IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received 26 July 2024, accepted 7 August 2024, date of publication 9 August 2024, date of current version 19 August 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3441240

RESEARCH ARTICLE

Automated Intel SGX Integration for Enhanced Application Security

LEONARDO REGANO[1] **AND DANIELE CANAVESE**[2]
[1]Dipartimento di Ingegneria Elettrica ed Elettronica, Università degli Studi di Cagliari, 09123 Cagliari, Italy
[2]Centre National de la Recherche Scientifique (CNRS), Institut de Recherche en Informatique de Toulouse (IRIT), 31062 Toulouse, France

Corresponding author: Leonardo Regano (leonardo.regano@unica.it)

This work was partially supported by project SERICS (PE00000014) under the NRRP MUR program funded by the EU - NGEU.

ABSTRACT Nowadays, many chip manufacturers offer various Trusted Execution Environment (TEE) implementations to protect the critical data and the algorithms in hardware. One of Intel's answers to the TEE race is SGX (Software Guard Extensions), which enables the creation of hardware-encrypted memory areas known as enclaves. Although it promises a high-security level, it still requires expertise, effort, and time to convert a traditional application into an SGX-enabled one. This paper proposes a novel approach to generate enclaves from existing C/C++ applications automatically. Our strategy involves annotating the sensitive code to be protected, which is then statically analyzed and modified to comply with all the SGX requirements. Our approach does not require the user's prior knowledge of the SGX platform. The framework automatically identifies and implements all the required modifications of the target application source code to make it compatible with the SGX toolchain. In addition, it is fast and can port big applications containing hundreds of functions in mere minutes, as we proved experimentally.

INDEX TERMS SGX, software security, static analysis, TEE, usable security.

I. INTRODUCTION

In today's digital landscape, preserving the confidentiality of sensitive data is an increasingly critical concern. Storing critical information in the local memory without any protection, even for a minimal time, can pose a significant risk. This approach may be dangerous, with the pervasive use of third-party services that have opened up new routes for attackers to steal private data. To avoid such perils, the processes that manipulate sensitive data should do so in a TEE (Trusted Execution Environment), a shielded memory area, usually safeguarded by some hardware protection technique. Many manufacturers ship TEE-enabled processors using technologies such as ARM's TrustZone[1][2] or AMD's SEV[3] (Secure Encrypted Virtualization). One of Intel's answers to the "TEE-race" is SGX (Software Guard eXtensions).[4]

The SGX technology allows a process to create TEEs named *enclaves*, which are AES-encrypted memory areas. The access to the enclaves is entirely transparent to the legitimate process, but any unauthorized access will only be able to get the encrypted data. This technology protects sensitive code and data written in the C and C++ programming languages. Open source project extends SGX support for other programming languages, including Go[5] and Rust.[6] It not only requires a compatible processor but also needs the code to be correctly organized and designed. Any SGX-enabled application code can be divided into two parts. The trusted part contains the sensitive data and algorithms to be safeguarded, while the untrusted part contains the unprotected code.

The associate editor coordinating the review of this manuscript and approving it for publication was Sedat Akleylek.

[1]https://www.arm.com/technologies/trustzone-for-cortex-a
[2]https://www.arm.com/technologies/trustzone-for-cortex-m
[3]https://www.amd.com/en/developer/sev.html
[4]https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html
[5]https://github.com/edgelesssys/ego
[6]https://github.com/apache/incubator-teaclave-sgx-sdk

Developing an SGX-enabled application from scratch or porting an existing one is not trivial. First, the developer must know the SGX architecture and its internal mechanics. Then, the trusted part must be identified, and the code must be modified to comply with all the Intel SGX requirements. These requirements include managing the communication between the trusted and untrusted parts, handling the errors, and performing the I/O without compromising the confidentiality of the data. These operations are complex and expensive to carry out manually, particularly for large projects, so an automatic conversion tool for pre-existing applications is desirable.

Our approach and Proof of Concept (PoC) implementation allow us to effortlessly transform a traditional application into an SGX-enabled one without requiring the developer to know the internal workings of the Intel technology. The developer only has to annotate the sensitive information, and our framework will do the rest. Our tool will statically analyze the code, modify it to create an enclave, and perform various sanity checks to ensure that the confidentiality of the data is not compromised. The scientific literature provides other approaches for automatically adapting applications to employ Intel SGX. However, to the best of our knowledge, our work is the first to adopt a function-driven approach, which is particularly suited to safeguard the Intellectual Property of algorithms contained in software. We elaborate more on this in Section V.

In summary, our main contributions include:

- a system that allows non-experts to easily enable the support for SGX enclaves in traditional C/C++ applications by simply annotating the code areas to be protected;
- introducing an automatic system able to statically analyze an application and verify its conformity to all the SGX requirements to avoid security issues and information leakage.

This paper is structured as follows. Section II contains a brief background on SGX and the static analysis tools used in our framework. Section III is the core of this document and describes in detail the internal workings of our tool. Section IV illustrates the experimental results of testing our system on different applications. Section V contains the related works, and Section VI concludes this paper.

II. BACKGROUND

This section introduces the Intel SGX technology, its limitations, and its vulnerabilities. In addition, we also briefly describe the static code analysis tools that our framework leverages.

A. INTEL SGX

Software Guard Extensions [1] (SGX) is a code protection solution introduced in 2015 by Intel Corporation in several processors to protect critical data and algorithms. SGX allows an application to allocate private regions of memory called enclaves. These memory areas are protected from access even by processes running at higher privilege levels by an encryption mechanism performed in hardware. The Intel SGX architecture allows the allocation of multiple enclaves for a single application. Dividing sensitive parts of an application into smaller enclaves reduces the attack surface and makes the code more manageable and reusable.

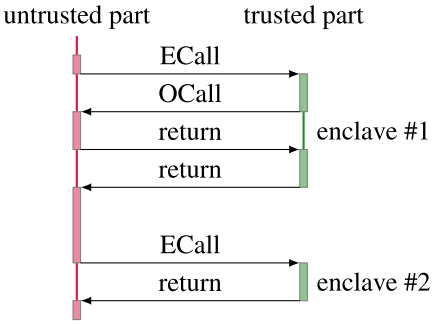


FIGURE 1. Workflow of a typical SGX application.

Data protection is ensured by an AES encryption/decryption mechanism transparent to legitimate applications. SGX supports two ways to encrypt data in the enclaves: a unique key for each enclave or a shared key for multiple enclaves. On the other hand, data integrity is guaranteed by computing and checking a truncated Wegman-Carter MAC for every enclave. Although the Memory Encryption Engine (MEE) circuit [2] performs the encryption and integrity checks directly in hardware, these security mechanisms still introduce a non-negligible time overhead. For this reason, enclaves are recommended only for manipulating sensitive information such as account credentials, banking data, or medical records.

1) DEVELOPMENT

The developer wanting to convert a traditional application into an SGX-enabled one by hand has several tasks to perform. First, the developer has to register on the Intel website and request a certificate to sign the SGX binary later. Then, he has to write an *EDL* (Enclave Description Language) file [3]. This file describes not only the enclaves but also the function calls used by the untrusted code to enter into an enclave (called *ECalls*) and the function calls used by an enclave to pass data to the untrusted code (called *OCalls*). Then, the developer must edit the C/C++ source code to fulfill all the SGX requirements. These requirements include, for instance, replacing all the calls to various standard C functions (e.g., malloc) with their SGX equivalent. Some functions and operations are prohibited in an enclave (e.g., strcat), so they must be appropriately replaced. Finally, the developer must run the Intel SGX SDK toolchain to compile everything, and the result will be a binary containing the untrusted code and a compiled dynamic library containing the encrypted and signed enclave code.

2) EXECUTION

When an SGX-enable application is launched, the untrusted part executes first. When the first ECall is encountered, the dynamic library is loaded into the memory, and the secure code inside the enclave is run. If the enclaves contain some OCalls, they are executed, temporarily switching the run-time to the untrusted part. This flow is executed until the enclave ends. Then, the untrusted execution can resume until the next ECall or the program termination. Figure 1 depicts this workflow.

3) LIMITATIONS

The Intel SGX must be enabled in BIOS/UEFI of a compatible processor and supported by the operating system. Although promising, this technology has many limitations, such as:

- Power events (e.g., system sleep and hibernation) can cause data loss in an enclave. If the event occurs during the execution of the untrusted part, the application can manage it as it seems most suitable. On the contrary, the data loss will be completely unrecoverable if the event arises during an ECall or OCall.
- The use of enclaves slows down performance due to function calls and encryption/decryption processes. The performance loss is mainly due to the context change procedures for transitioning between the code inside and outside the enclave.
- Function calls in an enclave have several limitations. The Intel SGX SDK provides several ad-hoc redefined libraries, exposing only some standard functions and removing the unsafe ones (which cannot be used). I/O interfaces and specific OS calls are not allowed, but this can circumvented by using OCalls to user-defined functions as a bridge between the protected and non-standard library functions.

4) ATTACKS

The Intel architecture SGX is susceptible to a variety of attacks. On the other hand, attackers can also use enclaves to distribute more robust malware. Some threats related to this technology include:

- *Enclave call ordering* [4]: An attacker can change the ECalls' order in an application's untrusted part. By altering such ordering, he can indirectly change the data injected inside an enclave without directly tampering with it.
- *Enclave replay attacks* [4]: An attacker can intercept the data exchanged with an enclave and execute another unexpected ECall using old messages as its parameters. This allows the potential reuse of an enclave many times if no countermeasures are put in place (e.g., using nonces to validate the freshness of the ECalls).
- *Iago* [5]: These attacks are not directly related to enclaves but can still be mounted against the untrusted part of an SGX-enabled application. In a Iago attack scenario, the kernel is malicious and returns a corrupted value when a victim application executes a system call. Enclaves cannot directly use system calls, but they can use them indirectly via OCalls, allowing a tampered kernel to taint an enclave.
- *Prime+Probe* [6]: In 2019, Schwarz et al. developed the Prime+Probe attack using Intel SGX to hide a malware that can recover RSA keys stored in other enclaves. The attack first locates the processor cache portion of the victim enclave. Then, it uses high-resolution time measurements to create a partially recovered RSA key. Multiple executions of these attacks allow the full recovery of a private key. The attack managed to extract 96% of a 4096-bit RSA private key with only 11 executions.
- *SGXPectre* [7]: Chen et al. have found that Spectre attacks, adapted for enclave execution, can exploit branch prediction to reconstruct the content of the encrypted memory areas of an enclave. Most of the SGX libraries are vulnerable to these types of attacks. Developers are advised to follow the Intel SGX guidelines to reduce such risks [8].
- *SGX ROP* [9]: This attack uses Return-Oriented Programming (ROP) to create dangerous unauthorized code into an enclave that is then inadvertently executed by the SGX-enabled application. This attack is hard to detect since it starts in an encrypted memory region and also to avoid since it completely bypasses ASLR (Address Space Layout Randomization), stack canaries, and address sanitization techniques.
- *Dark ROP* [10]: This attack exploits a memory corruption vulnerability in enclaves via ROP techniques. The researchers showed that they could exfiltrate the enclave's code and data into a shadow application, emulating a proper enclave. The shadow application is under the complete control of the attacker and uses only an enclave to perform some SGX-related operations, such as reading the SGX cryptographic keys.
- *SGX Bomb* [11]: RowHammer is a family of attacks exploiting unwanted side effects in DRAM cells to corrupt a memory cell. SGX Bomb is a variation of RowHammer targeting an enclave. It involves numerous attempts to access random addresses of an enclave to cause some bit flipping. This intentionally causes the processor to activate its drop-and-lock policy, causing the system to ignore the interrupts, thus initiating a denial-of-service.
- *Plundervolt* [12]: is an attack compromising the integrity of Intel SGX enclave computations based on an undocumented Intel Core voltage scaling interface. Plundervolt manipulates the processor's supply voltage during an enclave computation, inducing predictable faults within the processor package that can be used to recover cryptographic keys or create memory vulnerabilities in an enclave code.

Intel addressed the vulnerability through a microcode update.[7]

B. SOURCE CODE ANALYSIS

Static code analysis is an automated procedure to analyze source code without running the program, thus relying only on parsing and analyzing the code. Our approach uses two static analysis tools to analyze the C/C++ code to convert: Ctags and Frama-C.

C. CTAGS

Ctags[8] is a command-line tool that creates a summary file of a source code file. This summary contains information about the functions, variables, methods, classes, macros, and other definitions. It supports various programming languages like C, C++, Java, and Python.

We used this tool to find the functions to move into enclaves, their signature, and their location in the source files.

D. FRAMA-C

Frama-C[9] is an extensible framework for static code analysis of C applications, available as a command-line tool or with its graphical user interface. It has a plugin architecture that enables developers to create new modules and extend its functionality. The plugins work together in a pipeline to analyze and, if needed, modify code.

In particular, we used Frama-C to compute the call graph between the functions. The analysis is performed syntactically and semantically by analyzing pointers and estimating their values without running the application. This approach can significantly enhance the accuracy of the call graph without resorting to debugging the application under test.

III. OUR APPROACH

This section presents our PoC framework, implementing our approach for automatically protecting existing applications leveraging SGX. As previously stated, the framework protects sensitive parts of code by moving them to a protected memory area called an enclave. Its current iteration supports automatic modification of source code written in the C language to make it compatible with Intel SGX. The user should only annotate in the source code the sensitive areas of code that must be executed in an SGX enclave. The framework is entirely written in Python 3.8.1 and employs various external libraries, e.g. the static code analyzers Ctags and Frama-C. The framework is compatible with both Linux and Windows.

Following, we describe the workflow of our approach, from the test application source code to the generation of the SGX-protected application binaries. Figure 2 shows the main phases of the workflow.

[7]https://www.intel.com/content/www/us/en/support/articles/000094219/processors.html

[8]https://github.com/universal-ctags/ctags

[9]https://frama-c.com/

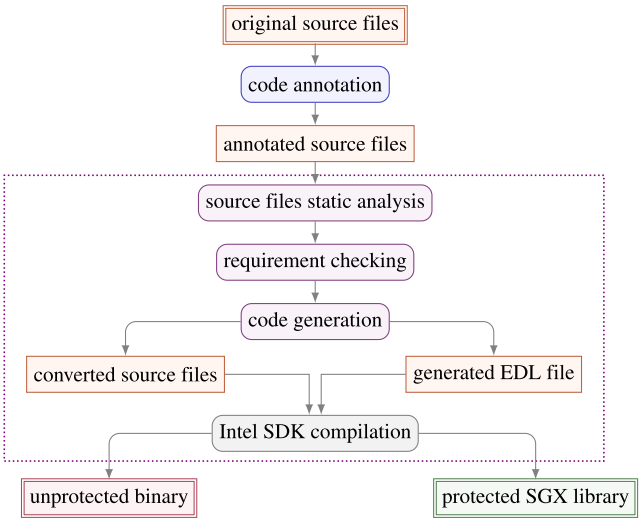


FIGURE 2. Workflow of our approach.

1) CODE ANNOTATION

Before executing the framework, the user must annotate the target application code in order to mark the functions that must be executed in SGX enclaves. These annotations contain the information the framework needs to generate an EDL file properly.

In particular, the user should mark the functions containing ECall or OCall calls, indicating in the annotations how parameters are passed in such calls. If the call parameters include buffers, their size should also be defined in the annotation following the EDL syntax. The latter is required by the SGX implementation: buffers passed as arguments in ECall or OCall calls are copied by SGX libraries to limit the attack surface, averting attacks based on invalid pointer dereferencing. Furthermore, buffer copying is needed when passing buffers in OCall calls since data contained in the buffer should be copied in an unencrypted memory area to allow the callee function to read it. Alternatively, pointers can be passed without changes.

The EDL syntax requires specifying how parameters are passed for single pointers and static size arrays. The syntax, abided also by our annotations, defines the following passing methods:

- i: when the callee execution starts, the caller's external buffer is copied in a new buffer allocated in the callee's region;
- o: when the execution returns to the caller function, the internal buffer of the callee is copied to the caller's external buffer;
- b: the caller's external buffer is copied to the callee's internal buffer call, and when the execution returns to the caller function, the callee internal buffer is copied into the caller's external buffer;
- u: performs no operations and passes the pointer to the call.

For the `u` option, the user should specify only the argument name using the following notation: `[argumentname,`

u]. The other options require the user to specify another function parameter containing the buffer size, following the syntax [argumentname, copy_option, size]. The size parameter takes on a different meaning depending on the pointed data type:

- if the pointer is of a defined type with computable size, the parameter specifies the number of elements of the vector;
- if the pointer has no type (e.g. void *), the parameter indicates the buffer size in bytes.

Furthermore, the user should also declare in each annotation the name of the wrapper function that will be stated in the EDL file and will be automatically generated by the Intel SDK. The framework will automatically change the calls to ECall and OCall target functions in order to redirect them to the SDK-generated wrappers.

```
#define sgx_ecall_calc([n, b], [list, i, size], [
    buf, u])
int calc(int n[100], int* list[10], int size, void
    * buf) {
...
}

#define sgx_ocall_print_text([text, i, size])
int print_text(char* text, int size) {
    return printf("%.*s", size, text);
}
```

Listing 1: Example of annotated functions.

Listing 1 shows an example of an ECall and OCall annotations. In the first case, the function calc is marked by the user as an entry point for an SGX enclave, i.e. is a target of an ECall call from unprotected code. The annotation contains the name of the wrapper function that the Intel SDK must generate, i.e. sgx_ecall_calc, and the passing method for each of the parameters of the ECall call. In this case, the user does not need to indicate the size for the n parameter since it is an array of fixed size. The size is instead indicated for the list parameter since it is a buffer passed by reference with the i passing mode. The parameter buf is also a buffer, but since the user has indicated the passing method u, where no buffer copying is performed by SGX, it is not necessary to indicate the size of this buffer. In the second case, the user marks the function print_text as a target for an OCall call. Since the text parameter is passed by reference, the annotation also states that the i passing method shall be used when calling the function, and the other parameter size indicates the size in byte of the text parameter.

2) PROTECTED CODE GENERATION

After annotating the source files of the target application, the user may execute the framework, which will analyze the files and modify them to ensure their compliance with SGX. Furthermore, the framework will generate the EDL file according to the information declared by the user in the code annotations. As described at the end of this section, the EDL file must be given as input to the Intel SDK compiler to build the protected application from the modified source code files.

The framework starts by parsing all the source files using CTags and a set of regular expressions in order to build a model of the application source code that will be leveraged in the subsequent phases of the workflow. In particular, the source model comprises:

- header information: included external libraries, type definitions, structures, unions, and enumerations;
- global variables;
- for each function: return type, name, arguments, local variables, and source code;
- ECall and OCall annotations with the related data;
- the application call graph, obtained via Frama-C.

The framework then analyzes the model in order to infer the required modifications to the application source files. The analysis is performed in the following steps. First, the framework verifies that, as SGX requires, all caller and callee functions of ECall and OCall calls have unique names. Then, it identifies the code that should be moved inside the enclave to safeguard the annotated functions. In particular, this analysis is needed to avoid information leaks due to calls from enclaves to unprotected code that have not been properly marked as OCall calls. For each function marked as an ECall target, the framework identifies the minimum subset of functions that must be mandatorily moved to the enclave. Minimizing the amount of code moved to enclaves is important since, as discussed in Section IV-B, SGX introduces a non-negligible computational overhead. Algorithm 1 is used to identify the functions that must be moved to the enclaves.

Figure 3 shows an example call graph of a target application, with the functions that must be moved to a SGX enclave highlighted in green. The algorithm starts navigating the call graph from the function f_0^e, targeted by an ECall call. This function will be marked to be moved to an enclave. Then, the algorithm will traverse the rest of the call graph, marking all the encountered functions (f_3,f_4,f_5) for movement to the enclave. The call graph traversal will stop when a function targeted by an OCall call is found, such as function f_6^o. As said before, the algorithm can also find incoherent situations in the code structure that may cause information leakage. For example, looking again at the example in Figure 3, if function f_1, which is not marked for safe execution in the enclave, is called by both function f_6^o (executed outside the enclave) and function f_4 (executed inside the enclave), the algorithm will stop, since the untrusted function f_1 may access privileged information inside the enclave through the trusted function f_4. In such cases, the framework will return a message to the user with an explanation of the error so that it can fix the code accordingly (e.g. annotating f_1 as an ECall target or removing the call from f_1 to f_4).

The framework will then identify the external library headers that should be included in the enclave code. Since SGX requires to include in enclave code only the ad-hoc redefined libraries described in Section II-A, the framework will analyze the external calls in the code to include only the

Algorithm 1 Call Graph Scanner

Input: the set of ECalls E, the set of OCalls O, and the set of all the functions F
Output: the set of functions X and libraries L to move into an enclave

$X \leftarrow \varnothing$
$L \leftarrow \varnothing$
foreach $e \in E$ **do**
  $R \leftarrow$ set of functions that can be reached starting from e
  $seen \leftarrow \varnothing$
  $todo \leftarrow \varnothing$
  $todo \leftarrow todo \cup \{e\}$
  while $todo \neq \varnothing$ **do**
    $now \leftarrow$ any element in $todo$
    $now \leftarrow todo \setminus \{now\}$
    $seen \leftarrow seen \cup \{now\}$
    if $edge.source = now$ **then**
      foreach $edge \in R$ **do**
        $todo \leftarrow todo \cup \{edge.dest\}$
      end
    end
    $todo = todo \setminus seen$
    $todo = todo \setminus O$
  end
  $X \leftarrow X \cup seen$
  $G = seen \setminus F$
  $L \leftarrow L \cup$ libraries defining the functions G
  $X = X \setminus G$
  return (X, L)
end

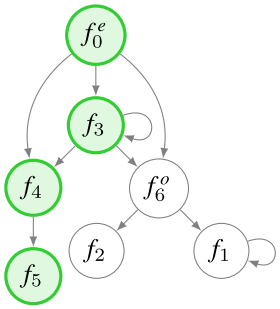


FIGURE 3. Call graph example.

required SGX libraries. The framework will also check if the user has mistakenly included calls to external functions in the enclave code that are not included in the SGX-redefined ones.

Finally, the framework will generate the SGX-compliant application code. First, the code for all the functions marked as trusted is moved to a separate source file containing all the enclave code. Then, the framework will add a wrapper function in the untrusted code for each trusted function targeted by an ECall call. The ECall calls in the untrusted code will be modified to target the corresponding wrapper function. These wrappers are needed to automatically handle the additional parameters needed for SGX ECalls, e.g. the buffer parameters passing modes and sizes indicated by the user in the ECall annotations, and the trusted function return value. The latter is used by SGX to signal runtime errors when creating enclaves and executing the related code and is automatically checked by the wrapper, aborting the application execution in case of SGX errors. The framework will then generate the EDL file using the information contained in user annotations and, as a final step, will include additional untrusted code to manage the enclave lifecycle and verify the compatibility of the hardware running the application with SGX.

3) BINARY COMPILATION

The protected binary can be generated using GNU make or Microsoft Visual Studio. Intel SGX supports different operating modes for application execution: debug, pre-release, release, and simulated. Both GNU make and Visual Studio support these modes. However, Visual Studio generates the necessary files for signing the enclave, making it easier for the user to compile the program.

a: VISUAL STUDIO

Intel created a plugin for Visual Studio that simplifies building solutions with Intel SGX. It allows the user to add enclave files and, depending on the selected application execution mode, specify the private key to sign the enclave code. The result is a protected solution that can be imported into a Visual Studio project. The user will only need to include in the project the source files containing the untrusted code and build the solution to obtain the protected binaries.

b: GNU MAKE

SGX applications can be compiled on Linux OS machines using GNU make. Libraries and tools included in the Intel SGX SDK are needed to compile the project. The framework includes a user-configurable GNU makefile to guide the user in compiling the application. The user should edit the makefile to specify the desired application execution mode, and, if needed, the RSA keypair that must be used to sign the enclave. The makefile includes the calls to the Intel SGX SDK tools used to compile and sign the enclave code.

A. LIMITATIONS

Following, we describe some limitations of our PoC framework, which we plan to address in its future iterations.

a: EXTERNAL FUNCTION SUPPORT

Functions within an enclave can only call other functions in the enclave or functions in SGX libraries. Calls to external libraries are prohibited unless redefined in SGX libraries. The conversion process checks for this and stops execution if any external calls are made. Data passed to external functions is not evaluated for sensitivity.

b: STATIC VARIABLES

Global variables cannot be shared between the enclave and untrusted code in SGX. Indeed, data handled in the enclave must be protected and not readable from the outside: sharing data between the two environments would make enclave data protection pointless. However, it could be possible, in theory, to automatically adapt the ECall/OCall calls to pass the global variable value when needed.

c: INTEGER RETURN TYPE

Enclave entry and exit functions must return an integer value because they are encapsulated in wrapper functions that handle related errors. The community prefers passing values, including return values, as pointer parameters to functions for more secure execution. More information is available in Section 2.2.2.

d: PARAMETERS

Enclave functions should use simple or user-defined types for arguments. This makes it easier to find the needed structures. Currently, only basic types like int, float, char, and void* are allowed for ECall and OCall function arguments. More types can be added by specifying them in a configuration file to import the correct SGX library. See Appendix A.1.6 for all allowed types.

e: ERRORS

The framework automatically generates protected code and adds error-handling code to ensure consistency. Errors related to hardware compatibility, enclave allocation/destruction, and call errors are handled by printing a console message and calling the exit function. Errors within the protected code are handled with the abort function. Special functions can be used to handle errors, such as retrying a call or writing a log to a file. The framework could be improved by adding an option to reallocate the enclave and retry the call in case of suspension errors.

f: ENCLAVE ALLOCATION

Enclave allocation is done once during the first call. Allowing users to specify the creation time can help anticipate this operation and stop the program when incompatible with SGX. The framework doesn't directly call code to destroy the enclave, so an option to specify when to destroy it would release resources when no longer needed. Otherwise, enclave resources are only released at the end of the program.

IV. EXPERIMENTAL RESULTS

This section discusses the tests we conducted to evaluate the performance of our PoC implementation. Table 1 reports the computer's specifications we used to perform the tests.

We computed the time needed to convert an application and its execution overhead using the SGX technology.

TABLE 1. Specifications of our test platform.

model	Asus UX510UXK
CPU	Intel Core i7-7500U @ 2.70 GHz
RAM	16 GiB DDR4
OS	Microsoft Windows 10 Pro

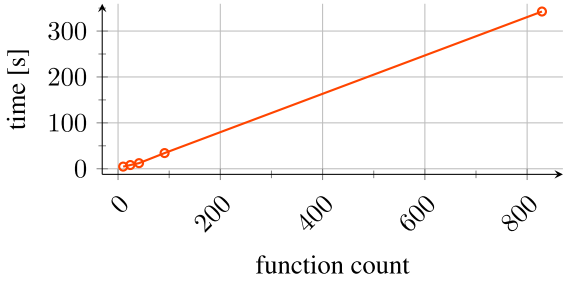


FIGURE 4. Conversion times.

A. CONVERSION TIMES

To test our framework's speed, we developed an ad-hoc Python script to generate fully working C applications with a fixed number of functions. Figure 4 reports various C applications' conversion times (in seconds) in multiple applications with an increasing function count.

The analysis and final code generation are quite efficient and linear in the number of functions. Faster performance can be achieved by disabling the semantic analysis by Frama-c and using only its syntactical version at the expense of a potentially less accurate code examination.

B. EXECUTION OVERHEAD

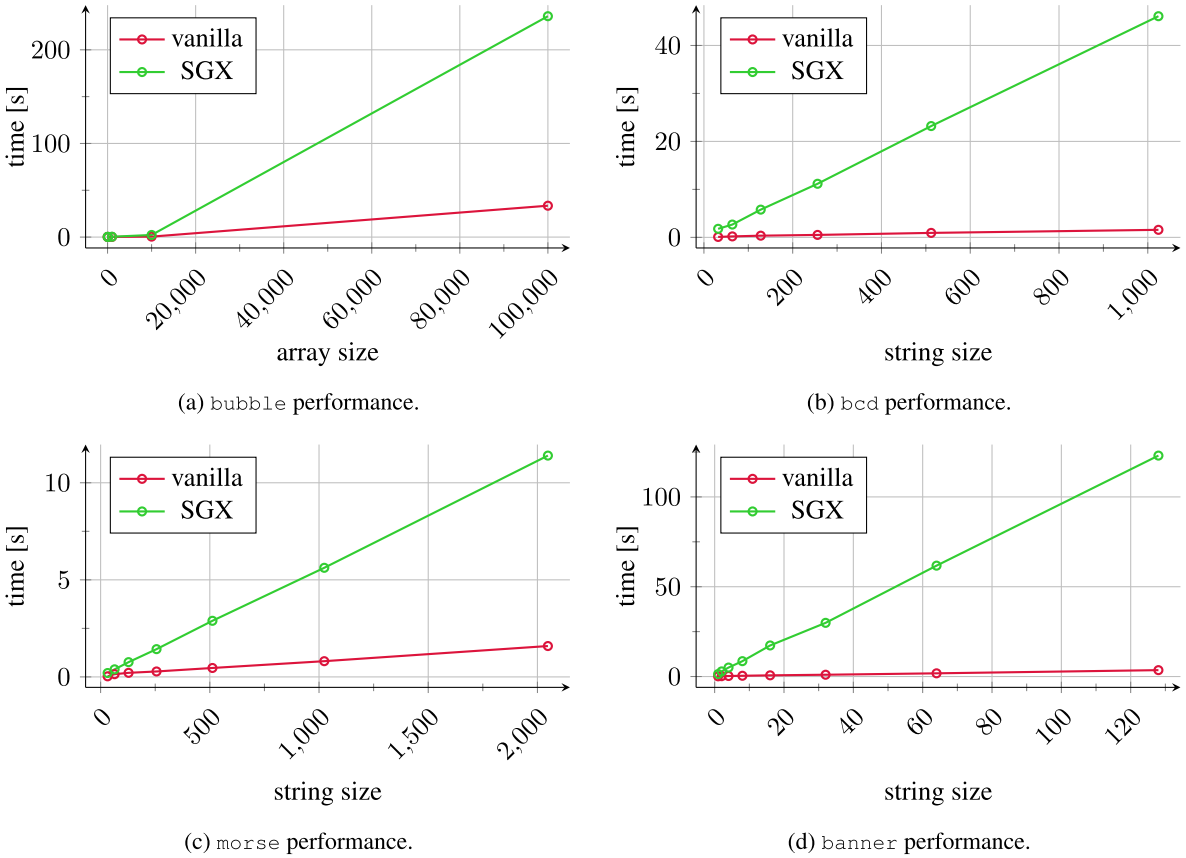
The enhanced security of an SGX-enabled application comes at a price: increased overhead. This behavior is shown in Figure 5, where we show the original and SGX version execution times of four different applications varying the input sizes. We picked four different programs to conduct our tests:

- bubble is a trivial bubble sort implementation in C working on integer arrays;
- bcd is part of Debian's bsdgames package [10] and converts a string in an ASCII art representation of a punched card;
- morse, also part of Debian's bsdgames package, convert a string into its Morse code representation;
- banner[11] reads a string and prints it as a large ASCII art banner on the screen.

As expected, the usage of enclaves introduces a significant slowdown in the applications. The bigger the data used (i.e. the array and string sizes), the slower the application is since the MEE will have to encrypt/decrypt more bits.

It is also interesting to note that the bubble sort algorithm has a quadratic complexity, while all the other applications have a linear complexity (depending on the input size). Note

[10] https://packages.debian.org/buster/bsdgames
[11] https://packages.debian.org/buster/sysvbanner



(a) `bubble` performance.

(b) `bcd` performance.

(c) `morse` performance.

(d) `banner` performance.

FIGURE 5. Execution times.

that introducing SGX does not alter this behavior; it just makes the overall execution times significantly slower.

V. RELATED WORKS

Although SGX has been around for almost ten years, the scientific community has seen very few tools for (semi-)automatically deploying enclaves. This section will briefly discuss some of the most interesting and complete frameworks to enable SGX in existing applications.

Glamdring [4] is a framework that aims to protect C language applications using Intel SGX. First, the developer specifies which data is sensitive by annotating the variables in the source code. Then, a static code analysis phase is executed (via a technique known as backward slicing) to find the lines of code and functions affected by the annotated variables. Finally, Glamdring moves the affected functions into an enclave to protect them and, indirectly, the chosen variables. In addition, Glamdring also adds some new code by implementing cryptographic operations when the data is received and returned by the enclave. This approach protects the enclave against attacks such as call ordering, Iago, and replay (see Section II-A). The Glamdring project and our framework are designed to automate the modifications necessary to protect source code with Intel SGX technology; however, they differ in a critical aspect. Glamdring uses a data-driven approach (the developer specifies the variables to protect by moving them inside an enclave), while our approach is function-driven (the developer specifies the code regions to move inside an enclave). According to the application to port in the SGX world, one approach or the other can be the best choice. Glamdring is more appropriate to safeguard sensitive data, while our framework is better suited to protect critical algorithms.

Enarx[12] is an open-source framework that enables an application to leverage a variety of TEEs without modifying the source code. It supports Intel SGX for securing functions via enclaves and AMD SEV (Secure Encrypted Virtualization) for safeguarding entire virtual machines. When a developer wants to create a TEE, he must convert its code into WebAssembly (via an automatic tool) and invoke the Enarx framework. The WebAssembly format makes the TEEs independent of the CPU architecture, although the significant drawback is increased execution overhead. In addition to confidentiality, Enarx also allows the integrity of the TEEs to be checked via a remote attestation procedure.

Occlum [13][13] is an open-source Library Operating System that lets legacy applications leverage the security of

[12]https://enarx.dev/
[13]https://occlum.io/

SGX's enclaves. To convert a traditional application with Occlum, the developer has to feed the application source code to the Occlum toolchain, based on the LLVM compiler, which generates an ELF binary compatible with the Occlum Library Operating System. This differs from our approach, which modifies the source code to make it compatible with the standard Intel SGX toolchain. Furthermore, using Occlum, the whole target application must be executed inside the enclave. Instead, using our approach, the software developer may freely choose the functions that must executed inside the enclave (e.g. security-sensitive functions), potentially limiting computational overhead.

SAPPX [14] is a method for automatically partitioning Commercial Off-the-Shelf binaries to ensure that sensitive operations are protected within an enclave while maintaining program semantics. Differently from our source-to-source approach, SAPPX takes as input the target application binary, performs binary analysis to determine the appropriate execution location for different parts of the application (i.e., in user space or within an enclave), and outputs the user binary and the enclave library. The method encompasses dependency analysis, taint analysis, and boundary adjustment to optimize the partitioning process. Additionally, it carefully manages functions incompatible with SGX or those incurring high overhead. The authors tested their approach on OpenSSL, the thttpd web server, and the SPECCPU 2017 benchmarking suite, introducing an average performance overhead of 19%.

Montsalvat [15] is a practical approach for partitioning Java applications to enable secure execution within Intel SGX enclaves. Through the use of annotations, Montsalvat divides Java applications into trusted and untrusted components, facilitating secure and efficient execution. It utilizes GraalVM native-image for ahead-of-time compilation to ensure the inclusion of only necessary methods in the secure enclave, effectively reducing the Trusted Computing Base (TCB). Montsalvat addresses challenges such as inter-object communication and synchronized garbage collection by implementing an RMI-like mechanism and a dedicated Garbage Collector helper. This approach ensures consistent object management across the trusted and untrusted runtimes. The implementation has been tested on LinkedIn's PalDB [14] and GraphChi [16], boosting the performance respectively up to 6.6 and 2.2 times, compared to running such applications entirely within an enclave.

The study by Dreissig et al. [17] delves into compiler-level tools for partitioning programs into trusted and untrusted enclaves using Rust. It introduces Cadote, a solution that automates SGX enclave generation from Rust programs. Developers can mark functions as trusted, and these functions are then executed within an enclave. Cadote leverages LLVM compiler passes and Rust's memory safety to securely transfer function parameters across enclave boundaries. It also supports SGX-compatible standard and third-party libraries and enables trusted functions to utilize most of the Rust standard library. The authors evaluated this approach using a set of microbenchmarks, with a particular focus on assessing the effect of alternating ECalls and OCalls. They utilized a recursive implementation of Euclid's algorithm to calculate the greatest common divisor as a test case, with the worst-case scenario tests reporting a performance overhead factor of 1700 times.

[14]https://github.com/linkedin/PalDB

VI. CONCLUSION

This paper presents a new automatic protection solution for Intel SGX technology. The technology allows for secure code execution using enclaves, ensuring data and algorithm confidentiality. Enclaves prevent other processes from reading or modifying the code stored within them. However, the developer wanting to create an SGX-enable application from a traditional one has to be familiar with this technology and potentially rewrite a significant portion of the code, define the interface functions with the enclaves, and appropriately handle the errors.

Our approach alleviates the developer's burden by automatizing most code conversion procedures. The developer has to appropriately annotate the code, marking the functions that need to be moved into an enclave, and our framework will do the rest. It will analyze the code statically to find the function boundaries and their call graph and generate the correct code and (encrypted) binaries to launch a protected snippet into an SGX enclave. Our experimental results show that the conversion time is highly dependent on the call graph depth, and although it can be in the order of minutes, it is a one-time operation only.

In the future, we plan to add support to our framework for the local and remote attestation procedures. These techniques allow the addition of strong security integrity checks to enclaves, further strengthening the safety of the SGX's TEEs. Furthermore, we also intend to integrate this work into the ESP (Expert system for Software Protection). The ESP [18] is a system leveraging AI and formal models [19] to automatically protect or suggest what software protections should be placed on the assets to safeguard.

REFERENCES

[1] V. Costan and S. Devadas, "Intel SGX explained," *IACR Cryptol. ePrint Arch.*, vol. 86, pp. 1–118, Jan. 2016. [Online]. Available: https://eprint.iacr.org/2016/086

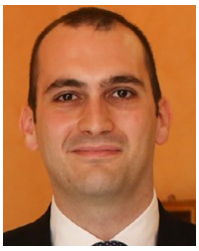
[2] S. Gueron, "A memory encryption engine suitable for general purpose processors," *IACR Cryptol. ePrint Arch.*, vol. 204, pp. 1–14, Feb. 2016. [Online]. Available: https://eprint.iacr.org/2016/204

[3] Intel Corporation. (Apr. 2020). *Intel Software Guard Extensions Developer Reference for Linux* OS*. [Online]. Available: https://download.01.org/intel-sgx/sgx-linux/2.9.1/docs/Intel_SGX_Developer_Reference_Linux_2.9.1_Open_Source.pdf

[4] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P. L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic application partitioning for Intel SGX," in *Proc. USENIX Annu. Tech. Conf. (ATC)*, Jul. 2017, pp. 285–298. [Online]. Available: https://www.usenix.org/system/files/conference/atc17/atc17-lind.pdf

[5] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *SIGARCH Comput. Archit. News*, vol. 41, pp. 253–264, Mar. 2013.

[6] M. Schwarz, S. Weiser, D. Gruss, C. Maurice, and S. Mangard, "Malware guard extension: Using SGX to conceal cache attacks," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Heidelberg, Germany: Springer, 2017, pp. 3–24.
[7] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai, "SgxPectre: Stealing Intel secrets from SGX enclaves via speculative execution," in *Proc. IEEE Eur. Symp. Secur. Privacy (Euro&SP)*, Jun. 2019, pp. 142–157.
[8] Intel Corporation. (Feb. 2018). *Intel Software Guard Extensions (SGX) SW Development Guidance for Potential Bounds Check Bypass (CVE-2017-5753) Side Channel Exploits*. [Online]. Available: https://software.intel.com/sites/default/files/180204_SGX_SDK_Developer_Guidance_v1.0.pdf
[9] M. Schwarz, S. Weiser, and D. Gruss, "Practical enclave malware with Intel SGX," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment (DIMVA)*. Heidelberg, Germany: Springer, 2019, pp. 177–196.
[10] J. Lee, J. Jang, Y. Jang, N. Kwak, Y. Choi, C. Choi, T. Kim, M. Peinado, and B. B. Kang, "Hacking in darkness: Return-oriented programming against secure enclaves," in *Proc. USENIX Secur., USENIX Secur. Symp.*, Aug. 2017, pp. 523–539.
[11] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-bomb: Locking down the processor via rowhammer attack," in *Proc. 2nd Workshop Syst. Softw. Trusted Execution (SysTEX)*, Oct. 2017, pp. 1–6.
[12] 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. Secur. Privacy (SP)*, May 2020, pp. 1466–1482.
[13] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of Intel SGX," in *Proc. 25th Int. Conf. Archit. Support Program. Lang. Operating Syst. (APLOS)*, Mar. 2020, pp. 955–970.
[14] J. Huang, H. Han, F. Xu, and B. Chen, "SAPPX: Securing COTS binaries with automatic program partitioning for Intel SGX," in *Proc. IEEE 34th Int. Symp. Softw. Rel. Eng. (ISSRE)*, Oct. 2023, pp. 148–159.
[15] P. Yuhala, J. Ménétrey, P. Felber, V. Schiavoni, A. Tchana, G. Thomas, H. Guiroux, and J.-P. Lozi, "Montsalvat: Intel SGX shielding for GraalVM native images," in *Proc. 22nd Int. Middleware Conf.*, New York, NY, USA, Dec. 2021, pp. 352–364.
[16] A. Kyrola, G. Blelloch, and C. Guestrin, "GraphChi: Large-scale graph computation on just a PC," in *Proc. 10th USENIX Symp. Operating Syst. Design Implement. (OSDI)*, 2012, pp. 31–46.
[17] F. Dreissig, J. Röckl, and T. Müller, "Compiler-aided development of trusted enclaves with rust," in *Proc. 17th Int. Conf. Availability, Rel. Secur. (ARES)*, New York, NY, USA, Aug. 2022, pp. 1–10.
[18] C. Basile, B. De Sutter, D. Canavese, L. Regano, and B. Coppens, "Design, implementation, and automation of a risk management approach for man-at-the-end software protection," *Comput. Secur.*, vol. 132, Sep. 2023, Art. no. 103321.
[19] C. Basile, D. Canavese, L. Regano, P. Falcarin, and B. De Sutter, "A meta-model for software protections and reverse engineering attacks," *J. Syst. Softw.*, vol. 150, pp. 3–21, Apr. 2019.



LEONARDO REGANO received the M.Sc. and Ph.D. degrees in computer engineering from the Politecnico di Torino, in 2015 and 2019, respectively. He was a Research Assistant with the Politecnico di Torino, for eight years. He is currently an Assistant Professor with the Dipartimento di Ingegneria Elettrica ed Elettronica, Università degli Studi di Cagliari, Italy. His current research interests include software security; artificial intelligence; and machine learning applications to cybersecurity, security policies analysis, and software protection techniques assessment.

DANIELE CANAVESE received the M.Sc. and Ph.D. degrees in computer engineering from the Politecnico di Torino, in 2010 and 2016, respectively. He was a Research Assistant with the Politecnico di Torino, for more than ten years. He is currently a Postdoctoral Researcher with the Centre National de la Recherche Scientifique (CNRS), Institut de Recherche en Informatique de Toulouse (IRIT), Toulouse, France. His current research interests include using artificial intelligence and machine learning techniques for security management, software protection systems, public-key cryptography, and models for network and traffic analysis.

Open Access funding provided by 'Università degli Studi di Cagliari' within the CRUI CARE Agreement