DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel(R) SGX Enclaves

Rodolfo Silva*, Pedro Barbosa† and Andrey Brito‡
Universidade Federal de Campina Grande
*rodolfomarinho@copin.ufcg.edu.br
†pedroyossis@copin.ufcg.edu.br
‡andrey@computacao.ufcg.edu.br

***Abstract*—Intel(R) Software Guard eXtensions (SGX) is a hardware-based technology for ensuring security of sensitive data from disclosure or modification that allows user-level applications to allocate protected areas of memory called enclaves. Such memory areas are cryptographically protected even from code running with higher privilege levels. This memory protection can be used to develop secure and dependable applications and systems. On the other hand, this technology has some limitations: (i) the code of an enclave is completely visible, (ii) libraries used by the code must be statically linked against it and (iii) the protected memory size is limited, demanding page swapping to be done when the limit is exceeded.**

In this paper we present DynSGX, a privacy preserving tool that allows users and developers to dynamically load and unload code to be executed inside SGX enclaves. Moreover, we present a series of experiments that have been carried out in order to assess how applications dynamically loaded by the proposed tool perform in comparison to statically linked applications that might demand page swapping and disregards privacy of the code running in the enclave.

1. INTRODUCTION

We live in an ever more connected world, where people are constantly uploading personal data to environments that cannot be controlled by them. Sometimes this data is meant to become public, but in most cases they are required to be kept private and/or secure. In that sense, developers need to find ways to protect their users' data from theft or improper modification at all costs.

Besides that, developers and companies often host their applications in public cloud environments in order to increase their availability and scalability, or even to lower costs spent on physical infrastructure [1]. In such environments, multiple applications from different owners may reside in the same physical server, making it possible for rogue cloud users to exploit existing security breaches in order to obtain secret data from other users.

Security of data based solely on software, in most cases, falls short due to vulnerabilities existing in the application developed or in libraries/resources used by the application. Considering that, higher security levels have been demanded.

Over the course of the past decade, several efforts were made to define and implement a security enabler called Trusted Execution Environment (TEE) [2], [3] [4], [5]. To put in a few words, TEE is a secure area of the main processor that guarantees that code and data stored inside it will not be modified or disclosed without permission. It was first specified by GlobalPlatform [6], and since then has attracted attention from several companies that have provided their own implementations [7], [8], [9], [10], [11].

One of these implementations is an Intel technology known as Intel SGX [10]. It is available on off-the-shelf processors of the 6th generation Intel Core family based on the Skylake microarchitecture or newer and on Xeon v5 processors or newer, and already has a wide variety of research publications related to its applicability on real world case scenarios [12]. This technology allows user-level code to allocate and to be executed inside protected areas of memory called enclaves. It also provides means for users/applications to verify if a given application is running inside an SGX enclave, and even if the code of the application is indeed the one expected to be running there; a process called Remote Attestation [13]. Despite these advantages, Intel SGX has some limitations regarding memory usage and code privacy that may raise some concerns when executing them in cloud environments. Regarding memory, only a very limited area can be protected by the processor. Currently, this limit is of up to 128MB. When this limit is reached, data need to be swapped to/from the unprotected DRAM, generating an overhead. Concerning code privacy, SGX programming model does not prevent code disclosure, since all enclave code is loaded unencrypted into memory. These limitations will be better discussed in the next sections of this paper.

In order to overcome these limitations, we propose DynSGX[1], an open source privacy preserving toolset for dynamically loading functions into and unloading functions from SGX enclaves. Our toolset enables developers to better manage the scarce memory resources they have available to use with Intel SGX, as well as to keep their applications private even when being used on cloud environments.

1. https://github.com/rodolfoams/DynSGX

The rest of the paper is divided as follows: in Section 2 we get into more details about Intel SGX, its limitations and possible vulnerabilities. We continue by introducing DynSGX in Section 3. Further on, in Section 4, we present an evaluation of our proposed solution. Finally, in Section 6 we draw our conclusions, and describe some possible future work to further improve our toolset.

2. INTEL SGX

Intel Software Guard eXtensions (SGX) can be described as a new set of instructions and changes in memory access mechanisms recently added to the Intel Architecture. It is a hardware-based technology that, like other TEE implementations, is used for ensuring security of sensitive data from disclosure or modification [10]. SGX works as an "inverse sandbox" mechanism, where code can be sealed inside an enclave (i.e., private regions of memory). Inside the enclave, code, data and stack are protected by hardware enforced access control policies which prevent attacks against the enclave's content even when these originate from privileged software such as virtual machine monitors, BIOS, or operating systems.

The protection of the enclaves is ensured by the creation of a reserved area of memory called Processor Reserved Memory (PRM), which is reserved by BIOS at boot time. Inside the PRM lies another region of memory known as Enclave Page Cache (EPC), also configured by BIOS at boot time, where the enclaves' pages actually reside. Access to these pages is controlled by the processor and protected by mechanisms such as the Memory Encryption Engine (MEE).

SGX also allows users to perform a Remote Attestation process (i.e. cryptographically verify if an application is running inside an SGX enclave, and if it is indeed the application that was expected). The generation of the hardware-based material used in the Remote Attestation process is also enabled by the SGX instructions [13]. The Remote Attestation process can be used to establish a secure communication channel between an application enclave and a user, by sharing a symmetric key via an Elliptic Curve Diffie-Hellman (ECDH) protocol. Users can than use this key to securely exchange messages with an application running inside an SGX enclave.

Possible applications of Intel SGX have been discussed in [12], where examples of applications that make use of the capabilities of Intel SGX were presented, as well as an application architecture including an application split between components requiring security protection which should run within enclaves, and components that do not require protection and can therefore be executed outside enclaves. In [14] SGX is used for securely communicating and processing fine-grained smart metering data in a cloud environment.

2.1. SGX Components

The Intel SGX solution comprises four main components: (i) the set of instructions in the processor, (ii) the operating system driver, (iii) the Software Development Kit (SDK), and (iv) the Platform Software (PSW).

2.1.1. SGX Instructions Set. Intel SGX instructions set is available on off-the-shelf processors based on the Skylake microarchitecture or newer, starting from the 6th Generation Intel Core family and on Xeon v5 processors.

It consists of 17 new instructions that can be classified into the following functions [10]:

- **Enclave build/teardown**: Used to allocate protected memory for the enclave, load values into the protected memory, measure the values loaded into the enclave's protected memory, and tear down the enclave after the application has finished.
- **Enclave entry/exit**: Used to enter and exit the enclave. An enclave can be entered and exited explicitly. It may also be exited asynchronously due to interrupts or exceptions. In the case of asynchronous exits, the hardware will save all secrets inside the enclave, scrub secrets from registers, and return to external program flow. It then resumes where it left off execution.
- **Enclave security operations**: Allow an enclave to prove to an external party that the enclave was built on hardware which supports the SGX instruction set.
- **Paging instructions**: Allow system software to securely move enclave pages to and from unprotected memory.
- **Debug instructions**: Allow developers to use familiar debugging techniques inside special debug enclaves. A debug enclave can be single stepped and examined. A debug enclave cannot share data with a production enclave. This protects enclave developers if a debug enclave should escape the development environment.

2.1.2. SGX Driver. The SGX *drivers* enable OSs and other softwares to access the SGX hardware. Intel SGX drivers are available both for Windows (via Intel Management Engine) [15] and for Linux* [16] platforms. As any other device driver, they serve as an abstraction to allow developers to write higher-level code for using the device capabilities.

2.1.3. SGX SDK. The SGX Software Development Kit (SDK) is a collection of APIs, sample source code, libraries and tools that enable software developers to write and debug SGX applications in C/C++. Intel SGX SDK is available both for Windows [17], and for Linux* [18] platforms.

2.1.4. SGX PSW. The SGX *Platform Software* (PSW) is a collection of special SGX enclaves, and an Intel SGX Application Enclave Services Manager (AESM), provided along with the SGX SDK. These special enclaves and AESM are used when loading enclaves, retrieving cryptographic keys, and evaluating the contents of an enclave.

2.2. SGX Memory Management

Some features in Intel SGX, specially its memory management model, make it very useful for providing data secu-

rity. The main aspects of the memory model are discussed below:

- **Enclave Page Cache**: The Enclave Page Cache (EPC) is protected memory used to store enclave pages and SGX structures. The EPC is divided into 4KB chunks called EPC pages. EPC pages can either be valid or invalid. A valid EPC page contains either an enclave page or an SGX structure.
  Each enclave instance has an enclave control structure, SECS. Every valid enclave page in the EPC belongs to exactly one enclave instance. System software is required to map enclave virtual addresses to a valid EPC page.
- **Memory Encryption Engine**: Memory Encryption Engine (MEE) is a hardware unit that encrypts and protects the integrity of selected traffic between the processor package and the main memory (DRAM). The overall memory region that an MEE operates on is called an MEE Region. Depending on implementation, the PRM is covered by one or more MEE regions.
  Intel SGX guarantees that all the data that leaves the CPU and is stored in DRAM is first encrypted using the MEE. Thus, even attackers with physical access to DRAM will not be able to retrieve secret data protected by SGX enclaves from it.
- **Memory Access Semantics**: CPU memory protection mechanisms physically block access to PRM from all external agents, by treating such accesses as references to non-existent memory. To access a page inside an enclave using MOV and other memory related instructions, the hardware checks the following:
  - Logical processor is executing in "enclave mode".
  - Page belongs to enclave that the logical processor is executing.
  - Page accessed using the correct virtual address.

  If any of these checks fails, the page access is treated as reference to nonexistent memory, or by signaling a fault. This guarantees that even a process with higher privilege levels will not be able to access enclave's memory.

2.3. SGX Limitations

Before designing a new secure application using SGX, there are some limitations that need to be kept in mind by developers, in order to avoid having security flaws or big overheads due to memory swapping. The main limitations are the following:

- **Code privacy**: The entire enclave code is loaded into memory unencrypted. The code is protected from modification, but it does not allow developers to maintain their code private. There are many scenarios where developers want to keep their code private, so this poses as a serious privacy drawback, once attackers could reverse engineer the enclave code by disassembling it, or even generating a pseudocode very similar to the original one, through tools like IDA [19].
- **Static linking**: Applications that use third-party libraries need to statically link these libraries against their enclaves. This may result in generating a big footprint for enclaves and, consequently, waste space in memory – a scarce resource for SGX.
- **Memory size**: When starting a machine, BIOS needs to reserve a portion of memory to the processor (PRM). Also, the entire EPC must reside inside the PRM. In the current version of SGX, this portion of memory is limited to only 128MB in size per machine. If the space needed is more than the space available, a big overhead in processing time is added, due to the need to encrypt the data before swapping from EPC to DRAM and decrypt the data after swapping from DRAM to EPC. This memory access overhead is better described in section 4.1.

2.4. Vulnerabilities

SGX protects against many types of attacks, even from privileged users and softwares. However, a side-channel adversary is able to gather statistics from the CPU regarding execution and may be able to use them to deduce characteristics of the software being executed (side-channel analysis). Examples of analyses may be: power statistics; performance statistics including platform cache misses; branch statistics via timing; and information on pages accessed via page tables. It is well documented that SGX does not defend against side-channel adversaries [20], [21].

SGX components are complex and unlikely to be bug-free, like any other software. There are drivers, libraries, dependencies, and complex instructions available for developers. Moreover, enclave developers may make mistakes and even the so called protected areas may contain common vulnerabilities like stack buffer overflows and uncontrolled format strings. This problem is boosted by the limited portion of memory because it affects the effectiveness of the address space layout randomization (ASLR).

ASLR is a security technique involved in protection from buffer overflow attacks. In order to prevent an attacker from reliably jumping to, for example, a particular exploited function in memory, ASLR randomly arranges the address space positions of key data areas of a process, including the base of the executable and the positions of the stack, heap and libraries. With SGX, because the memory space for an enclave is quite small, a simple brute forcing mechanism can easily identify the correct address. In our experiments, we observed that for different executions, an element has its addresses changed by only two bytes, meaning that the randomization is for approximately only 65536 possibilities. This is very small, considering that an attacker can, for example, increase the attack success probability through the injection of NOP slides before a malicious code.

3. DYNSGX: DYNAMICALLY LOADING FUNCTIONS INTO SGX ENCLAVES WITH PRIVACY GUARANTEES

DynSGX aims at providing an alternative to the conventional SGX programming model, where the entire code of the application to be run inside an enclave needs to be included in or linked against the enclave at build time. Doing so could cause the enclave size to rapidly exceed the available memory of the EPC and, consequently, cause a big overhead in processing time due to the need of page swap. The conventional SGX programming model would also cause the application code that runs in the enclave to be completely visible after loading the enclave into memory.

Instead, DynSGX toolset allows users to start their secure application with a very small enclave, and dynamically load functions into and remove functions from the enclave at runtime as needed. This allows users and developers to better manage the amount of memory that is occupied by the secure application code. Also, by using the Remote Attestation process to establish a secure communication channel and using SGX capabilities to protect memory, DynSGX also guarantees the complete privacy of the code running inside the enclave.

3.1. DynSGX TCB

In DynSGX we limit the trusted computing base (TCB) to the Intel SGX SDK/PSW plus a set of only six SGX-compatible C/C++ functions, yielding an enclave with an initial size of only 1.4MB. Such functions are used for (i) performing the SGX Remote Attestation process, (ii) loading functions into the enclave, (iii) running loaded functions inside the enclave and (iv) unloading functions from the enclave.

For the Remote Attestation process, only the *enclave_ra_init* function needs to be placed inside the enclave. This function internally calls the *sgx_ra_init* function from the SGX SDK, which starts the Remote Attestation process. To load functions into the enclave, two functions are provided and placed inside the enclave: *enclave_get_fas*, which is responsible for providing a list of functions that is already registered inside the enclave, and *enclave_register_function*, which is responsible for loading new functions into the enclave. The *enclave_execute_function* can be used to securely execute the functions that were loaded into the enclave. Finally, DynSGX provides *enclave_unregister_function* and *enclave_clear_functions* that developers/users can use to unload functions from the enclave.

3.2. DynSGX Programming Model

DynSGX, as many other cloud-based tools, follows the client-server model. The DynSGX enclave should run in the server side, while developers/users interact with it from the client side.

DynSGX does not require developers to know how to develop SGX applications. Instead, developers can write their functions as they were writing a regular C function. After writing their function, the tool compiles the $.c$ file that contains the function and then retrieve the bytes that compose the compiled function (this step can be done by using a tool called *bytes_extractor* provided as part of DynSGX). The bytes of the function can then be sent to be loaded into the enclave, and later on be executed inside the enclave. In the current state of DynSGX, developers are required to write only one function per $.c$ file. This requirement is due to the way the *bytes_extractor* tool is implemented. It looks for the first occurrence of the function name inside the compiled file in order to extract its bytes. If a case where a function is called before it has been defined happened, the tool would not work properly. Another important note is that the $.c$ files are compiled with the $-c$ option in order to not need a main method in the file, and with the $-fPIC$ flag so that the compiled code is position independent.

Let us consider an example where a developer/user wants to securely process a function to sum two integer numbers. This *sum_function* is depicted in Listing 1.

Listing 1. Sample C function for summing two integer numbers.

```
1  int sum(int a, int b) {
2    return a + b;
3  }
```

Listing 2. Corresponding assembly for the sum function.

```
1  push %rbp
2  mov  %rsp,%rbp
3  mov  %edi,-0x4(%rbp)
4  mov  %esi,-0x8(%rbp)
5  mov  -0x4(%rbp),%edx
6  mov  -0x8(%rbp),%eax
7  add  %edx,%eax
8  pop  %rbp
9  retq
```

After compiling the file containing this function, the *bytes_extractor* tool extracts the function bytes from the assembled code (Listing 2), resulting in the following *hexstring*: `\x55\x48\x89\xe5\x89\x7d\xfc\x89\x75\xf8\x8b\x55\xfc\x8b\x45\xf8\x01\xd0\x5d\xc3`. This *hexstring* can be loaded into the DynSGX enclave, where it will be registered and stored in the heap, a protected memory area. When the developer/user needs to execute this function, it will be casted to a regular function. The developer/user can unload their function from the enclave when it is no longer needed, in order to free memory.

3.3. Application Lifecycle

DynSGX enclaves are started with only a limited number of essential functions inside it. After the enclave is loaded, developers/users can contact it to dynamically provision their functions to the enclave. After sending the functions, developers can execute them inside the DynSGX enclave, and after even unload them afterwards. The steps needed to complete this processes are illustrated in Figure 1 and described as follows:

1) The client communicates with the DynSGX enclave and performs the Remote Attestation process to verify the identity of the enclave and establish a secure communication channel between both.
2) The developer/user writes the code of their function.

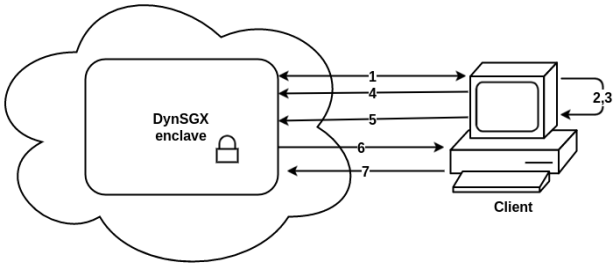


Figure 1. DynSGX Software Lifecycle.

3) The client compiles and extracts the bytes of the compiled function.
4) The client uses the secure communication channel to provide the enclave with the developer/user function bytes.
5) The developer/user uses the client and the secure communication channel to send any parameters needed by the function and requests the enclave to execute it.
6) DynSGX enclave uses the secure communication channel to send the result of the processing back to the developer/user through the client.
7) Developer/user uses the client to request the DynSGX enclave to unload the function to free memory space.

3.4. Distributed Linking

For a self-contained function (i.e., does not use external elements), compiling and sending the bytes of the assembled code are enough. However, if the function uses external elements, it is necessary a distributed mechanism to map these elements into their corresponding addresses at the enclave side. Examples of external elements may be:

- Library functions, such as from the tlibc (trusted libc, from the SGX SDK);
- Other functions previously defined by the user;
- Global variables.

We implemented a mechanism that allows the enclave to send to the client a JSON like the one presented in Listing 3, after the Remote Attestation process is completed.

Listing 3. JSON mapping the external elements that can be used by a function into their corresponding addresses.

```
1  {
2    "snprintf":  "(*(int(*)(0x7f1e438176f0)))",
3    "vsnprintf": "(*(int(*)(0x7f1e4381d770)))",
4    "strcmp":    "(*(int(*)(0x7f1e438179a0)))",
5    ...
6  }
```

Therefore, at the client side, for a function like the one presented in Listing 4, the `strcmp` at line 3 is replaced by `(*(int(*)(0x7f1e438179a0)))`. This works because it is the same of casting and calling a function pointer. The compiler does not know what will be in this address, but in runtime, the `strcmp` function will be at this address within the enclave.

Listing 4. Example of a function that uses an external element (the strcmp function).

```
1  int check_password(char* input) {
2    char password[] = "topsecret1234";
3    return !strcmp(input, password);
4  }
```

3.5. Requirements

To run the DynSGX enclave and load functions into in runtime, three requirements need to be met:

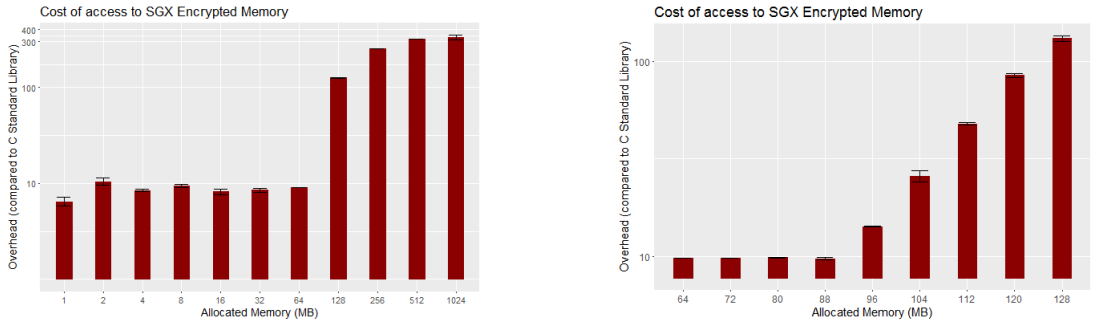
- **SGX-capable hardware**: SGX technology must be available and enabled by BIOS. Such hardware is commercially available since the third quarter of the year 2015.
- **SGX driver**: SGX driver [16] must be installed in order to enable the OS and other softwares to access the SGX hardware.
- **SGX PSW**: SGX PSW [18] is used to launch SGX enclaves, and also to generate data necessary for the Remote Attestation process. DynSGX requires a small modification in the regular PSW. This modification regards the option to make the program heap executable and is done by applying a patch to the SGX PSW code [22].

4. EVALUATION

As we have seen in Section 2, three main limitations of SGX may raise concerns for using it in cloud environments. Our solution enables developers to overcome these limitations, and to better manage the memory resource which is scarce to SGX. We consider these limitations here one by one.

- **Code privacy**: The code that the developers/users want to execute inside DynSGX enclave is not included at build time. Instead, the enclave initially contains only the code of the DynSGX solution. User code is only received afterwards and is kept secure and private with the use of SGX capabilities.
- **Static linking**: The code to be executed inside the DynSGX enclave does not need to be statically linked against it. In our solution, developers/users can dynamically load functions into and unload functions from the enclave as they are needed.
- **Memory size**: Although we can not modify the total amount of memory available for the EPC, DynSGX allows users to better manage the amount of memory that is consumed by their applications. This code management is important because of the high overhead added when memory needed exceed the memory available for the EPC. This overhead is better shown in Section 4.1.

For these reasons we consider the presented solution a much more appropriate fit for secure processing data in cloud environments than the regular SGX programming model. The security risks of DynSGX are discussed in Section 4.2.



(a) Memory size range from 1MB to 64MB

(b) Memory size range from 64MB to 128MB

Figure 2. SGX memory access overhead compared to pure C memory access.

4.1. SGX Page Swap Overhead

In order to evaluate how big is the overhead due to page swapping, we conducted a couple of experiments described as follows: we created two applications, one to be run as a regular C program, and other to be run inside a SGX enclave. Both applications worked exactly the same way, (i) by allocating arrays of various sizes, (ii) accessing random positions of the array, and (iii) clearing the array. For fairness of comparison, in our evaluation we only considered the time taken in step (ii) of both applications. We then computed the overhead of the program running inside SGX with the C program. In the first experiment, we varied the sizes of the array from 1MB to 1024MB. The results are depicted in Figure 2a. As we can see, there is a big increase in the overhead when the space needed goes from 64MB to 128MB. To better at what point the overhead starts to increase, we ran the second experiment, varying the array size from 64MB to 128MB as shown in Figure 2b. We can notice that the overhead starts to increase when the size of our array is bigger than 86MB, so we estimate that the memory size that we have available for loading our enclaves is approximately 90MB.

4.2. Vulnerabilities

Apart the side channel attacks and the vulnerabilities that an enclave's code may have, DynSGX introduces a new attack surface: the function sent by the developer/user. To provide its features, DynSGX disables two security protections: stack canaries at the user's functions, and non-executable heap.

Stack canaries are used to detect a stack buffer overflow before execution of malicious code can occur. This method works by placing a small integer in the memory, the value of which is randomly chosen at program start, just before the stack return pointer. Most buffer overflows overwrite memory from lower to higher memory addresses in order to overwrite the return pointer (and thus take control of the process). The canary value must also be overwritten. This value is checked to make sure it has not changed before a routine uses the return pointer on the stack. If the value changes, the function `_stack_chk_fail` from the libc is called. This function and its call, however, is introduced by the compiler. A C programmer is not able to access it, neither in the enclave side to get its address, nor in the client side to replace its calling form. Therefore, the technique described in Section 3.4 should not work for stack canaries.

Non-executable heap is a security protection that helps to prevent certain exploits from succeeding, particularly those that inject and execute code in the heap. With DynSGX, developer/user's functions are dynamically loaded into the heap space. Therefore, it was necessary to disable the non-executable heap protection, more specifically, using the configuration `<HeapExecutable>1</HeapExecutable>` at the Enclave.config.xml file.

Given the limitation of the ASLR mechanism due the small memory space, and the disabling of stack canaries and non-executable heap protections, it is very important to put more effort into developing secure codes. Listing 5 presents an example of a vulnerable code that could be dynamically loaded using DynSGX.

Listing 5. Example of a function vulnerable to stack buffer overflow.

```
1 int check_password(char *input) {
2     char buffer[15];
3     char password[] = "topsecret1234";
4     strncpy(buffer, input, strlen(input));
5     return !strcmp(buffer, password);
6 }
```

This function is vulnerable to stack buffer overflow. If the attacker provides an input longer than 15 bytes, it will overwrite other values that could be in the memory, like the value in the stack base pointer and the return address. Considering that `allow_access` is an enclave function at the address 0x7ffff580160d, an attacker could have access overwriting the return adddress using the following payload: `AAAAAAAAAAAAAAAAAAAAAAAA\x0d\x16\x80\xf5\xff\x7f\x00\x00`.

Another example of vulnerability is uncontrolled format string. A format function is a special kind of C function that takes a variable number of arguments, from which one is the so called format string. If an attacker is able to provide the format string to a C format function in part or as a

whole, a format string vulnerability is present. By doing so, the attacker can read and write to anywhere. Listing 6 presents an example of a code with uncontrolled format string vulnerability.

Listing 6. Example of a function vulnerable to uncontrolled format string.

```
1  int check_password(char *input) {
2      char password[] = "topsecret1234";
3      int result = !strcmp(input, password);
4      if (!result) {
5          char error[30];
6          snprintf(error, 30, input);
7          strncat(error, " is incorrect!", 14);
8          log_msg(error);
9      }
10     return result;
11 }
```

In a normal situation, the effect of this function is the same of the one presented in Listing 4, but with the additional feature of logging an error message. The uncontrolled format string vulnerability is at line 6, and an attacker can provide the following payload: `%10$p %11$p`. Therefore, the function will log the following message: `0x6572636573706f74 0x34333231 is incorrect!`. The numbers in hexadecimal are the password representation in little endian format.

Given the limitations in the security protections, it is very important to write the code carefully and do regular code reviews. The usage of tools that examine source code and report possible vulnerabilities (usually sorted by risk level) may be useful. Flawfinder [23], Cppcheck [24] and CheckConfigMX [25] are examples of static analysis tools that can be integrated with DynSGX for quickly finding and removing at least some potential security problems before sending a code to the enclave.

4.3. Tool Assessment

For assessing the capabilities of DynSGX, two implementations of an application that needs to execute a large number of functions inside an SGX enclave were made, the first one using the conventional SGX programming model, and the second using DynSGX. For comparison purposes, in both implementations the functions and parameters processed inside the enclave were exactly the same.

In the first implementation all the functions to be used are included in the enclave at build time, and called as any regular function inside an SGX enclave. In the second one, the enclave is built with only the functions that compose DynSGX. After that, the user establishes a secure communication channel with the enclave via the Remote Attestation process, and uses it to securely load the functions into the enclave. The functions are then stored in a C++ standard *map*, and called via the *enclave_execute_function* function. The last implementation is very similar to the second one, but uses a C++ standard *unordered_map* to store the loaded functions.

For communication between the server side and the client side of the application, a socket is created with ZeroMQ [26]. ZeroMQ has libraries available to be used in several different programming languages. The ZeroMQ socket was used both for performing the Remote Attestation process (unencrypted messages) and for exchanging the other messages between the client and the server (encrypted messages). All the messages are represented by a JSON - a data-interchange format.

The tests were conducted on computers with one Intel i7-6700 SGX capable processor and 8 GB of RAM available.

4.3.1. Discussion. With DynSGX we were able to successfully load, unload and execute code inside SGX enclaves at runtime. By using the Remote Attestation process we were able to securely transfer the code into the enclave, no longer having to worry about threats to the privacy of the code. Thus, we consider that DynSGX enhances the usability of Intel SGX in real-world cloud environments.

5. RELATED WORK

Intel has recently introduced SGX2, that extends the SGX instruction set, adding support for dynamic memory management from inside SGX enclaves [27]. The new instructions that will be available with SGX2 could be used in a similar way as the modification we made to the SGX PSW in the DynSGX solution in order to be able to execute code stored on the heap.

In [28] SCONE is proposed. SCONE enables developers to compile their C applications into Docker containers protected with the SGX capabilities. Developers can simply compile their C applications with a special compiler that is part of the SCONE toolset, and deploy compiled application in a SCONE client, and it will transparently be protected by SGX capablities.

SecureWorker [29] is a NPM package that allows JavaScript code to be run inside SGX enclaves. It is still under development, and many important features (e.g.: Remote Attestation) are yet to be implemented.

Both SCONE and SecureWorker solutions, like DynSGX, seem to ease the development of SGX solutions. On the other hand, both SCONE and SecureWorker lack the capabilities of dynamically loading code into SGX enclaves and preserving the privacy of such code.

6. CONCLUSION

Intel SGX has been considered to be one of the most promising TEE technologies. However, its limitations and code privacy concerns have drawn its applicability in cloud environments into question. This paper presented DynSGX - a toolset that enables users to dynamically deploy their functions into enclaves at runtime. It also ensures the privacy of such functions, without requiring many modifications to the SGX components provided by Intel. Our evaluation shows that DynSGX successfully loads/unloads functions into/from enclaves dynamically, enabling users to better manage the memory resource, resulting in the privacy the code executed in the enclave being preserved, and a potential

gain in performance when comparing to the conventional SGX programming model.

As future work we intend to extend DynSGX to provide support for other programming languages other than C. Also, we want to provide the developer/user with the current amount of memory that the enclave is using, in order to know precisely at what moment do some functions need to be unloaded from the enclave to free memory.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica *et al.*, "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[2] "Advanced trusted environment: Omtp tr1," accessed: 2017-04-16. [Online]. Available: http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

[3] T. Logic, "Trusted foundations by trusted logic mobility."

[4] "Trustonic secured platform," accessed: 2017-04-16. [Online]. Available: https://www.trustonic.com/wp-content/uploads/2017/02/trustonic-secured-platform-datasheet-feb-2017.pdf

[5] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki, "Flicker: An execution infrastructure for tcb minimization," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 315–328.

[6] G. Platform, "The trusted execution environment: Delivering enhanced security at a lower cost to the mobile market," *White Paper February*, 2011.

[7] A. Inc., "ios security," Apple Inc., Tech. Rep., October 2014. [Online]. Available: https://www.apple.com/br/privacy/docs/iOS_Security_Guide_Oct_2014.pdf

[8] D. Kaplan, J. Powell, and T. Woller, "Amd memory encryption," *White paper, Apr*, 2016.

[9] "Arm trustzone(r)," accessed: 2017-04-16. [Online]. Available: http://www.openvirtualization.org/open-source-arm-trustzone.html

[10] 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 *HASP@ ISCA*, 2013, p. 10.

[11] "Amd memory encryption," accessed: 2017-04-16. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[12] M. Hoekstra, R. Lal, P. Pappachan, V. Phegade, and J. Del Cuvillo, "Using innovative instructions to create trustworthy software solutions." in *HASP@ ISCA*, 2013, p. 11.

[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*, vol. 13, 2013.

[14] L. Silva, R. Silva, J. L. Vivas, and A. Brito, "Security and privacy preserving data aggregation in cloud computing," in *Proceedings of the 32nd Annual ACM Symposium on Applied Computing*. ACM, 2017.

[15] "Intel sgx sdk windows," accessed: 2017-04-16. [Online]. Available: https://downloadcenter.intel.com/pt-br/product/80895/Intel-Software-Guard-Extensions-Intel-SGX-for-Windows-

[16] "Intel sgx sdk windows," accessed: 2017-04-16. [Online]. Available: https://github.com/01org/linux-sgx-driver

[17] "Intel sgx sdk windows," accessed: 2017-04-16. [Online]. Available: https://software.intel.com/en-us/sgx-sdk

[18] "Intel sgx sdk and psw linux," accessed: 2017-04-16. [Online]. Available: https://github.com/01org/linux-sgx

[19] "Ida disassembler and debugger," accessed: 2017-04-17. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[20] Y. Xu, W. Cui, and M. Peinado, "Controlled-channel attacks: Deterministic side channels for untrusted operating systems," in *Proceedings of the 2015 IEEE Symposium on Security and Privacy*, Washington, DC, USA, 2015, pp. 640–656.

[21] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A. Sadeghi, "Software grand exposure: SGX cache attacks are practical," *CoRR*, vol. abs/1702.07521, 2017.

[22] "Sgx psw patch for executable heap," accessed: 2017-04-16. [Online]. Available: https://patch-diff.githubusercontent.com/raw/01org/linux-sgx/pull/63.patch

[23] "Flawfinder," accessed: 2017-04-17. [Online]. Available: https://www.dwheeler.com/flawfinder/

[24] "Cppcheck - a tool for static c/c++ code analysis," accessed: 2017-04-17. [Online]. Available: http://cppcheck.sourceforge.net/

[25] L. Braz, R. Gheyi, M. Mongiovi, M. Ribeiro, F. Medeiros, and L. Teixeira, "A change-centric approach to compile configurable systems with #ifdefs," in *ACM GPCE'16*, 2016, pp. 109–119.

[26] "Zeromq distributed messaging," accessed: 2017-04-16. [Online]. Available: http://zeromq.org/

[27] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas, "Intel® software guard extensions (intel® sgx) support for dynamic memory management inside an enclave," in *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016, p. 10.

[28] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell *et al.*, "Scone: Secure linux containers with intel sgx," in *12th USENIX Symp. Operating Systems Design and Implementation*, 2016.

[29] "Npm secureworker," accessed: 2017-04-16. [Online]. Available: https://www.npmjs.com/package/secureworker