Trust Beyond Border: Lightweight, Verifiable User Isolation for Protecting In-Enclave Services

Wenhao Wang, Weijie Liu, Hongbo Chen, XiaoFeng Wang, *Fellow, IEEE*, Hongliang Tian, and Dongdai Lin

Abstract—Due to the absence of in-enclave isolation, today's trusted execution environment (TEE), specifically Intel's Software Guard Extensions (SGX), does not have the capability to securely run different users' tasks within a single enclave, which is required for supporting real-world services, such as an in-enclave machine learning model that classifies the data from various sources, or a microservice (e.g., data search) that performs a very small task (within sub-seconds) for a user and therefore cannot afford the resources and the delay for creating a separate enclave for each user. To address this challenge, we developed *Liveries*, a technique that enables lightweight, verifiable in-enclave user isolation for protecting time-sharing services. Our approach restricts an in-enclave thread's privilege when configuring an enclave, and further performs integrity check and sanitization on critical enclave data upon user switches. For this purpose, we developed a novel technique that ensures the protection of sensitive user data (e.g., session keys) even in the presence of the adversary who may have compromised the enclave. Our study shows that the new technique is lightweight (1% overhead) and verifiable (about 3200 lines of code), making a step towards assured protection of real-world in-enclave services.

Index Terms—Trusted execution environment, cloud computing, in-enclave isolation, security

1 Introduction

Trusted execution environment (TEE) technologies have gained traction recently, thanks to ever-growing demands for privacy protection, wide deployment of Intel's Software Guard Extensions (SGX) [1] and aggressive development of new TEE solutions like AMD's Secure Encrypted Virtualization (SEV) [2] and open-source designs [3], [4]. Cloud service providers like Azure and Google have started providing TEE-based computing support [5], [6]. Major technical companies (Intel, Google, Microsoft, IBM, etc.) have recently joined forces to accelerate the adoption of TEE-based confidential computing, through the Confidential Computing Consortium [7].

However, today's TEEs (e.g., Intel SGX) are limited in the capability to support an Internet-wide service like machine learning inference as a service (MLaaS) and microservices of cloud applications, which become increasingly popular today [8]. These services, once protected by TEEs, often cannot afford to create a separate enclave for each user, due to resource constraint (e.g., a model with millions of parameters for each user) or unbearable delay (e.g., seconds for creating and attesting a microservice enclave that runs only tens of milliseconds). Meanwhile, lack of in-enclave isolation makes it hard to execute different users' tasks securely in the same enclave, which impedes the application of TEEs to protect real-world services.

Challenges in Protecting In-Enclave Service. More specifically, Intel SGX is designed to provide a private memory region – *enclave* to its user for protecting her data and computing task from unauthorized parties, such as the untrusted OS. To this end, the TEE hardware needs to report to the user the measurement of the enclave's initial state, including the code running inside the protected memory, through *remote attestation*, and negotiate a session key with the user for the follow-up communication (e.g., uploading her data to the enclave). Once the user verifies the measurement (which may go through the TEE vendor such as Intel) and is convinced that the enclave code is trusted, she can then allow the enclave to process her code and data.

This model, however, no longer works when the enclave runs a service that involves *multiple* users. Such multi-user support is important due to the cost incurred by creating, maintaining, and destroying an enclave (hundreds of milliseconds of delay, intensive use of memory and CPU) and performing remote attestation (cross-Internet communication, signature verification, etc.). As mentioned earlier, to serve a large number of users with requests for small tasks (e.g., labeling a small batch of images) that involve a large amount of shared resources (e.g., a large machine learning model), keeping an enclave for each of them is neither necessary nor realistic.

Sharing an enclave across multiple users, however, is fraught with security risks: no longer does the execution of

- *Wenhao Wang and Dongdai Lin are with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China. E-mail: {wangwenhao, ddlin}@iie.ac.cn.*
- *Weijie Liu, Hongbo Chen, and XiaoFeng Wang are with Indiana University Bloomington, Bloomington, IN 47405 USA. E-mail: {weijliu, hc50}@iu.edu, xw7@indiana.edu.*
- *Hongliang Tian is with Ant Group, Beijing 100020, China. E-mail: tate.thl@antfin.com.*

Manuscript received 7 April 2021; revised 22 October 2021; accepted 22 December 2021. Date of publication 28 December 2021; date of current version 16 January 2023.
This work was supported in part by the National Key R&D Program of China under Grant 2020YFB1805402, and in part by the National Natural Science Foundation of China under Grant 61802397. The authors from Indiana University Bloomington were supported in part by NIH under Grant R01HG010798 and in part by NSF under Grant CNS-1838083.
(Corresponding author: Wenhao Wang.)
Digital Object Identifier no. 10.1109/TDSC.2021.3138427

a user's task solely rely on the enclave's initial state and her data, since it could be affected by other users' tasks; in the absence of proper isolation among users, one would have no confidence that her private information would not be leaked to other users. More specifically, a malicious user can take control of the enclave through a memory corruption exploit, and steal the secret from other users. Existing solutions [9], [10], [11] compartmentalize users' tasks with software fault isolation. This approach, however, tends to be heavyweight, requiring to audit all memory reference and control flow instructions, which introduces large performance and memory overheads and a large trusted computing base (TCB) that is hard to verify (Section 8).

In-Enclave Isolation. In this paper, we made a step toward lightweight, verifiable in-enclave isolation for confidential multi-user services. Our approach, called *Liveries*, is designed for a *sequential service model* in which an enclave hosts a service that serves a group of users one at a time, with each user issuing a batch of tasks intermittently. The batches arriving at the same time will either be queued and processed one by one or be handled by another enclave. Only after the task batch from one user finishes, will the enclave state be cleaned for serving the next user. Such a sequential service model is characterized by "statelessness" and "short-living", as demonstrated by many real-world services and microservices, which help simplify user isolation: we could focus on the integrity of in-enclave service states to prevent the service program infected by one user from victimizing another user, and on the confidentiality of a user's data left in the enclave to protect it from leaking to the subsequent user receiving the service.

However, such protection still turns out to be nontrivial, in the presence of the adversary – a malicious user who compromises the service program using his task and gains control of the entire enclave. When this happens, not only do we need to ensure non-bypassability of the protection and its correct execution, but we also need to safeguard enclave-wide secrets and individual user privacy from other users: e.g., a user's session key needs to be kept in the enclave for her follow-up communication with the service, which could be exposed to another user if he manages to compromise the service and control the enclave. Another problem is that an in-enclave service's state changes during its operations so protection should be in place to ensure the correctness of the enclave's runtime state, particularly during a user switch.

To address these challenges, *Liveries* takes a strategy that both statically limits an in-enclave program's privilege when building an enclave and dynamically controls its operations during runtime. The strategy is implemented on SGX in our research through an *Enclave Security Configurator* (*ESC*) that minimizes an in-enclave service's privilege during enclave creation and a *Security Monitor* (*SM*) that inspects and sanitizes the service's runtime state in user switch. More specifically, the ESC is designed to configure security policies (e.g., invocation of the SM during ECALLs and write-protection of its code) of an enclave that cannot be changed during its lifetime. Particularly, the ESC ensures that executions of *security critical instructions*, such as `ENCLU`, as well as the `EGETKEY` and `EREPORT` leaf functions are restricted: each of them can only operate within an *sblock*, an atomic code block that once invoked will run to completion or be rollbacked, for preventing abuse of the instruction. For example, the sblock of `EGETKEY` ensures that the seal key it returns will be removed after encryption or decryption, so it will not be observed to a compromised thread using the instruction. In this way, we can leverage these restricted instructions to protect sensitive data (e.g., users' session keys and the keys for key agreement). Under this configuration protection, the SM is designed to check the integrity of security-critical data uploaded during the service's operations (e.g., parameters of a machine learning model) and remove all other user's data before the service runs a different user's task batch. As a result, *Liveries* ensures that each user cannot see others' tasks and data during the service, even under a collusive OS (e.g., issuing exceptions).

Implementation. We implemented *Liveries* over the official SGX SDK. The software TCB of the ESC and the SM has only 3,200 lines of code. As a first step towards formal verification, we presented preliminary results of sblocks verification using the modular software verification toolchain SMACK [12], and the model checking tool Spin [13].

We evaluated the performance of *Liveries* by running KANN [14], a lightweight C library for artificial neural networks in an SGX enclave, over the tasks such as multi-layer perceptron, variational autoencoder, convolutional neural networks, ResNet, character-level text generation with recurrent neural networks and genome sequence prediction. Our experiments show that the overhead introduced by our approach is small (geometric mean 1%, and $< 2\%$ in most cases), way below the cost for serving each user with a separate enclave and the overhead of SFI-based solutions.

Contributions. The contributions are outlined as follows:

- *Lightweight User Isolation*. We designed the first lightweight isolation solution for in-enclave multi-user services. Our approach does not rely on software fault isolation and instead, utilizes security configuration to limit enclave code's privilege and runtime control to time-share the enclave across users, separating their operations from each other. We show that this approach helps achieve small TCB and efficient protection.
- *Security Analysis and Verification*. We thoroughly analyzed the security guarantee of Liveries. The small TCB makes formal verification feasible and we provided preliminary verification results using automatic verification tools.
- *Implementation and Evaluation*. We prototyped the design on both SGX1 and SGX2 platforms, and evaluated the implementation on typical machine learning tasks, demonstrating that our approach incurs only a small overhead.

2 Background

In this section we briefly introduce the basics of Intel Software Guard Extensions (SGX) together with the threats it faces, which are essential to the design and implementation of *Liveries* on the SGX platform. Also we explain the assumptions made in our research.

2.1 Intel SGX Enclave Life Cycle

Intel SGX is a set of x86 instruction extensions that offer hardware-based memory encryption and isolation for application code and data. The protected memory area (called an

TABLE 1
The EGETKEY and EREPORT Leaf Functions

Instruction	EAX	RBX	RCX	RDX
EGETKEY	0x1	Address of KEYREQUEST	Address of OUTPUTDATA	
EREPORT	0x0	Address of TARGETINFO	Address of REPORTDATA	Address of OUTPUTDATA

enclave) resides in an application's address space, providing confidentiality and integrity protection even in the presence of the privileged adversary who controls the OS, BIOS, etc. More specifically, the memory of an enclave is mapped to a special physical memory area called Enclave Page Cache (EPC), which is encrypted and cannot be directly accessed by other system software, firmware (SMM) code, and DMA.

Enclave Leaf Instructions. Intel SGX instructions are organized as leaf functions under two instruction mnemonics: ENCLS (ring 0) and ENCLU (ring 3). Software specifies the leaf functions by setting the appropriate values in the register EAX as input. The semantics of EGETKEY and EREPORT are shown in Table 1 and described in Section 2.2 in more details.

Enclave Creation. An enclave is created by running the ECREATE instruction, which initiates an SGX Enclave Control Structure (SECS). SECS carries enclave meta-data used by TEE hardware and is not directly accessible to software. Particularly, an SECS field, called MRENCLAVE, stores the enclave build measurement value and is updated by EADD and EEXTEND instructions. Here, EADD loads an application's initial code and data into the enclave, copying a source page from non-enclave memory into the EPC and storing the security attributes of the page in Enclave Page Cache Map (EPCM). After all enclave content is ready, EINIT is executed to initiate the enclave and generate MRENCLAVE.

Synchronous Enclave Enter and Exit. Once an enclave is created, a thread can "enter" the enclave through EENTER. Later, the thread can exit the current enclave using EEXIT to transfer the control of the CPU to an unprotected program. A data structure that manages these operations is Thread Control Structure (TCS). TCS contains meta-data used by SGX hardware to save and restore thread-specific information when entering/exiting an enclave. Each thread within the enclave is associated with a TCS. Pages holding TCS are not directly accessible. The number of TCS for an enclave is fixed when the enclave is created on SGX1, which can be used to restrict the maximum number of threads concurrently running the enclave code. When EENTER is invoked, an available TCS is located and based upon its OENTRY field, the application jumps to a pre-determined entry point, transferring the execution to the enclave. TCS can be created dynamically after the enclave is initialized on SGX2. See Section 5 for the detailed design on SGX2.

Asynchronous Enclave Exit and Resume. When an interrupt or an exception occurs in the enclave mode, the processor performs an Asynchronous Enclave Exit (AEX) that saves an in-enclave thread's execution context to its current state saving area (SSA) frame. The AEX then invokes system software's exception handler from the TCS field AEP (Asynchronous Exit handler Pointer). After processing the exception, the system software executes the ERESUME instruction to get back to the enclave mode and restore the execution context. To support the exception handling, at least one *available* SSA frame should exist when entering the enclave with EENTER. The NSSA field of the TCS specifies the number of the SSA frames for a thread, and the CSSA field points to the current SSA frame. CSSA is incremented by an AEX and decremented by an ERESUME.

2.2 Key Derivation and Attestation

Key Derivation. The enclave secret keys, such as the sealing key and report key, are derived from a master derivation key stored in the processor and the identity information in the current enclave's SECS. As shown in Table 1, EGETKEY accepts the address of the KEYREQUEST structure (in RBX) and returns a 128-bit secret key to the address specified in RCX (while RAX selects the EGETKEY leaf)). Both the addresses in RBX and RCX are locations *inside* the enclave.

Local and Remote Attestation. The goal of remote attestation is to generate an attestation signature (signed with the attestation key), which includes the enclave's MRENCLAVE measurement and additional data provided by the enclave. A privileged Quoting Enclave (QE) issued by Intel, which can access the attestation key, is provided for the process.

The enclave to be attested first proves its identity to the QE through *local attestation*, by using the EREPORT instruction to produce a report that cryptographically binds the enclave's measurement and the additional report data. As shown in Table 1, EREPORT accepts 3 operands as input (while EAX selects the EREPORT leaf): RBX is the address of the MRENCLAVE value of the *target enclave* which authenticates the report (i.e., QE for remote attestation), RCX is the address of 64 bytes REPORTDATA structure which contains the additional data, and RDX is the address where the report will be output. The instruction validates that the 3 operands (RBX, RCX, RDX) are *inside* the enclave and computes a cryptographic hash over the attested enclave's SECS data (where MRENCLAVE measurement is included) and the additional data. The instruction generates the cryptographic hash with CMAC using a report key, which as specified by RBX can only be authenticated by the target enclave, as only the target enclave can get the report key using EGETKEY.

With the local attestation, the QE verifies and signs the report using the attestation key, and returns the generated *quote*. We omit the details of the remote attestation protocol [15], [16], and look more into the integrated Elliptic-curve Diffie–Hellman (ECDH) key agreement protocol: the private key and public key pair a and ga are generated inside the enclave, and after receiving the public key gb, the enclave can compute the shared secret and derive the key derivation key KDK. Then the additional report data H is computed as follows (where VK is short for verification key and $CMAC$ is cryptographic MAC algorithm):

$$VK = CMAC_{KDK}(0x01||''\text{VK}''||0x00||0x80||0x00),$$
$$H = SHA256(ga||gb||VK).$$

As a result, the quote enables cryptographic binding of the public key ga and the enclave's MRENCLAVE measurement, and the user can be convinced the private key a is generated in the attested enclave.

2.3 Scope and Design Goals

It has been demonstrated that enclave code may contain memory safety vulnerabilities, such as buffer overflows [17]. Hiding the enclave content or randomizing the enclave layout does not stop memory corruption attacks in SGX [18], [19]. Even fine-grained control flow integrity (CFI) can still be bypassed, e.g., through counterfeit object-oriented programming (COOP) [20] or non-control data-only attacks [21], [22].

In this paper, we assume that the service code itself is not malicious but potentially vulnerable to exploitation, e.g., due to a memory-safety violation. In the following we summarize the attack vectors that can be exploited after the adversary controls the enclave, and the goal of the paper is to provide a lightweight solution to protect the users from these attacks.

- *Man-in-the-Middle Attacks Against Remote Attestation*. As shown in previous studies [18], [19], a corrupted enclave can act as an oracle to assist a malicious program running outside the enclave to generate a "legal" cryptographic report authenticated and signed by the QE. In this way, the adversary can emulate the SGX environment with software and steal all the data from subsequent users. We attribute such a threat to the losing control of *critical data* (i.e., the private and public keys may be generated outside the enclave) and *critical instructions* (i.e., the `EREPORT` instruction is abused to generate the report on untrusted data).
- *Snooping on Concurrently Running Tasks or Exposing a Prior User's Data*. With the arbitrary memory read/write capability, the adversary can get and leak data from other users' tasks served concurrently in an enclave or the residue data from prior users (session keys, intermediate results etc.).
- *Victimizing Subsequent Users by Tampering With Enclave Data*. The adversary can tamper with enclave control data such as function pointers to redirect control flows, or non-control data to interfere with the service to subsequent users' requests or leak their data.

2.4 Threat Model

We assume a strong adversary with the following capabilities: (1) full knowledge about the source and binary code of the in-enclave service and the memory layout of the enclave, (2) arbitrary memory read and write, and (3) manipulation of the enclave's control flow. In the meantime, we assume that the adversary cannot circumvent the hardware protection of TEE. We consider (PKI-based) user authentication orthogonal to our work: e.g., the user's data can be signed by her private key certified by the service provider and therefore does not need to be sent to the service provider for signing. We assume that the source code is available and leave the adoption when only binary code is present as future work (e.g., using binary rewriting techniques [23]).

We designed Liveries to work on both SGX1 and SGX2 platforms. On SGX1, page properties cannot be changed after an enclave is built. Therefore, we consider the situation where the users and the service provider only upload data into the enclave so no page bears both writable and executable permissions. We show this design can be extended to SGX2 (Section 5). We trust the enclaves provided by Intel, such as QE. We do not consider side channel attacks (e.g., speculative execution attacks) and voltage manipulation based fault attacks [24], [25] in the current design and leave them for future work (see Section 8 for a brief discussion on side channels). Denial-of-service attacks are out of the scope.

3 Design of Liveries

3.1 Overview

Design Requirements. Liveries is built to meet the rising privacy demands for today's online services like deep learning inference as a service [26], [27], personal health analysis [28] and natural language processing [29]. These services involve sensitive personal data, which their owners often do not want to share with the service providers: e.g., users of an image classification API might be reluctant to expose their photos to the machine-learning-as-a-service platform. A solution is expected to protect the user data from both untrusted providers and other users, and ensure the correctness of the service results: the outcome returned to a user should only depend on her data, the service data (e.g., model parameters) and service code, not other users' data. Given the large amount of resources these services involve (e.g., gigabytes of model parameters) and the popularity of the microservice architecture today (serving each user in subseconds), running a dedicated enclave for each user becomes unviable. So, the new solution is expected to enable effective in-enclave isolation for serving different users.

Straw-Man Solutions. Keeping an enclave for each user is unlikely to scale (to millions of users). Creating an enclave when a user's request comes and destroying it after the service is done would incur hundreds of milliseconds of delay for each request.[1] Maintaining an enclave service pool cannot address the problem since each user's request still needs to go through remote attestation to authenticate the enclave and establish a secure channel, which may take seconds just for communicating with the attestation service. Even worse, the man-in-the-middle attacks on remote attestation still work when critical data and critical instructions are unprotected.

Our Design. As mentioned earlier, our approach is designed for the *sequential service model*. In this model, the enclave serves a group of users but processes the tasks from the same user at a time. To use the service, each user needs to run remote attestation with the enclave and exchange a session key the first time when she interacts with the enclave. We refer the interested readers to a detailed description of the remote attestation protocol provided by Intel [30]. Then, she can utilize the same key (securely stored in the enclave) to communicate with the in-enclave service and send in subsequent task batches. Only after the task batch from one user finishes, will the enclave be sanitized for serving the next user. Under this model, different users' tasks are either queued and processed sequentially at the enclave or handled concurrently by different enclaves. We show that the *sequential* service model, characterized by "*statelessness*" and "*short-living*" as demonstrated by many real-world services and microservices, helps to simplify user isolation, bringing in benefits such as low performance/memory overhead and an extremely small TCB, as compared with software fault isolation (SFI, see Section 7.3).

1. According to a recent study, many production services in Microsoft relying on DNN inference are often constrained to single-digit millisecond latency budgets [26].

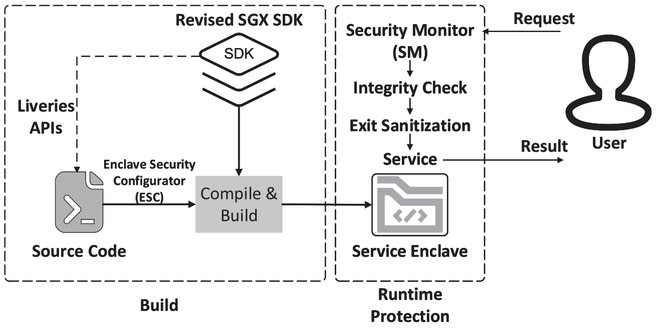


Fig. 1. Design overview.

Isolation under the service model is expected to prevent a prior user from stealing a later one's data through the compromised service and prevent the later user from collecting his predecessor's content left in the enclave. For this purpose, the enclave is built with an Enclave Security Configurator (ESC) to "hardcode" it with protection on critical program components and data (e.g., session keys) and restrict the use of critical instructions (e.g., those that may change enclave security settings). This approach limits what an in-enclave thread can do and thereby what an adversary could achieve by hijacking the thread's control flow: even after compromising the enclave, still the adversary cannot run instructions in violation of the configured security policies. Under this built-in protection, further, we use a Security Monitor (SM) to perform runtime checks on the integrity of the critical data uploaded by the service during its operations and to clean up the sensitive content left by a user before the next one's tasks come in. Fig. 1 illustrates the architecture of our system. At the center of the ESC is the technique for restricting the ways critical instructions can be used, which is elaborated under the context of SGX in Section 3.2.

3.2 Restricted Instruction Execution

Restriction With Sblock. The key to limiting an enclave program's privilege is to restrict the use of security-critical instructions. For example, EREPORT binds a public key generated for attestation and key exchange with the measurement of an SGX enclave state and can be abused by a malicious enclave program to link an attestation report to a key controlled by the adversary for a man-in-the-middle attack [18], [19]. As another example, EGETKEY returns a 128-bit secret key from the processor-specific key hierarchy and can be used to hide secrets inside an enclave if it is protected from a compromised enclave program. By setting enclave page properties to either writable or executable but not both (Section 4.1) through the ESC, we can ensure that these critical instructions cannot be brought into the enclave by the adversary. So, the only attack surface left is to leverage the instructions already in the enclave code segments (called *gadgets*), through the techniques such as return oriented programming (ROP).

To control those instructions, the ESC first finds them from all the code in the enclave (service, SM and SGX SDK), and then builds a protective *sblock* for each of them. An sblock is a set of instructions with an entry and an exit instruction, such that if the execution begins at the entry instruction, it will leave the sblock from the exit instruction before running any enclave instruction outside the sblock. Therefore we can use this *atomicity* property to ensure that all protected instructions can only be used in an authorized way. For example, EREPORT can only take an authorized public key as input; the secret key fetched by EGETKEY will be deleted immediately after decryption, to avoid exposing to the instruction outside the sblock.

To achieve this atomicity, a simple solution seems to utilize transactional synchronization extensions (TSX) [31], [32]. This approach, however, does not work since a transaction is always aborted by ENCLU. Our design takes a unique strategy: the ESC sets a flag (e.g., by storing a specific value to a register) at the beginning of the block, which is checked at the end of the block and resets if correct (checking and clearing the register); otherwise, the sblock terminates the whole enclave service. For security and performance purposes, the flag shall possess the following properties:

- *Flag Setting Can be Controlled*. Since we assume an adversary with an arbitrary memory read and write capability, a memory variable cannot be used as the flag.
- *The Use of the Flag can be Easily Eliminated in Normal Code*. Since the adversary can manipulate enclave code's control flow, any code gadget that can set the flag, except for those at an sblock entry, needs to be removed from the binary. Unaligned code gadgets that may reside within instructions or cross different instructions also need to be removed.

Instead of using general purpose registers which are frequently used in normal code, our approach takes advantage of segment registers, i.e., GS and FS base registers supported in the 64-bit mode which can be configured by the ring-3 enclave code. While FS is used by the SGX SDK to support thread-local storage, GS is unused and therefore perfect for hosting the flag. Moreover, the value of GS can be accessed only with special instructions, i.e., WRGSBASE and RDGSBASE to write and read GS respectively. These instructions are not supposed to appear in normal user programs, making it easy to remove their occurrences by scanning and rewriting the enclave code (evaluating Q3 in Section 7.2).

Sblock Definition and Properties. Here we describe sblock and its protection for security-critical instructions when multi-threading is disabled. We discuss multi-threading support in Section 5.

Definition 1 (Sblock). *Consider a block of enclave instructions that can be represented as a triplet (C, $entry$, $exit$), where C is the instruction block, $entry$ is the unique entry-point instruction for the block and $exit$ is the unique exit-point instruction. Such a block is an sblock if and only if it is self-contained and atomic, with the following two properties when its $entry$ is run: (1) the control flow of C is independent of the enclave's memory and architectural states (e.g., the content of registers) set by instructions outside C; and (2) $exit$ is the last instruction before the processor executes other enclave instructions that do not belong to C.*

Fig. 2 shows examples and counter-examples of sblock within the control flow graph (CFG) of an enclave program. For example, basic block 3 and 4 together form an sblock, with the entry at the beginning of basic block 3, and the exit at the end of basic block 3. Basic block 1 does not form an sblock, since the control flow may go to basic block 2 or 3

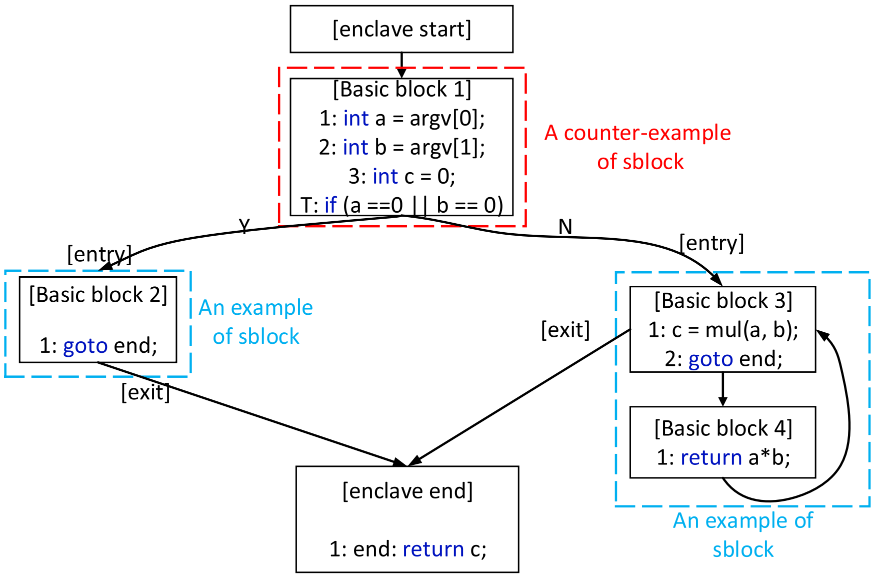


Fig. 2. Sblock examples and counter-example(s).

depending on the values of a and b, which are set outside the basic block.

It is important to note that even a self-contained code block can be interrupted. In our design for SGX1, the ESC disables multi-threading in enclave (see $P1$ in Section 4.1) and enforces an exception policy ($P2$) to ensure that the thread can only reenter the enclave through `ERESUME`, so the execution will resume from where the interrupt happens with all enclave state intact.

Under these security policies, we can control the use of security critical instructions (such as `EGETKEY` and `EREPORT`) with sblocks. Formally, let $I_{k=1\cdots n}$ be a sequence of critical instructions. We can build a sequence of sblocks ($C_{j=1\cdots n+1}$, $entry_j$, $exit_j$) as follows (Fig. 3): $entry_1$ sets a flag through the instruction `set_flag` (e.g., using the `GS` register); $exit_{n+1}$ checks the flag and when fails, throws an exception (e.g., destroying the enclave); $exit_j$ and $entry_{j+1}$ ($1 \leq j \leq n$) are the same instruction (i.e., I_j). Together they form a concatenated chain of sblocks.

Proposition 1. *With the protected flag (`set_flag` only run by $entry_1$), if any protected instruction I_j is executed, either (1) instructions in all sblocks $C_{1\cdots n+1}$, from $entry_1$ to $exit_{n+1}$, have correctly run to completion before the execution moves outside the sblocks, or (2) when $exit_{n+1}$ is reached, flag-checking will fail and an exception will be thrown, and in this case, no enclave instructions outside the sblocks will be run before $exit_{n+1}$. Either way, the flag will be unset.*

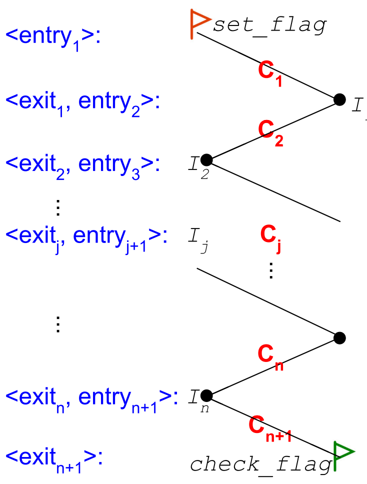


Fig. 3. Example of an sblock chain.

Proof. To run I_j, the execution either starts at (1) $entry_1$ or (2) another instruction in $C_{1\cdots n+1}$. In the case (1), according to Definition 1, if in the end the execution moves outside C_1, $exit_1$ must be reached (the second property of the definition). Since $exit_{j=1\cdots n}$ is also $entry_{j+1}$, so the execution will continue within the sblocks (from C_1 to C_{n+1}), until $exit_{n+1}$, which checks and resets the flag. In the case (2), if I_j is executed, since I_j is $entry_{j+1}$, the execution will continue within the sblocks (from C_{j+1} to C_{n+1}), until $exit_{n+1}$. Given that the flag has not been set at $entry_1$, the check at $exit_{n+1}$ will fail and an exception will be thrown. □

Discussion. As mentioned earlier, our design for SGX utilizes the `GS` register for flag. Access to the register is restricted by the compiler and further checked to ensure that no gadget in the enclave can be leveraged to violate the policy through binary code inspection. Also note that `set_flag` is always placed at $entry_1$, which will be reset at $exit_{n+1}$. So the adversary cannot use `set_flag` at one sblock chain to circumvent the protection at another.

To prevent ABI poisoning attacks [33], the processor extended state is sanitized on enclave entry. Under the control of multi-threading and exception handling (Section 4.1 for single threading and Section 5 for multi-threading support), the sblock chain described above can achieve atomicity in protection: the enclave adversary will only observe the enclave state after the whole chain has been executed if a protected instruction is run. Using the technique to protect key instructions (`EREPORT` and `EGETKEY`, Section 4.1), the code around these instructions should form sblocks, both self-contained and atomic. This has been verified (Section 7.1) to ensure that (1) the targets of all direct branches are inside the block; (2) the code does not contain indirect jumps and calls; and (3) the code does not contain memory safety flaws to prevent overwriting the return addresses.

4 Enabling Liveries on SGX1

4.1 Enclave Security Configurator (ESC)

The ESC is designed to configure the protection of an in-enclave service before creating its enclave according to a set of security policies. The configurations are meant to limit the adversary's capability even after he controls the entire enclave and cannot be changed during the enclave's lifetime. The policies enforced through the configurations include those for memory protection ($P0$), sequential service model ($P1$ and $P2$), instruction protection and data protection ($P3$ and $P4$) using sblocks, and SM protection ($P5$). Following we elaborate on these policies and their enforcement.

P0: W⊕X Permissions on Enclave Pages. To prevent an adversary who controls the enclave from running arbitrary code, the W⊕X policy needs to be enforced on all regular enclave pages (with page type PT_REG). This can be easily done and verified to the enclave user on the SGX1 platforms since page properties are part of the enclave measurement and cannot be changed after the enclave is initialized. We discuss enforcing $P0$ on SGX2 platforms in Section 5.

P1: Single-Threaded Enclave. The enclave is configured to disable multi-threading by setting TCSNUM to 1 (Section 2) on the SGX1 platform. We discuss how to support multi-threading on SGX2 in Section 5, which uses of a separate single-threaded enclave to keep secrets and a multi-threaded enclave to provide the service.

P2: Secure Exception Handling. Another threat to the atomicity of sblock is exceptions, which can be triggered by the event both inside and outside an enclave (e.g., a page fault). If an exception occurs during an sblock's runtime, all its context (memory and register content) will be saved by an AEX to the state saving area of the running thread (SSA), which is both readable and writable to another thread later entering the enclave for exception handling. A simple solution is to disable the exception handling altogether, forcing any entry back into the enclave to go through ERESUME that returns to the position where the exception happens. This can be configured to the enclave by setting the number of SSA (NSSA) to 1: given that each thread needs an SSA to run, any attempt to return to the enclave can only resume the interrupted thread, since there is no SSA available to a new exception handling thread.

If there might be situations when exception handling is needed, we could disable it *only when running a protective sblock*. To this end, we set the entry of EENTER to the SM ($P5$), which whenever triggered by the execution of the instruction, first checks the flag (by retrieving the GS register from the SSA): if set (an sblock is interrupted), it immediately exits the enclave. This ensures during an sblock's operation the enclave can only be entered through ERESUME, while exception handling in other situations will proceed as normal. The enclave exception handler is checked not to change the value hosted at the SSA address reserved for GS through code review, to prevent overwriting the register after ERESUME is executed.[2] The restriction does not affect exception handling, as GS is not supposed to be modified in any normal cases.

2. On the hardware available to us GS is loaded from the SSA (see Section 5 for more details).

```
DECLARE_GLOBAL_FUNC do_eaccept
                       SE_PROLOG
<entry1>               mov $0xdeadbeaf, %eax
                       wrgsbase %rax
                       mov $SE_EACCEPT, %eax
<exit1>, <entry2>      ENCLU
<exit2>                rdgsbase %rax
                       cmp $0xdeadbeaf, %eax
                       je CHECK_SUCCESS
                       ud2
                       CHECK_SUCCESS:
                       xor  %rax, %rax
                       wrgsbase %rax
                       ...
```

Fig. 4. Restriction on the use of ENCLU.

Note that we include in all sblock chain instructions to clean up the SSA before leaving the chain, as the content in that area will always be there should an exception happen.

P3: Protection of Critical Instructions. As mentioned earlier, Liveries protects critical instructions like EGETKEY and EREPORT using sblock chains. On the binary code level, however, these instructions are all compiled into ENCLU, which invokes different SGX non-privilege leaf functions (EGETKEY, EREPORT, etc.) based upon the parameters set in the register EAX [34]. So, not only do we need to protect EGETKEY and EREPORT, but proper guard should be in place for all other occurrences of ENCLU to ensure that its input parameters have not been changed to run a different instruction (e.g., EREPORT) or when this happens, the abuse will be detected by the flag checking. For this purpose, a simple sblock chain shown in Fig. 4 is adequate in most cases, given the fact that most instructions built on top of ENCLU do not cause any control-flow change to the code block starting with ENCLU and ending at the flag check, which fits well into the sblock definition. The only exception is EEXIT, which moves a thread outside the enclave. To prevent the adversary from abusing its underlying ENCLU, the ESC simply adds an ud2 instruction right after EEXIT (as already did in the official SGX SDK): if the instruction is properly run and causes EEXIT to happen, ud2 will never be taken, since the current thread leaves the enclave; if the adversary changes EAX outside the block and jumps to the ENCLU to create EGETKEY or others, ud2 will be run and the enclave will be terminated.

Recalling that the operands of EGETKEY and EREPORT should be addresses *inside* the enclave (Section 2.2), the instruction output will not be leaked to outside the enclave before the flag check is performed (see Section 8 for a brief discussion on transient execution attacks). Unlike the simple protection in Fig. 4, sblock chains for EREPORT and EGETKEY are more complicated. These two instructions (both built on top of ENCLU) are meant to safeguard critical enclave data, which we elaborate below.

P4: Protection of Critical Data. Most challenging in user isolation is to protect the confidentiality and integrity of critical enclave data from the adversary in full control of the enclave. Such data include the private key and the public key used in the ECDH key agreement and the session key negotiated with each user. Protection of such data is of critical importance: e.g., if the adversary can get his hands on a user's session key, he will be able to steal her data from her

```
<entry>:
    set_flag();

    // select curve and generate key pair
    curve ← uECC_secp256r1();
    (public_key, private_key) ← make_keypair(curve);

    // seal encrypted data in memory with EGETKEY
    // two sblocks with ENCLU for EGETKEY as exit
    seal_data(private_key);
    seal_data(public_key);

    // clear plaintext data
    // EGETKEY key already cleared
    clear_data(private_key);
    clear_data(public_key);
<exit>:
    check_flag();
```

(a) Seal critical data, i.e., the ECDH key pair.

```
<entry>:
    set_flag();

    // unseal public key and clear secret key
    // 1 sblock with ENCLU for EGETKEY as exit
    public_key ← unseal_data();

    // generate report data using sha256
    report_data ← {{0}};
    report_data ← sha256(public_key);

    // 1 sblock with ENCLU for EREPORT as exit
    report ← create_report(report_data);

    // clear plaintext data
    clear_data(public_key);
<exit>:
    check_flag();
```

(b) Unseal critical data and create enclave report.

Fig. 5. Pseudocode for the protection of critical data with sblocks.

follow-up task; if the public key for the key agreement is replaced, the adversary can launch a man-in-the-middle attack, impersonating the enclave to the user [18], [19].

Liveries addresses those risks by utilizing EGETKEY to derive a secret key from SGX hardware and encrypting critical data (session keys, public/private key pairs) with the key. So it is important to ensure that the enclave adversary cannot run the instruction to get the key or directly steal it from the memory or registers. To this end, we build an sblock chain that derives the key, (en)decrypts secret data under the key, performs desired operations with the secret (e.g., decrypting a user's data with her session key) and then removes both the derived key and the secret. In case that the compiler might spill some of the secret to the stack, all registers and stack need to be cleaned before exiting the sblock chain. Fig. 5a illustrates such an example. Leveraging the atomicity of the sblock chain, our approach ensures that the adversary can only observe the states before and after the execution of the chain and therefore will not be able to compromise the confidentiality of the data.

Further by linking EGETKEY and EREPORT together, we can protect the integrity of the public key when generating a cryptographic report for remote attestation with a new service user. Specifically, the public/private key pair for the attestation is sealed under the hardware derived key. To produce the report, Liveries instruments the enclave code to embed an sblock chain (Fig. 5b), which first runs EGETKEY to retrieve the key to unseal the public key before removing the derived key, and then delivers the public key to EREPORT for binding it to the report. During this process, the atomicity of the chain ensures that only the right public key is used in the report, thereby defeating the man-in-the-middle attack [18], [19].

P5: Non-Bypassablity of the Security Monitor. In addition to the ESC, Liveries also runs an SM inside the enclave to check the integrity of the service data and remove sensitive user content during every user switch. To defend against an enclave adversary, the SM's code is protected by the page permission and its data is sealed using the protected EGETKEY. Further, we need to ensure that the SM will be run upon each switch. For this purpose, we set the entry of the enclave (saved in the TCS and immutable during the enclave lifetime) to point to the SM, so it will be invoked each time when EENTER is used to deliver a new task into the enclave through an ECALL. However, when the in-enclave service is compromised, the adversary might directly read in the user's *encrypted* data without going through ECALL. To defend against this attack, we design the SM to incorporate integrity check, memory sanitization, and decryption of the user data together into an sblock chain to ensure its atomic execution (Section 4.2).

4.2 Security Monitor

The goal of the SM is to protect the enclave data that cannot be secured by the enclave configurations during user switches. These data include the function pointers, dynamically loaded parameters (e.g., those for a machine learning model), a user's task data, and all temporary data generated during the processing of her task. The SM is invoked when an ECALL is made, which first finds out whether an exception is triggered inside the enclave. This is determined by monitoring the SSA. Specifically, the SM sets a marker in the SSA for the thread, e.g., writing 0 to the address within SSA that reserved to store rip. The value will be overwritten if an exception happens. If an exception is detected, the SM performs exception management according to $P2$. Otherwise, it retrieves the user identity and pointers to the encrypted data provisioned by the user from the ECALL to prepare for the invocation of the new task. For this purpose, the SM first performs an *integrity check* to ensure that critical system parameters of the service have not been tampered with, and an *exit sanitization* to "reset" the service state and remove all the exiting user's data. These operations, once successful, are followed by decryption of the new user's data. All such operations are protected with an sblock chain to ensure atomic execution.

Following we present the design of these components. For the ease of presentation, we describe as MEASURE_AREA the memory location keeping data to be measured for the integrity check (e.g., machine learning parameters), TMP_AREA hosting temporary and intermediate user data (e.g., the input image), and ENC_AREA holding critical data under encryption (e.g., session keys). Particularly, MEASURE_AREA is separated from other areas based on different API functions used by the service developer (Section 6.2).

Integrity Check. The integrity check proceeds by computing a hash measurement of MEASURE_AREA and comparing it with the expected value (denoted by RUNTIME_HASH). If the check fails, the service is terminated. Otherwise, the task data is decrypted and handed over to the server.

Generation and protection of RUNTIME_HASH are straightforward when *all service parameters are pre-determined, before initialization of the enclave*. For example, biomedical services such as mapping genome reads to a reference genome (which is public and embedded in the enclave binary) and generating sequence alignment map files do not rely on additional input from the service provider during the enclave's runtime. In this case, we can simply compute the RUNTIME_HASH of its parameters and keep it on *read-only* pages while building the enclave. More complicated is the situation where *service parameters are uploaded during the enclave's runtime and could be updated by the service provider*. An example is machine learning inference as a service, in which neural network parameters may be loaded after the enclave is initialized and could be changed by the service provider later. When this happens, RUNTIME_HASH can only be generated during the runtime and needs to be protected from unauthorized modification. Following is our solution.

We hardcode the service provider's public key in the service code for authenticating the parameters uploaded into the enclave. Whenever the SM receives an update request from ECALL, it first runs the integrity check using the protected RUNTIME_HASH calculated from the initial state of the enclave and cleans up all registers and memory, and then verifies the signature on the parameters from the service provider. If correct, the SM uploads the parameters to MEASURE_AREA, generates its RUNTIME_HASH, and encrypts the hash using the derived key from EGETKEY. All these operations are protected by an sblock chain. Using the hash for the integrity check, the SM needs to retrieve the key to decrypt RUNTIME_HASH, which is guarded together with exit sanitization and user data decryption by a different sblock chain.

Exit Sanitization. The purpose of exit sanitization is to reset the service state and clean up the previous user's data. Except for the persistent data of the user (e.g., the session key) protected in ENC_AREA, all data in TMP_AREA is cleaned up, together with the data of the SSA and registers.

Atomic Execution Protection. As mentioned earlier, the SM uses sblock chains to ensure the atomicity of its operations: only after the integrity check is passed and all memory states are cleaned up, will it decrypt new user's data or the new service parameters; also all critical data, such as derived key and RUNTIME_HASH, will be either removed or encrypted.

5 Multi-Threading on SGX2

SGX supports multi-threading, which however poses a new challenge to protecting in-enclave services. Specifically, under the surveillance of a concurrently running thread, the execution of an sblock can no longer ensure confidentiality of the data involved: for example, the secret key derived from EGETKEY (for protecting critical data) can be exposed to the monitoring thread while being used; this risk cannot be addressed by keeping the secret key in registers, as the attacker may trigger an AEX so that the register values are offloaded to the memory (i.e., SSA) by the hardware. In this section, we present the design of secure multi-threading with minimal hardware extensions over SGX2.

SGX2 Instructions. SGX2 has introduced a new set of leaf functions with the following capabilities: (1) dynamic creation and addition of a page to an already initialized enclave; (2) update of an EPC page's permissions; (3) change of an EPC page's type. These operations take effect only when they are directly invoked by the enclave code through user leaf functions, or by the untrusted driver through supervisor leaf functions and the changes they cause are accepted by the enclave code (using EACCEPT). This guarantees the changes made by the leaf functions will not weaken the protection of the enclave.

We can protect the leaf functions (e.g., EACCEPT) using sblock as described in Section 3.2. The measurement hash RUNTIME_HASH can be protected by setting the RUNTIME_HASH page to read-only after it is created. On SGX2, Liveries can support dynamically loadable code by loading the code to pages with read and write permissions and further checking the nonexistence of ENCLU and WRGSBASE gadgets and then granting the pages with read and execute permissions (which cannot be later altered by unauthorized enclave code since EACCEPT is protected).

Protection of Critical Data (P4). Leveraging the SGX2 features, we are able to protect data confidentiality by forcefully downgrading the enclave to single-threading when running the sblock chain involving sensitive data, and later resuming the multi-threading mode. This is done by dynamically adjusting the number of TCS (thread control structure) pages. More specifically, when the enclave is about to handle secret data (e.g., decryption of users' session keys using keys derived by EGETKEY), our design guarantees the enclave has only 1 TCS page and is thus single-threaded. After all data for the user's task batch is prepared and all secret data is cleared, we add more TCS pages back to the enclave and allow multiple threads to process the data in parallel.

When a user's service request enters the enclave through EENTER, the SM of the enclave ensures that no other user's thread is also running inside the enclave before turning the enclave into the single-thread mode: it removes all the TCS pages of the enclave except the one used by the current thread. Then it performs integrity check and exit sanitization, as mentioned earlier (Section 4.2). In this way, the SM either obtains the user's session key through remote attestation (for the first time user) or retrieves the user's (encryption-protected) session key by calling EGETKEY, allowing it to further decrypt the data for the task batch using the key. After that, it clears up all the secret data, such as the session key and the public/private key pair used for remote attestation. At last, it adds new TCS pages to switch the enclave to the multi-thread mode. All the above operations are protected by sblocks to ensure atomicity.

Protection of Critical Instructions (P3). Similar to the Liveries protection on SGX1, we can use GS to host the flag for sblock. GS is saved to memory on an AEX and restored on ERESUME. In the multi-threading setting, it is important that the memory cannot be changed by another thread after an thread leaves the enclave through AEX. Otherwise, GS can be set without using WRGSBASE, which would invalidate the sblock protection.

Since FS/GS are designed to support the features like thread-local storage, they are not expected to be modified by normal enclave code or the exception handler. Therefore GS should be restored from the TCS on EENTER and ERESUME.

Indeed this is the case according to the official manual [34, Volume 3, 40.4], which states "(on ERESUME,) new values (of FS and GS) are constructed using TCS.OFSBASE/GSBASE (32 and 64-bit mode)". Since TCS cannot be accessed by enclave code, the flag cannot be modified by the running thread inside the enclave. On our SGX2 NUC Kit, however, we found that it does not follow the documentation: GS is loaded from the TCS in the 32-bit mode, but from the SSA in the 64-bit mode, which may be modified by an in-enclave thread. We expect the inconsistency could be easily fixed.[3]

Since Liveries detects the violations of the security policies at the end of the sblocks, we need to prevent the output data of EGETKEY and EREPORT from being exposed to a concurrently running thread, before the security check is conducted by the exit instruction. In the current SGX design, the output data is directly written to the enclave memory (Table 1). This could be easily secured by separating the semantics of EGETKEY (or EREPORT) to a 2-step procedure: first the output data is stored to the TCS by the EGENERATEKEY (or EGENERATEREPORT) instruction, which is not directly accessible by enclave code; then the output data is visible to the enclave code by running the ERETRIVEKEY (or ERETRIVEREPORT instruction, which moves the output data to the enclave memory. In this design, a security check after the EGENERATEKEY (or EGENERATEREPORT) instruction can prevent a concurrently running thread from observing the output data.

6 Implementation

We prototyped the design on an Intel Xeon E3-1585L v5 CPU which supports SGX1, and on an Intel NUC Kit NUC7PJYH with Pentium Silver J5005 CPU, which supports SGX2.

6.1 Implementation Details

Using the RDGSBASE and WRGSBASE Instructions. The FSGSBASE instruction set is introduced since Ivybridge to allow access to FS and GS from any privilege. The availability of the instruction set is determined by the control register CR4, and is enabled by default since Linux kernel version 5.9-rc1. In our research, we also built a kernel module to enable them in ring-3 enclave code. If this attempt is stopped by the untrusted OS, an invalid opcode exception (#UD) will be thrown to halt the enclave. Also note that GS can only be changed by WRGSBASE using the content of general purpose registers at the beginning (setting GS flag) and the end (clearing GS flag) of an sblock. So if the adversary intends to set the flag for jumping to the middle of an sblock, he can only use the WRGSBASE at an sblock's exit (line 8 in Fig. 6), due to sblock's atomicity. To defeat this attempt, our approach adds an RDGSBASE right after the WRGSBASE at the exit, to ensure that the content of GS is indeed cleared (line 9). Other occurrences of the (unaligned) WRGSBASE (and ENCLU) gadgets are eliminated from enclave code with the help of the gadget finding tool (evaluating Q3 in Section 7.2).

Protecting Critical Data and Instructions. As discussed earlier, the private key, public key, and the shared key are

```
1        rdgsbase %rax
2        cmp $0xdeadbeaf, %eax
3        je CHECK_SUCCESS
4    CHECK_FAIL:
5        ud2
6    CHECK_SUCCESS:
7        xor  %rax, %rax
8        wrgsbase %rax
9        rdgsbase %rax
10       cmp $0, %rax
11       jne CHECK_FAIL
```

Fig. 6. Clear GS after the flag check.

critical data that need protection. However, the code in the SGX SDK has a lot of dependencies which makes it difficult for verification. Instead, we found standalone implementations for the elliptic-curve Diffie-Hellman (ECDH), SHA-256, and AES algorithms (with AES-NI instruction). The same ECC curve as used by the official SGX SDK (i.e., secp256r1) was implemented. We also located a minimum code base for invoking the EGETKEY and EREPORT instructions. We rewrote the code to make them conform with the definition of sblocks (e.g., by avoiding the use of heap and global memory and removing the use of function pointers). These code segments are used in our implementation and are verified (Section 7.1). Note that our current prototype was implemented as a proof of concept for the Liveries technique, which may not provide strong side channel resistance, a feature currently outside the scope of our research.

Integrity Check and Exit Sanitization. Runtime measurement is implemented using the AES-NI instruction. To ease the operation for integrity check and exit sanitization, we create separated heap regions used for different purposes by revising the SGX tlibc library implementation and reserving additional mspaces (i.e., memory space) in the heap. Besides the malloc and free function which allocates and de-allocates memory regions on the normal heap (used for the MEASURE_AREA), we added tmp_malloc and tmp_free function for memory operations on the TMP_AREA.

Controlling TCS Pages Using SGX2 Instructions. Our implementation works as follows. To remove a TCS page, the SM changes the TCS page to trimmed page and then notifies the driver to remove the page. We confirmed that the enclave crashes if the TCS page being removed is *in use by another thread*, e.g., if a malicious thread is still running within the enclave. To add a TCS page, the SM adds the page as regular page and fills the TCS fields. After that, the SM changes the page type from regular page to TCS page. These changes are made by the untrusted driver (through OCALLs) and confirmed by the enclave using EACCEPT. The SM guarantees that the execution continues at the next instruction when the OCALL returns, conforming with the sblock definition, since no other enclave instructions can be executed before exiting the sblock. As mentioned in Section 5, these operations are protected by sblocks.

6.2 Liveries: Programming Model

We applied Liveries to the official SGX SDK by securing the occurrences of the ENCLU instruction. Our current implementation is built to protect critical data for remote attestation, including the key pairs for key negotiation and users' session keys.

To use Liveries, the enclave developer first needs to identify the data that requires integrity protection (e.g., machine

3. Actually, other secure storage can also host the flag in the multithreading mode, e.g., the enclave shadow stack of control-flow enforcement technology (CET) can only be written with a specific instruction (WRUSS) and cannot be modified by other threads, since each thread operates on a separate shadow stack.

learning parameters) and the data that requires exit sanitization (e.g., temporary user inputs) upon a user switch. These data should then be stored in separated memory regions using Liveries APIs. On such data, the SM performs integrity check or exit sanitization according to the types of memory regions. We also provide APIs for the developers to encrypt (and decrypt) enclave users' intermediate results using their session keys (with version numbers to prevent replay attacks), which are then kept in the `ENC_AREA`. After such configuration, the enclave can be built by linking our revised SGX SDK library. We show this programming model can easily support common in-enclave services such as machine learning (only 23 LoCs are modified to protect KANN [14], *cf.* evaluating Q2 in Section 7.2). The last step is to remove the (unaligned) occurrences of `ENCLU` and `WRPKRU` instructions (except for those used for sblock protection) through source code or binary code rewriting. As shown in Section 7.2, such occurrences are rare in common enclave libraries.

7 Analysis and Evaluation

7.1 Security Analysis

TCB Analysis. Compared with software fault isolation (SFI), Liveries is characterized by its much smaller TCB, which does not include most of the SGX SDK library code and introduces only a small software stack, as described below.

• Two sblock chains for protecting critical data (enforcing $P4$). This adds about 300 lines of C code (LoCs) for the AES implementation and unsealing/sealing with `EGETKEY`.

• An sblock chain for generating the ECDH key pair. This brings in about 1,100 LoCs for the ECC implementation by tailoring [35].

• Two sblock chains for creating the enclave report and generating `MSG3` in the remote attestation protocol. This adds about 1,000 LoCs, including 200 for the SHA-256 implementation by tailoring [36] and 500 for creating the enclave report.

• An sblock chain for calculating `RUNTIME_HASH` and an sblock chain for integrity check using `RUNTIME_HASH`. This introduces about 500 LoCs, excluding the AES implementation (since it is already counted above).

• Three sblock chains for protecting the `EREPORT` and `EACCEPT` instruction on SGX2 machines. This includes about 200 LoCs.

• The gadget finding tool that we built contains less than 100 LoCs.

Together, our software TCB contains about 3,200 LoCs (1,600 LoCs are for cryptographic implementations) with no dependencies.

Analysis on the Protection of `GS` . According to the Intel software development manual, `EENTER` always loads the value to `GS` as specified in the `OGSBASGX` field of the `TCS` [34, Volume 3, Sec. 40.4]. `OGSBASGX` is part of the `TCS` and is measured during enclave creation. As such the SGX design prevents the code outside the enclave from setting the `GS` value for the enclave. Other instructions (besides `WRGSBASE` instructions) to modify the segmentation state inside the enclave (e.g., `MOV` or `POP`) will cause the #UD exception [34, Volume 3, Sec. 41.3.4], leading to the termination of the enclave in our design. Another attack avenue is that the adversary may interrupt the enclave and then utilize the exception handler to modify the `SSA`, so as to later run `ERESUME` to reenter the enclave to load the content of the `SSA` to `GS`. To block this avenue, Liveries ensures that the code of the exception handler does not touch the `GS` field of the `SSA` (enforcing P2 in Section 4.1). Lastly, Liveries ensures that `WRGSBASE` after the flag check cannot be reused to manipulate `GS`, and other (unaligned) `WRGSBASE` gadgets are eliminated (Section 6).

Sblocks Verifications. The small TCB size makes formal verification feasible, which will be our future work. Here we report the preliminary results of sblocks verification, in terms of their atomicity and functionalities.

• *Atomicity Verification*. According to its definition, an sblock is a chunk of code whose control flow never transfers to a non-sblock instruction before the $exit$ instruction is executed (Section 3.2). To verify each sblock's conformation with this atomicity property, we ran a script to scan the binary code of the sblocks to confirm that they do not contain indirect jumps/calls, and the target addresses of direct control transfers are within their individual blocks.

To prevent the adversary from redirecting the control flow, e.g., by overwriting the function return addresses through buffer overflow, we performed memory safety verification using the SMACK verification tool [12], [37]. SMACK builds a memory model, provides memory access check assertions, and instruments these assertions at the LLVM IR level to keep track of memory operations. Further, SMACK translates from the LLVM IR to the Boogie intermediate verification language (IVL). Ultimately, the verification is done at the Boogie IVL level with the Corral verifier [38] using an underlying SMT solver. Verification at the Boogie IVL level is similar to direct verification of machine code [39], which has been widely used in the prior research, e.g., analyzing constant time implementations of cryptographic algorithms [39], [40] and safety of smart contract [41]. We found that all the memory operations in our implementation were covered by SMACK, and thus utilized the tool to verify the absence of memory errors in the sblock code. Altogether, our verification ensures that each block on the chains indeed has a single exit point through the $exit$ instruction. Table 2 presents the software TCB components we have verified.

• *Functional Verification*. We further performed a preliminary functional verification on the sblocks using (exhaustive) model checking (Spin [13]), which proves that the model derived from each sblock has desired properties. Specifically, we manually built the model for each sblock operation using Promela [42], together with a set of attack threads.

As an example, Fig. 7 illustrates the model for the sblock chain that generates and protects the ECDH private key. Under the model, all security critical data are represented as global variables (line 1), since they will be stored to the `SSA` and become accessible to an in-enclave adversary whenever the adversary triggers an AEX. Also, the whole sblock operation is modeled as an *atomic* sequence in Promela (line 4 to line 21), given the protection put in place by $P1$ and $P2$ (Section 4.1). For the attack threads, with the atomicity of the sblock code, all they can do is just to cause the execution of some enclave instructions to fail (e.g., `rdrand` may fail in generating random numbers due to insufficient entropy source [34, Volume 1, Sec.7.3.17.1]). To describe such an attack, we built Promela processes to simulate the

TABLE 2
Summary of Verification Results

Source file	Description	LoCs	Static loop bound	Recursion bound	Verification time
aesni.c	AES implementation using AES-NI	< 200	30	5	0.877s
uECC.c	A small and fast ECDH implementation by tailoring [35]	< 1100	30	4	4371s
hash.c	AES based hash implementation used for generating `MEASURE_HASH` and integrity check	< 400	30	5	151.3s
sha256.c	SHA-256 implementation based on [36]	< 200	30	3	3.38s
getkey_report.c	Implementation for key derivation and creating report using `EGETKEY` and `EREPORT`	< 500	30	5	16.2s

We set recursion bound to 3 and 4 for verifying SHA-256 and the ECDH implementations, as it did not finish in 10 hours for larger recursion bound. According to the Corral paper [38], setting recursion bound to 3 find almost all the memory bugs in a test suite of real-world programs.

manipulation of instruction return states (line 2 and line 24 to line 35). The property the sblock model is expected to hold can then be described as follows: whenever the attack threads read from the critical data, they always get an empty result (since the data have been removed before being accessed by the threads, see line 36 to line 41).

7.2 Performance Evaluation

The evaluations were conducted on an Intel Xeon E3-1585L v5 CPU (supporting SGX1) with hyper-threading enabled, equipped with 16 GB physical memory. It ran the Ubuntu 16.04 operating system with kernel version 4.14.20. We used version 2.5 of the SGX SDK. The enclaves were built with GCC version 5.4.0 and optimization level "-O2". The evaluations were conducted in SGX hardware mode to answer the following questions:

```
1  bit privatekey, sgx_seal_key
2  bit flag_rdrand_ret, flag_egetkey_ret
3  active proctype sgx_ra_get_ga() {
4     atomic {
5        if
6        :: flag_rdrand_ret == 1 ->
7           goto exit
8        :: else -> skip
9        fi
10       privatekey = 1 // ECDH key generated
11       if
12       :: flag_egetkey_ret == 0 ->
13          sgx_seal_key = 1
14       :: else ->
15          privatekey = 0
16          goto exit
17       fi
18       skip // egetkey succeeds
19       sgx_seal_key = 0
20       privatekey = 0
21    }
22 exit:
23 }
24 active proctype attacker1() {
25    flag_rdrand_ret = 0
26 }
27 active proctype attacker2() {
28    flag_rdrand_ret = 1
29 }
30 active proctype attacker3() {
31    flag_egetkey_ret = 0
32 }
33 active proctype attacker4() {
34    flag_egetkey_ret = 1
35 }
36 active proctype assertion1() {
37    assert(sgx_seal_key == 0)
38 }
39 active proctype assertion2() {
40    assert(privatekey == 0)
41 }
```

Fig. 7. Modeling the generation and protection of ECDH keys using Spin.

Q1. How much is the performance gain brought in by the design, compared with the baseline of creating and destroying enclave, and performing remote attestation for every service request?

Q2. What is the overhead brought by the SM (i.e., integrity check and exit sanitization) on real-world case studies?

Q3. How many `ENCLU` and `WRGSBASE` gadgets are there in the enclave binary? If any, what is the overhead for removing them?

Evaluating Q1. Considering the baseline design of instantiating an enclave for every service request, the time for each request includes the delay for creating and destroying an enclave (which depends on the enclave sizes), along with the time for remote attestation. The remote attestation needs to connect to the Intel attestation server and may take seconds. While in our design, when a past user requests the service, the delay includes that for integrity check and exit sanitization, which is also related to the enclave sizes. In the evaluation, we divided the heap region into 2 parts, one half for the `MEASURE_AREA`, and the other half for the `TMP_AREA`.

Fig. 8 shows the comparison of the delay for creating and destroying the enclave with that for integrity check and exit sanitization (averaged over 1,000 measurements). Even without taking the time for remote attestation into account, our approach is still orders of magnitudes faster: e.g., when the heap size is 1 MB, our approach is 827× faster. When the heap size is larger, the performance gain becomes smaller (52× for 256 MB heap), which may be caused by page swapping.

On SGX2 platforms and with the support of Data Center Attestation Primitives (DCAP) [43], the time for enclave creation/destroy and remote attestation could be reduced. However, Liveries provides additional protection from the adversary who can control the enclave through a malicious request that exploits vulnerable enclave code and launch the man-in-the-middle attacks against remote attestation [18], [19]. While the threat might be mitigated using a different enclave image for each user (so that the enclaves' measurements would be different), it introduces the burden that the service providers may be reluctant to bear.

Evaluating Q2 With Case Studies. We ported KANN [14], a lightweight library in C for artificial neural networks into SGX. As did in the previous evaluation, we divided the heap into 2 halves for the `MEASURE_AREA` and the `TMP_AREA` respectively. To perform inference tasks with trained neural networks, the model was loaded into the `MEASURE_AREA`, while the input data to be processed was

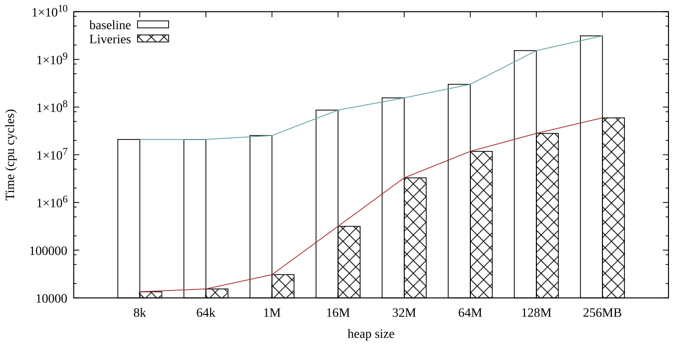


Fig. 8. Comparison with the baseline (without counting the time for remote attestation). The stack size is set to 64 KB.

TABLE 3
Multi-Layer Perceptron on the MNIST Dataset

	MEASURE_AREA size	TMP_AREA size S0	Batch size	Baseline time	Overhead
S1	256 KB	128 KB	5	0.15 ms	5.9%
S2	256 KB	128 KB	10	0.26 ms	3.3%
S3	256 KB	128 KB	20	0.48 ms	1.6%
S4	256 KB	256 KB	50	1.12 ms	0.95%

TABLE 4
Variantional Autoencoder on the MNIST Dataset

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	1 MB	128 KB	5	0.34 ms	2.8%
S2	1 MB	128 KB	10	0.64 ms	1.4%
S3	1 MB	128 KB	20	1.22 ms	0.74%
S4	1 MB	256 KB	50	2.97 ms	0.35%

TABLE 5
CNN on the MNIST Dataset

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	4 MB	128 KB	5	4.64 ms	0.22%
S2	4 MB	128 KB	10	9.18 ms	0.07%
S3	4 MB	128 KB	20	18.8 ms	0.03%
S4	4 MB	256 KB	50	46.0 ms	0.00%

TABLE 6
CNN on the ImageNet Dataset

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	64 MB	16 MB	5	183 ms	3.4%
S2	64 MB	16 MB	10	354 ms	1.6%
S3	64 MB	16 MB	20	702 ms	0.85%
S4	64 MB	32 MB	50	2.14 s	0.97%

TABLE 7
ResNet50 on the ImageNet Dataset

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	256 MB	16 MB	5	840 ms	3.22%
S2	256 MB	16 MB	10	1.88 s	2.98%
S3	256 MB	16 MB	20	3.81 s	0.62%
S4	256 MB	16 MB	50	9.50 s	0.53%

loaded to the TMP_AREA. In total we modified 23 lines of code for loading the data to appropriate memory regions for the evaluation. For this evaluation, we considered that a batch of one user's data was sent for processing, with various batch sizes. We measured the overhead introduced by the integrity check and exit sanitization upon user switches. Since the overhead is highly related to the sizes of MEASURE_AREA and TMP_AREA, we evaluated the overhead on typical machine learning workloads, with varied sizes of MEASURE_AREA and TMP_AREA suitable to support the computing tasks. The stack size was set to 16 KB. All results were averaged over 10,000 measurements.

- *Multi-Layer Perceptron*. Multi-layer perceptron (MLP) is a class of feed-forward artificial neural networks and has been applied to diverse fields such as speech/image recognition and machine translation. We used the MNIST dataset [44] and a small network with 1 hidden layer and 64 neurons for the evaluation. The results are shown in Table 3. It shows the overhead is from 0.95% to 5.9%.
- *Variantional Autoencoder*. Variational autoencoder (VAE) is a kind of deep generative model, which combines ideas from deep learning with statistical inference. It is shown effective in generating many kinds of complicated data, including handwritten digits, faces, and predicting the future from static images. In our evaluation, the VAE has 64 hidden neurons for the encoder and decoder. We used the MNIST dataset for the evaluation (Table 4, showing that the overhead is less than 2.8%.
- *CNN for MNIST*. In this test, a convolutional neural network (CNN) was used for inference tasks on the MNIST dataset. The network has 8 hidden layers. The results are shown in Table 5. It demonstrates that the overhead is less than 0.22%.
- *CNN for ImageNet*. We created a CNN with 2 convolution layers followed by 5 fully-connected layers and used the ImageNet dataset [45] for testing the network. The results (Table 6) show that the overhead is less than 3.4%.
- *ResNet for ImageNet*. For evaluation purposes, we added the support of ResNet in KANN. We tested ResNet50 using the ImageNet dataset. As shown in Table 7, the overhead is small (less than 3.22%). It significantly outperforms the solution that creates a new enclave and loads the model for every task batch (147% slowdown when the batch size is 5 according to our evaluation).
- *Character-Level Text Generation With RNN*. In the evaluation, we used a recurrent neural network (RNN) for character-level text generation. The network has 3 hidden layers, each of which has 256 neurons. The lengths of generated text were 300, 500, 1,000, and 1,500 respectively. The results are presented in Table 8, showing that the overhead is negligible.
- *Model and Predict Short DNA Sequence Features*. In this evaluation, we used a deep-learning model to identify the alpha satellite repeats on DNA sequences [46]. The evaluation was performed on short DNA sequences with batch size of 100, 200, 500, and 1,000 respectively. The result (Table 9) shows the overhead is less than 0.51%.

Evaluating Q3. We built our own gadget finding tool to find the occurrences of both the ENCLU and the WRGSBASE

TABLE 8
RNN on Text Generation

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	6 MB	12 KB	300	124 ms	0.01%
S2	6 MB	12 KB	500	193 ms	-0.01%
S3	6 MB	12 KB	1000	367 ms	0.00%
S4	6 MB	12 KB	1500	574 ms	0.01%

gadgets in popular enclave binaries and further validated our findings using the ROPgadget tool [47]. For this purpose, we collected 10 popular open-source SGX projects (same as [48]) as well as SGX-nbench [49] and lmbench [50], compiled with the default configurations, as listed in Table 10. Also listed are the numbers of instructions within the enclave binaries, and the numbers of (unaligned) `ENCLU` and `WRGSBASE` gadgets found. For the only `ENCLU` gadget found in Intel-SGX-SSL, we confirmed the gadget can be removed by simply adding a `NOP` instruction using inline assembly without affecting the functionalities, and recompiling the code. As such we believe that it is easy to eliminate such gadgets in normal enclave code, and removing the relevant gadgets only introduces a negligible performance overhead on common enclave programs.

7.3 Comparison With SFI-Based Solutions

Software Fault Isolation (SFI) [51] is a software instrumentation technique for sandboxing untrusted modules. In-enclave isolation can be achieved with SFI for confinement [9], [10], [11]. However the design goals of SFI and Liveries are different: SFI can be used to sandbox concurrent tasks within an enclave, while Liveries is for lightweight user isolation under the *sequential* service model, which is simpler and enables highly-efficient protection.

More specifically, SFI needs to instrument the enclave code to inspect the memory reference and control flow instructions, which incurs a large performance and memory overhead, and a large TCB for verifying SFI compliance, including the disassembler and binary analyzer. In comparison, Liveries simplifies user isolation under the sequential service model, bringing in advantages such as low performance/memory overhead (1% on average) and extremely small TCB (3200 LoCs).

We evaluated the performance of SFI-based solutions using two state-of-the-art tools: Occlum[4] [10] and WebAssembly Micro Runtime (WAMR) [52], both of which support SFI inside SGX enclaves. We ran unmodified KANN using Occlum and WAMR, and collected the performance statistics over the same machine learning tasks. As shown in Fig. 9, in most cases running with Occlum and WAMR is much slower than the baseline (about 5.21× and 1.51× slowdown on average respectively).

7.4 Discussions and Limitations

Model Rollback Attacks. The attacker may install an older version of the model which is correctly signed by the service provider. Such attacks are generally applicable to in-enclave services and are not specific to our design. It is possible to mitigate these attacks with the support of monotonic counters (MCs) [53] or by involving a challenge-response protocol with the service provider to load the model.

Single-Threaded Enclave. Our design on SGX1 platforms restricts the enclave to running only in single-thread mode. It is worth noting that running more threads will pose a big pressure on the available EPC memory. We evaluated the speedup of running multi-threaded enclave over single-threaded enclave, in terms of the throughput on the computing tasks. As shown in Fig. 10, using 4 threads achieves a speedup ratio of about $2.64\times$.

On the other hand, running multi-threaded enclaves enlarges the attack surface, opening door to the AsyncShock attack [54], the game of threads attack [55] and the concurrent calls vulnerability (a type of COIN attacks [48]). It also makes possible the in-enclave speculative execution attacks [56], which do not need to cross the enclave boundary.

Exception Handling. If in-enclave exception handling is not needed, we can simply disable exception handling by setting `NSSA` to 1. Indeed, both the Fortanix Rust EDP [57] and Alibaba Inclavare [58] have taken this approach.

Supporting exception handling breaks the atomicity of the enclave. As shown by the recent SmashEx attack [59], in the absence of atomicity, the exception handling in SGX is complicated and can be prone to re-entrancy vulnerabilities.

The Use of `FS` and `GS`. The `FS` and `GS` registers are used to determine the location of thread-local storage (TLS). Most executables use either `FS` or `GS` for this purpose (such as Intel SGX SDK), and we can use the other one to host the flag. If both registers are in use, we could resort to general purpose registers (such as $\%r15$). We need to reserve the register (e.g., using the GCC `-ffixed-reg` option) and

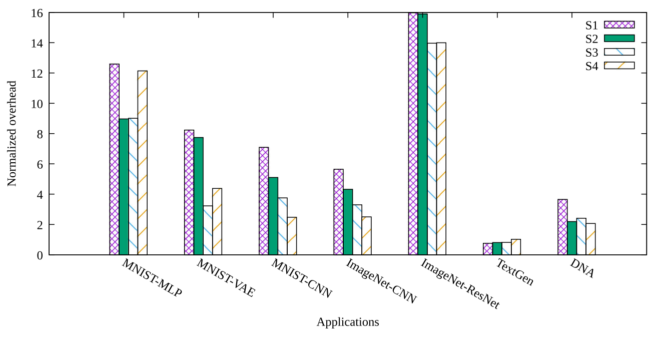
TABLE 10
Evaluating Q3 With Open-Source SGX Projects

Projects	Binary size	#Inst	ENCLU gadgets	WRGSBASE gadgets
mbedltls-SGX	2.2 M	201453	0	0
SGX-Tor	5.8 M	698965	0	0
TaLoS	9.8 M	348017	0	0
Bolos-enclave	1.4 M	260835	0	0
Intel-SGX-SSL	3.3 M	421696	1	0
SGX_SQLite	1.3 M	192518	0	0
SGX-Migration	1.8 M	243030	0	0
SGX-Wallet	344 K	64154	0	0
SGX-Reencrypt	411 K	81819	0	0
SGXCryptoFile	322 K	62208	0	0
SGX-nbench	171 K	25868	0	0
lmbench	417 K	60859	0	0

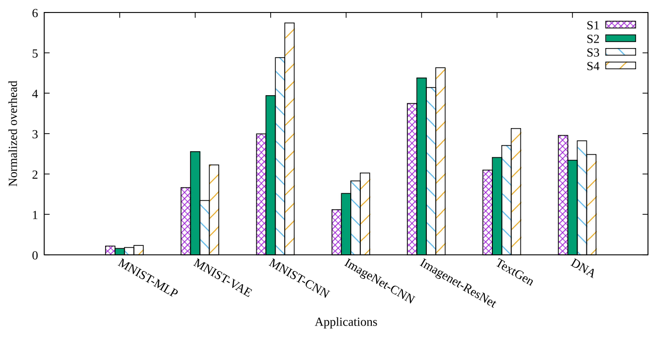
TABLE 9
DNA Sequences Feature Prediction

	MEASURE_AREA size	TMP_AREA size	Batch size	Baseline time	Overhead
S1	256 KB	64 KB	100	1.52 ms	0.51%
S2	256 KB	64 KB	200	3.02 ms	0.29%
S3	256 KB	64 KB	500	7.22 ms	0.10%
S4	256 KB	64 KB	1000	14.3 ms	0.05%

4. Since the latest Occlum does not support SFI any more, we used the artifact evaluation version presented at ASPLOS 2020.



(a) Evaluation of Occlum over the baseline.



(b) Evaluation of WAMR over the baseline.

Fig. 9. Performance evaluation of SFI-based solutions using the KANN and the same machine learning tasks with varied task batch sizes. We conducted the evaluation using the same settings (i.e., S1 $\sim$ S4) for all tasks as in Section 7.2.

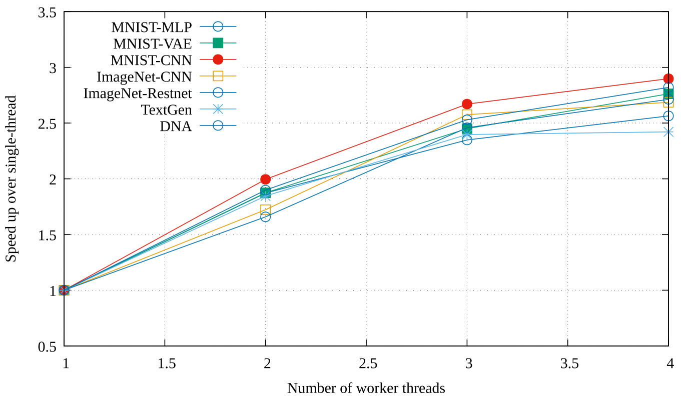


Fig. 10. Evaluation of the speedup of using multi-threaded enclave. The testbed has 4 physical cores.

remove the occurrences of gadgets that update the register, using similar techniques to SGX-Shield [60] and G-Free [61].

8 Related Works

Intra-Space Privilege Separation. A problem that has long been studied is how to isolate components in the same address space to restrict attack surface and prevent cross-component memory corruption, e.g., intra-space separation within the process [23], [62], [63], [64], [65], [66], [67], browser [51], operating system kernel [68], [69], bare-metal embedded systems [70], nested enclave [71] etc. These solutions are all built with hardware supports such as virtualization, segmentation, Memory Protection Keys (MPK), and ARM memory domains. However, none of the hardware features is available within the enclave,[5] while randomization can be easily circumvented by the adversary with arbitrary read and write capabilities.

Memory Attacks and Defenses. A recent study shows memory corruption vulnerabilities are easily introduced into enclave code [17]. It is possible to apply memory safety protection mechanism inside an enclave, such as address space layout randomization (ASLR) [60] and tagged pointers [72]. However, control flow hijacking attacks are still possible without knowing the enclave content [18] and fine-grained ASLR is applied [19]. Rust-SGX SDK [73] enables the development of application layer memory safety, however the lower layer SGX SDK libraries (i.e., unsafe code) are not enhanced.

5. MPK can be used in the enclave, however the protection keys are set by the untrusted OS, making it insecure for in-enclave isolation [34, Volume3, Sec. 4.6.2].

Side Channels. Side channels in SGX have been extensively studied [74], [75], [76], [77], [78], [79], [80], [81]. While it is not the focus of the paper, we believe that side channel resistant implementation [82] is important, which can be used to protect secret keys involved in the cryptographic operations on sblocks. Also to prevent transient execution attacks (e.g., Spectre attacks [56] from users in the same enclave), we can insert the serializing instructions (`lfence`) after all critical instructions.

Control Flow Attestation. Control flow attestation [83], [84] was proposed to detect runtime memory attacks on remote embedded devices. Without hardware support in current TEEs, control flow attestation incurs significant performance penalties (several seconds for each attestation [83]) even for small applications.

9 Conclusion

In this paper, we proposed the design of Liveries, a lightweight and verifiable method for users isolation in supporting in-enclave services. We demonstrated that the TCB of Liveries is small, containing about 3,200 lines of code in our current implementation. We provided preliminary results of the sblocks verification using automatic verification tools. Evaluations showed that the overhead in supporting in-enclave user isolation is small.

References

[1] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative technology for CPU based attestation and sealing," in *Proc. 2nd Int. Workshop Hardware Architectural Support Secur. Privacy*, 2013, vol. 13, pp. 1–7.
[2] D. Kaplan, J. Powell, and T. Woller, "AMD memory encryption," White Paper, 2016. [Online]. Available: https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf
[3] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal hardware extensions for strong software isolation," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 857–874.
[4] D. Lee, D. Kohlbrenner, S. Shinde, K. Asanović, and D. Song, "Keystone: An open framework for architecting trusted execution environments," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–16.
[5] Introducing Azure confidential computing. Accessed: Feb. 13, 2020. [Online]. Available: https://azure.microsoft.com/en-us/blog/introducing-azure-confidential-computing/
[6] G. C. Blog, "Advancing confidential computing with Asylo and the confidential computing challenge," Accessed: Feb. 13, 2020. [Online]. Available: https://cloud.google.com/blog/products/identity-security/advancing-confidential-computing-with-asylo-and-the-confidential-computing-challenge
[7] Confidential computing consortium, 2019. [Online]. Available: https://confidentialcomputing.io/

[8] J. Ma *et al.*, "S3ML: A secure serving system for machine learning inference," 2020, *arXiv:2010.06212*. [Online]. Available: https://arxiv.org/abs/2010.06212

[9] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel, "Ryoan: A distributed sandbox for untrusted computation on secret data," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, 2018, Art. no. 13.

[10] Y. Shen *et al.*, "Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 955–970.

[11] A. Ahmad, J. Kim, J. Seo, I. Shin, P. Fonseca, and B. Lee, "CHANCEL: Efficient multi-client isolation under adversarial programs (to appear)," in *Proc. 28th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2021, pp. 1–18.

[12] Z. Rakamarić and M. Emmi, "SMACK: Decoupling source language details from verifier implementations," in *Proc. Int. Conf. Comput. Aided Verification*, 2014, pp. 106–113.

[13] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. Softw. Eng.*, vol. 23, no. 5, pp. 279–295, May 1997.

[14] A lightweight C library for artificial neural networks," Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/attractivechaos/kann/

[15] S. P. Johnson, V. R. Scarlata, C. V. Rozas, E. Brickell, and F. McKeen, "Intel SGX: EPID provisioning and attestation services," *Intel*, 2016. [Online]. Available: http://www.intel.com/content/dam/develop/public/us/en/documents/ww10-2016-sgx-provisioning-and-attestation-final.pdf

[16] Intel software guard extensions remote attestation end-to-end example. Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/intel/sgx-ra-sample/

[17] T. Cloosters, M. Rodler, and L. Davi, "TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves," in *Proc. 29th USENIX Secur. Symp.*, 2020, pp. 841–858.

[18] J. Lee *et al.*, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 523–539.

[19] A. Biondo, M. Conti, L. Davi, T. Frassetto, and A.-R. Sadeghi, "The guard's dilemma: Efficient code-reuse attacks against Intel SGX," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 1213–1227.

[20] F. Schuster, T. Tendyck, C. Liebchen, L. Davi, A.-R. Sadeghi, and T. Holz, "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications," in *Proc. IEEE Symp. Secur. Privacy*, 2015, pp. 745–762.

[21] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proc. 14th Conf. USENIX Secur. Symp.*, 2005, vol. 5, Art. no. 12.

[22] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *Proc. IEEE Symp. Secur. Privacy*, 2016, pp. 969–986.

[23] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1221–1238.

[24] P. Qiu, D. Wang, Y. Lyu, and G. Qu, "VoltJockey: Breaking SGX by software-controlled voltage-induced hardware faults," in *Proc. Asian Hardware Oriented Secur. Trust Symp.*, 2019, pp. 1–6.

[25] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based fault injection attacks against Intel SGX," in *Proc. IEEE Symp. Security Privacy*, 2020, pp. 1466–1482.

[26] J. Soifer *et al.*, "Deep learning inference service at Microsoft," in *Proc. USENIX Conf. Operational Mach. Learn.*, 2019, pp. 15–17.

[27] NVIDIA deep learning inference platform. Accessed: Feb. 13, 2020, 2020. [Online]. Available: https://www.nvidia.com/en-au/deep-learning-ai/inference-platform/

[28] 23andMe: DNA genetic testing & analysis. Accessed: Feb. 13, 2020. [Online]. Available: https://www.23andme.com/

[29] Cloud natural language API by Google cloud. Accessed: Feb. 13, 2020. [Online]. Available: https://cloud.google.com/natural-language/

[30] Code sample: Intel software guard extensions remote attestation end-to-end example, 2018. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/code-sample-intel-software-guard-extensions-remote-attestation-end-to-end-example.html

[31] R. Rajwar and M. Dixon, "Intel transactional synchronization extensions," in *Intel Developer Forum San Francisco*, vol. 2012, 2012. [Online]. Available: https://www.intel.com/content/dam/develop/external/us/en/documents/sf12-arcs004-100-393551.pdf

[32] M. Muench, F. Pagani, Y. Shoshitaishvili, C. Kruegel, G. Vigna, and D. Balzarotti, "Taming transactions: Towards hardware-assisted control flow integrity using transactional memory," in *Proc. Int. Symp. Res. Attacks Intrusions Defenses*, 2016, pp. 24–48.

[33] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens, "Faulty point unit: ABI poisoning attacks on Intel SGX," in *Proc. Annu. Comput. Secur. Appl. Conf.*, 2020, pp. 415–427.

[34] Intel 64 and IA-32 architectures software developer's manual, combined volumes: 1,2A,2B,2C,3A,3B,3C and 3D, 2019. [Online]. Available: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1–2abcd-3abcd.pdf

[35] A small and fast ECDH and ECDSA implementation. Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/kmackay/micro-ecc/

[36] A portable byte-oriented SHA-256 implementation. Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/ilvn/SHA256/

[37] M. Carter, S. He, J. Whitaker, Z. Rakamaric, and M. Emmi, "Smack software verification toolchain," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 589–592.

[38] A. Lal, S. Qadeer, and S. K. Lahiri, "A solver for reachability modulo theories," in *Proc. Int. Conf. Comput. Aided Verification*, 2012, pp. 427–443.

[39] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *Proc. 25th USENIX Secur. Symp.*, 2016, pp. 53–70.

[40] C. Sung, B. Paulsen, and C. Wang, "CANAL: A cache timing analysis framework via LLVM transformation," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 904–907.

[41] S. Kalra, S. Goel, M. Dhawan, and S. Sharma, "ZEUS: Analyzing safety of smart contracts," in *Proc. 25th Netw. Distrib. Syst. Secur.*, 2018, pp. 1–15.

[42] R. Gerth, "Concise promela reference, 1997, " 2017. [Online]. Available: http://cm. bell-labs. com/cm/cs/what/spin/Man/Quick. html

[43] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, "Supporting third party attestation for Intel® SGX with Intel® data center attestation primitives," *White Paper*, 2018.

[44] Y. LeCun, C. Cortes, and C. Burges, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[45] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2009, pp. 248–255.

[46] Model and predict short dna sequence features with neural networks. Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/lh3/dna-nn/

[47] ROPgadget tool. Accessed: Feb. 13, 2020. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget/

[48] M. R. Khandaker, Y. Cheng, Z. Wang, and T. Wei, "COIN attacks: On insecurity of enclave untrusted interfaces in SGX," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2020, pp. 971–985.

[49] SGX nBench. 2017. [Online]. Available: https://github.com/utds3lab/sgx-nbench/

[50] lmbench for Intel SGX. 2020. [Online]. Available: https://github.com/vsecurity-research/sgx-bench/tree/master/lmbench/SGX/

[51] B. Yee *et al.*, "Native client: A sandbox for portable, untrusted x86 native code," in *Proc. 30th IEEE Symp. Security Privacy*, 2009, pp. 79–93.

[52] Webassembly micro runtime. Accessed: Oct. 13, 2021. [Online]. Available: https://github.com/bytecodealliance/wasm-micro-runtime/

[53] S. Cen and B. Zhang, "Trusted time and monotonic counters with Intel software guard extensions platform services," 2020. [Online]. Available: https://community.intel.com/legacyfs/online/drupal_files/managed/1b/a2/Intel-SGX-Platform-Services.pdf

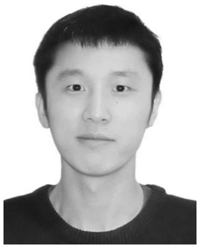
[54] N. Weichbrodt, A. Kurmus, P. Pietzuch, and R. Kapitza, "AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2016, pp. 440–457.

[55] J. R. Sanchez Vicarte, B. Schreiber, R. Paccagnella, and C. W. Fletcher, "Game of threads: Enabling asynchronous poisoning attacks," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Oper. Syst.*, 2020, pp. 35–52.

[56] P. Kocher *et al.*, "Spectre attacks: Exploiting speculative execution," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 1–19.

[57] Fortanix rust enclave development platform. Accessed: Oct. 13, 2021. [Online]. Available: https://github.com/fortanix/rust-sgx/

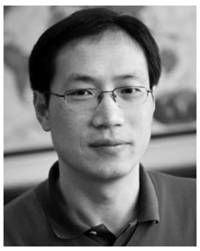
[58] Inclavare containers. Accessed: Oct. 13, 2021. [Online]. Available: https://inclavare-containers.io/
[59] J. Cui, J. Z. Yu, S. Shinde, P. Saxena, and Z. Cai, "SmashEx: Smashing SGX enclaves using exceptions," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2021, pp. 779–793.
[60] J. Seo *et al.*, "SGX-Shield: Enabling address space layout randomization for SGX programs," in *Proc. Netw. Distrib. Syst. Secur.*, 2017, pp. 1–15.
[61] K. Onarlioglu, L. Bilge, A. Lanzi, D. Balzarotti, and E. Kirda, "G-Free: Defeating return-oriented programming through gadget-less binaries," in *Proc. 26th Annu. Comput. Secur. Appl. Conf.*, 2010, pp. 49–58.
[62] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged CPU features," in *Proc. 10th USENIX Symp. Oper. Syst. Des. Implementation*, 2012, pp. 335–348.
[63] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *Proc. IEEE Symp. Security Privacy*, 2016, pp. 56–71.
[64] T. Frassetto, P. Jauernig, C. Liebchen, and A.-R. Sadeghi, "IMIX: In-process memory isolation extension," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 83–97.
[65] M. Hedayati *et al.*, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *Proc. USENIX Annu. Tech. Conf.*, 2019, pp. 489–504.
[66] Z. Wang *et al.*, "SafeHidden: An efficient and secure information hiding technique using re-randomization," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1239–1256.
[67] Z. Wang *et al.*, "SEIMI: Efficient and secure SMAP-enabled intra-process memory isolation," in *Proc. IEEE Symp. Security Privacy*, 2020, pp. 592–607.
[68] N. Dautenhahn, T. Kasampalis, W. Dietz, J. Criswell, and V. Adve, "Nested kernel: An operating system architecture for intra-kernel privilege separation," in *Proc. 20th Int. Conf. Architectural Support Prog. Lang. Oper. Syst.*, 2015, pp. 191–206.
[69] M. Sung, P. Olivier, S. Lankes, and B. Ravindran, "Intra-unikernel isolation with Intel memory protection keys," in *Proc. 16th ACM SIGPLAN/SIGOPS Int. Conf. Virt. Execution Environ.*, 2020, pp. 143–156.
[70] A. A. Clements *et al.*, "Protecting bare-metal embedded systems with privilege overlays," in *Proc. IEEE Symp. Security Privacy*, 2017, pp. 289–303.
[71] J. Park, N. Kang, T. Kim, Y. Kwon, and J. Huh, "Nested enclave: Supporting fine-grained hierarchical isolation with SGX," in *Proc. 47th Int. Symp. Comput. Archit.*, 2020, pp. 776–789.
[72] D. Kuvaiskii *et al.*, "SGXBOUNDS: Memory safety for shielded execution," in *Proc. 12th Eur. Conf. Comput. Syst.*, 2017, pp. 205–221.
[73] H. Wang *et al.*, "Towards memory safe enclave programming with rust-SGX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2019, pp. 2333–2350.
[74] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proc. IEEE Symp. Security Privacy*, 2015, pp. 640–656.
[75] W. Wang *et al.*, "Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2421–2434.
[76] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside SGX enclaves with branch shadowing," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 557–574.
[77] J. Van Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx, "Telling your secrets without page faults: Stealthy page table-based attacks on enclaved execution," in *Proc. 26th USENIX Secur. Symp.*, 2017, pp. 1041–1056.
[78] J. Van Bulck *et al.*, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 991–1008.
[79] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software grand exposure: SGX cache attacks are practical," in *Proc. 11th USENIX Workshop Offensive Technol.*, 2017, Art. no. 11.
[80] M. Hähnel, W. Cui, and M. Peinado, "High-resolution side channels for untrusted operating systems," in *Proc. USENIX Annu. Tech. Conf.*, 2017, pp. 299–312.
[81] E. Dmitry, R. Ryan, N. C. Abu-Ghazaleh, and D. Ponomarev, "BranchScope: A new side-channel attack on directional branch predictor," in *Proc. 23rd Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2018, pp. 693–707.
[82] M. Adalier *et al.*, "Efficient and secure elliptic curve cryptography implementation of curve P-256," in *Proc. Workshop Elliptic Curve Cryptography Standards*, 2015, vol. 66, pp. 446–456.
[83] T. Abera *et al.*, "C-FLAT: Control-flow attestation for embedded systems software," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 743–754.
[84] G. Dessouky *et al.*, "LO-FAT: Low-overhead control flow attestation in hardware," in *Proc. 54th Annu. Des. Autom. Conf.*, 2017, pp. 1–6.



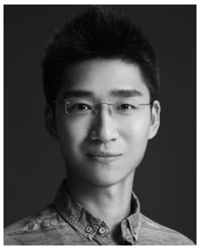
Wenhao Wang received the BS degree from the Ocean University of China, China, in 2009, and the PhD degree from the University of Chinese Academy of Sciences, China, in 2015. He is currently an associate professor in the Institute of Information Engineering, Chinese Academy of Sciences, China. His research interests include cryptography, system security, cloud security and trusted execution technologies.

Weijie Liu received the BS and PhD degrees from Wuhan University, China, in 2012 and 2018, respectively. He is a postdoctoral researcher at Indiana University, Bloomington, Indiana. His research interests include virtualization security and trusted execution technologies.

Hongbo Chen received the bachelor's degree from Xi'an Jiaotong University, China, in 2018. He is currently working toward the PhD degree at Indiana University Bloomington, Bloomington, Indiana. He mainly focuses on system security research in trusted execution environment.



XiaoFeng Wang (Fellow, IEEE) is a James H. Rudy professor of computer science and engineering at Indiana University Bloomington, Bloomington, Indiana. His research focuses on system security and data privacy with a specialization on security and privacy issues in mobile and cloud computing, and privacy issues in dissemination and computation of human genomic data.



Hongliang Tian received the PhD degree in computer science from Tsinghua University, China. He is currently a system architect at Ant Group, China. His primary research interest is confidential computing, system software, and system security. Before joining Ant Group, he worked at Intel Labs, focusing on Intel SGX.

Dongdai Lin is a professor in Institute of Information Engineering, Chinese Academy of Sciences, China, and in the University of Chinese Academy of Sciences, China. His research interests include cryptology and security protocols.