an enclave still needs to inform the OS about domain-ID modification and then the OS can help set the new domain-ID in the page table.

However, there exist two more attack vectors. The first is caused by adding new pages. The OS may utilize *EAUG* to add new enclave pages: one TCS page (i.e., adding a new enclave entry point) and some malicious code pages. A compromised light-enclave can adopt these pages through executing *accept instructions*. Then, the OS can let one thread enter

the enclave through the new TCS with any PKRU value (any domain access permission) and execute the malicious code. Eliminating *accept instructions* in untrusted light-enclaves cannot close this attack vector. This is because such instructions must exist in the enclave (e.g., secure monitor) if dynamic paging is needed and a malicious light-enclave is still possible to execute them by using Return-Oriented Programming (ROP). Even if there are identity checks immediately after the *accept instructions*, one malicious thread (T) has already accepted the new pages for other malicious threads. The vulnerable window is open before T is caught. More seriously, the untrusted OS can frequently interrupt T aiming to enlarge the window. The second arises from modifying domain-IDs. A light-enclave can modify the domain-IDs (set to its private domain-ID) of other light-enclaves' pages by executing *EMODPE* and asking the OS to modify the domain-IDs filled in the page table. Then, it can access any sensitive page.

Authorized dynamic paging based on privilege separation. Original SGX enclave lacks the mechanism of privilege separation. For example, all the enclave threads can always access the same memory pages and execute the same instructions. LIGHTENCLAVE aims to build isolated light-enclaves within one hardware enclave. The above-proposed SGX extensions make it possible to enforce different memory access permissions within one hardware enclave. Nevertheless, missing the hardware capability of restricting instruction execution leads to the above two attack vectors.
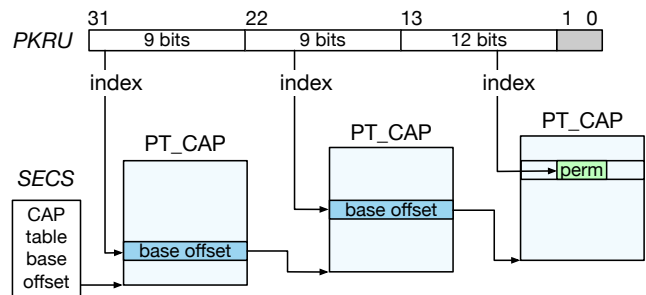

Figure 3. The structure of per-enclave capability table.

We propose a per-enclave capability table to specify which *ENCLU* instructions (SGX user-mode instructions) can a light-enclave execute. Specifically, each light-enclave has a unique identity stored in the PKRU register when it executes. The capability table uses the light-enclave identity (the PKRU register) as the index and stores the instruction permission of the identity.

There can be up to 2^{32} identities since PKRU has 32 bits. To minimize the table's memory space, we refer to the page table structure. As shown in Figure 3, the capability table has three levels. The first level has only one page whose address is specified in the enclave's *SECS*. The first-level page can contain up to 512 entries which point to 512 second-level

Procedures	Effects	Brief Description of the Extensions
Initialization	*EADD*	Set the page's domain-ID in EPCM according to the new *SECINFO*.
	ECREATE	Include the new fields in *SECS* into the enclave measurement.
Swapping	*EWB*	The domain-ID of an enclave page will be used to update the hash (*PCMD.MAC*).
	ELD	Ensure the domain-ID of a loaded-back page is unchanged.
Dynamic Paging	*EAUG*	Similar to *EADD*. Set the domain-ID for a new enclave page.
	EMODPE	Change the domain-ID of an enclave page in the EPCM at runtime.
	EACCEPT(COPY)	Check whether a page's domain-ID in EPCM matches the desired one.
Runtime	*EENTER/ERESUME*	Check whether MPK is enabled as required in *SECS*.
	Memory Access	Ensure the domain-IDs in the EPCM and in the page table match before accessing.
	ENCLU Execution	Check the capability table when executing sensitive *ENCLU* instructions.

Table 1: Major SGX extensions introduced by LIGHTENCLAVE

pages. Similarly, one second-level page can point to 512 third-level pages. Each table entry stores the relative offset from the enclave base address. One PKRU value is divided into four parts: the first two parts (each part takes 9 bits) are used as indexes to locate next-level pages; the third part (12 bits) is to locate the 1-byte permission in the last-level page; the fourth part (the least significant 2 bits) should both be 1, indicating light-enclaves cannot access domain-0. If the last 2 bits of PKRU are not 1 (i.e., secure monitor), the capability table takes no effect. Every used light-enclave identity corresponds to one 1-byte permission, one bit meaning whether a *sensitive ENCLU* instruction can be executed. Such instructions are *EACCEPT*, *EACCEPTCOPY*, *EMODPE*, and *EGETKEY* because they are related to security (the last one is used to retrieve the enclave encryption key). The rest four bits of the permission are reserved. The type of capability table pages is *PT_CAP*, which is a new enclave page type and writable to the privileged secure monitor.

There are over 3,000 reserved bytes in *SECS*, which is enough for storing the base offset of the capability table. Besides, considering the backward compatibility, *SECS* could include an extra control flag to configure whether to enable the capability table. The content of the capability table should also be included in the enclave measurement for attestation. When light-enclaves execute sensitive instructions, the CPU transparently checks the capability table to avoid arbitrary execution of sensitive instructions. Although checking the capability table may involve several memory accesses, such sensitive instructions are infrequently executed and usually executed during complex operations like dynamic paging. Thus, this security enhancement will barely cause a performance slowdown. Moreover, the multi-level design of the capability table can scale to a wider PKRU (support more memory domains) and more sensitive instructions.

4.4 Hardware Extensions Summary

Table 1 summarizes the hardware extensions introduced by LIGHTENCLAVE. First, to securely leverage MPK-based memory isolation inside an SGX enclave, LIGHTENCLAVE deprives the ability of arbitrarily modifying domain-IDs from the untrusted OS through recording the domain information during enclave initialization and letting the MMU automatically check the information for memory accessing during enclave runtime. Second, to keep the domain-IDs unchanged during enclave page swapping, the swapping-related SGX instructions save and restore the domain-IDs when evicting and reloading enclave pages. Third, to support dynamic paging and domain-ID modification, *EAUG*, *EMODPE*, and *accept instructions* are extended to specify the domain-IDs of enclave pages. Fourth, security checks are also added to prevent the untrusted OS from disabling MPK when entering a hardware enclave and allow fine-grained privilege separation according to a per-enclave capability table when executing four sensitive *ENCLU* instructions. Last but not least, our hardware extensions are inspired by current SGX implementation, namely, microcode, which is feasible to be integrated into existing SGX hardware designs.

5 Software Design

The software part of LIGHTENCLAVE contains code outside and inside an enclave. The former is an extension based on the existing Intel SGX SDK, which is responsible for generating the enclave memory layout (with the pages' domain-IDs) specified by programmers and some glue code for easing the development of light-enclaves. The latter is mainly about the secure monitor and light-enclave-gates. § 5.1 introduces the programming model. Based on the hardware extensions, the privilege of a light-enclave is determined by the specific PKRU value to which it binds. To enforce the privilege separation, LIGHTENCLAVE must further ensure a light-enclave is always bound to the unique PKRU (§ 5.2).

5.1 Programming Model

As shown in Figure 4, LIGHTENCLAVE allows building either mutual-distrusted light-enclaves. For instance, light-enclave-
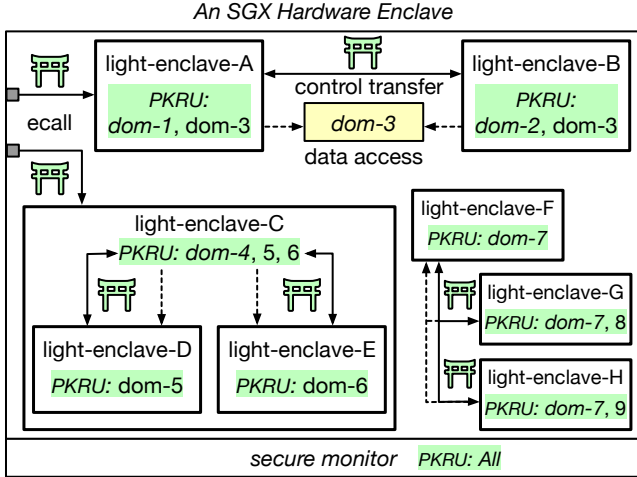
Figure 4. Logical view of light-enclaves: they can be mutually distrusted or have hierarchical organization.

A and light-enclave-B are mutually distrusted, and their private memory domains are domain-1 and domain-2, respectively. To facilitate data exchange between them, domain-3 is allocated as a shared memory domain for them. Thereby, light-enclave-A can access domain-1 and domain-3 while light-enclave-B can access domain-2 and domain-3. To support interaction between them, LIGHTENCLAVE can designate a *light-enclave-gate* that is responsible for transferring the control flow between the two light-enclaves (changing the identity of light-enclave).

LIGHTENCLAVE inherits and extends the original SGX programming model. In the original model, enclave developers use the interface definition language (IDL) [33] to specify *ecall*s and *ocall*s. The former ones are interfaces used by the untrusted application part to invoke functions provided by the enclave. The latter ones are reverse. In LIGHTENCLAVE, the light-enclave developers can still use the same IDL for defining the interfaces between the light-enclave and the untrusted application part. Nevertheless, when automatically generating the trampoline code for the interface, LIGHTENCLAVE ensures the PKRU register is properly set to the light-enclave's identity (declared by developers) before transferring the control flow into the light-enclave.

For the interfaces between different light-enclaves, LIGHTENCLAVE allows developers to use IDL similarly. A light-enclave can expose two types of interfaces: one is public (any light-enclave can invoke), the other is provided for another specific light-enclave. In contrast, the interface provided by an original SGX enclave (*ecall*) is always callable. For data transferring between an enclave and the outside-enclave part, the enclave should always be the data mover because it can access both the inside-enclave and outside-enclave memory. Nevertheless, LIGHTENCLAVE allows more data movement mechanisms, e.g., shared memory between two light-enclaves.

Two light-enclaves (e.g., light-enclave-A and light-enclave-B) can directly exchange data through the shared memory (e.g., domain-3) since it is inaccessible by the untrusted application part and other light-enclaves. Compared with two hardware enclaves, the interaction between two light-enclaves is more efficient for both control and data transfer.

Mutual-distrusted light-enclaves have some potential usage scenarios like accommodating different third-party libraries or different processes of a multi-process application. In addition, LIGHTENCLAVE allows hierarchical light-enclaves. As demonstrated in Figure 4, light-enclave-C takes domain-4 as its private domain while having access to domain-5 and domain-6 that are the private domains of D and E. In this case, C is more privileged than both D and E and can directly access their private memory, which may be suitable for auditing or secure multi-party computation scenarios. Another hierarchical case is light-enclave-F, G, and H. Their private domains are domain-7, 8, and 9, separately. In this case, G and H can be considered as more secure execution environments than F, which is similar to an application (F) creating two hardware enclaves (G and H).

In brief, compared to the original SGX programming model, LIGHTENCLAVE shares the similarity of using IDL for declaring the interfaces, which can benefit adoption and usability. LIGHTENCLAVE requires developers' minor efforts to declare the light-enclave identity, i.e., the memory access permission. Furthermore, developers can declare the instruction execution permission as well. A light-enclave cannot execute sensitive instructions by default. LIGHTENCLAVE abstracts away other details such as memory partitioning through *automatically* setting the domain-IDs, generating light-enclave-gates, and configuring the capability table.

5.2 PKRU Binding

LIGHTENCLAVE must prevent unauthorized modifications of the PKRU register. Otherwise, a malicious light-enclave can promote its privilege by modifying PKRU to access others' memory or execute disallowed instructions. To this end, LIGHTENCLAVE deprives light-enclaves of their ability to (arbitrarily) change the PKRU register. There are four ways a light-enclave may get an illegal PKRU value and thus achieve privilege escalation. LIGHTENCLAVE prevents all of them.

First, *EENTER* instruction does not change the PKRU register so that a light-enclave can inherit the PKRU value configured outside the SGX enclave. To avoid it, LIGHTENCLAVE configures the PKRU register before transferring the control flow to a light-enclave. Specifically, *EENTER* only transfers the control flow to fixed enclave entries specified by TCS and thus LIGHTENCLAVE can carefully set the PKRU register at these entries (inside the hardware enclave) for *ecall*s targeted light-enclaves (LIGHTENCLAVE generates the related code). Thereby, a light-enclave cannot inherit the PKRU value set outside the enclave (before invoking *EENTER*). Besides, since

EEXIT instruction also does not change the PKRU register when the control flow transfers from SGX enclave inside to outside, LIGHTENCLAVE saves the outside PKRU at entries and restores it at exits.

Second, the *AEX* (Asynchronous Enclave Exit) procedure automatically saves PKRU in SSA (State Save Area), and *ERESUME* restores it. If the saved PKRU in SSA is maliciously manipulated, a light-enclave may get a manipulated PKRU value after being resumed by *ERESUME*. LIGHTEN-CLAVE prevents this by making all the SSA pages reside in domain-0. Domain-0 is only accessible to the trusted secure monitor and inaccessible to all the light-enclaves. In other words, malicious light-enclaves cannot modify the PKRU value saved in the SSA. It is noted that the *AEX* procedure can always access SSAs regardless of MPK checks.

Third, *WRPKRU* instruction is specialized for modifying PKRU, and light-enclaves may execute it to change PKRU and achieve higher privilege. To prevent this, LIGHTENCLAVE leverages binary scanning and rewriting to guarantee there is no *WRPKRU* instruction in the code of light-enclaves, similar to [28, 56]. Since x86 instructions have variable lengths, *WRPKRU* could appear as a part of long instructions or span several instructions. LIGHTENCLAVE scans the binary code byte-by-byte to locate WRPKRU, and if exists, it replaces the related instructions with semantically-identical ones. Thus, compromised/malicious light-enclave cannot find and execute illegal WRPKRU instructions even with return-oriented programming (ROP).

```
... // save and clear the caller's state
1  mov $SECRET_TOKEN, %r15
2  xor %ecx, %ecx
3  xor %edx, %edx
4  rdpkru
5  cmp $PKRU_CALLER, %rax
6  jne handle_abuse
7  mov $PKRU_CALLEE, %eax
8  WRPKRU
9  cmp $SECRET_TOKEN, %r15
10 jne handle_abuse
11 xor %r15, %r15
... // callee executes
```

Figure 5. A light-enclave-gate example.

Nevertheless, executing *WRPKRU* instruction is necessary for the interaction between different light-enclaves as they have different PKRU values. LIGHTENCLAVE utilizes light-enclave-gates for the interaction. A light-enclave-gate is a piece of code generated by LIGHTENCLAVE and contains *WRPKRU*. Figure 5 shows an example of the light-enclave-gate. Generally, the gate saves the execution states of the caller light-enclave, switches the light-enclave identity to the callee, and restores the execution states of the callee light-enclave (e.g., let the callee execute the corresponding function). The gate is designed to enforce two security requirements: A light-enclave *req-1*) cannot abuse the *WRPKRU* instruction for privilege escalation, *req-2*) cannot arbitrarily use one gate for invoking other light-enclaves. Specifically, Line-2, Line-3 and Line-7 prepare the parameters for the *WRPKRU* instruction as it requires that *eax* stores the new PKRU value and *ecx* and *edx* are both 0. Line-1 and Line-9 can ensure their wrapped code piece must be executed line-by-line. This is because *SECRET_TOKEN* is unknown to light-enclaves. If a light-enclave wants to pass the check at Line-9, it must execute Line-1 first and thus go through the whole code piece, which guarantees Line-8 (*WRPKRU*) cannot be abused (e.g., a malicious light-enclave cannot directly jump to Line-8 with an illegal PKRU value in *eax*) (meeting *req-1*). Light-enclaves cannot know *SECRET_TOKEN* for two reasons: first, the secure monitor randomly generates and fills *SECRET_TOKEN* when initializing the hardware enclave; second, light-enclave-gates are located in domain-0 and thus are *execute-only* for light-enclaves. Moreover, as mentioned in § 5.1, a light-enclave can expose an interface only for a specific light-enclave. Line-4 to Line-6 are used to authenticate the identity of the caller light-enclave (meeting *req-2*).

```
... // save the current states
1  mov $SECRET_TOKEN, %r15
2  mov $bitmap_low, %eax
3  mov $bitmap_high, %edx
4  xrstor64 (%rdi)
5  cmp $SECRET_TOKEN, %r15
6  jne handle_abuse
7  xor %r15, %r15
... // continue the execution
```

Figure 6. Avoid the abuse of *XRSTOR*.

Fourth, *XRSTOR/XRSTORS* can restore the processor extended states that can also include the PKRU register. Both Intel SGX SDK and some LibOSes for SGX utilize *XRSTOR* when re-entering an enclave. Similarly, LIGHTENCLAVE first ensures no occurrence of these instructions in light-enclaves by using binary inspection and then wraps the necessary instructions as shown in Figure 6. By clearing bit-9 in *bitmap_low*, *XRSTOR* will not modify the PKRU register.

By configuring PKRU at enclave entries/exits and forbidding light-enclaves to access SSA pages, LIGHTENCLAVE ensures light-enclaves cannot change the PKRU during the hardware enclave transition. By elaborately controlling the existence of PKRU modifying instructions, LIGHTENCLAVE disallows light-enclaves to modify the PKRU arbitrarily. Therefore, LIGHTENCLAVE can ensure a light-enclave is always bound to its designated PKRU during execution.

5.3 Secure Monitor

Each hardware enclave contains one secure monitor that works as the control plane. It is trustworthy and can access all the memory domains and execute all the SGX-compatible instructions. During initialization, it has two responsibilities. The first is remote attestation that reports the measurement of the whole hardware enclave to remote users. The second is filling random tokens in the light-enclave-gates. During runtime, it serves light-enclaves, interacts with them through light-enclave-gates. The first service is dynamic enclave page management: it allocates enclave pages with desired domain-IDs for different light-enclaves. The second service is dynamic light-enclave management: it supports building new light-enclaves, reclaiming existing light-enclaves, adding new light-enclave-gates, and allocating shared memory domain. Besides, it also updates the enclave capability table.

6 Implementation

6.1 Integration with SGX LibOSes

Existing SGX library OSes (LibOSes) can be extended to work as the secure monitor in LIGHTENCLAVE. Occlum [53] is a LibOS that runs inside an SGX enclave and supports multiple tasks running on it. To enforce isolation amongst tasks, it enforces MPX-based bounds checking on memory accesses. We add/modify about 1,100 lines of codes (LOC) in Occlum (commit 0a06c898) to integrate the mechanism of LIGHTENCLAVE into it. We name the modified Occlum as Iso-Occlum. Iso-Occlum leverages the abstraction of light-enclave to run different tasks and abandons the original time-consuming bounds checking. Graphene-SGX [55] is another LibOS that runs inside an SGX enclave to support unmodified Linux applications. It can support multi-process applications by creating multiple hardware enclaves. We add/modify about 4,900 LOC to incorporate LIGHTENCLAVE with Graphene-SGX (commit 9c226c9a). The modified one, named Iso-Graphene-SGX, creates light-enclaves instead of hardware enclaves.

6.2 Analysis of the Hardware Proposal

We validate most hardware extensions presented in § 4.2 and § 4.3 on the Intel SGX emulator, i.e., Intel SGX SDK simulation mode. For extensions on data structures, we add new attributes to SECINFO (Domain-ID) and SECS (PKE bit), which use the reserved fields in SECINFO and SECS implemented in the SDK; we also extend the EPCM entry to record Domain-ID. For extensions on instructions, we modify or add the emulation routines of the SGX instructions related to the extended data structures; Since the user-level *EENTER* and *ERESUME* routines need to read *CR4*, we add an *ioctl()* in the SGX driver for returning the value of *CR4*, which, however, is unnecessary if implemented in the microcode. We evaluate the correctness of the implementations on the emulator from two aspects. First, enclave applications can seamlessly run with a newer memory layout configuration (with Domain-IDs specified). Second, tampering with any Domain-ID of an enclave page can be detected during the page adding or swapping time. However, the emulator cannot emulate the hardware operations of memory access validation. Thus, we *cannot* validate the runtime check which aims to ensure the domain-IDs in the EPCM and the page table match before filling the TLBs (*this Non-emulated Check is one limitation of our work*).

We argue that our hardware proposal is feasible through analyzing the hardware, performance, and memory overhead.

① *Hardware Overhead.* Most SGX implementation is based on CPU microcode [8, 24], and the microcode is much easier to update than the hardware [8, 18]. Hence, the updates of instructions summarized in Table 1 can all be applied using SGX microcode without any significant change in the CPU hardware logic. The checking procedure of the enclave memory access also needs one extra validation step, i.e., the *Non-emulated Check*. According to an in-depth SGX analysis [24] (§ 6), two SGX patents [35, 41], and a recent study [47], the checking procedure only happens when TLB misses occur, and is (very likely) implemented in microcode as well, which means adding our extra validation step may also only need microcode update and require *no* hardware (e.g., MMU) modifications.

② *Performance Overhead.* The required changes to SGX instructions are mostly lightweight: in the emulator, the modifications incur 0.6%-0.7% overhead for constructing an enclave (1MB-1GB), and negligible overhead for enclave swapping/paging. Yet, checking the capability table may add the most overhead if the table is swapped out, i.e., adding up to 30,000 CPU cycles due to swapping three enclave pages in the worst case (adding tens of cycles without swapping). Such a potential cost can be avoided if only the secure monitor executes sensitive ENCLU instructions (infrequently needed) because it does not refer to the capability table. The *Non-emulated Check* needs to compare the Domain-ID in the page table entry (PTE) and the EPCM. Since the PTE and EPCM are accessed in the original checking procedure, our extra check may only add one cycle of comparison when TLB misses, which is also negligible.

③ *Memory Overhead.* Four extra (or reserved) bits in the EPCM entry are needed for each 4K enclave page as Domain-ID. Other extensions in SECS and SECINFO incur zero overhead by utilizing previously-unused fields. If required, the capability table usually takes at most several 4K pages for each enclave.

6.3 Limitations

Besides the above Non-emulated Check, another limitation of LIGHTENCLAVE is the number of distrusted light-enclaves

is limited to 16 (although the hierarchical ones can be more) because MPK currently only supports up to 16 domains. There are three potential future solutions. First, prior work [48, 52, 58] proposes orthogonal approaches to extend the number of MPK domains. Second, there are still unused bits in page table entries. It is possible for Intel to allow more bits (even configurable) to be used as domain-ID (2 more bits can increase the number to 64). Third, using multiple enclaves could be one simple solution, i.e., 2 SGX enclaves or 2 nested-enclaves [47] can offer 32 domains, which, however, increases the overhead when cross-enclave interaction is needed. This is also left as our future work.

7 Security Analysis

When two mutually distrusted light-enclaves run in one SGX enclave, LIGHTENCLAVE promises the same security guarantees as running them in separated SGX enclaves. **The untrusted OS cannot compromise a light-enclave.** Since the light-enclave is guarded inside an SGX enclave, be default the untrusted software including the OS cannot access its memory and execution states. One difference made by LIGHTEN-CLAVE is enclave pages are tagged with different domain-IDs, which, however, does not bring new attack vectors for two reasons. First, light-enclave's pages are always located in EPC (i.e., inaccessible to the OS) whichever their domain-IDs are. Second, the OS can only cause DoS attacks by manipulating the domain-IDs in the page table because the correct ones are securely recorded in EPCM.

An untrusted neighbor cannot compromise a light-enclave. A light-enclave may co-run with compromised or malicious neighbor light-enclaves in the same SGX enclave. Suppose light-enclave-A intends to attack light-enclave-B. It has four major attack vectors. First, A may try to directly access B's memory pages, but LIGHTENCLAVE prevents A from accessing B's memory by using MPK-based isolation. Second, A may try to access B's execution states in SSAs. Nevertheless, the SSA pages are located in domain-0 that is inaccessible to light-enclaves. Third, A may try to modify the PKRU register that determines the domain access permission. LIGHTENCLAVE carefully wraps the instructions that can modify PKRU (see Section 5.2), and consequently A cannot leverage these instructions to modify its own PKRU. Moreover, LIGHTENCLAVE locate all the code pages in in domain-0 so that light-enclaves cannot either modify its code nor add new code (e.g, via *EAUG*) to modify the the PKRU register. Fourth, A may deliberately trigger exceptions to crash the execution of the whole enclave, leading to a DoS attack. LIGHTENCLAVE requires the OS to report the unexpected exceptions to the secure monitor that can then inspect the SSAs for locating A. Thereby, A cannot conduct DoS attacks without the help of OS. If the OS does not cooperate, a DoS may happen. Yet, the OS can also easily launch DoS attacks without LIGHTENCLAVE.

The Collusion attack cannot compromise a light-enclave. The untrusted OS can neither set the PKRU register for the malicious light-enclave due to light-enclave-gates at enclave entries, nor modify the domain-IDs of the sensitive enclave pages due to MMU checks according to EPCM. Another concern is two light-enclaves share the same memory encryption key because of running in the same enclave. Nevertheless, SGX encryption mechanism involves the virtual addresses, which prevents potential information leakage under physical attacks.

8 Performance Evaluation

In this section, we seek to answer the following questions:

1. How fast is LIGHTENCLAVE in terms of light-enclave creation and communication? (§ 8.1)

2. How much overhead does the intra-enclave isolation of LIGHTENCLAVE cause to applications? (§ 8.2)

3. How much performance improvement can LIGHTEN-CLAVE bring to existing LibOSes? (§ 8.3)

4. How much performance improvement can LIGHTEN-CLAVE bring for FaaS scenarios? (§ 8.4)

We do not use the emulator for evaluation since it doesn't emulate SGX memory encryption engine and thus shows much better performance than the real SGX. Instead, we evaluate LIGHTENCLAVE on the real machine with Intel i7-10700 IceLake CPU (2.9 GHz, SGXv1, MPK) and Linux kernel 5.4.110 by adding the estimated overhead (similar to [39, 47]). According to § 6.2, the proposed hardware extensions on enclave initialization and memory management incur negligible performance overhead; two of three added hardware checks at runtime are both negligible because either extremely lightweight (the Non-emulated Check) or too infrequent (the capability table check, not required in the following experiments); the last added check in *EENTER* and *ERESUME* is on the critical path. Thus, we only add the estimated overhead (19 CPU cycles) on *EENTER/ERESUME*, which simulates accessing one SECS field (10 cycles) and reading CR4 (9 cycles). The former one is the latency of accessing L1 cache since the cacheline of the SECS field is originally accessed by the two instructions while the latter one is measured by repeating accessing CR4 and getting the average cost.

To show the feasibility and performance benefits of LIGHT-ENCLAVE, we leverage it to isolate third-party libraries (§ 8.2), integrate it into Graphene-SGX [55] and Occlum [53] (§ 8.3), and apply it in the serverless scenario (§ 8.4).

8.1 Microbenchmarks

Fast creation of light-enclaves. When Graphene-SGX needs to spawn a new process, it follows the standard FORK semantic and thus launches a new enclave that contains exactly

App	helloworld	SQLite3	cc1
Grapene-SGX	1773	1762	1888
Iso-Grapene-SGX	32	37	54
Occlum	0.163	9.414	65.86
Iso-Occlum	0.162	6.259	42.55

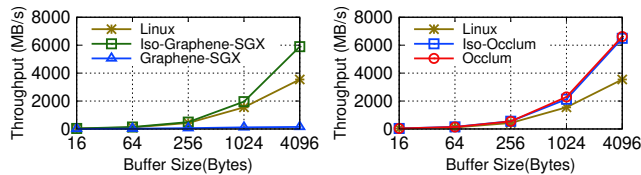Table 2: Task creation latency (ms).

Figure 7. Performance of Pipe-based communication.

the same contents. Nevertheless, Checkpointing and transferring existing states (opened file tables, memory layout, etc.) are not free. With LIGHTENCLAVE, Iso-Graphene-SGX effectively addresses this performance issue because it can construct multiple light-enclaves (as separated processes) in the same hardware enclave without creating a new hardware enclave from scratch. The first test spawns (light-)enclaves initialized from different sizes of ELFs (include the 2MB LibC): helloworld (2MB), SQLite3 (3MB) and cc1 (13MB), and measures their creation latency. As shown in Table 2, Iso-Grapene-SGX can reduce application creation time by 97% as it does not construct a new hardware enclave and fork a new LibOS instance. The latency of spawning helloworld or SQLite3 is dominated by the complex process creation logic of Graphene-SGX and thus close to each other.

We also compare the process spawn time between Occlum and Iso-Occlum. Iso-Occlum can reduce application creation time by up to 35% because it eliminates code instrumentation and thus avoids bloating the ELF size. For example, the size of *cc1* is 17MB with instrumentation and 11MB otherwise. The ELFs are initially stored in Occlum's encrypted file system. A smaller ELF means loading less content from the file system when spawning.

Fast communication between light-enclaves. Graphene-SGX uses multiple SGX enclaves for isolating different entities, running the LibOS instance in each enclave. It supports enclave communication through *pipe*. For creating one pipe, two LibOS instances negotiate the encryption/decryption key via cryptographic methods. When the enclaves send/receive messages through the pipe, the messages are encrypted/decrypted, which is time-consuming. In Iso-Graphene-SGX, the modified LibOS works as the secure monitor and can allocate a shared memory domain within the hardware enclave for light-enclaves to exchange data. Thereby, the implementation of pipe in Iso-Graphene-SGX requires no message encryption/decryption. We measure the through-

put of message passing over pipe under different buffer sizes. As shown in Figure 7, compared with Graphene-SGX, pipe throughput in Iso-Graphene-SGX is 7x to 40x higher. It is even higher than that in Linux because system calls to LibOS are more lightweight. Differently, Occlum and Iso-Occlum achieve similar throughput because they both leverage the shared memory provided by the LibOS for message passing.
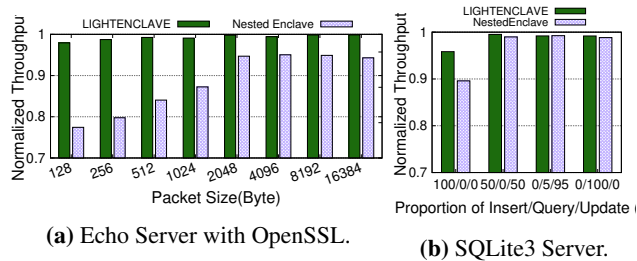

(a) Echo Server with OpenSSL.

(b) SQLite3 Server.

Figure 8. Performance of OpenSSL and SQlite3 servers.

8.2 Isolating Third-Party Libraries

In this subsection, we show LIGHTENCLAVE can be used to apply the multi-level security principle to applications in SGX with low performance overhead. We also compare LIGHTENCLAVE with Nested Enclave [47]. Since hardware extensions of Nested Enclave are also unavailable, we emulate its performance by invoking an empty *ocall/ecall* when *n_ocall/n_ecall* happens between inner enclaves and the outer enclave.

Echo Server with OpenSSL. Isolating sensitive code and third-party code is good for security. For example, prior work [56] suggests isolating the session keys as well as the related SSL library code accessing the keys in an isolated execution environment. Accordingly, we deconstruct the SGX-OpenSSL [13] library and make the minimal code that allocates and accesses the session keys as light-enclave-S (secure) while leaving the rest code as light-enclave-NS (non-secure). Then, we implement a simple echo server linked with the SSL library, and it also runs in light-enclave-NS. Light-enclave-S is more privileged than light-enclave-NS, and any vulnerability in the latter one cannot leak the secrets in the former one. The two light-enclaves run in a hardware enclave as a server, and a client running in another hardware enclave invokes the corresponding service. The client and server use a session key for symmetric encryption/decryption during communication. Since all the session keys are allocated in Light-enclave-S, the echo server in Light-enclave-NS needs to switch to Light-enclave-S for encrypting and decrypting packets. Thus, light-enclave switches happen when sending and receiving packets. For using the approach of Nested Enclave, the secure code should run in an inner enclave while the insure code in the outer enclave.

We gradually increase the size of packets exchanged by the client and the server and present the throughput in Figure 8a.

LIGHTENCLAVE introduces up to 2% overhead compared with the baseline (no isolation). There is nearly zero overhead when the packet size is larger than 2kB, since content encryption/decryption and sending/receiving dominate the time. The overhead of Nested Enclave is between 2% to 23%. LIGHTENCLAVE can outperform it by up to 27%, indicating the interaction between isolated entities in LIGHTENCLAVE is more efficient.

SQLite3 Server. We further show an example of confining a whole third-party library in one light-enclave. We build a simple, secure key-value store server that uses the SQLite3 (v3.23.0) engine. The enclave is divided into one trusted light-enclave and one untrusted light-enclave, while the untrusted one is for the third-party library, SQLite3. A client running in a normal process keeps sending requests to the server. The server and client communicate through the local network. The server invokes the interfaces of SQLite3 to perform Insert, Update, and Query operations. The server also encrypts the data before storing it into SQLite3 and decrypts the data after retrieving it from SQLite3. We evaluate both LIGHTENCLAVE and Nested Enclave under different workloads. Figure 8b shows the normalized server throughput (baseline has no isolation). For the 100% Insert workload, Nested Enclave shows 11% overhead while LIGHTENCLAVE only incurs 4% overhead. For other workloads, the overhead is negligible.

8.3 Optimizing Applications on LibOSes

In this subsection, we show LIGHTENCLAVE can improve the performance of SGX-oriented LibOSes, Graphene-SGX and Occlum (without degrading the security).
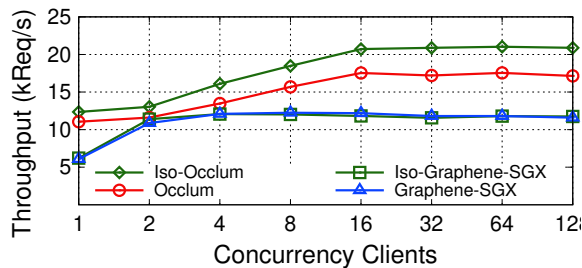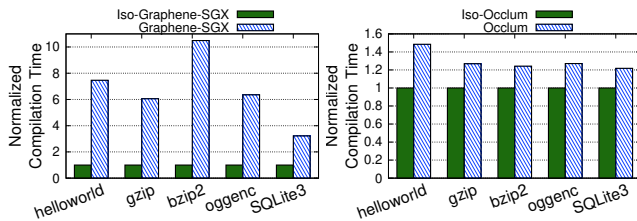

Figure 9. The performance of Lighttpd.

Figure 10. The performance of GCC.

Lighttpd. Lighttpd [10] is a widely-used multi-process web server. We configure the Lighttpd server (v1.4.40) with two workers running in two light-enclaves, and use ApacheBench as clients to fetch 10KB web pages. The server and client are connected with the local network. We increase the concurrency of clients gradually to get the peak throughput. Figure 9 shows the results. The difference of peak throughput between Iso-Grapene-SGX and Graphene-SGX is only about 2% because there is no process spawn and there is little communication between different workers. Nevertheless, the peak throughput in Iso-Occlum is 1.2x higher than that in Occlum, owing to no boundary checking.

GCC. GCC [7] (v4.4.5) spawns and executes *cc1*, *as*, *collect2* and *ld* for compiling programs from C codes to ELF. We evaluate the performance of GCC compilation (CPU-intensive) on both Iso-Grapene-SGX and Iso-Occlum to show the performance improvement brought by LIGHTENCLAVE. We use GCC to compile five files with various lines of codes: *helloworld* with 5 LOC, *bzip2* with 5K LOC, *gzip* with 5K LOC, *oggenc* with 50K LOC and *SQLite3* with 130K LOC. The normalized compilation time is shown in Figure 10. Iso-Graphene-SGX optimizes the process creation time and thus achieves 5.11x to 10.5x speedup compared with Graphene-SGX. Iso-Occlum eliminates the overhead of SFI instrumentation, which is 1.22x to 1.49x faster than Occlum.

LibOS	Processing Time
Graphene-SGX	15.7 s
Iso-Graphene-SGX	910.9 ms
Occlum	16.3 ms
Iso-Occlum	12.7 ms

Table 3: The benchmark of Fish Shell.

Fish Shell. Fish shell [6] is a smart and user-friendly command-line shell. When executing new commands, Fish shell spawns a new process. The intermediate results are transferred using pipe. BusyBox [5] is a command-line tool that combines tiny versions of many common UNIX utilities. We use Fish shell (v3.0.0) and BusyBox (v.1.23.1) to show the performance improvement brought by LIGHTENCLAVE to applications that need to spawn new processes frequently. We execute a test script based on byte-unixbench [12] in Fish Shell, which invokes several BusyBox commands (*od*, *sort*, *grep*, *wc* etc.) to handle text files. Table 3 shows the script's execution time. Iso-Graphene-SGX is 17.2x faster than Graphene-SGX as it reduces the overhead of process spawns, and Iso-Occlum is 1.28x faster than Occlum as time-consuming bound check instructions are not needed in Iso-Occlum.

8.4 Optimizing Serverless Functions

SGX can also be used to protect private data in serverless scenarios. However, the slow enclave initialization may incur substantial overhead because serverless functions usually

execute for a short period. To solve it, some studies deploy serverless functions with SFI [50] or language-based [21] sandbox in SGX enclaves. But they introduce runtime overhead due to software instrumentation or restrict the function language. LIGHTENCLAVE has the potential to reduce such overhead since the construction of light-enclaves is efficient. Specifically, it can initialize a light-enclave inside an existing enclave instead of creating a new hardware enclave for executing one function. We evaluate the whole latency of functions with three approaches: COLD, creating enclave on demand (upon a new request); WARM, maintaining enclave pools to serve requests (avoid enclave creation); LIGHTENCLAVE, creating light-enclaves on demand.

Functions	Description	Runtime
auth	login authentication	Node.js v8.10.0
crypto	file encryption	Node.js v8.10.0
face_detect	face image detection	Python v3.7
sentiment	text sentiment analysis	Python v3.7

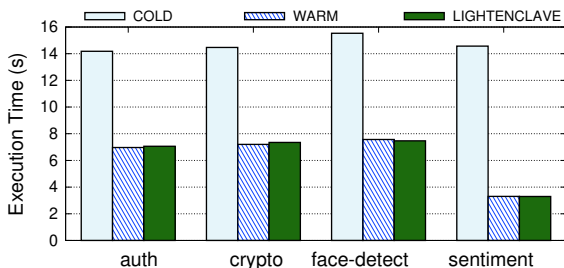Table 4: Serverless function benchmarks.

Figure 11. The latency of executing serverless functions.

Table 4 lists the evaluated serverless functions from [39]. Node.js and Python environments are deployed in Iso-Graphene-SGX and Iso-Occlum, respectively. Figure 11 shows the evaluation results. LIGHTENCLAVE decreases the latency by 50% to 77% compared with COLD, and achieves similar latency with WARM but with significant fewer resources (e.g., one hardware enclave can concurrently run 16 instead of 1 functions). Moreover, skipping runtime initialization is known to decrease further the latency of serverless functions [46], which is orthogonal to our work. Besides fast booting of light-enclaves, the fast interaction between light-enclaves may also decrease the communication overhead between chained serverless functions. In the example of the Sequence-chained test of ServerlessBench [11] (a chain of 5 ephemeral functions takes 105 ms), LIGHTENCLAVE can reduce 12% overall latency by decreasing the communication latency from 3.0 ms to 2.1 us.

9 Related Work

Intel SGX + Intel MPK. Two prior studies [23, 57] leverage Intel MPK to confine a malicious enclave's behaviors such that an enclave cannot access the host regions arbitrarily. Their design goals are different from LIGHTENCLAVE. Nevertheless, LIGHTENCLAVE can be deployed together with such systems by carefully setting the domain access permission. EnclaveDom [43] is a pioneer in using MPK for intra-enclave isolation, and LIGHTENCLAVE shares a similar goal and design. Yet, it is envisioned with an aligned threat model for MPK and is meant to showcase the performance. The hardware proposals of LIGHTENCLAVE can also enhance the security for it.

Intra-enclave isolation. Most prior studies achieve intra-enclave memory isolation by instrumenting memory access instructions and ensuring the instrumentation is non-bypassable, such as Occlum [53], CHANCEL [15], Spons & Shields [51], which inevitably causes non-negligible runtime overhead. Nested Enclave [47] proposes architectural extensions to allow an SGX enclave to be an outer enclave of multiple other SGX enclaves as inner enclaves. The outer enclave cannot access the inner enclaves while all the inner ones can access the outer one. Differently, LIGHTENCLAVE is flexible to provide configurable light-enclave hierarchies.

SGX-oriented LibOSes. To ease programming of SGX enclaves, there is a line of LibOSes work in the context of SGX. Haven [19] is the first LibOS with multiple shim layers (e.g., network, filesystem, console) to run unmodified Windows applications. Graphene-SGX [55] and SCONE [17] runs unmodified Linux applications and Docker images, respectively, by containing the corresponding OS components within the LibOS. Ryoan [31] further implements an in-memory file system. All of these LibOSes introduce relatively large TCB. LIGHTENCLAVE provides a light-enclave abstraction which may help compartmentalize the LibOS into multiple domains to bring security and reliability benefits.

Future prospects of LIGHTENCLAVE. TEEs like SEV [14] and TrustZone [40] can use a trusted OS to provide isolated processes as enclaves. Nevertheless, there also lacks intra-process (intra-enclave) isolation, and decoupling an application into different processes (e.g., for isolating third-party libraries) will incur non-trivial performance overhead due to costly inter-process communication. The idea of LIGHTENCLAVE could be generalized to introduce fine-grained isolation to such monolithic enclave models because the hardware feature of memory protection key (MPK) is not unique to Intel. Examples include ARM memory domains [4], Apple APRR [2], AMD MPK [1], and RISC-V Donky [52].

10 Conclusion

This paper presents LIGHTENCLAVE, a hardware-software co-design for efficient intra-enclave isolation, which compartmentalizes an SGX enclave into isolated light-enclaves to reduce the overall TCB with low runtime overhead. LIGHTENCLAVE can also be integrated into state-of-the-art SGX LibOSes to achieve high-security and high-efficiency.

11 Acknowledgement

References

[1] AMD memory protection key. https://www.phoronix.com/scan.php?page=news_item&px=AMD-PRM-PCID-PKEY.

[2] Apple APRR. https://blog.svenpeter.dev/posts/m1_sprr_gxf/.

[3] ARM Confidential Compute Architecture. https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture/.

[4] ARM memory domain. https://arm-software.github.io/CMSIS_5/Core_A/html/group__CMSIS_DACR.html.

[5] BUSYBOX. https://www.busybox.net/.

[6] Fish Shell. https://fishshell.com/.

[7] GCC, the GNU Compiler Collection. https://www.gnu.org/software/gcc/.

[8] Intel software developer's manual. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html. Referenced December 2021.

[9] Intel® Trust Domain Extensions (Intel® TDX). https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html.

[10] LIGHTTPD, fly light. https://www.lighttpd.net/.

[11] Serverlessbench. https://github.com/SJTU-IPADS/ServerlessBench. Referenced December 2021.

[12] byte-unixbench. https://github.com/kdlucas/byte-unixbench, 2018.

[13] Sgx-openssl. https://github.com/sparkly9399/SGX-OpenSSL, 2021.

[14] Inc Advanced Micro Devices. Amd secure encrypted virtualization (sev). https://developer.amd.com/sev/.

[15] Adil Ahmad, Juhee Kim, Jaebaek Seo, Insik Shin, Pedro Fonseca, and Byoungyoung Lee. CHANCEL: efficient multi-client isolation under adversarial programs. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*, 2021.

[16] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *ARM White Paper*, 3(4):18–24, 2004.

[17] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, David Goltzsche, David Eyers, Rudiger Kapitza, Peter Pietzuch, and Christof Fetzer. Scone: Secure linux containers with intel sgx. In *Proc. the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Nov. 2016.

[18] Andrew Baumann. Hardware is the new software. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems*, HotOS '17, page 132–137, New York, NY, USA, 2017. Association for Computing Machinery.

[19] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems*, 33(3):8, 2015.

[20] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: Sgx cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, 2017.

[21] Stefan Brenner and Rüdiger Kapitza. Trust more, serverless. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 33–43, 2019.

[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019.

[23] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX. In *31st USENIX Security Symposium (USENIX Security 22)*, Boston, MA, August 2022. USENIX Association.

[24] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptol. ePrint Arch.*, 2016(86):1–118, 2016.

[25] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[26] Erhu Feng, Xu Lu, Dong Du, Bicheng Yang, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. Scalable memory protection in the penglai enclave. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 275–294, 2021.

[27] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, pages 1–6, 2017.

[28] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[29] Stephen Herwig, Christina Garman, and Dave Levin. Achieving Keyless CDNs with Conclaves. In *Proc. of the USENIX Security Symposium*, 2020.

[30] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proc. of the Hardware and Architectural Support for Security and Privacy (HASP)*, 2013.

[31] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proc. the OSDI*, Nov. 2016. DOI:10.1145/3231594.

[32] IBM. Data-in-use protection on ibm cloud using intel sgx. https://www.ibm.com/cloud/blog/data-use-protection-ibm-cloud-using-intel-sgx, 2018.

[33] Intel Corp. https://software.intel.com/en-us/sgx-sdk. *Intel Software Guard Extensions SDK*.

[34] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[35] Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata, Francis X McKeen, and Carlos V Rozas. Technique for supporting multiple secure enclaves, March 3 2015. US Patent 8,972,746.

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[37] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: an open framework for architecting trusted execution environments. In *Proc. of the ACM European Conference on Computer Systems (EuroSys)*, 2020.

[38] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 557–574, Vancouver, BC, August 2017. USENIX Association.

[39] Mingyu Li, Yubin Xia, and Haibo Chen. Confidential serverless made efficient with plug-in enclaves. In *Proc. of the International Symposium on Computer Architecture (ISCA)*. IEEE, 2021.

[40] ARM Ltd. Arm trustzone technology. https://developer.arm.com/ip-products/security-ip/trustzone.

[41] Francis X McKeen, Carlos V Rozas, Uday R Savagaonkar, Simon P Johnson, Vincent Scarlata, Michael A Goldsmith, Ernie Brickell, Jiang Tao Li, Howard C Herbert, Prashant Dewan, et al. Method and apparatus to provide secure application execution, July 21 2015. US Patent 9,087,200.

[42] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. Intel® Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proc. of the Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[43] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. Enclavedom: Privilege separation for large-tcb applications in trusted execution environments, 2020.

[44] Microsoft. Azure confidential computing. https://azure.microsoft.com/en-us/solutions/confidential-compute/, 2017.

[45] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1466–1482, 2020.

[46] Edward Oakes, Leon Yang, Dennis Zhou, Kevin Houck, Tyler Harter, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sock: Rapid task provisioning with serverless-optimized containers. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '18, page 57–69, USA, 2018. USENIX Association.

[47] Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, and Jaehyuk Huh. Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX. In *Proc. of the International Symposium on Computer Architecture (ISCA)*, 2020.

[48] 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 Conference on Usenix Annual Technical Conference*, USENIX ATC '19, page 241–254, USA, 2019. USENIX Association.

[49] Christian Priebe, Divya Muthukumaran, Joshua Lind, Huanzhou Zhu, Shujie Cui, Vasily A Sartakov, and Peter Pietzuch. Sgx-lkl: Securing the host os interface for trusted execution. *arXiv preprint arXiv:1908.11143*, 2019.

[50] Weizhong Qiang, Zezhao Dong, and Hai Jin. Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave. In *International Conference on Security and Privacy in Communication Systems*, pages 451–470. Springer, 2018.

[51] Vasily A. Sartakov, Daniel O'Keeffe, David M. Eyers, Lluís Vilanova, and Peter R. Pietzuch. Spons & shields: practical isolation for trusted execution. 2021.

[52] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[53] 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 *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2020.

[54] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. Panoply: Low-tcb linux applications with sgx enclaves. In *Proc. the Annual Network and Distributed System Security Symp.(NDSS)*, Feb. 2017.

[55] Chia-Che Tsai, Donald E Porter, and Mona Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proc. the USENIX Annual Technical Conference (ATC)*, page 8, July 2017.

[56] 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 *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[57] Samuel Weiser, Luca Mayr, Michael Schwarz, and Daniel Gruss. Sgxjail: Defeating enclave malware via confinement. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 353–366, Chaoyang District, Beijing, September 2019. USENIX Association.

[58] Yuanchao Xu, ChenCheng Ye, Yan Solihin, and Xipeng Shen. Hardware-based domain virtualization for intra-process isolation of persistent memory objects. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 680–692, 2020.