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

Rodolfo Silva, Pedro Barbosa, Andrey Brito
Universidade Federal de Campina Grande
Campina Grande, PB, Brazil
{*rodolfoams, pedroysb, andrey*}*@lsd.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. This technology has some limitations though: (i) the code of an enclave is visible at load time, (ii) libraries used by the code must be statically linked and (iii) the protected memory size is limited, demanding page swapping to be done when this limit is exceeded. We present DynSGX, a privacy preserving tool that allows users and developers to dynamically load and unload code to be executed inside SGX enclaves. Such a technology makes possible that developers use public cloud infrastructures to run applications based on sensitive code and data. Moreover, we present a series of experiments that assess how applications dynamically loaded by the proposed tool perform in comparison to statically linked applications that disregard privacy of the enclave code at load time.**

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 usually falls short due to vulnerabilities existing in the application developed, in libraries/resources used by the application, or even in the operating system/virtualization technology [2]. Considering that, higher security levels have been demanded.

In response to this demand, over the course of the past decade, several efforts were made to define and implement a security enabler called Trusted Execution Environment (TEE) [3], [4], [5]. To put in a few words, TEE is a secure area of the main processor, or a separate, dedicated 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].

One of these implementations is an Intel technology known as Intel SGX [8]. It is available on recent off-the-shelf processors based on the Skylake microarchitecture or newer, and already has a wide variety of research publications related to its applicability on real world scenarios [10]. This technology allows user-level code 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 Intel SGX enclave, and even if the code of the application is indeed the one expected to be running there; a process called Remote Attestation [11].

Despite these advantages, Intel SGX has some limitations regarding memory usage and code privacy that may raise some concerns when using it in cloud environments. Regarding memory, only a very limited area, up to 128MB in size, can be protected by the processor [12]. 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 can be viewed from the executable file stored in the file system. These limitations will be better discussed in the next sections of this paper.

In order to overcome these limitations, we propose DynSGX, a privacy preserving toolset for dynamically loading functions into and unloading functions from Intel 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 loaded on cloud environments.

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. In section 5 we discuss published work related to our toolset. 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 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 [8]. SGX works as an "inverse sandbox" mechanism, where code can be sealed inside an enclave (i.e., private region of memory). Inside the enclave, code and data 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 and 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 (RA) process (i.e. cryptographically verify if the desired application is running inside an SGX enclave). The generation of the hardware-based material used in the RA process is also enabled by the SGX instructions [11]. The RA 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 then use this key to securely exchange messages with an application running inside an enclave.

Possible applications of SGX have been discussed in [10], where examples of applications that make use of the SGX capabilities 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 [13], SGX is used for securely communicating and processing fine-grained smart metering data in a cloud environment.

2.1. SGX Components

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

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

It consists of 17 new instructions that can be classified into the following functions [8]:

- **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 Drivers. 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) [14] and for Linux* [15] platforms. 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, and for Linux* platforms [14].

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 security. 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

As with most technologies, Intel SGX has some limitations that need to be considered by developers. The main ones are the following:

- **Code privacy**: The entire enclave code can be viewed from the executable file stored in the file system. 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 [16]. We performed this attack and reverse-engineered an enclave (*.so* file) containing a recursive Fibonacci function, as depicted in Figure 1.
- **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. In [13] experiments are presented, showing that randomly accessing memory is more than 100 times more costly in SGX enclaves that need 128MB, which results in exceeding the PRM size, in comparison with unprotected applications.

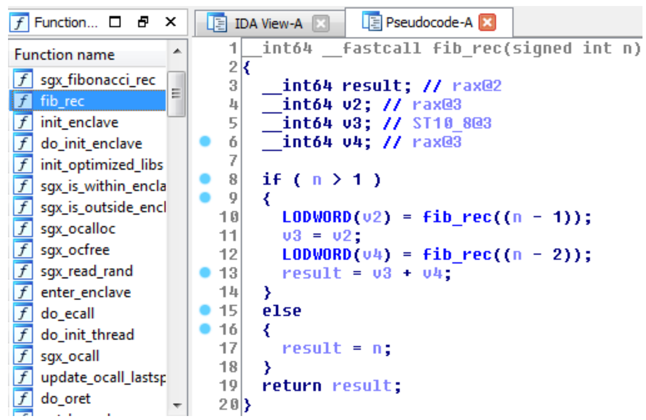


Figure 1. Reversing engineering the enclave, a recursive Fibonacci function.

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 are 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 [17], [18].

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 based 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 many types of 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 application code. Also, by using the RA process to establish a secure communication channel and using SGX capabilities to protect memory, DynSGX provides privacy guarantees for 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 RA process, (ii) loading functions into the enclave, (iii) running loaded functions inside the enclave and (iv) unloading functions from the enclave.

For the RA 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 RA 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 are 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 the functions *enclave_unregister_function* and *enclave_clear_functions* that developers 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 runs in the server side, and developers 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 programs. After writing their functions, the tool compiles the *.c* file that contains the functions and then retrieves the bytes that compose the compiled functions (this step can be done by using a tool called *bytes_extractor* provided as part of DynSGX). The bytes of the functions can then be sent to be loaded into the enclave, and later on be executed inside it. An important note is that the *.c* files are compiled 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.

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

Listing 2. Corresponding assembly for the *sum_function*.

```
push %rbp
mov  %rsp,%rbp
mov  %edi,-0x4(%rbp)
mov  %esi,-0x8(%rbp)
mov  -0x4(%rbp),%edx
mov  -0x8(%rbp),%eax
add  %edx,%eax
pop  %rbp
retq
```

After compiling the file containing this function, the *bytes_extractor* tool extracts the function bytes from the assembled x86-64 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. After sending the functions, users can execute them inside the DynSGX enclave, and even unload them afterwards. The steps needed to complete this processes are illustrated in Figure 2 and described as follows:

1) The client communicates with the DynSGX enclave and performs the RA process to verify the identity

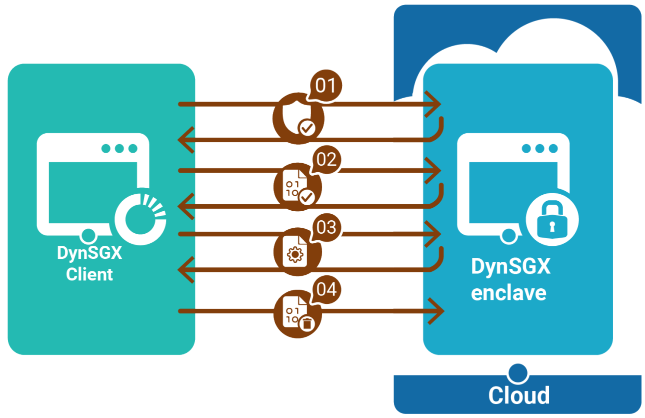


Figure 2. DynSGX Software Lifecycle.

of the enclave and establish a secure communication channel between both;

2) The client compiles the user provided functions, extracts the bytes of the generated binary file using the *bytes_extractor* tool, and sends them to the enclave via the secure communication channel. As a response, it obtains an identifier of the loaded function;
3) Via the secure channel, the client requests the DynSGX enclave to execute a dynamically loaded function, along with any user provided function parameter. As a result, the client receives the result of the execution;
4) Developer/user uses the client to request the DynSGX enclave to unload the functions 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, a distributed mechanism is needed 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.

DynSGX has an internal mechanism that retrieves the addresses and return types of all functions available to the enclave (i.e.: *tlibc* functions, functions loaded by the user, etc.) and sends to the client a JSON like the one presented in Listing 3, after the RA process is completed.

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

```
{
  "snprintf":  "(*(int(*)(0x7f1e438176f0)))",
  "vsnprintf": "(*(int(*)(0x7f1e4381d770)))",
  "strcmp":    "(*(int(*)(0x7f1e438179a0)))",
  ...
}
```

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[] = "topsecret123";
3    return !strcmp(input, password);
4  }
```

3.5. Requirements

To run the DynSGX enclave and load functions into it 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 [15] must be installed in order to enable the OS and other softwares to access the SGX hardware.
- **SGX PSW**: SGX PSW [14] is used to launch SGX enclaves, and also to generate data necessary for the RA 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 [19].

3.6. 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). Therefore, the canary value is also 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.

```
void check_password(char *input) {
    char buffer[16];
    char password[] = "topsecret123";
    strncpy(buffer, input, strlen(input));
    if (!strcmp(buffer, password))
        access();
}
```

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 the address of `access` is 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  void check_password(char *input) {
2      char password[] = "topsecret123";
3      if (!strcmp(input, password)) {
4          access();
5      } else {
6          char error[30];
7          snprintf(error, 30, input);
8          strncat(error, " is incorrect!", 14);
9          log_msg(error);
10     }
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 7, and an attacker can provide the following payload: `%10$p %11$p`. Therefore, the function will log the following message: `0x6572636573706f74 0x33323174 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 [20], Cppcheck [21] and CheckConfigMX [22] 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. EVALUATION

To assess the performance of applications that use the DynSGX toolset, we performed a series of experiments that compare it to the pure C and regular SGX and programming models. In this section we describe the experiments performed, and present a discussion based on the results obtained.

4.1. Experiments Setup

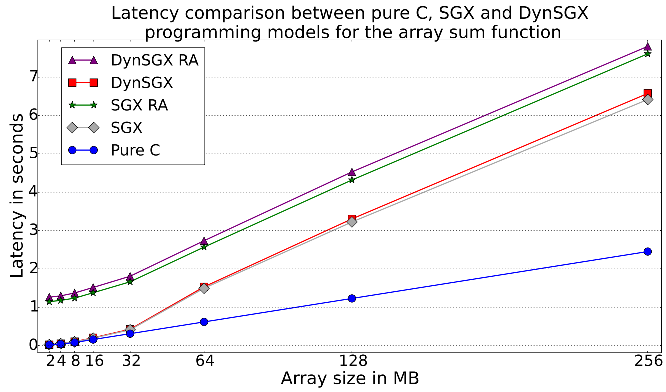
The experiments were conducted in a machine running Ubuntu Linux 16.04 with one Intel i7-6700 SGX capable processor and 8GB of RAM. The server side of the application is written in C++ and the client side is written in Python 2.7.

In our experiments, we measured the latency that represents the total time taken from the client requesting the server to process a function until the client receives the response from the server. The latency also includes the time taken to perform the RA process when RA is needed, and to encrypt and decrypt data both in the client and in the server to ensure privacy and security of data exchanged between client and server. The latency was calculated based on two different functions. The first one is the *sum_array* function, which iterates over an array of integers and computes the sum of all of them, and the second one is the *recursive_fibonacci* function, which recursively calculates the n-th term on a Fibonacci sequence. These two functions were selected to illustrate a memory-intensive application (a known limitation of SGX) and CPU-intensive application.

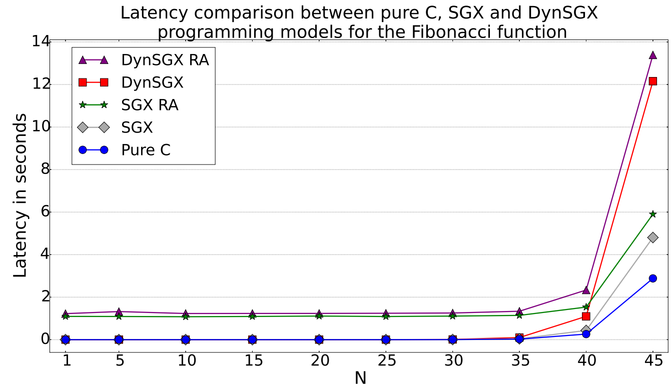
Three different implementations were made. The first does not consider any security and privacy guarantees. The second uses regular SGX programming model, with privacy and security guarantees of the data being processed but no guarantees regarding the privacy of the code. The third uses the DynSGX programming model, which enables better manageability of the memory consumed by enclave functions and privacy of the code being executed inside the enclave. For the SGX and DynSGX implementations, we considered both the cases where the RA process is needed to establish a secure communication channel, and where the secure communication channel had already been established.

4.2. The Experiments

The experiments consisted of two parts. The first part aimed to compare the latency for the computation of the



(a) Latency of *sum_array* function



(b) Latency of *recursive_fibonacci* function

Figure 3. Latency comparison between pure C, Intel SGX and DynSGX implementations

sum_array function for arrays with 2, 4, 8, 16, 32, 64, 128, and 256 MB in size. Figure 3a depicts the median values of latency obtained in this part. This experiment is useful to understand the behavior of using DynSGX with iterative functions and with higher memory usage.

The second part aimed to compare the latency for the computation of the *recursive_fibonacci* for n values of 1, 5, 10, 15, 20, 25, 30, 35, 40, and 45. Figure 3b depicts the median values of latency in this part. This experiment is important to understand the behavior of using DynSGX with recursive functions.

In both parts of the experiment, for each array size and n value, 30 runs were executed.

4.3. Discussion

As can be seen in Figure 3, both SGX and DynSGX implementations always exhibit considerable overheads when compared to the unprotected pure-C implementation. This is due to two main factors: (i) the time taken to perform the RA process and (ii) the need to encrypt and decrypt data.

From the experiments results we can also observe that for processing iterative functions, DynSGX has only a small overhead compared to the regular SGX use. This behavior can be observed both when the RA process is needed and when a secure channel has already been established. In this case, we consider the benefits of using DynSGX significant because of the code privacy guarantees and the possibility to manage memory occupied by code.

Nevertheless, it is important to notice that DynSGX may not perform as well when processing recursive functions. Figure 3b depicts this situation, where the *recursive_fibonacci* function grows exponentially in number of recursive calls. In this case, DynSGX performs much slower than the regular SGX implementation. This is due to the fact that with DynSGX the function code resides in the heap, which is considered a data segment, and competes with other data areas (such as the stack frames of each recursive call) for a space in the processor cache. Modern processors have different caches for instructions and data, hence, the advantage of regular SGX with code at the instructions segment. This behavior is not limited to DynSGX; a pure C or regular SGX implementation of a recursive algorithm that stores code in the heap also suffers some overhead.

Table 1 contains a comparison of advantages and disadvantages of each of the three implementations considered in our experiments.

TABLE 1. PERFORMANCE COMPARISON BETWEEN PURE C, SGX AND DYNSGX IMPLEMENTATIONS

	Integrity		**Privacy**		**Performance**	
Impl.	**data**	**code**	**data**	**code**	**iterative**	**recursive**
Pure C					High	High
SGX	✓	✓	✓		Medium	Medium
DynSGX	✓	✓	✓	✓	Medium	Low

5. RELATED WORK

Intel has recently introduced SGX2, which extends the SGX instruction set by adding support for dynamic memory management from inside SGX enclaves [23]. SGX2 will allow lazy loading enclave code into the EPC, instead of loading it at once at enclave load time. This will avoid the need of DynSGX to make the modification to the SGX PSW described on Section 3. On the other hand, SGX2 does not address any of the code privacy concerns addressed by DynSGX. Besides that, SGX2 does not aim at enabling developers to create new functions and loading them into enclaves after they have been built, but only to dynamically load and unload code that is already linked against enclaves.

In [24] 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 the compiled application in a SCONE client, and it will transparently be protected by SGX capablities.

SecureWorker [25] is an 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.

On the one hand, both SCONE and SecureWorker solutions, like DynSGX, 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.

A known approach for code privacy is obfuscation [26]. It makes the code difficult for humans to understand and programmers may deliberately obfuscate code to conceal its purpose (security through obscurity), primarily, in order to prevent reverse engineering. However, Barak *et al.* [27] perform a theoretical investigation and prove that, secure obfuscation is impossible. It is known that a talented hacker can reverse engineer code even after being obfuscated.

6. CONCLUSION

Intel SGX has been considered to be one of the most promising TEE technologies. TEEs can enable application with sensitive data to run on the cloud. However, its limitations and code privacy concerns have drawn its applicability in cloud environments into question. This paper presented a toolset that enables users to dynamically deploy their functions into enclaves and also ensures the privacy of such functions. Our evaluation shows that DynSGX enables developers to load/unload functions into/from enclaves and can be used to guarantee the privacy of code to be executed inside SGX enclaves with a small overhead compared to the regular SGX programming model. We have also shown that recursive functions with many recursive calls may not be suitable for use with DynSGX.

As future work we plan to extend DynSGX to provide support for programming languages other than C. In addition, we want to provide the developer/user with functions that enable a more detailed monitoring and, consequently, a more efficient management of the enclave 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] A. Gkortzis, S. Rizou, and D. Spinellis, "An empirical analysis of vulnerabilities in virtualization technologies," in *Cloud Computing Technology and Science (CloudCom), 2016 IEEE International Conference on*. IEEE, 2016, pp. 533–538.

[3] "Advanced trusted environment: Omtp tr1," accessed: 2017-08-06. [Online]. Available: http://www.omtp.org/OMTP_Advanced_Trusted_Environment_OMTP_TR1_v1_1.pdf

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

[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, 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] "Arm trustzone(r)," accessed: 2017-08-06. [Online]. Available: http://www.openvirtualization.org/open-source-arm-trustzone.html

[8] 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.

[9] "Amd memory encryption," accessed: 2017-08-06. [Online]. Available: http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf

[10] 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.

[11] 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.

[12] "Intel(r) sgx sdk for linux os," accessed: 2017-08-06. [Online]. Available: https://download.01.org/intel-sgx/linux-1.9/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.9_Open_Source.pdf

[13] 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*, 2017.

[14] "Intel sgx sdk," accessed: 2017-08-06. [Online]. Available: https://software.intel.com/en-us/sgx-sdk/download

[15] "Intel sgx driver linux," accessed: 2017-08-06. [Online]. Available: https://github.com/01org/linux-sgx-driver

[16] "Ida disassembler and debugger," accessed: 2017-08-06. [Online]. Available: https://www.hex-rays.com/products/ida/index.shtml

[17] 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.

[18] 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.

[19] "Sgx psw patch for executable heap," accessed: 2017-08-06. [Online]. Available: https://patch-diff.githubusercontent.com/raw/01org/linux-sgx/pull/63.patch

[20] "Flawfinder," accessed: 2017-08-06. [Online]. Available: https://www.dwheeler.com/flawfinder/

[21] "Cppcheck - a tool for static c/c++ code analysis," accessed: 2017-08-06. [Online]. Available: http://cppcheck.sourceforge.net/

[22] 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.

[23] 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.

[24] 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.

[25] "Npm secureworker," accessed: 2017-08-06. [Online]. Available: https://www.npmjs.com/package/secureworker

[26] 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*, ser. CCS '03. New York, NY, USA: ACM, 2003, pp. 290–299.

[27] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," vol. 59, pp. 6:1–6:48, May 2012.