SgxElide: Enabling Enclave Code Secrecy via Self-Modification

Erick Bauman
The University of Texas at Dallas
Richardson, Texas, USA
erick.bauman@utdallas.edu

Huibo Wang
The University of Texas at Dallas
Richardson, Texas, USA
huibo.wang@utdallas.edu

Mingwei Zhang
Intel Labs
Hillsboro, Oregon, USA
mingwei.zhang@intel.com

Zhiqiang Lin
The University of Texas at Dallas
Richardson, Texas, USA
zhiqiang.lin@utdallas.edu

Abstract

Intel SGX provides a secure enclave in which code and data are hidden from the outside world, including privileged code such as the OS or hypervisor. However, by default, enclave code prior to initialization can be disassembled and therefore no secrets can be embedded in the binary. This is a problem for developers wishing to protect code secrets. This paper introduces SgxElide, a nearly-transparent framework that enables enclave code confidentiality. The key idea is to treat program code as data and dynamically restore secrets after an enclave is initialized. SgxElide can be integrated into any enclave, providing a mechanism to securely decrypt or deliver the secret code with the assistance of a developer-controlled trusted remote party. We have implemented SgxElide atop a recently released version of the Linux SGX SDK, and our evaluation with a number of programs shows that SgxElide can be used to protect the code secrecy of practical applications with no overhead after enclave initialization.

CCS Concepts • **Security and privacy → Digital rights management; Software reverse engineering;**

Keywords SGX, self-modifying code, code obfuscation

ACM Reference Format:
Erick Bauman, Huibo Wang, Mingwei Zhang, and Zhiqiang Lin. 2018. SgxElide: Enabling Enclave Code Secrecy via Self-Modification. In *Proceedings of 2018 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'18).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3168833

CGO'18, February 24–28, 2018, Vienna, Austria

ACM ISBN 978-1-4503-5617-6/18/02…$15.00
https://doi.org/10.1145/3168833

1 Introduction

Most software today is delivered in the form of binary executables, which are normally executed on a platform out of the control of software developers. Secrets in the form of binary code are not directly visible to most users. However, curious or malicious attackers can still reverse engineer the executable to uncover secrets (e.g., particular algorithms, data structures, or values of variables) inside the software, since they control the entire computing stack, including the operating system, libraries, runtime environment (e.g., debuggers), and the application code of the software, and they can easily disassemble the binary, or monitor (e.g., trace, or debug) the execution of the binary. Keeping code and data secret in a compiled executable has long been a challenge. Many interesting and important applications require such a capability.

One area in need of the ability to hide code secrets is in defending against cheating in computer games [15]. The popularity of multiplayer games, including highly competitive games that offer substantial monetary rewards in tournaments, or games with economies that allow for the trading of virtual items for real money, gives players incentive to cheat the system for personal gain. Many techniques have been developed to cheat in games [23]. For instance, attackers can reverse-engineer the game code to find exploits, search game memory for certain game values that can be changed to benefit the attacker, or use automated tools that can perform better than a human (such as improved reflexes).

In addition, vendors of software, including games, have strong incentives to prevent users from running unlicensed copies of their software. Over the past few decades, vendors have developed a wide range of Digital Rights Management (DRM) [32] techniques to protect the unauthorized redistribution of their software. Historically, there has been no way to trust a remote machine, and therefore the only way to enable DRM has been through various obfuscation strategies [19, 28], including control flow flatting, signal-based control flow hiding (e.g., [31]), code encryption and packing, etc. Therefore, to prevent illegal copying DRM must remain confidential, because once the DRM is reverse-engineered,

attackers can circumvent the copy protection and upload executables with the DRM removed onto the Internet, or rebuild new software using the stolen algorithm. This results in an escalating arms race between application crackers and developers, with each side trying to outwit the other.

In the past several years, we have also witnessed the growth of cloud computing, in which cloud providers host a platform for selling computing power to third parties. But unfortunately, the customer must trust the cloud provider with their confidential code or data. This prevents certain industries, such as hospitals, from being able to use the (public) cloud, due to strict privacy legislation. In addition, companies with software that may contain valuable trade secrets cannot use such platforms lest a malicious cloud provider extract and sell their secrets. This is also a concern for software that runs on an end user's machine, as secrets can also be extracted from more traditional software.

Therefore, there is a need to provide true confidentiality guarantees. This issue has only recently begun to be solved, and one important step forward is Intel's Software Guard eXtensions (SGX) [22], which allow code to execute in a secure "enclave" opaque to any other code. This allows applications to hide their secrets from all other software on a machine, including privileged code such as the OS or hypervisor. As was shown by Haven [17], this can be extended to protect arbitrary legacy code from being spied on by a cloud provider. SGX is already being deployed in consumer-grade hardware starting with Intel's Skylake processor. This means that eventually a high percentage of users may own SGX-enabled hardware, allowing software developers to take advantage of its features. This may enable much stronger copy protection and trade secrets by hiding crucial code and data in SGX enclaves.

However, despite the substantial benefits provided by SGX, there remains an unfortunate omission in what SGX currently protects, as the SGX SDK Guide explicitly states: "The enclave file can be disassembled, so the algorithms used by the enclave developer will not remain secret" [9]. Therefore, while SGX does protect code and data *integrity*, it only protects the confidentiality of data, while code confidentiality is not assured. SGX does not protect code and data until after enclave initialization. This, combined with the fact that enclave code must be signed and cannot be modified before being loaded, means that there is no straightforward way to hide the actual enclave code from would-be attackers.

To address this shortcoming of SGX's programming model, we introduce SgxElide, a framework that leverages SGX to provide complete confidentiality and integrity for both code and data. The key idea is to treat program code as data, and dynamically decrypt the secret code or retrieve the secret code from the trusted remote parties after an enclave is initialized. SgxElide can be almost transparently integrated into any enclave code, providing a mechanism to securely decrypt or deliver the secret code with the assistance from a trusted remote party controlled by the developer. We have implemented SgxElide atop Intel's Linux SGX SDK [12], and our evaluation with a number of programs shows that SgxElide can be used to protect code secrecy for applications of interest without introducing any additional runtime overhead for the enclave code.

In short, we make the following contributions:

- We present a systematic protection model for software that uses SGX, and show that the default SGX programming model lacks an oblivious mechanism to provide *code confidentiality*.
- We design a framework for providing *code confidentiality* to SGX applications, thereby providing both code and data integrity and confidentiality.
- We have implemented our framework on the Linux platform and demonstrate that it is easy to add to existing SGX projects and effective at protecting the confidentiality of code, without introducing any runtime overhead to the applications.

2 Background and Motivation

2.1 Intel SGX

At a high level, SGX allows an application or part of an application to run inside a secure *enclave*, which is an isolated execution environment. SGX hardware, as a part of the CPU, protects enclaves against malicious software, including the operating system, hypervisor, or even low-level firmware code (e.g., SMM) from compromising its confidentiality and integrity. At a low level, Intel SGX is an extension to the x86 instruction set architecture (ISA).

SGX Enclaves. The first step in creating an enclave is to call the instruction `ECREATE`. This allocates memory inside the Enclave Page Cache (`EPC`) to hold enclave code and data [22]. EPC memory is encrypted by the Memory Encryption Engine (MEE) and decrypted when accessed by enclave code. Enclave contents are added with the `EADD` instruction, which copies a 4KB page from ordinary memory into an `EPC` page [29].

However, SGX must also calculate the enclave's *measurement*, a cryptographic hash that is used for remote attestation. This is done with the `EEXTEND` instruction. Every time `EEXTEND` is executed, it measures 256 bytes, and therefore it must be executed 16 times to cover a full page [29].

The enclave cannot be entered until it has been initialized with the `EINIT` instruction. However, unless the enclave's measurement matches the original measurement calculated by the enclave's creator, the hardware will not initialize it. The creator of the enclave provides the measurement inside the `SIGSTRUCT` data structure, which the creator signs with their private key and provides along with the enclave.

Attestation. Once an enclave has been created, a signed report can be obtained from the processor by using the new `EREPORT` instruction. This report provides a guarantee to enclaves on the same machine as the enclave. This allows

enclaves to perform local attestation and to establish secure channels between each other.

A special platform enclave called the quoting enclave is used to support remote attestation—authenticating an enclave to a remote server. This enclave signs reports with a device-specific key, producing a structure called a quote. The device-specific key is only visible to the quoting enclave, and therefore a remote server can trust quotes signed with this key. The actual key itself is embedded in the processor by Intel and therefore Intel is the root of trust [11, 22].

After remote attestation is complete, a server is assured that the enclave it is talking to matches its declared measurement, and a secure channel has been established between them, allowing the server to provide secrets to the enclave.

Sealing. In order to save an enclave's state, SGX provides a sealing mechanism to save and restore data from disk. Using hardware-derived keys unique to every processor and enclave, secrets can be decrypted and restored [11, 22].

Bridge Functions. At the assembly level, enclaves have only a single entry point. In addition, data must be copied to and from an enclave since enclave memory cannot be read from outside the enclave. Therefore, Intel provides a mechanism to create bridge functions in their SGX SDK. At the enclave entry point there is a function that dispatches calls into the enclave (ecalls) to the correct enclave functions, which allows an enclave to enforce a set of functions (specified by the developer) that can be called from untrusted code and vice versa (i.e., ocalls, untrusted functions that can be called from trusted code). Functions are identified by indexing into a table of function pointers. The bridge functions (both ecalls and ocalls) automatically handle copying the contents of buffers across the enclave boundary and allow for the illusion that enclave functions are being directly called from outside the enclave [11, 22].

2.2 The Default Protection Model of SGX

While SGX has been presented as an environment guaranteeing both confidentiality and integrity within enclaves, it is possible to analyze and inspect all enclave code; the code loaded into an enclave can be disassembled prior to enclave initialization [29]. Therefore, despite the claims of SGX, there is no framework for code confidentiality by default.

- **Data Integrity.** Since the code in an enclave can be attested, it is possible to ensure all code that modifies data is trusted. Data in an enclave can be encrypted when it is sent to a remote server. This, combined with a message authentication code (MAC), ensures that an attacker can never modify data without it being detected. This is important in preventing software tampering, because it prevents an attacker from directly changing any data.
- **Code Integrity.** Similarly to data integrity, the integrity of the code in an enclave can be assured via remote attestation. The original vendor of software can ensure that the code that the client is running is identical to the code provided by the vendor. This also defends against any tampering of program code.
- **Data Confidentiality.** While an enclave is running, all data inside it is invisible to all other software running on the machine, including the OS or hypervisor. Since data sent to and from enclaves can be encrypted, an enclave can communicate with a trusted server without any chance of eavesdropping. This helps enable DRM and protection of secrets by making it impossible to perform runtime analysis on an enclave.
- **Code Confidentiality.** In contrast to runtime data, the code (and static data) of an enclave is in plain text and can be disassembled and statically analyzed. There are no current defenses in place to prevent this. This means that it is possible to reverse-engineer algorithms or DRM mechanisms by analyzing the enclave binary. Our framework aims to provide a default solution to cover this crucial aspect of enclave security.

3 Overview

In this section, we provide an overview of our SgxElide framework. We first describe the challenges we will be facing in §3.1, then present our key insights and solutions in §3.2. Next, we describe how SgxElide works in §3.3, and finally present how a developer would use our framework in §3.4.

3.1 Challenges

Our objective is to ensure the confidentiality of code inside an enclave. We focus on compiled C/C++ code (instead of scripting languages) running inside enclaves, meaning enclaves must be self-modifying in order to run code that was not present when it was first initialized. Therefore, any enclave contents that we want to keep confidential must not be visible in the initial enclave binary, and said contents must be decrypted and restored at runtime. While adding code encryption and decryption to enclaves seems obvious and straightforward in theory, there are subtle complications and challenges in practice, particularly from the constraints of enclave code execution.

- **Enclaves must be signed and cannot be modified until after they are initialized**. We therefore must have self-modifying enclaves. In addition, since enclaves cannot be modified until after loading, we must set the code pages as writable before signing or after loading. However, it turns out that dynamically setting page permissions for an enclave at runtime is not permitted by the hardware.
- **The entire enclave cannot be encrypted**. An intuitive and simple solution may be to simply encrypt the entire enclave, leaving only a tiny bootstrap function to decrypt the rest of the code after the enclave is started. Unfortunately, calls in and out of the enclave

must pass through bridge functions created during compilation by the SGX SDK. If we blindly encrypt these bridge functions, control will never reach our decryption function, as the application will crash when trying to run the encrypted entry point. In order to make our solution work with the official SDK, we must be careful not to disrupt any of the bridge functions.

- **The decryption key or plaintext secret code must not be in the enclave**. There is no way for an enclave to securely decrypt itself; any key contained in the binary can be extracted by an attacker. While the SGX platform could be used to generate a key, each piece of SGX hardware has its own hardware key. This is perfect for an enclave on a specific machine to seal its own secrets, but insufficient for deploying a general application. Since there is no way to hide the key in the enclave, the enclave must retrieve the key or secret code from a trusted remote source.
  An important result of this is that a remote enclave on an untrusted machine is inherently vulnerable to denial-of-service attacks, because it requires a trusted remote server. If an attacker prevents the remote server from communicating with the enclave, it will not function. Since this is unavoidable, we can only seek to minimize the amount of communication required with the trusted server.
- **The toolchain should have minimal changes**. In order to make our solution usable, it must be compatible with arbitrary SGX applications, requiring no major restructuring of an application in order to work.

3.2 Key Insights and Solutions

Our key insights. One straightforward approach might be to encrypt the entire enclave and unpack it at runtime, similar to a binary unpacker [10]. However, this leads to more challenges: it is not possible to create and initialize one enclave from inside another (i.e. pages added before initialization should not be in the EPC), and each enclave is designed with bridge functions designed to enter and exit the enclave, complicating unpacking.

Therefore, instead of taking inspiration from packers, we instead try a different approach: redacting confidential enclave functions and then restoring them later. This can be thought of as "sanitizing" the enclave. Next, we have to consider the best way to specify the functions to be redacted, since we are now focused on redacting certain functions in an enclave. There are two ways to approach this: with a blacklist or a whitelist.

- **Blacklist**. A blacklist-based approach involves specifying which functions should be redacted. One way this could be implemented is for the developer to annotate each function that they want to keep confidential. While this can keep the number of functions that are redacted to a minimum, it puts the burden of securing the enclave on the developer. It requires the developer to decide which functions are secret and mark them accordingly, leading to potential mistakes.
- **Whitelist**. A more general approach is using a whitelist. Instead of having to specify which functions to redact, we instead specify which functions we should *not* redact. This information is applicable to any enclave, and does not need to be specified by a developer because it consists of only the functions required to restore redacted functions, meaning we can redact all other functions.

Initial Approach. We explored several solutions before our final design. Since the objective is to provide confidentiality for code considered sensitive, we at first considered it important to encrypt only secret functions, since there may be many functions in an enclave that are safe being public and need no encryption. Therefore, we originally used a blacklist approach, requiring secret functions to be annotated. Those functions were then placed in a special text section separate from the public functions, and that section was then encrypted. At runtime, the secret functions were restored within this special memory region. This meant that the only self-modifying code was the secret text section (a smaller attack surface) at the expense of requiring developers to determine which functions were confidential. We eventually decided that greater transparency was more important, and therefore we decided not to use this approach.

Our Solution. We therefore use a whitelist. Since we would like our framework to be as transparent as possible to a developer, it is tempting to simply place the code for restoring the encrypted functions in the enclave's initialization code, thus requiring no changes in how an enclave is initialized. However, there are a number of things that could go wrong when trying to restore the encrypted data. Therefore, we designed SgxElide so that a developer must call a single function to restore all encrypted functions. This allows them to handle errors in a way unique to their application. We used the following strategies to design SgxElide.

- **Sign a dummy enclave and restore all secrets after initializing**. We can create and sign a dummy enclave with most functions redacted. All functions for restoring redacted code are untouched. The dummy enclave restores the redacted code at runtime.
- **Encrypt all nonessential functions**. The enclave requires the functions required to retrieve and decrypt secret code from a server. By whitelisting all necessary functions and leaving them intact in the dummy enclave, the code will be able to initialize successfully. All other functions can be redacted.
- **Use remote attestation**. An SGX enclave can guarantee to a server it was unmodified before it was initialized via remote attestation. This means that the decryption key or code can be held remotely and is

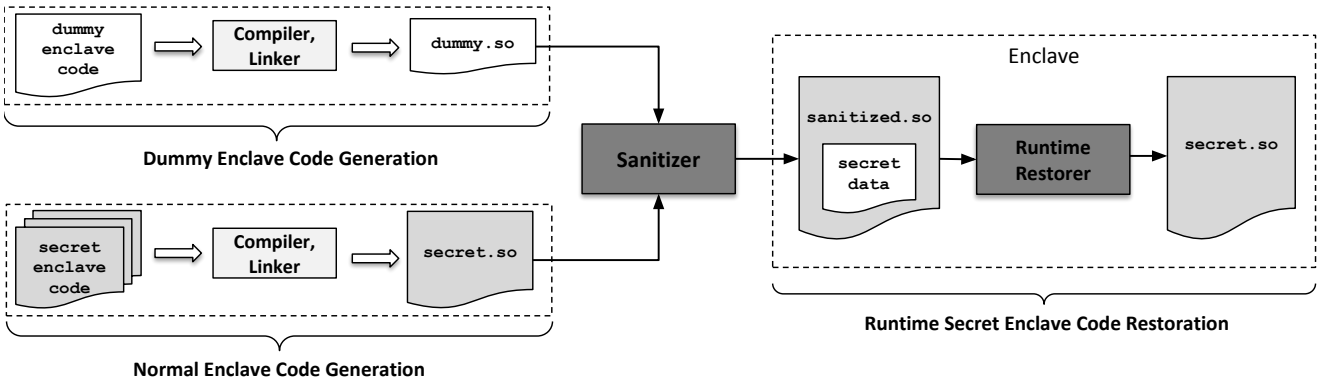

Figure 1. Overview of Our SgxElide Framework.

only ever provided over a secure channel to an enclave that has attested it is running the developer's trusted dummy enclave.

- **Use both local and remote storage**. If stored locally, the secret code must be stored encrypted either in the dummy enclave or as an encrypted file on disk. However, there is no reason that the code must be stored locally. It could also be sent from a trusted server over an encrypted connection, and thus the code itself can simply be loaded after it is obtained from the server. However, this introduces a tradeoff between size overhead on the client and network overhead. If the secret code is included encrypted with the enclave, then the enclave will only need to retrieve the decryption key from the server, whereas if the code is stored remotely, then the encrypted code must be sent to the client before the enclave can be used.

3.3 SgxElide Overview

An overview of SgxElide is provided in Figure 1. From a developer's perspective, there are just two components: a *Sanitizer* and a *Runtime Restorer*. To use SgxElide, an enclave programmer will develop the (secret) enclave code as usual, integrate the regular enclave code with our library, and compile it to produce `secret.so`.

Next, our *Sanitizer* takes `secret.so` and redacts all user defined functions in this shared library. To know precisely which function is user defined, our *Sanitizer* uses a whitelist our framework provides. All functions not on the whitelist are considered user functions and will be sanitized. Our *Sanitizer* produces `sanitized.so` with all secret functions redacted. We place metadata in a `enclave.secret.meta` file and the original raw bytes in `enclave.secret.data`.

At runtime, `sanitized.so` is initialized as usual. Then `elide_restore` is invoked by developer code to perform remote attestation and then retrieves the secrets from a trusted party through a secure channel. Then our *Runtime Restorer* restores the secret enclave code. Note that the implementation of the *Runtime Restorer* is embedded in `sanitized.so`.

3.4 How to Develop Secret Enclave Code

As stated earlier, we aim to require as few changes as possible to an SGX application, and our solution was to sanitize all developer functions. This requires no input from the developer, as the whitelist is identical for all SGX applications. However, we do require a developer to call `elide_restore` in order to restore enclave functions. We require this instead of automatically restoring the enclave because the enclave is not necessarily entered when created. One solution would be to insert a call to `elide_restore` at the top of all ecalls before the original functions are restored, meaning the first ecall to be called would restore the enclave before continuing. However, this would result in unpredictable latency from the first ecall invocation. In addition, by explicitly having developers call `elide_restore`, they can handle various errors the enclave might encounter (e.g., a network error).

Therefore, the only changes a developer must make to the enclave application are adding the library and a single call to `elide_restore`. However, the library also requires an authentication server to give an attested enclave the data it needs to restore its functions. Our framework contains a very small number of public API functions: only one ecall (`elide_restore`) and two ocalls (`elide_server_request` and `elide_read_file`). The ocalls are automatically called by our library, so the required developer effort is minimal.

Finally, in our framework, the server stands alone and requires no developer input, but in many applications it may be desirable for the developer to add custom functionality between enclave and server. However, for the task of simply attesting that an enclave is running on real SGX hardware, the process can be automatic. Thus our framework only requires a server with access to the secret data and metadata that the enclave requires.

4 Detailed Design

As outlined in Figure 1, our system's operation is divided into three main stages: whitelist generation (§4.1), enclave sanitization (§4.2), and runtime code restoration (§4.3).

4.1 Whitelist Generation

The first stage involves building a dummy enclave (`dummy.so`) containing only SgxElide helper functions and required SGX libraries. This stage is required to generate the whitelist of functions not to be sanitized. Building the dummy enclave is quite straightforward; we write an enclave containing the helper APIs introduced by our framework and link with the libraries they require (e.g., SGX crypto libraries).

This base enclave is analyzed by our *sanitizer* to extract the whitelist of functions that do not need to be redacted. Normal users of our framework will never touch `dummy.so`. Also, note that our *sanitizer* does not need to analyze `dummy.so` to sanitize an enclave, as the extracted whitelist can be reused across all developer enclaves.

4.2 Sanitized Enclave Generation

Our *sanitizer* component takes an unsigned enclave as input and outputs an unsigned enclave with the contents of all functions not on the whitelist removed. In order for this to work, the developer must build the enclave with the SgxElide library and their own functions. Then, before signing the enclave, they pass it to the *sanitizer*, which strips out the content of all their functions based on the whitelist functions from `dummy.so`, and outputs a sanitized enclave and two files pertaining to the secrets that were stripped out.

The first file produced is `enclave.secret.meta`, which contains information about the data itself such as its size and whether it is encrypted. If the data is encrypted, the metadata also includes the decryption key. This file must never be distributed with the enclave and only reside on the authentication server. The second file contains the confidential data (i.e., `enclave.secret.data`), and must be encrypted (if delivered with the enclave) or kept only with the authentication server so that its secrets cannot be leaked. An extra flag tells the sanitizer whether to encrypt the data.

4.3 Runtime Secret Restoration

The *restoration* component must run before any of the encrypted functions can be executed. These runtime libraries are compiled into the enclave and application when the developer adds SgxElide to their project. As shown in Figure 2, the library can operate in two ways depending on whether the data is stored locally or remotely. However, both approaches share common steps.

- **Step ❶**: The application calls `elide_restore`. This is the single enclave call that the library requires to perform the entire process.
- **Step ❷**: The enclave calls `elide_server_request` with `REQUEST_META`. This function connects to the server and requests the metadata about the stored secrets that the the enclave requires.
- **Step ❸**: The server responds with the metadata. Since this data is sent over a secure connection, it may contain the decryption key for the secret data.

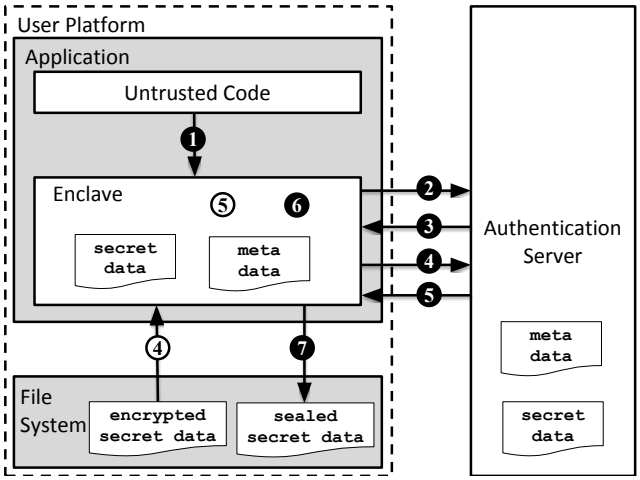

Figure 2. Runtime operation of SgxElide, showing how the enclave communicates with a server. **Step ❹** and **Step ❺** demonstrate communication with a server holding the secret data, while **Step** ④ and **Step** ⑤ demonstrate communication with a server holding the decryption key, with the encrypted secrets stored locally.

After this point, the behavior diverges based on the metadata contents. If the metadata states that the secret data is encrypted, then the data is stored encrypted locally. Otherwise, the server has the data. Note that the enclave does not need to contact the server every time. SGX's sealing mechanism provides the ability for the enclave to seal data to disk using an enclave-specific key derived from the SGX hardware key and unseal the data later, therefore allowing all accesses to the secret code after the first to require no network communications at all.

Remote Data. If the secret data is stored on the authentication server, it can simply be sent over the secure connection directly to the enclave, where it will be stored in its secure memory. This approach corresponds to the continued communications with the server in Figure 2.

- **Step ❹**: The enclave calls `elide_server_request` with `REQUEST_DATA`.
- **Step ❺**: The server responds with the data. As with the other approach, this secret data is sent over a secure channel.

Local Data. If the data is stored locally, then it must be encrypted, and the metadata will contain the decryption key for the encrypted data. This allows the enclave to finish without needing to contact the server again. The details for this approach are shown in Figure 2.

- **Step** ④: The enclave calls `elide_read_file` to load the encrypted secret data file into its memory.
- **Step** ⑤: The enclave calls the SGX library's standard decryption functions to decrypt the secret data.

The final two steps are identical for both approaches in Figure 2.

- **Step ❻**: The enclave copies the original bytes over the sanitized ones. After this point, all previously encrypted functions can now be called by other enclave functions, or they can be called from outside the enclave via a corresponding bridge function.
- **Step ❼**: Before shutting down, the enclave seals the secret data with its sealing key so that it will not need to contact the server in the future.

These two approaches represent a tradeoff between local storage and data transmitted over the network. Storing the enclave data locally requires less data to be sent from the server, but takes up more initial disk space, while storing the enclave on the server requires the server to send the redacted enclave content to the enclave.

5 Implementation

We have implemented SgxElide, which is made publicly available, in C/C++ and python on Linux. Code inside the enclave is C/C++, but the sanitizer and server components can be implemented in any language. Therefore, we wrote the sanitizer and server in python, while the enclave helper functions are written in C/C++. For enclave encryption and decryption we used the SGX SDK crypto libraries, while we used the *cryptography* package for the python server. Below we provide some implementation details of how we implement SgxElide.

Sanitizer. We provide a list of our library and default SGX functions from the dummy enclave to the sanitizer. Every time an enclave (e.g., `enclave.so`) is compiled, it is passed to the sanitizer. The sanitizer parses the ELF section headers and enumerates through each function in the shared object. Any function not on the whitelist is sanitized by overwriting its contents with zeroes. Once we finish sanitizing the original shared object, we save the original contents of the text section to an `enclave.secret.data` file. If the sanitizer is told to encrypt the data, it encrypts the original text section with a new encryption key and writes metadata to an `enclave.secret.meta` file.

Server Protocol. Communication between our client and server is simple. The client sends a single byte request representing what resource it requires (i.e., `REQUEST_META` in **Step ❷**, and `REQUEST_ DATA` in **Step ❹**, respectively), and the server responds with the data. The client and server communicate using AES GCM encryption, and if the secret data is encrypted on disk it also uses AES GCM. The metadata provided by the server consists of the data length, offset, whether it is encrypted, and (if encrypted) its encryption key, initialization vector (IV), and MAC. The offset value is the offset of the `elide_restore` function from the start of the text section.

Enclave Self-Modification. There is in fact some challenge involved to enable a self-modifying enclave. An intuitive approach to this is to change the permissions of all pages in the text section as writable right before writing to those pages. In Linux this can be done with the `mprotect` system call, although the call itself would have to be be performed outside the enclave. However, this approach does not work, as the SGX hardware enforces the original permissions.

Our solution to this is to set the permissions of the text section's pages statically when we sanitize the enclave. In ELF files, the executable file format for Linux, segments specify which parts of the file are to be loaded at certain addresses. In the program headers table in an ELF file, there is an entry for each segment, and each segment has a `p_flags` field specifying the permissions for the pages in that segment. We therefore modify the `p_flags` field for the program header entry for the text section in the enclave `.so` file; we `or` the existing field's value with `PF_W` (the flag specifying a segment is writable), making the section writable. Note that this makes the section writable through the enclave's lifetime. We discuss ways to mitigate this in §7.

When it is time to restore the enclave's contents, we load the secret data retrieved either from the server or from disk and containing the exact contents of the original text section, which we then copy over the sanitized version of the text section in memory. We use position-independent code to do this by taking the offset value from the metadata (offset of `elide_restore` from the text section's start) and subtracting it from the address of `elide_restore`. This gives the starting address of the text section, so we can copy the original text section contents directly over the sanitized contents.

This approach could be made more space efficient by keeping track of the ranges of each sanitized function and storing only that data in `enclave.secret.data`, but this would produce a more complicated implementation not necessary for a proof-of-concept. We therefore use the simple approach of saving and restoring the entire original text section instead of the individual sanitized functions.

6 Evaluation

In this section, we present how we evaluate SgxElide. We describe how we created the benchmarks in §6.1 and how SgxElide performs over the benchmarks in §6.2.

6.1 Experiment Setup

Since SGX is a new platform and requires development effort to create new SGX software, we had no benchmarks available to evaluate SgxElide. While it is possible to use a library OS (e.g., Haven [16]) to directly run legacy applications atop SGX, it does not offer the full benefits of enclave protection. Therefore, we must first develop applications that use SGX. It turns out this is actually non-trivial, because we have to select the appropriate programs in which to hide secrets and port them into SGX.

Benchmark Creation. We selected seven open source programs and ported them to SGX. We then inserted our framework into each ported application. As shown in Table 1, we selected four cryptographic algorithms, two games, and one reverse engineering challenge program. Our choice in selecting games is obvious, since games are frequent targets of reverse engineering. We use the cryptographic functions for illustrative purposes since their implementations are public and have no need to be hidden.

At a high level, an SGX application needs to be divided into trusted and untrusted components. The trusted component, containing all application secrets, will be executed inside the enclave. The rest of the application, including all runtime libraries, belongs to the untrusted component. According to the SGX developer manual [11], a developer should make the trusted component as small as possible because larger enclaves could have more vulnerabilities.

Therefore, to port an application to SGX, we must first find the secret functions we aim to protect, put them into the trusted component, and leave the rest in the untrusted component. Take the first benchmark, AES, as an example: we protected its 4 encryption and decryption related functions. However, we also needed to port another 11 functions into the enclave as they are transitively called by the first 4 functions. Partitioning a process into untrusted and trusted components is often the most difficult step because most applications are not designed to be partitioned in this way.

When porting these functions inside the enclave, we have to declare bridge functions: ecalls for the untrusted component to call enclave code (e.g., cryptographic functions), and ocalls for enclave code to call untrusted functions (e.g., system calls). Note that this is often tedious due to the strong dependency between trusted and untrusted code. We may end up with many ecalls/ocalls, depending on the secret we aim to protect. The sizes of the ported benchmarks are shown in Table 1. Details are elided for brevity.

The secrets we aim to protect are application specific. For the cryptographic functions, the secrets are the corresponding algorithms. `crackme` is similar. The secrets for the games are code that loads/decrypts the assets from disk to defeat reverse engineering.

Having ported the regular programs to SGX, we next add the protection of SgxElide. For each ported program, we simply recompile them with our framework code with no enclave code modifications. We manually insert an explicit `elide_restore` call into the untrusted component. Therefore, as illustrated in the 5th and 6th columns in Table 1, the final untrusted code size is always 50 LOC more (the call to `elide_restore` and our library's ocalls), and the trusted component is always 113 LOC more (our library's ecalls and additional library helper functions).

Our above description demonstrates that it is indeed very convenient to use our framework; we simply add one function call in the untrusted component of any SGX program, and then recompile both the trusted and untrusted component with our library to get a new SGX program protected by SgxElide. The most tedious work lies in creating the SGX programs themselves. This explains why all of our original benchmark programs are mostly small to middle sized programs (from 412 LOC to 3523 LOC).

Environment Configuration. Our experiment ran on an Ubuntu 14.04.4 LTS machine with 64GB RAM and a 3.40Ghz Intel i7-6700 Skylake CPU. We compiled our benchmarks with `gcc/g++` 4.8.4 and the Linux SGX SDK [12].

6.2 Experimental Result

Sanitizer. After we have compiled and linked each enclave's code, our *sanitizer* takes the enclave `.so` as input and produces a sanitized enclave. As reported in Table 1, for each enclave binary, we sanitize any function not on the required function whitelist. Our *sanitizer* sanitizes all functions except the 170 on the whitelist. Note we have 170 unsanitized functions due to many statically linked library functions (e.g., `sgx_rijndael128GCM_decrypt`) in our dummy enclave, in addition to our framework's functions. These whitelist functions consist of all the functions within a minimal enclave containing only our restoration code and standard SGX libraries, and provide the functionality for the enclave to restore the developer's functions.

We also measured how long it takes to sanitize an enclave, as shown in Table 2. We ran the sanitizer 10 times per benchmark, then took the average and standard deviation. The sanitization time for each enclave is around 0.09 ms for remote data and 0.15 ms for local data as they are all of similar size. The process takes less time for remote data because in that case the sanitizer does not encrypt the secret data until runtime, when the server sends it to the enclave. Sanitization will take longer for larger enclaves, but we emphasize that this occurs offline, without any impact on runtime execution.

Runtime Restorer. When called, our runtime restorer will contact the server to retrieve and restore the sanitized functions. We measured the overhead for restoration, performing restoration 10 times for every benchmark, with the result also presented in Table 2. We ran the enclave application and server on the same machine connecting over network sockets, so there was very little network latency. In our testing environment the restoration process took less than 5 ms. The overhead of using remote data was very similar to local data, with only slightly more benchmarks taking longer with remote than local; the difference is minimal. Also, note that such overhead only occurs once (when the enclave is first created), and is fixed for each specific enclave.

Overall Performance Overhead. Finally, we also measured the performance overhead of SgxElide over the SGX-only versions. Since the games run forever, we did not measure their overhead and instead measured the four cryptographic programs and `Crackme`. We used the built-in test suites for

Table 1. The ported benchmarks, each divided into an Untrusted Component (UC) and Trusted Component (TC).

Benchmarks	Original LOC	LOC w/ SGX		LOC w/ SgxElide		TC Functions	TC Bytes	Sanitized Functions	Sanitized Bytes
		UC	TC	UC	TC				
AES [1]	802	472	427	522	540	185	75999	15	3840
DES [2]	473	463	372	513	485	179	75455	9	3296
Sha1 [3]	315	423	251	473	364	179	73791	9	1632
Shas [4]	2417	1529	1240	1579	1353	224	80127	54	7968
2048 [5]	413	551	192	601	305	208	76351	38	4448
Biniax [6]	3523	3582	193	3632	306	208	76351	38	4448
Crackme [7]	48	316	93	366	206	182	73711	12	1536

Table 2. Sanitization/restoration execution time (ms) with remote/local data.

Benchmarks	Remote Data				Local Data			
	Sanitize Time	Stand. Dev.	Restore Time	Stand. Dev.	Sanitize Time	Stand. Dev.	Restore Time	Stand. Dev.
AES	0.09	0.01	4.06	0.54	0.15	0.01	3.76	0.20
DES	0.09	0.01	3.99	0.52	0.14	0.01	3.97	0.75
Sha1	0.09	0.01	3.67	0.35	0.14	0.01	3.97	0.98
Shas	0.09	0.00	4.06	0.53	0.15	0.01	4.26	0.97
2048	0.09	0.01	3.78	0.52	0.15	0.01	3.73	0.28
Biniax	0.09	0.00	4.44	0.61	0.15	0.01	4.32	0.92
Crackme	0.09	0.01	3.53	0.28	0.15	0.00	3.54	0.78

the cryptographic programs for testing and directly execute `Crackme` since it does not require input. We ran the selected benchmarks 10 times each. The normalized average performance overhead for this measurement is presented in Figure 3 and Figure 4. The runtime overhead increase is tiny (all $< 3\%$ over the SGX version for both remote and local data). This is expected because all SgxElide applications have fixed startup overhead from restoring the enclave functions, determined by the amount of code restored; after that point the code is identical to the plain SGX version, and the runtime is dominated by identical enclave computations.

7 Discussions

Security Implications. The goal of SgxElide is to offer developers an almost transparent approach (requiring only one line of code to call `elide_restore`) to provide code secrecy. No prior works, nor Intel, offer such a capability. While the high level concept may appear trivial, many seemingly obvious approaches are unfeasible. We had to examine various alternatives and experiment with what is and is not allowed in enclaves. The result is a combination of existing concepts such as self-modifying code.

By making enclaves self-modifying, SgxElide introduces new security challenges and changes assumptions about how enclave code can be vetted—code screening as in Apple's iPhone app store will not work since SgxElide enclaves are self-modifying. This also introduces the security issue of whether a platform owner (e.g., a cloud provider) should trust enclave code. For instance, an enclave can pass an initial scan for malicious code, but later unpack a malicious payload. Therefore, there is a need for new research to search for solutions in defending against malicious enclaves. However, enclaves are isolated and depend on the OS and host application to interact with the outside world, so a security policy could restrict an enclave's capabilities. Also, developers must sign enclaves before distributing them, so there is a degree of attribution that may make it possible to blacklist or identify malicious developers.

However, security concerns go beyond intentionally malicious enclaves. Since SgxElide makes the enclave text section writable, certain vulnerabilities in an enclave could result in an attacker inserting arbitrary malicious code into the enclave. We added an `mprotect` call revoking `PROT_WRITE` for the enclave text section immediately after restoring the enclave code. However, `mprotect` must be called outside the enclave, so this would not defend against a malicious OS or host application. Note that this still requires a vulnerability in the enclave to allow an attacker to actually modify enclave code, and this can be protected against by using software-based DEP, in which the enclave code is compiled with memory write instructions that can never write to the text section [34]. Only the restoration instructions would be allowed to overwrite code. Also, while there is no way to securely change runtime permissions in the currently available SGX-v1, SGX-v2 will provide this ability [8].

The recent discovery of the powerful controlled-channel attacks against SGX showed that enclave code could potentially leak extensive amounts of data to a malicious OS [39].

However, our solution is in fact an excellent defense against such attacks, because controlled-channel attacks require knowledge of the code in order to extract secrets. If the code itself is hidden, an attacker will not have this information.

Limitations and Future Work. We have demonstrated that we can use SgxElide to protect enclave code secrecy, but there are several ways to improve our work. One is to make our framework totally transparent if a user does not mind unpredictable runtime latency imposed during restoration. As discussed in §3.4, we decided to have developers explicitly call `elide_restore`; we could remove this explicit call by having restoration occur the first time an ecall is made.

Second, we only ported a handful of benchmarks to evaluate our framework. Our immediate task is to investigate using SgxElide to protect large scale and more practical software. Another valuable research direction lies in developing automatic techniques to partition code [26], which could significantly boost the speed of enclave code development.

Third, we currently only focus on code secrecy against existing reverse engineering techniques. SgxElide does not protect against data leakage vulnerabilities. We will look into how to secure the enclave against other attacks.

Finally, we did not implement a few details of our framework, such as the final sealing step or performing full attestation. These would be important for an actual production system, but our implemented framework is sufficient to demonstrate the effectiveness of our approach.

8 Related Work

Binary Code Protection. The goal of code obfuscation is to disrupt analysis of the code and deter reverse engineering efforts. In general, there are three types of widely used binary analysis platforms: disassembler, debugger, and VM. Consequently, obfuscation techniques can be categorized into anti-disassembler, anti-debugger, and anti-VM. For each category, there exists a variety of techniques. For example in the anti-dissembler category, there exists the techniques of garbage code insertion (e.g., [27]), control flow obfuscation (e.g., [25]), instruction aliasing (e.g., [30]), or binary code compression and encryption (e.g., various packers such as UPX [10]).

SgxElide offers a new way of protecting binary code by using the security provided by enclaves. With hardware-enforced security provided by SGX, existing disassembler, debugger, and VM-based reverse engineering techniques will no longer work.

SGX Applications. Since SGX holds great potential to solve challenging security problems [13], many efforts have started to explore the potential of SGX. Haven [16], SCONE [14], and Graphene SGX [38] run native applications inside SGX enclave without modification. VC3 [33] demonstrated code and data confidentiality of MapReduce computation in the cloud. SGXRand [18] mitigated the side channel leakage of data analytics with SGX by using data noise. Most recently,

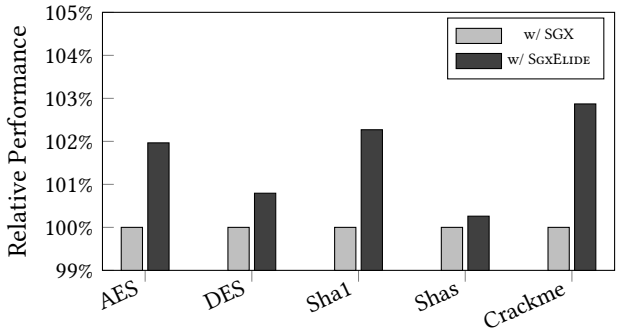

Figure 3. Overhead of running the SgxElide protected benchmark with remote data.

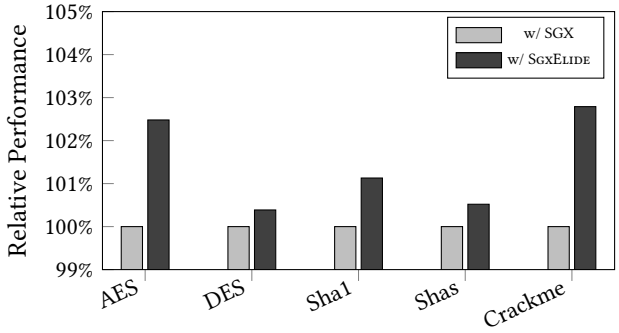

Figure 4. Overhead of running the SgxElide protected benchmark with local data.

SGX-BigMatrix [36] provided a framework to secure matrix operations for data analytics in the cloud.

Improving Security with SGX. SGX-LAPD [20] thwarts the controlled side channel attacks [39] via compiler extensions. T-SGX [37] defeats them via hardware transactional memory (HTM) and compiler extensions, and Cloak [21] defeats cache side channel attacks against SGX with just HTM. SGX-Shield [35] enables ASLR to defeat memory corruptions, and SGXBounds [24] provides memory safety for enclave programs.

9 Conclusion

In this paper, we presented SgxElide, a framework to ensure enclave code confidentiality. By treating code as data, we dynamically restore code at runtime by writing the decrypted code over the sanitized functions. SgxElide can easily be integrated with any SGX project to provide code secrecy, with secrets delivered by a developer-controlled trusted party. We have implemented SgxElide atop the Linux SGX SDK, and our evaluation with SGX programs shows that SgxElide can be used to protect the code secrecy of practical applications without any runtime overhead after the enclave is initialized.

A Artifact description

A.1 Abstract

The artifact contains the SgxElide framework and its associated benchmarks as described in our CGO 2018 paper *SGXElide: Enabling Enclave Code Secrecy via Self-Modification*. It requires an Intel Skylake processor (or newer) that supports Intel SGX instructions (or simply an x86-64 processor if running in simulation mode). The benchmarks were used to create the data for the performance evaluation in the paper.

A.2 Description

A.2.1 Checklist (Artifact Meta Information)

- **Program:** SgxElide framework written in C++ and Python, Benchmark programs written in C/C++
- **Compilation:** gcc/g++
- **Binary:** x86-64 executables for SGX-compatible hardware (Unless in simulation mode)
- **Run-time environment:** 64-bit Ubuntu with Intel SGX SDK installed
- **Hardware:** Any Intel Skylake processor (or newer), SGX-compatible BIOS (SGX hardware support not required for simulation mode)
- **Output:** SGX applications with encrypted enclaves
- **Experiment workflow:** Git clone; Install SGX SDK; Install dependencies; Compile BaseEnclave for whitelist; Compile SampleEnclave, start server.py and run app for simple example; Compile benchmarks; Start server.py for each benchmark; Run each benchmark for results
- **Publicly available?:** Yes

A.2.2 How Delivered

SgxElide and its associated benchmarks are hosted on Github at https://github.com/utds3lab/sgxelide.

A.2.3 Hardware Dependencies

SgxElide requires any Intel Skylake processor (or newer) and an SGX-compatible BIOS. If running in simulation mode (for testing purposes only), SGX hardware support is not required. However, it may be difficult to get SGX applications running in simulation mode on incompatible hardware, so running on a compatible system is strongly recommended.

A.2.4 Software Dependencies

SgxElide requires the Intel SGX SDK, Python 2.7, gcc, g++, and the *pyelftools* and *cryptography* python libraries. The two graphical game benchmarks also require `libsdl1.2`, `libsdl-image1.2`, and `libsdl-mixer1.2`.

A.3 Installation

Clone the git repository and install all dependencies. Depending on the SGX SDK version (or if running in software versus hardware mode), SgxElide's function whitelist may need to be re-generated, which is done by running make in the *BaseEnclave* directory. The resulting whitelist.json then needs to be copied to the other project directories.

To run an example of SgxElide, compile *SampleEnclave* with make, start `server.py`, and run the binary called `app`. The benchmarks are compiled the same as *SampleEnclave*. The python script `server.py` needs to be run before each SgxElide application, as they need to communicate with the server.

More detailed installation information can be found in the README in the git repository.

A.4 Experiment Workflow

When a benchmark is compiled, the sanitizer is run as part of the build process. The sanitization time is calculated by using the output of the `time` command when running the sanitizer.

In order to measure the performance of the benchmarks on remote data versus local data, a single change needs to be made to the Makefile of each benchmark. The sanitizer will encrypt enclave data if the `-c` flag is passed (local data), and not encrypt the data if no flag is passed (remote data).

Each benchmark once compiled can be run by first starting the benchmark's server with `python server.py` and then running the benchmark with `./app`. The benchmark will print timing information (the line that says "Time elapsed in enclave initialization") as well as perform its original functionality (e.g., run the game for the game benchmarks).

In order to measure the overall performance overhead of the non-game benchmarks, the `time` command was run for each benchmark.

The benchmarks without SgxElide (used for baseline performance) can be run with `./app`.

A.5 Evaluation and Expected Result

The sanitizer for each benchmark will run when compiling each benchmark and will print the time it took to sanitize the enclave. This was used to obtain the data for *Sanitize Time* in Table 2.

The timing information for the benchmarks both with and without SgxElide is printed when the benchmark is run. This was used to obtain data in Table 2, Figure 3, and Figure 4. Timing data will be different when running in simulation mode, so hardware mode should be used to compare results.

Acknowledgement

We thank the anonymous reviewers for their very helpful comments. This research was partially supported by NSF awards CNS-1453011, CNS-1564112, and CNS-1629951. Any opinions, findings, conclusions, or recommendations expressed are those of the authors and not necessarily of the NSF.

References

[1] https://github.com/kokke/tiny-AES128-C.

[2] https://github.com/tarequeh/DES.

[3] https://tools.ietf.org/html/rfc3174.

[4] https://tools.ietf.org/html/rfc6234.

[5] https://github.com/poupou9779/z2048.

[6] http://mordred.dir.bg/biniax/download2.html.

[7] https://exploit.ph/reverse-engineering/2014/05/11/an-easy-linux-crackme/.

[8] https://software.intel.com/en-us/forums/intel-software-guard-extensions-intel-sgx/topic/624878.

[9] Intel software guard extensions sdk for linux os. https://download.01.org/intel-sgx/linux-1.6/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.6_Open_Source.pdf.

[10] Upx: the ultimate packer for executables. http://upx.sourceforge.net/.

[11] Intel software guard extensions programming reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, Oct. 2014.

[12] Intel sgx for linux. https://github.com/01org/linux-sgx, June 2016.

[13] I. Anati, S. Gueron, S. Johnson, and V. Scarlata. Innovative technology for cpu based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, page 10, 2013.

[14] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. Stillwell, et al. Scone: Secure linux containers with intel sgx. In *OSDI*, pages 689–703, 2016.

[15] E. Bauman and Z. Lin. A case for protecting computer games with sgx. In *Proceedings of the 1st Workshop on System Software for Trusted Execution (SysTEX'16)*, Trento, Italy, December 2016.

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

[17] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3):8, 2015.

[18] S. Chandra, V. Karande, Z. Lin, L. Khan, M. Kantarcioglu, and B. Thuraisingham. Securing data analytics on sgx with randomization. In *Proceedings of the 22nd European Symposium on Research in Computer Security*, Oslo, Norway, September 2017.

[19] C. S. Collberg and C. Thomborson. Watermarking, tamper-proofing, and obfuscation-tools for software protection. *IEEE Transactions on software engineering*, 28(8):735–746, 2002.

[20] Y. Fu, E. Bauman, R. Quinonez, and Z. Lin. Sgx-lapd: Thwarting controlled side channel attacks via enclave verifiable page faults. In *Proceedings of the 20th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'17)*, Atlanta, Georgia. USA, September 2017.

[21] D. Gruss, J. Lettner, F. Schuster, O. Ohrimenko, I. Haller, and M. Costa. Strong and efficient cache side-channel protection using hardware transactional memory. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 217–233, Vancouver, BC, 2017. USENIX Association.

[22] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[23] G. Hoglund and G. McGraw. *Exploiting online games: cheating massively distributed systems*. Addison-Wesley Professional, 2007.

[24] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. Sgxbounds: Memory safety for shielded execution. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 205–221. ACM, 2017.

[25] T. László and Á. Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[26] J. Lind, C. Priebe, D. Muthukumaran, D. OâĂŹKeeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, et al. Glamdring: Automatic application partitioning for intel sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC 17), Santa Clara, CA*, 2017.

[27] C. Linn and S. Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[28] A. Majumdar, C. Thomborson, and S. Drape. A survey of control-flow obfuscations. In *International Conference on Information Systems Security*, pages 353–356. Springer, 2006.

[29] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8, Tel-Aviv, Israel, 2013.

[30] T. Mudge, C.-C. Lee, and S. Sechrest. Correlation and aliasing in dynamic branch predictors. In *Computer Architecture, 1996 23rd Annual International Symposium on*, pages 22–22. IEEE, 1996.

[31] I. V. Popov, S. K. Debray, and G. R. Andrews. Binary obfuscation using signals. In *Usenix Security*, 2007.

[32] W. Rosenblatt, S. Mooney, and W. Trippe. *Digital rights management: business and technology*. John Wiley & Sons, Inc., 2001.

[33] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy Data Analytics in the Cloud using SGX. 2015.

[34] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[35] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. Sgx-shield: Enabling address space layout randomization for sgx programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[36] F. Shaon, M. Kantarcioglu, Z. Lin, and L. Khan. A practical encrypted data analytic framework with trusted processors. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS'17)*, Dallas, TX, November 2017.

[37] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the 2017 Annual Network and Distributed System Security Symposium (NDSS), San Diego, CA*, 2017.

[38] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library os for unmodified applications on sgx. In *2017 USENIX Annual Technical Conference (USENIX ATC)*, 2017.

[39] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.