# Nested Enclave: Supporting Fine-grained Hierarchical Isolation with SGX

Joongun Park, Naegyeong Kang, Taehoon Kim, Youngjin Kwon, Jaehyuk Huh
School of Computing, KAIST
{jupark,ace9934,thkim,yjkwon,jhhuh}@casys.kaist.ac.kr

***Abstract*—Although hardware-based trusted execution environments (TEEs) have evolved to provide strong isolation with efficient hardware supports, their current monolithic model poses challenges in representing common software structures with modules produced from potentially untrusted 3rd parties. For better mapping of such modular software designs to trusted execution environments, it is necessary to extend the current monolithic model to a hierarchical one, which provides multiple inner TEEs within a TEE. For such hierarchical compartmentalization within a TEE, this paper proposes a novel hierarchical TEE called nested enclave, which extends the enclave support from Intel SGX. Inspired by the multi-level security model, nested enclave provides multiple inner enclaves sharing the same outer enclave. Inner enclaves can access the context of the outer enclave, but they are protected from the outer enclave and non-enclave execution. Peer inner enclaves are isolated from each other while accessing the execution environment of the shared outer enclave. Both of the inner and outer enclaves are protected from vulnerable privileged software and physical attacks. Such fine-grained nested enclaves allow secure multi-tiered environments using software modules from untrusted 3rd parties. The security-sensitive modules run on the inner enclave with the higher security level, while the 3rd party modules on the outer enclave. It can be further extended to provide a separate inner module for each user to process privacy-sensitive data while sharing the same library with efficient hardware-protected communication channels. This study investigates three case scenarios implemented with an emulated nested enclave support, proving the feasibility and security improvement of the nested enclave model.**

***Index Terms*—trusted computing, trusted execution environments, SGX enclave, multi-level security**

## I. Introduction

Hardware-based trusted execution environments (TEEs) such as Intel SGX and ARM TrustZone can provide strongly isolated execution supports by hardware-enforced access restriction and memory protection [5], [6]. Such trusted execution models have evolved from a single secure world model in TrustZone to multiple user-level *enclaves* in SGX. Using multiple independent enclaves, a prior study has shown that a task consisting of mutually untrusted components can be mapped to distributed enclaves communicating with channels in untrusted memory [24]. However, mapping a task to multi-tiered enclaves requires extensive refactoring of the current hierarchical software structure, in addition to relatively costly communication channels through untrusted memory region with potential vulnerability [44].

The limitation of the current enclave model stems from its monolithic design of enclaves. An enclave provides a single protection domain without any decomposition of security levels within an enclave. However, providing multiple levels of security within an enclave strengthens the security of applications with components developed by untrusted third parties. Applications are commonly developed by composing privately developed modules with third party libraries, and such external modules must be protected but cannot be entirely trusted, as vulnerabilities are often found even in these open source libraries [1], [2], [19], [35]. These vulnerabilities adversely affect not only the data and code inside of the libraries but also the entire application that uses their features. When an enclave includes the codes from untrusted 3rd parties, the current monolithic enclave design cannot provide an efficient representation of privilege isolation required for such software structures.

To express the hierarchical isolation, the classic multi-level security (MLS) concept [12], [13] can be applied to the current TEE model. To provide MLS within an enclave while supporting the procedural semantics of programming languages, this paper proposes to extend the current monolithic enclave design of SGX to *nested enclave*, which can support fine-grained hierarchical isolation within an enclave. With the nested enclave, an enclave (outer enclave) can contain multiple inner enclaves with the higher security level than the outer enclave. The execution environments of an inner enclave cannot be accessed by the outer enclave, while the inner enclave has full access permission to the outer enclave. Peer inner enclaves within an outer enclave are isolated with each other, while they can communicate efficiently through the hardware-protected memory of the outer enclave. In the MLS concept, the top secret corresponds to the inner enclave, while the lower-level secret corresponds to the outer enclave.

Figure 1 shows the concept of nested enclave compared to the current monolithic enclave model. In the current monolithic SGX model, multiple enclaves can exist in a system. However, communicating and transferring controls must be done through the unsecure environments with costly software-based encryption for communication. In the nested enclave model, hierarchical enclaves are supported, with multiple isolated inner enclaves sharing the same outer enclave. The figure shows the asymmetric access permission between the inner and outer enclaves.

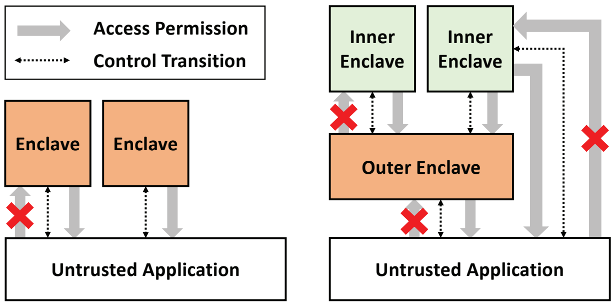
Nested enclave enables the efficient representation of call-based program structures with multiple security levels within an application. First, the hierarchical model provides natural support for the security isolation between the inner enclave with the higher level of permission and the outer enclave with the lower level. For example, the third-party library can exist in the outer enclave while the privacy-sensitive processing is done in the inner enclave. The combined execution environment of the inner and outer enclaves is protected from the vulnerable privileged software and hardware attacks while running on untrusted cloud environments. Second, multiple inner enclaves sharing the same outer enclave allow fine-grained isolation within an enclave. For example, if an application serves multiple users, the privacy-sensitive code for each user can run separately in an inner enclave, while only the sanitized data can be fed to the shared libraries in the outer enclave. Finally, fine-grained communication across inner enclaves becomes more efficient, as they access the protected enclave memory of the outer enclave. The enclave memory is efficiently protected by the hardware engine (Memory Encryption Engine in SGX), and thus the data can be securely transferred across inner enclaves without any software intervention for encryption.

While the nested enclave can provide the rich semantics of hierarchical module-based software structures with strong isolation, it does not require any significant new hardware changes. The internal meta-data of enclaves are extended to represent the nested relationship between inner and outer enclaves. The extra hardware changes for supporting nested enclaves are minimized mostly to the access control mechanism for the enclave memory region. In SGX, the access control is conducted during handling TLB (Translation Lookaside Buffer) misses, by validating translation entries to be inserted to the TLB. The required change is to allow an inner enclave to access the memory range of its outer enclave, but not vice versa.

To show how the nested enclave model can improve the restrictions of the current monolithic one, we implemented three case studies with an emulated nested enclave support by modifying the SGX emulation framework. The first case study uses nested enclave to isolate the 3rd party library from the rest of the critical code in an application. We use the OpenSSL library to show the effectiveness of isolation between the inner and outer enclaves. The second case study investigates how multiple inner enclaves sharing the same outer enclave can improve privacy protection with machine learning and database server applications. The last case study shows how data sharing can be facilitated by the outer enclave with multiple inner enclaves. The outer enclave becomes a secure communication channel across inner enclaves.

This study is one of the first studies for efficiently separating security levels within an enclave by hardware extension. The main contributions of this paper are as follows:

- It proposes a new semantic of enclave model to provide multiple levels of security within an enclave. The nested enclave model opens a new semantic expansion of hardware-based trusted execution.
- It shows the required hardware and software extensions for the nested enclave are minimal and feasible to add in the current SGX.



(a) SGX Enclave (b) Nested Enclave

Fig. 1. Overview of nested enclave (b) compared to SGX enclave (a)

- It investigates case studies proving the effectiveness of the proposed nested enclave with a prototype design by extending the enclave emulation framework.

The rest of the paper is organized as follows. Section II presents the background of Intel SGX. Section III discusses the motivation for hierarchical isolation within an enclave and threat model of this work. Section IV describes the design of nested enclave and its required hardware and software changes. Section V shows the methodology of this work, and Section VI explores the case scenarios with nested enclave. Section VII discusses the security guarantee of nested enclave, and Section VIII discusses potential extensions of nested enclave. Section IX presents the prior work, and Section X concludes the paper.

## II. Background

### A. Intel Software Guard Extensions (SGX)

Intel SGX provides a trusted execution environment called *enclave* protected from untrusted system software. SGX isolates the execution of an enclave through data protection and access restriction. The protected memory pages for enclaves are allocated in Enclave Page Cache (EPC) which is mapped to a reserved protected memory region, called Processor Reserved Memory (PRM). EPC pages are encrypted and integrity-protected by a hardware encryption engine in SGX-enabled processors [22]. The owner enclave code can access its EPC pages mapped to its virtual address space. However, non-enclave or non-owner enclave codes cannot access the EPC pages of another enclave, as a hardware-based access control technique blocks any unauthorized accesses during address translation. In addition, with attestation support, a user can verify the identity and measured digest of enclave itself and platform setting where the enclave runs.

The meta-data of an enclave is stored in its SGX Enclave Control Structures (SECS) allocated in EPC pages. SECS contains the required meta-data defining an enclave. Therefore, the physical address of an SECS structure is unique for each enclave. An enclave can have multiple threads, and the context of each thread is stored in Thread Control Structure (TCS) which is also allocated in EPC pages.

## *B. Memory Protection for Enclave*

EPC pages are protected both from physical attacks on the DRAM and from software attacks by user processes or the kernel running on the processor. The resilience to physical attacks is supported by the hardware memory encryption engine, which encrypts the reserved memory at cacheline granularity. EPC memory pages exist only as encrypted text in the physical DRAM. In addition, a variant of hash tree validates the integrity of data brought from the physical DRAM [22], [39], [46]. With the two mechanisms, the confidentiality and integrity of EPC pages are protected.

Memory access control from the software running in the processor is enforced by the address translation mechanism for virtual memory support. Two conditions for the security of an EPC page must be satisfied:

1) An EPC page must be accessed only by the owner enclave. Any access attempt by non-enclave execution or non-owner enclave must be blocked.
2) An EPC page must be mapped to the virtual address of the enclave, which is fixed at the enclave initialization[1]. It follows the virtual memory layout specified by the enclave author.

A key internal data structure for the protection of EPC pages is Enclave Page Cache Map (EPCM). For each EPC page, the corresponding entry in EPCM must contain two fields, the owner enclave ID and the virtual address mapped for the EPC page. EPCM indexed by physical addresses contains the reverse mapping from the physical page address to the virtual page address for all EPC pages.

The actual access validation is done during the address translation. For efficient validation, the access checking is conducted for each TLB (Translation Lookaside Buffer) miss. During the handling of a TLB miss, the validity of the access is checked. Once it is verified, the page translation entry is inserted into TLB. With the TLB-based validation, a key invariant for access control is that *TLB must always contain only valid translations*. To satisfy the invariant, TLB must be flushed during the transition between enclave and non-enclave environments.

A restriction on the virtual address layout for an enclave for efficient validation is that the virtual address range of an enclave must be contiguous, represented by the starting virtual address and size, called Enclave Linear Address Range (ELRANGE). The virtual address range (ELRANGE) is fixed during the initialization of an enclave, and cannot be changed. With the contiguous virtual address range (ELRANGE), whether an address is within the ELRANGE is quickly checked by a simple hardware component.

Figure 2 shows the validation steps during a TLB miss handling, which checks the validity of the corresponding page table entry. 1) It checks whether the core is running in enclave mode. 2) In enclave mode, it validates the enclave ID of the running enclave is equal to the one in the EPCM entry. 3) After verifying the enclave ID, it checks whether the virtual address of the EPC page matches the virtual address stored in the EPCM entry.

[1]SGX2 [33] allows dynamic EPC allocation to an existing enclave.

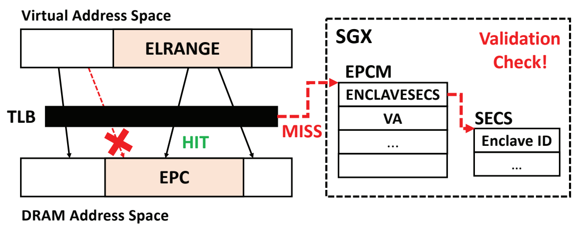


Fig. 2. TLB miss handling procedure of EPC pages. For a TLB miss, the translation from the page table is validated by the hardware component.

## *C. Attestation*

SGX enables an enclave to verify other enclaves in the same physical machine. When the code and data are loaded in an enclave, the enclave generates a digest by keyed-hashing the initial state of the enclave, including the initial meta-data, virtual address layout, code, and data contents, etc. In addition, the enclave certificate contains the expected digest of the enclave signed by the enclave author. During the initialization of an enclave, the author's signature is verified, and the hash of the signature is added to the enclave SECS to be used for later attestation (MRSIGNER). The expected digest by the author in the enclave certificate is also compared to the actual measurement of the enclave during the initialization. Once an enclave is properly initialized, the measurement of the enclave and the certificate hash are used for validating the enclave during local attestation procedures.

## III. Motivation and Threat Model

## *A. Motivation*

Modern applications are complex to make them secure. Applications rely on the existing untrusted, third-party libraries to reduce development complexity and handle a various range of data from the privacy-sensitive to the ordinary one. To secure such applications, a strong isolation mechanism is the cornerstone. With the isolation mechanism, developers compartmentalize a large application into smaller pieces and map them in different protection domains, confining unintended damages by malicious or misbehaving pieces. SGX provides an attractive way; developers deploy the compartmentalized pieces to each enclave, and the enclave provides the hardware-protected isolated environment. However, despite the strong isolation, the current SGX model may not provide sufficient mechanisms to meet applications' requirements for isolation.

**Need for confinement:** Intel SGX provides a monolithic protection model within an enclave, meaning the entire codes in an enclave has the same protection domain. However, application developers often need isolation between the codes written by themselves and third-party libraries [41]. In the current enclave application development, the in-enclave third-party libraries linked with user enclave codes run in the same enclave, which is vulnerable to data leak [19] or remote code execution attack [4]. One example is the HeartBleed attack [19] in the OpenSSL library, a collection of cryptographic functions and secure communication protocols widely used in practice. Due to a small bug in processing heartbeat messages to maintain OpenSSL sessions, attackers could leak information of arbitrary freed buffers from the applications linking the OpenSSL library.

**Need for multi-level protection:** Often information has different security levels as represented by multi-level security (MLS) [12]. According to the sensitivity, data should be labeled with different levels of security and protected by the access control mechanism supporting multiple security levels. Likewise, for strong isolation, application code should be separated by protection domains, and the code and data associated with the higher level of security must be protected from the code with the lower level of security permission, reflecting the hierarchical structure of MLS.

Unfortunately, the current monolithic enclave needs non-trivial efforts and costs to meet such applications' requirements. One possible way to isolate the third party libraries is separating the application code or third-party libraries to multiple peer enclaves. Ryoan [24] takes the approach; By Combining Google's NaCl [56] with Intel SGX, Ryoan separates modules with mutually distrusting domains. While effective, this approach adds the substantial amount of NaCL code to the trust computing base and converts each library call to expensive inter-enclave IPCs.

Nested enclave addresses the above limitations by extending the coarse-grained monolithic SGX design to the fine-grained hierarchical one.

### B. Threat Model

The trusted computing base (TCB) of SGX enclave is i) an SGX-enabled processor chip, ii) trusted code written by the developer and the statically linked libraries running in enclaves. Nested enclave shares the basic attack model of the monolithic enclave: the attacker has authority to i) access hardware outside the CPU package, ii) fully control system software and iii) exploit the vulnerability (if exist) of third-party libraries used in the target application.

Unlike the dichotomy of trustworthiness in the SGX enclave (trusted or untrusted), nested enclave is based on the MLS model and thus the security is represented in multiple levels, such as top secret, secret, and untrusted. Extending the threat model for MLS, nested enclave guarantees the integrity and confidentiality of code and data running in inner enclaves (top secret) from outer enclaves (secret) and the untrusted world (untrusted). Likewise, outer enclaves are protected from the untrusted world.

As nested enclave is based on SGX, it shares the same vulnerability for side-channel attacks as the monolithic enclave. Nested enclave is vulnerable to side-channel attacks such as foreshadow attacks [15] and cache side-channel attacks [17], [18], [43]. In addition, we do not consider availability attacks such as denial of service.

## IV. Design

### A. Overview

*Nested enclave* provides fine-grained, hierarchical isolated domains for trusted execution environments, supporting multi-level security semantics within an application. An enclave (*outer enclave*) can contain more than one enclave (*inner enclave*). An inner enclave can have full access permission to the execution environments and memory region of the outer enclave, while the inner enclave is completely isolated from the outer enclave. The asymmetric permission between the inner and outer enclaves inspired by multi-level security model [12], allows a security hierarchy inside a trusted execution environment. Peer inner enclaves sharing the same outer enclave are isolated from each other. The inner enclaves can communicate through the outer enclave. Such support for multiple inner enclaves within an outer enclave can provide fine-grained privilege isolation within an enclave.

The concept of nested enclave can be extended to any levels of nesting. As will be discussed in this section, the extra hardware support for nested enclave is to extend the access validation during TLB misses, and thus, arbitrary levels of nesting only increase the validation time without extra hardware complexity. However, for practical purposes, two levels of enclaves can represent many real-world problems, and thus in the rest of the paper, we use two levels (inner and outer) of enclaves. An inner enclave can be associated only with a single outer enclave in the same process, while an outer enclave can have multiple inner enclaves. Although the nested enclave design can be extended to allow an inner enclave to be associated with multiple outer enclaves, as will be discussed in § VIII, this paper will be focused on the single-outer per inner enclave model. Functions for crossing the boundary of inner and outer enclaves are pre-defined by the enclave author. By adding new instructions for transitioning between inner and outer enclaves, switching between inner and outer enclaves can be done directly between them, and it does not require to jump to the non-enclave context and to jump back to an inner or outer enclave.

Nested enclave can be used in several different ways. First, the nested structure can be used to separate the privilege between the privacy-sensitive execution from that with untrusted third-party libraries. The third-party library can only observe the sanitized data by the privacy-sensitive code running in an inner enclave. Second, when multiple users are served by the application, the sensitive processing for each user can be isolated within an inner enclave, as multiple separate enclaves share an outer enclave.

Finally, the communication of peer enclaves can be facilitated by the outer enclave, as the channel via the outer enclave is protected from the outside non-enclave contexts. In the current monolithic model, multiple enclaves can communicate via untrusted memory, and thus all the communicated data must be encrypted and integrity-protected by the enclave software. Such communication overheads can be reduced significantly by using the outer enclave as the communication channel.

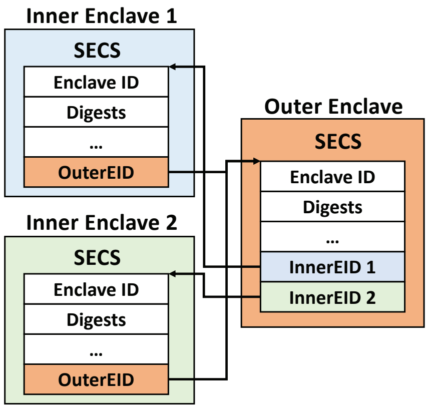


Fig. 3. Modified SECS structures for nested enclave

TABLE I
INSTRUCTIONS FOR SOFTWARE INTERFACE OF NESTED ENCLAVE

| Instructions | Privilege | Description |
|---|---|---|
| NEENTER | user | make transition to inner enclave from outer enclave |
| NEEXIT | user | make transition to outer enclave from inner enclave |
| NASSO | kernel | make inner-outer association |
| NEREPORT | user | get REPORT with inner-outer relations of the enclave |

The memory reserved for the outer enclave is protected by the hardware memory encryption engine (MEE), and thus the fine-grained data transfers are efficient with the cacheline-unit encryption by MEE. Furthermore, if the size is small, the data transfers can be done via the large on-chip last-level cache. In such cases, the encryption by MEE is not invoked as the data exist in plaintext within the CPU boundary.

### B. Meta-Data and New Instructions

**Enclave Meta-data:** Changes in the meta-data for supporting nested enclaves are limited to minor extra fields in the meta-data of each enclave, called SECS (SGX Enclave Control Structures) as described in Figure 3. The SECS of an inner enclave must have a pointer (*OuterEID*) to the SECS of its outer enclave if it exists. Otherwise, OuterEID is set to zero. The SECS of an outer enclave contains a list of pointers (*InnerEIDs*) for all SECSes of its inner enclaves. Once the initialization of each enclave is finalized by `EINIT`, an extra instruction is executed to associate a pair of inner and outer enclaves, which sets the corresponding OuterEID and InnerEIDs fields of their SECSes.

**New Instructions:** Table I shows extra instructions added to support nested enclave. `NASSO` associates a pair of inner and outer enclaves. When `NASSO` instruction is executed with a pair of enclaves, it reads the digest value (MRENCLAVE) and signature (MRSIGNER) from the SECS of each enclave. Those values of an outer enclave are validated against the expected values by the inner enclave, which are included in the inner enclave file, and vice versa. Once the validation successes, the SECSes of the paired enclaves are updated, by setting the OuterEID field of the inner enclave and the InnerEIDs field of the outer enclave.

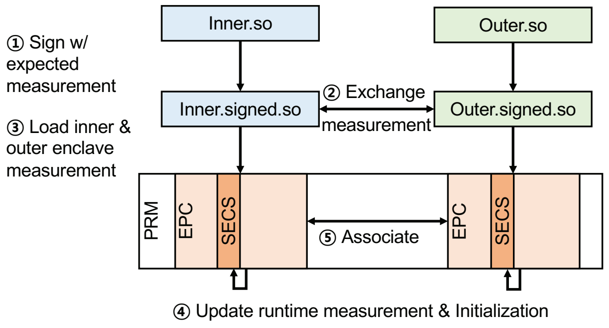

① Sign w/ expected measurement
③ Load inner & outer enclave measurement
Inner.so
Outer.so
② Exchange measurement
Inner.signed.so
Outer.signed.so
PRM
EPC
SECS
⑤ Associate
EPC
SECS
④ Update runtime measurement & Initialization

Fig. 4. Initialization steps for nested enclave

`NEENTER` is similar to `EENTER` in SGX, but it enters an inner enclave from its outer enclave. The original `EENTER` and `EEXIT` are used to transit from inner or outer enclaves directly to non-enclave mode. `NEENTER` jumps to the destination in the inner enclave specified in the input Thread Control Structure (TCS). Before the transition is performed, it checks whether the destination enclave exists and its TCS is currently idle. In addition, the core must be in the enclave mode of the outer enclave, and the destination TCS must belong to the inner enclave of the current enclave. If valid, it flushes the TLB, sets the TCS busy, and transfers control to the destination enclave's entry point. Any invalid invocation results in a general protection fault (GP).

`NEEXIT` exits an inner enclave to its outer enclave. It clears all the information of the inner enclave by flushing the TLB and setting 0s for all registers. In addition, it checks and updates TCS states as it does for `NEENTER`. Asynchronous enclave exits (AEX) from either inner or outer enclaves can occur for hardware exceptions. Unlike `NEEXIT`, in such cases, the processor exits the enclave mode and jumps to the exception handler. Figure 5 shows the transition among the protection boundaries. The semantics of `EENTER` (entering enclaves from untrusted code) and `EEXIT` (exiting to untrusted code) remains the same with the original SGX. To allow transition between outer and inner enclaves, `NEENTER` and `NEEXIT` are used.

`NEREPORT` is an instruction to report the measurement of current enclaves and their connectivity for attestations. The difference between `NEREPORT` from `EREPORT` of SGX is that `NEREPORT` includes the association relationship of two target enclaves.

### C. Initialization

Creating nested enclaves consists of two steps: 1) Each enclave is created with the current SGX enclave creation procedure. 2) After the creations of all enclaves are finished, the nested relationship is specified by `NASSO`. Each enclave is created by `ECREATE`, and its memory is added by `EADD`.

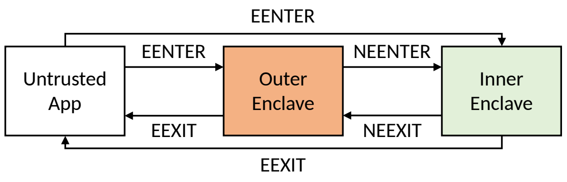


Fig. 5. Nested enclave state transition

During the addition of memory pages, the virtual memory layout must follow the specified layout by the author of the enclave. `ECREATE` and `EADD` measure the initial state and virtual memory layout, accumulating the hashed measurement. For each `EADD`, `EEXTEND` measures the content of the page. The enclave creation is finalized by `EINIT`.

The original SGX loads the enclave binary from a signed enclave file. To support nested enclave, the signed file of an inner or outer enclave must contain the expected measurement of the expected inner or outer enclave. During the execution of `NASSO`, the signed digest is compared to the one in the associated enclave. An inner enclave can be added dynamically to an outer enclave. However, the outer enclave file must contain the expected inner enclave signature for validation. Figure 4 presents the initialization steps for nested enclave.

**Building enclave binary:** To create a binary for an enclave, the enclave author defines Enclave Definition Language (EDL) files. Nested enclave extends the syntax and semantics of the original Intel SGX EDL. EDL files must include interface functions for transitions of (inner enclave, untrusted code), (outer enclave, untrusted code), and (inner enclave and outer enclave). Along with the EDL files, the programmer should supply source codes for untrusted context, inner, and outer enclaves. The signed files are created for inner and outer enclaves, which contain the virtual memory layout and the digest of the compiled binary.

`Ecall` (transition function to enclave mode) and `ocall` (transition functions to non-enclave mode) are supported in the same manner as the original Intel SGX. In addition, Nested enclave adds two new function interfaces: `n_ecall` and `n_ocall`. `n_ecall` allows an outer enclave to call a registered function in an inner enclave. `n_ocall` is for an inner enclave to call a function in the outer enclave. With the `n_ocall`, an application in an inner enclave can call library functions isolated in the outer enclave with the same procedure call syntax.

## D. Access Validation

Nested enclave requires only minor changes in the current hardware architecture supporting SGX. The major required change is to extend the memory access validation logic to consider the inner and outer enclave relationship. As discussed in (§ II-B), the access validation occurs during TLB miss handling. Once a validated entry is inserted into the TLB, no more validation is done until it is flushed. Note that the page table itself is not protected by SGX and thus the validation is required for every TLB miss.

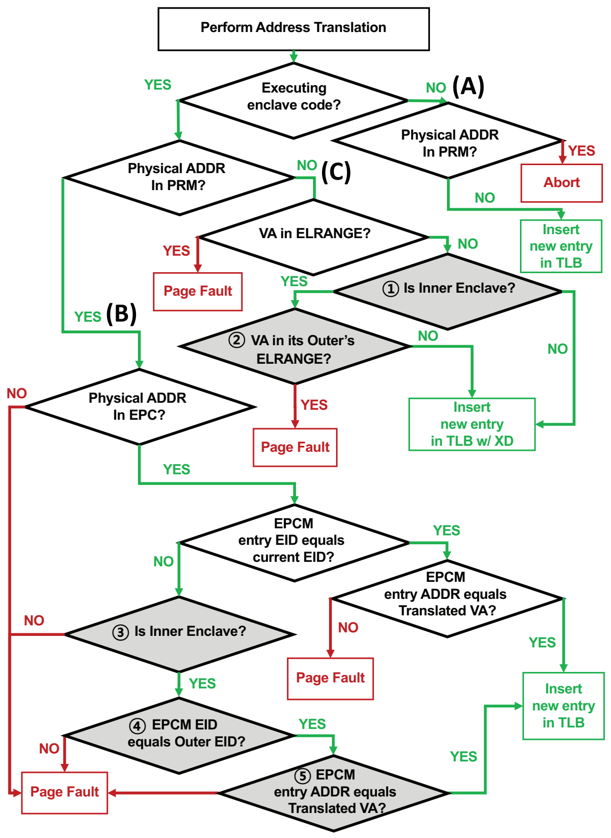


Fig. 6. Access control flows for nested enclave. Modifications on the original SGX access control are in shaded steps

In addition to the memory access validation by the current SGX architecture, the additional validation for nested enclaves is to allow an inner enclave to access the memory region of the outer enclave. During the translation entry validation, if access is valid by the current SGX step, the validation succeeds. For a validation failure, if the current enclave is an inner enclave, an additional step is added to check whether the access is to the outer enclave memory associated with the current inner enclave. For accesses from an inner enclave to its outer enclave, the validation step must succeed.

Figure 6 describes the access control logic with nested enclave. In the figure, the shaded parts represent the extra validation steps by nested enclave. For a TLB miss, the page table is accessed to retrieve the translation entry. First, if the current execution is not in enclave mode **(A)**, the physical page address is compared to the protected memory region (PRM in the figure). If it is not mapped to the protected memory, the entry is inserted into TLBs, which represent normal non-enclave accesses. If the physical address is pointing to the protected memory region, the validation is aborted.

Second, if the current execution is in enclave mode, and the physical page address is in PRM **(B)**, the corresponding EPCM entry is checked. Unlike SGX, if the enclave ID (EID) in the EPCM entry does not match, an additional step is taken. As shown in the step (3), (4), and (5), if the current enclave is an inner enclave, the validation checks the EPCM enclave ID matches its outer enclave ID. Additionally, the virtual address is validated against the stored virtual address in the EPCM entry.

Finally, if the current execution is in enclave mode, and the physical page address is *not* in PRM **(C)**, the validation step checks whether the virtual page address is within the range of the virtual address of the current enclave (ELRANGE). If yes, it causes a page fault, since the virtual page within the enclave does not have a matching EPC page. Such exception can occur for swapped-out EPC pages. If the virtual page address is *not* in the virtual address range of the current enclave, extra steps are necessary for nested enclaves. As shown by the step (1) and (2) in the figure, it checks whether the virtual page address is within the ELRANGE of the outer enclave, if any. If yes, an exception occurs, since the outer enclave virtual page was currently evicted. If the virtual page address belongs to neither of the virtual address ranges of the current and its outer enclave, it is a translation to an unsecure memory page accessed from an enclave. Its executable permission is disabled.

For supporting nested enclaves, the information in EPCM (Enclave Page Cache Map) does not change. Each EPC page belongs only to a single enclave at a time. For an EPC page of an outer enclave, its virtual address follows the specification of the outer enclave, and the virtual address does not change until the outer enclave is removed.

### E. Other Issues

**EPC page eviction:** SGX provides a safe mechanism to evict an EPC page to unsecure memory and reload to an EPC page later. To overcome the limited EPC capacity, such an EPC eviction mechanism allows the protected memory of an enclave to be extended beyond PRM (Processor Reserved Memory). However, the support for such page eviction requires careful coordination to prevent potential attacks exploiting the address translation mechanism. Page eviction changes the mapping between the virtual and physical addresses for an EPC page. Since the access validation is done only during TLB misses, TLB entries must be flushed if any changes in the virtual-to-physical mapping of EPC pages occur. In the current SGX, such mapping changes incur inter-processor interrupts to cause asynchronous enclave exits for all the active threads of the enclave, if the threads are scheduled and running in processors. While exiting from enclave mode by interrupts, TLBs are flushed.

An extra consideration required for nested enclave is for the eviction of EPC pages in an outer enclave. When an EPC page of the outer enclave is evicted, the translation entry for the page can exist not only in the processors running the threads of the outer enclave, but also the processors running its inner enclaves. Therefore, the thread tracking mechanism for handling EPC eviction must be extended to cause asynchronous enclave exits in the inner enclave threads. To support this extended tracking, the SECS of the outer enclave includes the list of the SECSes of the inner enclaves. A simplified, but potentially more costly solution is to send inter-processors to all the cores in the system. It can potentially cause exceptions even for unrelated cores, but the tracking becomes simpler.

**Remote attestation:** The current local and remote attestation only reports the measurement of an individual enclave. However, to support nested enclave, the attestation must be able to report the relationship between enclaves. To provide such a report attesting the relation attributes, the hardware support returns the measurements of all associated enclaves to attest nested enclave. An attestation to an outer enclave must report the measurements of all inner enclaves sharing the outer enclave, in addition to the measurement of the outer enclave. To provides the functionalities, nested enclave supports the `NEREPORT` instruction. Once `NEREPORT` is triggered by a challenger, it checks the association of enclaves and gets their measurements. This information will be sent back to the challenger.

### F. Summary of Hardware Changes

The majority of SGX implementation is known to be based on the microcode implementation [18]. Therefore, the addition of new instructions, the extra meta-data fields, and the extra validation step for access control are all added to the microcode implementation, without any significant change in the hardware logic of the processor.

The memory data encryption is done by the hardware memory encryption engine (MEE). However, such memory encryption is independent of the access validation for enclave memory, since the memory encryption is to protect the DRAM resident data from physical attacks. Therefore, MEE uses a shared key for encrypting the EPC pages for all enclaves. The support for nested enclaves does not add any new complexity to the memory encryption engine.

## V. Methodology

The testbed system consists of Intel i7-7700 processor with 64 GB DRAM. Nested enclave is implemented in the SGX Linux driver with Linux SDK version 1.9 running on Ubuntu 16.04 and Linux kernel 4.13.0. The SDK provides both hardware mode and simulation mode compilation. Nested enclave is built in the simulation mode since it needs to extend the hardware features.

In the emulation framework, we add new attributes (as shown in Figure 3) to SECS, which use reserved fields in SECS implemented in the SDK. One limitation of the simulation mode is that it does not emulate the hardware operations of memory access validation (§ IV-D) and MEE. The nested enclave design does not affect the operations of MEE, but the TLB miss handling time can be slightly increased when an inner enclave attempts to access the EPC region of the outer enclave. For fair comparison, we measure the performance of both of the baseline and nested enclave with the emulation-based evaluation. For the evaluation requiring the overheads of MEE operations as in (§ VI-C), the hardware mode execution is used in the testbed system.

TABLE II
AVERAGE LATENCY OF THE ENCLAVE TRANSITION FOR REAL HARDWARE, EMULATED SGX, AND EMULATED NESTED ENCLAVE.

| Mode | ecall | ocall |
|---|---|---|
| HW SGX ecall/ocall | 3.45us | 3.13us |
| Emulated SGX ecall/ocall | 1.25us | 1.14us |
| Emulated nested ecall/ocall (n_ecall/n_ocall) | 1.11us | 1.06us |

TABLE III
THE NUMBERS OF MODIFIED LINES OF CODE

| Name | Modification | Modified LOC | Original LOC |
|---|---|---|---|
| echo server | C/C++ code | 34 | 883 |
| | EDL | 10 | 28 |
| | SGX-OpenSSL | 0 | 507k |
| SQLite server | C/C++ code | 19 | 501 |
| | EDL | 5 | 30 |
| | SGX-SQlite | 0 | 127k |
| svm-predict | C/C++ code | 27 | 208 |
| | EDL | 10 | 49 |
| | SGX-LibSVM | 0 | 152k |
| svm-train | C/C++ code | 24 | 333 |
| | EDL | 10 | 41 |
| | SGX-LibSVM | 0 | 152k |

**Overhead emulation for transitions:** For inner and outer transitions, we add new instructions to SDK: *NEENTER* (enter to inner enclave) and *NEEXIT* (exit from inner enclave). We extend the ENCLU instruction to perform NEENTER and NEEXIT. We carefully emulate the transition instructions to satisfy the security requirements. When a transition is performed, nested enclave saves the current context including segment registers, registers for ABIs (i.e., parameter passing), stack pointer, and instruction pointer to a reserved stack frame of the entering inner enclave. To emulate the transition overhead, for every transition, nested enclave performs flushing TLBs, zeroing registers and flags, and saving and restoring contexts. To invalidate TLB entries, nested enclave invokes the SGX Linux driver through *ioctl* to perform TLB flushes because the TLB flush operation requires the kernel privilege in x86.

Table II presents the average execution time of enclave transition calls (`ecall`/`ocall` and `n_ecall`/`n_ocall`). We measured the cost with a microbenchmark performing transition calls for 1 million times. In the table, the emulated nested ecalls/ocalls show similar execution times to the emulated SGX ecalls/ocalls. However, the emulated transitions in SGX and nested enclave tend to underestimate the transition costs, compared to the real hardware measurement. Since nested enclave can run only with the emulated framework, the evaluations in this paper are measured with the emulated transition calls for both monolithic enclave and nested enclave for fair comparison.

**Porting Applications:** Table III shows the number of modified lines of codes needed for porting applications originally designed for the conventional enclave to nested enclave. Modifications for C/C++ are for initialization and substitution of library calls to ecalls/ocalls. Nested enclave additionally requires a new EDL to define interfaces among enclaves. We do not modify the SGX-enabled libraries used in the evaluation (OpenSSL, SQLite, LibSVM).

TABLE IV
THE TYPE OF THREE CASE STUDIES AND DATA CLASSIFICATION ACCORDING TO MLS MODEL. INNER ENCLAVES CAN READ TOP SECRET AND SECRET. THE OUTER ENCLAVE CAN READ SECRET ONLY.

| Type | Top secret (inner) | Secret (outer) |
|---|---|---|
| Confinement (§VI-A) | Data for main app. | Data for OpenSSL |
| Data protection (§VI-B) | Private data | Data allowed for ML |
| Fast Comm. (§VI-C) | Data not to expose | Data to communicate |

## VI. CASE STUDIES

Nested enclave provides the hierarchical model which allows applying computations according to the different security levels of data, inspired by the multi-level security model [12], [13]. In the hierarchical model, the data that need the highest level of confidentiality and integrity are performed in inner enclaves, and other data that still need to be protected from the untrusted world are processed in the outer enclave. To demonstrate how nested enclave's model can overcome the limitation of the current monolithic design, this section provides three case studies according to the multi-level security model. Each study provides different ways to map an application structure into inner and outer enclaves. Table IV shows the three case studies and data labeling according to the multi-level security model.

### A. Confinement

It is common in cloud services that servers are composed of the core software component developed and tested by the service provider and third-party components such as utility or communication libraries linked with the core component. As reported in the HeartBleed bug (CVE-2014-0160), a vulnerability in the untrusted library running in the same address space could reveal the memory contents of the main application. This case study demonstrates how to map the server components to inner and outer enclaves to confine untrusted third-party libraries.

For secure communication, the OpenSSL library provides fully-featured implementation and APIs for the Transport Layer Security (TLS) and Secure Sockets Layer (SSL) protocols. OpenSSL includes the SSL standard heartbeat option to verify that the communicating computer is online. Researchers found that, with cleverly crafted heartbeat messages, it is possible to expose information of the communicating application. A crafted heartbeat message can leak up to 4KB from the server-side heap memory, which is freed but might contain security-critical contents.

We tested the Heartbleed attack in SGX. To launch an application in enclave, Intel provides SDK, which includes SGX-OpenSSL, an OpenSSL library ported for SGX applications [28]. SGX-OpenSSL contains modified OpenSSL codes and necessary wrapper functions to be used for SGX enabled

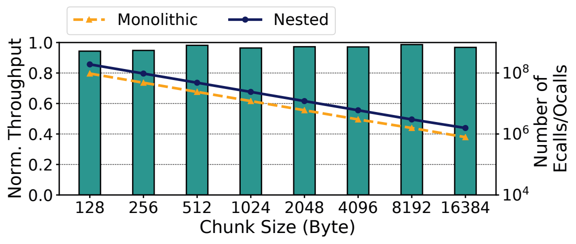


Fig. 7. Throughput of echo server with varying chunk sizes. Throughputs (bars) are normalized to those with the baseline monolithic model.

applications. For this experiment, we slightly modify the SGX-OpenSSL sources to make it behave like the vulnerable version of OpenSSL-1.0.1e. With the current monolithic enclave, SGX-OpenSSL and server code share the enclave, vulnerable to the HeartBleed attack. The secret data in a freed buffer of the server can be exposed as the heartbeat payload. We could reproduce the HeartBleed attack on the application running on the real SGX hardware and obtain secret data of the applications with the HeartBleed bug.

Nested enclave prevents the attack by segregating the untrusted libraries from the critical application domain. Specifically, we can set the SGX-OpenSSL library runs on the outer enclave, while the security-sensitive server code runs on the inner enclave. By the isolation guaranteed between the outer enclave and inner enclave, the flawed SGX-OpenSSL code cannot access data in the inner enclave application. The HeartBleed attack cannot leak any secrets from the inner enclave with the same attack code.

By confining the SGX-OpenSSL library to the outer enclave, function calls between the main application (inner) and SGX-OpenSSL turn to calls crossing the protection boundary, incurring extra TLB flushes and enclave context switches. To measure the performance overhead, we built a simple echo server, which communicates via SSL. We assume the key is distributed to the echo server and client. The encryption and decryption of messages are done in the inner enclave. The server can protect data from the outer enclave but still reuse rich security features of the standard SSL such as the secure handshake protocol to prevent the version rollback or the cipher suite rollback attack. For the comparison, we make the echo server reside in the same enclave with the SGX-OpenSSL library and use it as the baseline.

Figure 7 shows both the throughput of the echo server normalized to the baseline and the number of ecalls/ocalls with nested enclave. The client and server exchange messages with various chunk sizes from 128B to 16KB. The bars show the normalized performance, and the lines show the numbers of ecalls and ocalls for the baseline (`Monolithic`) and nested enclave. For nested enclave, the number of ecalls and ocalls includes `n_ocall` and `n_ecall` used between the inner and outer enclave. In the figure, the performance of nested enclave is only slightly lower than that of the baseline with about 2%∼6% performance degradation. The degradation is caused by additional `n_ocall` and `n_ecall` by nested enclave. The degradation is slightly higher in small chunk sizes because the number of additional calls increases as chunk size decreases.

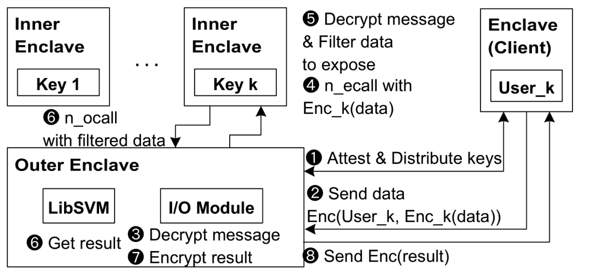

Inner Enclave
Key 1
...
Inner Enclave
Key k
❺ Decrypt message & Filter data to expose
❹ n_ecall with Enc_k(data)
Enclave (Client)
User_k
❻ n_ocall with filtered data
Outer Enclave
LibSVM
I/O Module
❶ Attest & Distribute keys
❷ Send data Enc(User_k, Enc_k(data))
❻ Get result
❸ Decrypt message
❼ Encrypt result
❽ Send Enc(result)

Fig. 8. LibSVM in the outer enclave. Each inner enclave has a separate key.

TABLE V
DATASETS USED FOR EVALUATING LIBSVM. '-' MEANS THERE IS ONLY TRAINING DATA FOR THAT DATASET. IN THAT CASE, TRAINING SET IS REUSED AS TEST SET.

| name | class | training size | testing size | feature |
|---|---|---|---|---|
| cod-rna [50] | 2 | 59,535 | - | 8 |
| colon-cancer [7] | 2 | 62 | - | 2,000 |
| dna [3] | 3 | 2,000 | 1,186 | 180 |
| phishing [37] | 2 | 11,055 | - | 68 |
| protein [52] | 3 | 17,766 | 6,621 | 357 |

## B. *Fine-grained Data Protection*

To show the hierarchical mapping of computation and data, we built a simple system for machine learning as a service [38] using LibSVM [16]. In the machine learning as service, as shown in Figure 8, the service provider opens APIs for trains and inferences to use their infrastructure. Clients feed data to the service provider and receive the computation results. While the clients leverage the service provider's infrastructure for machine learning computations, the clients do not want to expose their private data to the service provider. To protect data privacy, the service provider assigns inner enclaves to run a client's code, and the client can install small functions to perform the anonymization of data in the enclaves. In Figure 8, the inner enclaves decrypt data (the highest secret data) and filter private data not to expose them to the outer enclave. After that, the inner enclave launches LibSVM with the privacy-filtered data. The client can decide how to associate users with the inner enclaves, possibly assigning each inner enclave for a different user.

We report nested enclave performance of the LibSVM case study normalized to the performance of the monolithic design. Table V shows the dataset, some of which do not have testing data size. For such datasets, we run the prediction (inference) experiments with a fraction of their training dataset. Figure 9 shows the execution times for prediction and training, normalized to those of the monolithic enclave case which runs all operations in an enclave. Across all the datasets, nested enclave shows a similar performance to the monolithic enclave because a small number of extra transitions between the inner

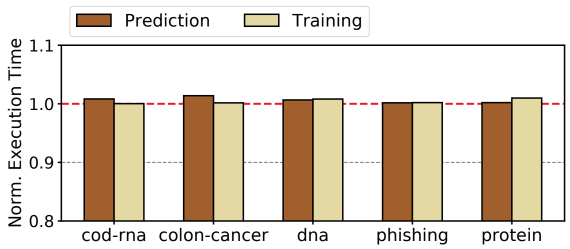


Fig. 9. Normalized execution time for training and prediction. Each execution time is normalized to that with the baseline run.

TABLE VI
SQLITE THROUGHPUT WITH YCSB (UNIFORM RANDOM REQUEST DISTRIBUTION)

| Workload | Normalized Throughput |
|---|---|
| 100% INSERT | 0.99 |
| 50% SELECT & 50% UPDATE | 0.99 |
| 95% SELECT & 5% UPDATE | 0.98 |
| 100% SELECT | 0.98 |

and outer enclaves do not add significant overheads in the LibSVM computations.

In addition, we ported SQLite to nested enclave and the baseline monolithic enclave. Similar to the libSVM scenario, a shared SQLite service runs in an outer enclave. A client sends queries to an inner enclave, the inner enclave parses the queries and encrypts data, and the inner enclave sends query requests to the SQLite service via ocall. Table VI shows normalized throughput for 10000 queries. Considering the entire query process time, the portion of additional data encryption time in inner enclaves is small, incurring less than 2% overheads compared to the monolithic enclave.

## C. Library Sharing and Fast Communication

This case study includes two useful scenarios leveraging the shared outer enclaves: library sharing and fast data communication.

**Library sharing:** In nested enclave, by locating shared codes in an outer enclave, application modules running in inner enclaves can share the outer enclave's code, avoiding unnecessary duplication of codes in enclaves. Such library sharing can reduce the memory footprint of applications and enable faster application launches. The shared library can reduce the memory footprint of inner enclaves, which are otherwise inflated by the duplicated static libraries, relieving memory pressure on the size-restricted SGX memory management. In addition, the binary footprint is one of the important factors for enclave launch performance because SGX verifies the entire binary when loading.

Figure 10 shows the loading times of the original enclave and nested enclave, and the total size of loaded enclaves in memory. The system runs a simple server using the OpenSSL library code (`SSL`) and application code (`App`), both of which are protected by enclaves. The memory footprint of the OpenSSL code is about 4MB, and that of the application

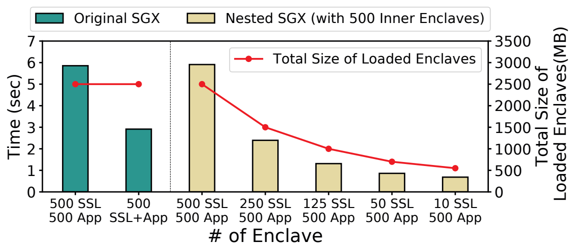


Fig. 10. Time to load enclaves running OpenSSL server.

codes is about 1MB. With the baseline SGX, we show two configurations. The first run launches separate 500 enclaves for OpenSSL and 500 enclaves for application code. The second run launches 500 enclaves which contain both of the library and application codes. The second run represents the current SGX application which put both of the codes in a single enclave. The first run shows the potential overheads of creating separate enclaves for `SSL` and `App`. Note that the memory sizes of the two runs in the baseline are similar, since both of `SSL` and `App` codes are loaded in the two runs.

On the other hand, the nested enclave setup always creates 500 inner enclaves for `App`, but the number of the outer SSL enclaves is varied. For example, when 500 `App` inner enclaves and 50 `SSL` outer enclaves are used, 10 inner enclaves share an outer SSL enclave. For the experiment, after we launch all the enclaves, we associate them at once. In the figure, nested enclave significantly reduces the memory footprints and shortens the loading times by sharing the OpenSSL library. Compared to the baseline 500 `SSL+App` run, nested enclave has shorter latencies except for the 500 `SSL` and 500 `App` configuration, which has a similar latency to the 500 `SSL` and 500 `App` run in the baseline. The results show that as more sharing is allowed, the benefits of reduced memory footprints increase. In addition, as the inner and outer are securely separated in nested enclave, it improves the security of the system as shown in § VI-A.

**Secure and fast communication:** Another useful scenario of using the shared outer enclave is that applications can leverage the shared memory of the outer enclave as a fast communication channel. The application running in an inner enclave assigns an outer enclave and loads trusted code (e.g., written by the same developer) for setting shared memory. Because the outer enclave is protected from the untrusted world, inner enclaves can build a fast message passing system among inner enclaves without encrypting/decrypting data. Still, the data is protected from the untrusted world and unauthorized enclaves. On the other hand, for enclave-to-enclave communication in the current SGX, the monolithic model requires message passing across the untrusted world, necessitating authenticated encryption mechanisms like AES-GCM to guarantee the confidentiality and integrity of the communicating data. In addition to higher throughput, the outer enclave can improve the security of channels, as will be discussed in § VII-B.

To show the potential performance advantage for com-

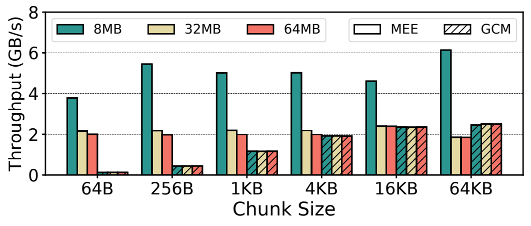


Fig. 11. Performance comparison for the intra-enclave communication protected by MEE (`MEE`) vs the enclave-to-enclave one with Rijndael AES-GCM encryption operation (`GCM`) supported by Intel(R) SGX SDK cryptography library. Both are measured in hardware mode with the testbed system. The legend on top indicates the total footprint of the communication.

TABLE VII
POSSIBLE ATTACKS FROM THE CASE STUDIES AND SECURITY ANALYSIS

| Attack Type | Protection |
|---|---|
| OpenSSL vulnerability leaks main application's memory (§VI-A) | Isolation between enclaves |
| LibSVM and SQLite can read privacy-sensitive data (§VI-B) | Isolation between enclaves |
| OS eavesdrops and controls inter-enclave communication (§VI-C) | Secure inter-enclave communication |

municating via the outer enclave over the software-based encrypted channel through untrusted memory in the current SGX, we compare the performance of intra-enclave communication to communication through the untrusted memory. Since the emulation-based run does not include the overheads of the hardware memory protection by MEE, we use this real hardware evaluation to show the potential benefit of the channel via the outer enclave. In the experiments, two threads in an enclave communicate directly by writing and reading the memory within the enclave to mimic the channel through the outer enclave. For the software-based encryption used by the baseline enclave-to-enclave communication, we use AES-GCM for the protected communication between monolithic enclaves.

Figure 11 shows that the throughput of the intra-enclave channel (`MEE`) is much higher than the conventional enclave-to-enclave channel via AES-GCM (`GCM`) especially when the footprint size is 8 MB, since memory encryption does not occur when the data fit inside the on-chip caches. Leveraging cache-sized messaging is a common technique for fast IPC [10]. On the other hand, AES-GCM needs to perform encryption even if the footprint size fits in the cache. For small chunk sizes, nested enclave performs up to 29.9 times better than the monolithic enclave. As chunk sizes increase, the software-based encryption is also improved as the costs are amortized by processing large chunks.

## VII. SECURITY ANALYSIS

This section first discusses the invariants for the security of nested enclave. Based on the security guarantees, it discusses security analysis for nested enclave by describing possible attack scenarios and how nested enclave can prevent the attacks with the proposed hardware and programming model. Table VII shows attack scenarios from the case studies and corresponding security analysis discussions.

### *A. Protection of Enclaves with Security Invariants*

For the secure protection of the enclave memory access, several invariants must be satisfied as discussed in [18]. The security invariants for nested enclave memory accesses are as follows.

1) If a processor is not in enclave mode, its TLB must not contain any entry with physical pages assigned to the reserved protected memory region.
2) When a processor is in enclave mode, if the requested virtual address is beyond the virtual address range (ELRANGE) of the enclave, it must not be translated to the pages in the reserved protected memory region.
3) When a processor is in enclave mode, if the requested virtual address is within the virtual address range (ELRANGE) of the enclave, the EPCM entry for the translated physical page contains the same enclave ID. In addition, the virtual page address must match the virtual address specified in the EPCM entry.
4) When a processor is in enclave mode, if the requested virtual address is within the virtual address range (ELRANGE) of its outer enclave, the EPCM entry for the translated physical page contains the outer enclave ID. In addition, the virtual page address must match the virtual address specified in the EPCM entry.

Invariant 1-3 are discussed by Costan and Devadas [18] for the current SGX, but Invariant 4 is added for nested enclave. Since nested enclave does not change the access validation steps for the cases related to Invariant 1-3, they are satisfied. As shown in § IV-D, Invariant 4 is satisfied, as the validation checks whether the EPCM entry belongs to the outer enclave.

### *B. Isolation and Secure Channel*

**Isolation between enclaves (case study VI-A and VI-B):** Memory isolation between inner and outer enclaves are fundamental to enforce multi-level security model. In nested enclave, inner and outer enclaves run in the same virtual address space. However, even if the code running in an outer enclave is compromised, it cannot read or modify the contexts of inner enclaves. In addition, even if an inner enclave runs a compromised code, the memory access control prevents any direct accesses to other inner enclaves. OS may create a fake EDL file describing interfaces between inner enclaves, but nested enclave never allow any direct calls among inner enclaves. With these security properties, a confined library in outer enclave cannot access data in any inner enclave (§ VI-A) and each inner enclave can keep its private data from being used by the other inner enclaves or the outer enclave (§ VI-B).
**Secure inter-enclave communication (case study VI-C):** Panoply [44] presents concrete attacks on inter-enclave communication. The attacks leverage the fact that current inter-enclave communications rely on the OS-controlled channel such as IPC primitives. OS, as an active attacker, can drop an IPC request selectively or create a fake or old message.

In the attack scenario, a target application, running in an enclave, asks the verification of an SSL certificate to a trusted certificate manager in another enclave. Using the OpenSSL standard interfaces, the target application registers a callback from the trusted certificate manager with an initialization call, and the callback function does the certificate check within an enclave. The communications are made via an IPC channel provided by the untrusted OS.

To bypass the certificate check in the callback function, OS drops the initialization call, and the target application never executes the callback function. The target application only handles the case where the validation check explicitly fails. The silent drop does not return any error values, making the target application proceed without ensuring that the certificate check is done.

This type of attack is not possible in nested enclave. In nested enclave, the target application and the certificate manager are in the same trusted boundary. Therefore, they run in inner enclaves respectively and share an outer enclave. The outer enclave provides a shared channel among authorized inner enclaves. Applications using the shared channel is completely isolated from the kernel by hardware. OS cannot watch and modify any communication messages transferred in the channel.

**Secure binding of inner and outer enclaves:** Nested enclave allows unrestricted accesses to an outer enclave from the inner enclaves that bind the outer enclave. By default, the inner enclaves can see the memory contents of the bound outer enclave as plaintext; it is at the applications' discretion to encrypt messages for its specific inner-to-inner communication channel. The nested enclave attestation mechanism prevents the unauthorized join of an inner enclave (§ IV-C). It computes the digest of the malicious inner enclave and verifies it with the authorized ones in SECS. If no matching one found, the nested enclave will reject the join; the hardware will not add the ID of the outer enclave to the SECS of the malicious inner enclave, and the memory protection logic will disallow the malicious inner enclave to access memory of the outer enclave.

## VIII. Extending Nested Enclaves

This study limited the nested enclave model to two-level nesting with a single outer per inner enclave. However, with relatively minor changes, further extensions are possible to represent more general relationship among enclaves in a process.

**Multi-level nesting:** Nested enclave can support multiple levels of nesting to represent more than two levels of security layers used in the general multi-level security model. The meta-data change is in the SECS of the outer enclave, which can include the pointer to the next level of outer enclave. In addition to the meta-data change, there are two required updates. In the access validation step in Figure 6, if the current enclave is an inner enclave, it needs to traverse to its outer enclave (step ① and ③). To support multi-level nesting, the traversal must be extended to follow the chain of inner-outer links. The second change is in the TLB flush tracking as discussed in § IV-E. For any virtual-to-physical mapping change in EPC pages of an outer enclave, TLB flush must be enforced for all inner enclaves by traversing multiple levels of inner-outer links. Alternatively, all the cores can be flushed to eliminate the tracking overheads.

**Multiple outer enclaves:** Nested enclave can be extended to allow multiple outer enclaves for an inner enclave. Such multiple outer enclaves allow a more general lattice model where an inner enclave can access the contexts of more than one outer enclave [20]. An example of usage is to support multiple private secure channels from an enclave to different peer enclaves. With multiple outer enclaves per inner enclave, an enclave can set up a separate secure communication channel for each individual enclave in a set of enclaves. Supporting multiple outer enclaves requires to add a list of outer enclaves (instead of a single outer enclave) in the SECS of an inner enclave. In addition, the access validation step in Figure 6 must be extended for step ② and ④ to consider more than one outer enclave.

## IX. Related Work

**Trusted execution with Intel SGX:** With the advent of Intel SGX, there have been many studies to build a secure execution environment with enclaves, supporting various security models and applications [8], [11], [24], [31], [40], [44], [49]. Haven [11], Graphene [49] and PANOPLY [44] allow running unmodified applications on isolated execution using a library OS. SCONE proposed a secure container running in an enclave, using an asynchronous system call of the container to interface with untrusted software [8].

**Secure partitioning for SGX:** PANOPLY [44] and Ryoan [24] divide an application into multiple enclaves to protect security-critical code and data from untrusted 3rd party libraries. However, nested enclave provides the hierarchical privilege separation withing an enclave. There are several prior works to efficiently partition an application into the untrusted domain and trusted domain [21], [32], [48], which can be used for decomposing codes in nested enclave.

**Secure memory protection and control transition:** There are several studies to alleviate the limitations of the current SGX. First, to overcome the memory limitation, several studies proposed to store data in untrusted memory with software encryption and integrity protection [9], [14], [29], [36]. Second, to reduce the performance overhead of crossing boundary, prior studies proposed techniques using additional threads in the untrusted context [8], [36], [54]. A recent version of Intel SGX SDK provides similar switchless calls to reduce the overhead of crossing boundary [47]. Lastly, to protect I/O path, recent studies proposed to establish trusted paths to I/O devices using a verified hypervisor or new hardware-software protection mechanisms [25], [51], [53].

**Privilege separation within TEE:** Prior work have studied to support multiple protection levels in a TEE environment. AEGIS provides several security types on the code and data, and supports multiple secure execution modes on an application [45]. vTZ supports multiple secure VMs using the secure world primitive of ARM TrustZone [23]. Keystone provides multiple privilege levels for the data with custom enclaves [30]. EnclaveDom [34] and Multi-domain SFI [42] support the privilege separation within an enclave by integrating Memory Protection Keys (MPK) and Memory Protection Extensions (MPX). Unlike these studies, nested enclave is implemented in a new hardware mechanism with a small modification, and provides hierarchical security domains. For partitioning at virtual machine granularity, hardware-based techniques isolate each virtual machine under the vulnerability of hypervisors [26], [27], [55].

## X. Conclusion

This paper proposed a new extension to the trusted execution environments for supporting multi-level security within TEE. Based on Intel SGX enclave, nested enclave can support the strong hierarchical isolation within an enclave, and efficient data sharing across enclaves by sharing the same outer enclaves. The paper showed that the proposed semantic extension can be added without significant HW changes in the current SGX support. With the new nested enclave model, the paper investigated three scenarios with an emulated nested enclave execution framework. The evaluation showed that the proposed model can prevent potential information leak, with negligible performance degradation, by separation between the inner and outer enclaves in addition to isolation among peer inner enclaves.

## Acknowledgment

This work was supported by National Research Foundation of Korea (NRF-2019R1A2B5B01069816) and the Institute for Information & communications Technology Promotion (IITP-2017-0-00466). Both grants are funded by the Ministry of Science and ICT, Korea.

## References

[1] "Ghost: glibc vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2015-0235, jan, 2015.

[2] "Heap-based buffer overflow in libavformat/http.c in ffmpeg," https://nvd.nist.gov/vuln/detail/CVE-2016-10190/, sep, 2017.

[3] "Statlog data set," https://archive.ics.uci.edu/ml/support/Statlog+Project.

[4] "Windows dhcp client remote code execution vulnerability," https://securingtomorrow.mcafee.com/other-blogs/mcafee-labs/dhcp-client-remote-code-execution-vulnerability-demystified, jan, 2019.

[5] "Security technology building a secure system using trustzone technology (white paper)," *ARM Limited*, 2009.

[6] "Intel(R) Software Guard Extensions for Linux* OS, linux-sgx," https://github.com/intel/linux-sgx, Intel Corp, 2016.

[7] U. Alon, N. Barkai, D. A. Notterman, K. Gish, S. Ybarra, D. Mack, and A. J. Levine, "Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays," *National Academy of Sciences*, 1999.

[8] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[9] M. Bailleu, J. Thalehim, P. Bhatotia, C. Fetzer, M. Honda, and K. Vaswani, "SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution," in *USENIX Conference on File and Storage Technologies (FAST)*, 2019.

[10] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania, "The multikernel: a new OS architecture for scalable multicore systems," in *ACM SIGOPS 22nd Symposium on Operating systems principles (SOSP)*, 2009.

[11] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with Haven," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

[12] D. E. Bell and L. J. LaPadula, "Secure Computer Systems: Mathematical Foundations," in *Report ESD-TR-73-275, MITRE Corp*, 1973.

[13] K. J. Biba, "Integrity considerations for secure computer systems," MITRE CORP BEDFORD MA, Tech. Rep., 1977.

[14] S. Brenner, C. Wulf, D. Goltzsche, N. Weichbrodt, M. Lorenz, C. Fetzer, P. Pietzuch, and R. Kapitza, "SecureKeeper: Confidential ZooKeeper Using Intel SGX," in *International Middleware Conference (Middleware)*, 2016.

[15] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *USENIX Security Symposium (USENIX Security)*, 2018.

[16] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, 2011.

[17] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu," in *ACM on Asia Conference on Computer and Communications Security (Asia CCS)*, 2017.

[18] V. Costan and S. Devadas, "Intel SGX Explained." in *IACR Cryptology ePrint Archive*, 2016.

[19] Z. Durumeric, F. Li, J. Kasten, J. Amann, J. Beekman, M. Payer, N. Weaver, D. Adrian, V. Paxson, M. Bailey *et al.*, "The matter of heartbleed," in *Internet Measurement Conference (IMC)*, 2014.

[20] A. Ferraiuolo, Y. Wang, D. Zhang, A. Myers, and G. Suh, "Lattice priority scheduling: Low-overhead timing-channel protection for a shared memory controller," in *IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2016.

[21] A. Ghosn, J. R. Larus, and E. Bugnion, "Secured Routines: Language-Based Construction of Trusted Execution Environments," in *USENIX Annual Technical Conference (ATC)*, 2019.

[22] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors," *Cryptology ePrint Archive*, 2016.

[23] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, and H. Guan, "VTZ: Virtualizing ARM Trustzone," in *USENIX Security Symposium (USENIX Security)*, 2017.

[24] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016.

[25] I. Jang, A. Tang, T. Kim, S. Sethumadhavan, and J. Huh, "Heterogeneous isolated execution for commodity gpus," in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2019.

[26] S. Jin, J. Ahn, J. Seol, S. Cha, J. Huh, and S. Maeng, "H-svm: Hardware-assisted secure virtual machines under a vulnerable hypervisor," *IEEE Transactions on Computers*, vol. 64, no. 10, pp. 2833–2846, 2015.

[27] S. Jin, J. Ahn, S. Cha, and J. Huh, "Architectural support for secure virtualization under a vulnerable hypervisor," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2011.

[28] S. Kim, J. Han, J. Ha, T. Kim, and D. Han, "Enhancing Security and Privacy of Tor's Ecosystem by Using Trusted Execution Environments," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017.

[29] T. Kim, J. Park, J. Woo, S. Jeon, and J. Huh, "ShieldStore: Shielded In-memory Key-value Storage with SGX," in *European Conference on Computer Systems (EuroSys)*, 2019.

[30] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting tees," 2019.

[31] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry, "Minibox: A two-way sandbox for x86 native code," in *USENIX Annual Technical Conference (ATC)*, 2014.

[32] J. Lind, C. Priebe, D. Muthukumaran, D. OKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, and et al., "Glamdring: Automatic Application Partitioning for Intel SGX," in *USENIX Annual Technical Conference (ATC)*, 2017.

[33] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, “Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave,” in *Hardware and Architectural Support for Security and Privacy (HASP)*, 2016.

[34] M. S. Melara, M. J. Freedman, and M. Bowman, “EnclaveDom: Privilege Separation for Large-TCB Applications in Trusted Execution Environments,” *ArXiv*, 2019.

[35] B. Möller, T. Duong, and K. Kotowicz, “This POODLE bites: exploiting the SSL 3.0 fallback,” *Security Advisory*, 2014.

[36] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, “Eleos: ExitLess OS Services for SGX Enclaves,” in *European Conference on Computer Systems (EuroSys)*, 2017.

[37] F. T. Rami Mustafa A Mohammad, Lee McCluskey, “Phishing websites data set,” https://archive.ics.uci.edu/ml/datasets/phishing+websites.

[38] M. Ribeiro, K. Grolinger, and M. A. Capretz, “Mlaas: Machine learning as a service,” in *International Conference on Machine Learning and Applications (ICMLA)*, 2015.

[39] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, J. A. Joao, and M. K. Qureshi, “Morphable Counters: Enabling Compact Integrity Trees for Low-overhead Secure Memories,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[40] 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 *IEEE Symposium on Security and Privacy (S&P)*, 2015.

[41] J. Seo, D. Kim, D. Cho, I. Shin, and T. Kim, “FLEXDROID: Enforcing In-App Privilege Separation in Android.” in *Network and Distributed System Security Symposium (NDSS)*, 2016.

[42] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan, “To Isolate, or to Share? That is a Question for Intel SGX,” in *Asia-Pacific Workshop on Systems (APSys)*, 2018.

[43] M.-W. Shih, S. Lee, T. Kim, and M. Peinado, “T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs,” in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[44] S. Shinde, D. L. Tien, S. Tople, and P. Saxena, “PANOPLY: Low-TCB Linux Applications with SGX Enclaves,” in *Network and Distributed*

[56] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: A Sandbox for

*System Security Symposium (NDSS)*, 2017.

[45] E. Suh, “AEGIS: A single-chip secure processor,” Ph.D. dissertation, MIT, 2005.

[46] M. Taassori, A. Shafiee, and R. Balasubramonian, “VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures,” in *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2018.

[47] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, “Switchless Calls Made Practical in Intel SGX,” in *Workshop on System Software for Trusted Execution (SysTEX)*, 2018.

[48] C.-C. Tsai, R. A. Popa, and E. Porter, “Civet: An Efficient Java Partitioning Framework for Hardware Enclaves,” in *USENIX Security Symposium (USENIX Security)*, 2020.

[49] C.-C. Tsai, D. E. Porter, and M. Vij, “Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX,” in *USENIX Annual Technical Conference (ATC)*, 2017.

[50] A. V. Uzilov, J. M. Keegan, and D. H. Mathews, “Detection of non-coding RNAs on the basis of predicted secondary structure formation free energy change,” *BMC bioinformatics*, 2006.

[51] S. Volos, K. Vaswani, and R. Bruno, “Graviton: Trusted Execution Environments on GPUs,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

[52] J.-Y. Wang, “Application of support vector machines in bioinformatics,” in *Master’s thesis, Department of Computer Science and Information Engineering, National Taiwan University*, 2002.

[53] S. Weiser and M. Werner, “SGXIO: Generic Trusted I/O Path for Intel SGX,” in *ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[54] O. Weisse, V. Bertacco, and T. M. Austin, “Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves,” in *International Symposium on Computer Architecture (ISCA)*, 2017.

[55] Y. Xia, Y. Liu, and H. Chen, “Architecture support for guest-transparent vm protection from untrusted hypervisor and physical attacks,” in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 246–257.

Portable, Untrusted x86 Native Code,” in *IEEE Symposium on Security and Privacy (S&P)*, 2009.