Spring 5-2020

A Performance Analysis of Hardware-assisted Security Technologies

Wenjie Qiu

Washington University in St. Louis

Washington University in St. Louis

McKelvey School of Engineering

Department of Computer Science and Engineering

Thesis Examination Committee:
Ning Zhang, Chair
Jonathan Shidal
Stephen Cole

A Performance Analysis of Hardware-assisted Security Technologies

by

Wenjie Qiu

A thesis presented to the McKelvey School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

Master of Science

May 2020
Saint Louis, Missouri

Contents

List of Tables

List of Figures

Acknowledgments

I would first thank to my parents for their wholehearted support over the last several years. They taught me how to overcome difficulties and gave the strength of moving forward.

I would like to thank to my committee members for their precious time and effort, their encouragement and feedback are quite useful and meaningful. I would like to thank to my research advisor Dr. Ning Zhang for giving me chance of doing this work. His encouragement, enlightening advice, and criticism are so beneficial for me.

I would like to thank to my friends. Their smile, support and companion made me feel alive. I would like to thank to the Forest Park in St. Louis. Personally, I really enjoyed having a walk in the park and I visited many interesting corners there. I especially love the October and November in the park, and in St. Louis.

I will not forget the time I spent in St. Louis, in WashU, in Olin Library, in Jolley Hall, and more importantly, in my tiny and cozy laboratory. I learned a lot in the past two years in the lab and really enjoyed understanding and finding novel knowledge, even it is so hard for the most of the times.

Wenjie Qiu

Washington University in Saint Louis
May 2020

ABSTRACT OF THE THESIS

A Performance Analysis of Hardware-assisted Security Technologies

by

Wenjie Qiu

Master of Science in Computer Science

Washington University in St. Louis, May 2020

Research Advisor: Professor Ning Zhang

Intel Software Guard Extensions (SGX) is a novel hardware-assisted security technology introduced by Intel Corporation. The ambition of Intel SGX is to provide an isolated and secure execution environment for user-space applications. Even if the BIOS is compromised, the protected applications remains secure. The isolated execution environment is located in a special memory region called the enclave.

Promoting and using a novel technology requires a good understanding of it. This thesis first contains a systematization of knowledge of the hardware-assisted security technologies, trusted computing and the Intel SGX. What is more, to have the best practice of using Intel SGX, we must understand its advantages and limitations, especially the performance issue. This thesis then has a discussion of where the performance overhead of Intel SGX comes from and how to evaluate and avoid them. In the final chapter of the thesis, we demonstrate how to secure a non-trivial application using Intel SGX and we have a performance analysis of the protected application.

Chapter 1

Introduction

1.1 Background and Motivation

Security has became the one of the most important concerns when building modern applications or systems. The ambition of this work is to have a better understanding of the hardware-assisted security technology proposed by Intel, i.e., the Intel Software Guard Extension (SGX)[29]. We quite know that to promote a novel technology, the technology itself must be efficient and easy to use, so we pay special attention to evaluate the performance and usability of Intel SGX and its platforms. We identify where the performance overhead of Intel SGX comes from and classify them into three categories. We show a demonstration of securing OpenCV[5] using SGX platform; and finally, performance analysis are performed to evaluate the usability of each OpenCV module and the entire library.

1.2 Problem Statement

The motivation of introducing hardware-assisted security technologies is to secure applications or systems against some traditional attacks. When a technology or a product is introduced, the first question is simple and clear:

- Q1: How to use hardware-assisted security technologies to secure applications?

Although the security guarantee is of the most important, we cannot ignore the performance problem of the Intel SGX and SGX platforms. If the performance overhead is totally unacceptable, convincing developers to switch to a novel security technology with great performance compromise is impossible. Thus, the second question is about the performance:

- Q2: How to evaluate the performance of SGX applications and platforms? Do they have acceptable performance overhead?

When using Intel SGX and its platform, there might be some difference between building traditional software and SGX applications, so the third question is about the best practice of using Intel SGX and its platforms:

- Q3: What kinds of applications are well suited for SGX environment? What do we need to pay attention with when building SGX applications?

1.3 Contributions

The major contributions of this work are listed below:

- A comprehensive introduction to Intel SGX, SGX programming model, and a simple tutorial of building SGX applications.
  In the chapter 2 we have detailed information on SGX and SGX programming, and this can answer Q1.
- An in-depth performance analysis of native SGX and SGX platfroms.
  In the chapter 3 we have a through evaluation of the Intel SGX and its platforms and we conclude some tips for developers, and this can answer Q2 and Q3.
- A secured version of OpenCV powered by Intel SGX and its performance analysis.
  In the chapter 4 we show how to secure OpenCV using a SGX platform and we evaluate the performance of each OpenCV module, this can answer Q1 and Q3.

Chapter 2

Technical Background

This chapter will introduce some technical background: basic security concepts, concepts and classification of trusted computing, and the Intel SGX. We will also go through a demonstration of how to build a simple SGX application from a developer's perspective.

2.1 Security Concepts

In this section, we will introduce some basic concepts in computer security, and these basic concepts will present throughout the entire thesis when describing the security of Intel Software Guard Extensions (SGX) technology. Basic goals which mostly accepted by security community are confidentiality, integrity and availability. We will also go through concepts of authenticity and anonymity.

2.1.1 Confidentiality

Confidentiality is the avoidance and prevention of unauthorized disclosure of information[28]. This involving protection of data, providing access to people who are allowed to use and see, while denying those who do not have access to it.

To achieve confidentiality, a significant amount of tools and methods were introduced by computer security researchers, and these tools support functionalities of encryption, access control, authentication and authorization.

2.1.2 Integrity

Integrity is another aspect of computer security, which means any information has not be modified by any people in any unauthorized means[28].

Ransomware is a great example of how information integrity can be compromised[35]. Fundamentally, ransomware is a form of malware that encrypts a victim's files and what an attacker wants is money. Typically, a victim will be taught how to make the payment, and told the encrypted files will not be recovered unless the payments is completed.

To achieve integrity, administrators must regularly verify the data has not been modified. As for verification, there are multiple ways such as using checksum and data correcting codes, and it is always recommended to backup data periodically for future recovery.

2.1.3 Availability

Availability is the property that information is available and modifiable in a timely manner when those who authorized request it[28].

Information can be considered safe when it is extremely difficult for anyone to get access; however, it cannot be considered practically secure in this case, since it violates the goal of availability. We all know that the quality of some information is directly associated with the availability, e.g., stock and currency. If we do not have access to the most up-to-date data and information, it will be difficult for us to evaluate the importance and correctness of them, therefore unable to make the correct decision.

Because availability is also extremely important, attackers might simply choose to break the availability when they cannot steal or modify it without being noticed. The interruption or significant slowdown of information access by overwhelming the target machine with high network load (e.g., a web server) can be classified as Denial-of-Service (DoS) attack[32], and it is a typical example of attack on availability.

To achieve and maintain availability, administrators must have plans even in emergency situations. For example, administrators maintaining critical web servers might use some

physical protections against earthquakes, storms and severe temperature. And they may also take advantage of redundant arrays of inexpensive disks (RAID) to store redundancies to keep data available if the main server failed.

2.1.4 Authenticity

Authenticity is the ability to determine that actions or permissions are genuine[28]. The property of authenticity is a must in a secure system, because any other protections will be meaningless if a single fabricated password is approved. To achieve authenticity, we sometimes use digital signatures.

2.1.5 Anonymity

Anonymity is the property of interacting and communicating without been traced and found the real identity[28]. In ideal situations, users can send information using a pseudonym and not be identified by system administrators. This is because combining and mixing of data from individuals makes it difficult for the administrator to keep track of a specific identity.

2.2 Trusted Computing

This section describes some basic concepts and technologies related to Trusted Computing (TC), including Trusted Platform Module (TPM), Trusted Execution Environment (TEE) along with its hardware requirement, and Trusted Computing Base (TCB).

Trusted computing is a technology which first introduced by the Trusted Computing Group (TCG)[30, 20]. Four basic components of trusted computing framework are attestation, isolation, secure storage and secure I/O. With the help of them, the computer will behave under expectation enforced by hardware and software security check. However, trusted computing is controversial because the enforcement either by software, hardware or their incorporation is not just secured for its owner, but also against its owner[1], which can cause

the usability problem. Besides, the loss of anonymity can also be a serious problem: trusted computing needs to identify the uniqueness of a computer, which means it is possible for those trusted computing solution providers to keep track of a user's identity and even more than this.

In general, trusted computing is a double-edged sword so that the decision of switching to trusted computing solution needs to be made carefully and in a case-by-case manner. System administrators are responsible for evaluating the needed security level, implementation complexity, usability and entire cost, then to choose the most appropriate solution. They also need to decide which part of system needs to be secured by trusted computing, because experience tells us it is a bad idea to put everything in a trusted environment. We will discuss more about this later in this section.

2.2.1 Trusted Platform Module

Trusted Platform Module (TPM) is a standard for a secure cryptoprocessor[37], and it is a fundamental component of a trusted platform and the core of the trusted computing. It was first standardized by ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) as ISO/IEC 11889[17].

There are five variations of TPM[3], namely discrete TPM, integreted TPM, firmware TPM, software TPM and virtual TPM. Each variation has its own security level, security features, implementation complexity, and of course, cost. Generally speaking, a TPM with higher security level has higher implementation complexity, and needs higher cost. Among these variations, firmware TPM has high (but not the highest) security level , and is the most suitable and most widely used solution for personal computers nowadays.

2.2.2 Trusted Execution Environment

Protected software and some special hardware components are needed for implementing firmware TPM, but a separate chip is not necessary. With firmware TPM, the program executes on the main CPU, or more specifically, in a special secure execution environment named

trusted execution environment (TEE)[39]. A TEE is separated from the traditional program execution environment and they cannot communicate with each other without special instructions. The isolation makes it possible for administrators to seal and store credentials (e.g., a private key) in a TEE and prevent unauthorized access from outside TEE, thus greatly enhances the security level of the entire computer system.

Vendors and researchers have developed numbers of hardware technologies to support TEE implementations including AMD memory encryption [23], ARM TrustZone[27], Intel Software Guard Extensions, and RISC-V MultiZone[21]. Each technology has its advantages and drawbacks. Among them, Intel SGX is the mostly used and many prior research work [22, 9, 2] have been done to enhance its usability. We will discuss more on Intel SGX in the next section.

TEE is not flawless: there is no perfect solution for trusted computing nowadays. Administrators experience the difficulty of setting up a TEE and users have to withstand the performance overhead. In this thesis, we will focus on analysis and evaluation of performance overhead on several frameworks and solutions based on Intel SGX.

2.2.3 Trusted Computing Base

For a computer system, the trusted computing base (TCB) is comprised of hardware, software and firmware components that are critical to security[30]. The major responsibility of a TCB is to maintain the confidentiality and integrity of data on a system. The TCB in a system typically has the highest level of trust, so if a design flaw exists in a TCB, the overall system may be compromised. Thus, it is extremely important to protect the TCB from any kind of attack or insecure implementation. In practice, it is not hard to expect that a TCB is easiest to protect if its size and complexity are minimized. To secure more, we start from trusting less, so the minimization of a TCB is of great importance.

2.3 Intel SGX Explained

The Intel SGX is a set of extension to the Intel architectures. It has the ambition of providing security guarantees to user-space applications with the help of isolated memory region, known as enclave. Even the privilege software components (BIOS, OS, hypervisor, etc) are compromised, protected applications in enclaves remains secure and safe[29].

SGX relies on software attestation, i.e., proves to a user that he or she is communicating with a specific and trusted software running in a secure environment which located in the trusted hardware[16]. The proof is a hash of the content of the secure environment. Although the software entity can be load by any software or services, it will refuse to reveal any private data if the signature does not match. However, SGX is not flawless, prior works show that SGX can be cracked by multiple side-channel attacks[6, 31].

Unlike discrete TPM, SGX has a smaller TCB. The method utilized by discrete TPM is to build a chain of trust and tries to include all the software running on a machine, while SGX only covers code and data that we truly want to protect. SGX encapsulates them into an enclave image, which can be safely loaded into the enclave memory for later execution. Another advantage of SGX over discrete TPM is that discrete TPM requires a independent hardware module soldered to the chip, while everything is inside the Intel CPU in the SGX solution.

2.3.1 Memory Management

SGX Physical Memory Organization

Figure 2.1 shows the SGX physical memory organization with several layers and we will discuss them individually.

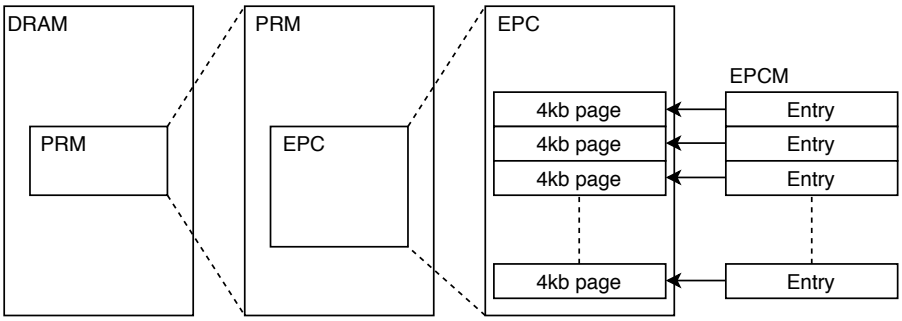


Figure 2.1: SGX physical memory organization

An Intel CPU with SGX allocates a reserved memory region with the help of BIOS, called the Processor Reserved Memory (PRM)[16]. The PRM is a contiguous subset of the DRAM and the size of PRM can be configured. The CPU protects the PRM from all memory access unless the the access is initiated directly from an enclave. Even accesses from the kernel or hypervisor are prohibited.

The Enclave Page Cache (EPC) is a subset of the PRM that store the content of enclave data and code. To support multiple enclaves on the machine at the same time, which can be utilized for data isolation, an EPC is split into multiple 4 KB pages. The size of 4 KB is not a random pick, as the virtual memory pages in an enclave are also set to 4 KB. This design is useful when memory address are translated from physical to virtual, and vice versa. Non-enclave software cannot directly access the EPC, as it is contained in the PRM. This restriction plays the core role in SGX security guarantees.

The SGX is not responsible for allocating EPC pages, and it requires the system software to allocate the EPC pages when creating enclaves. But the problem is the system software is not trusted by the SGX. In face, it may allocate some EPC pages that can compromise the SGX security guarantees. One simple example is that a memory allocator may allocate the same EPC page to two different enclaves because of the Copy-on-Write (CoW) convention, but this is malicious and will break Intel SGX security guarantees. To solve this, allocator's decision for each EPC page will be recorded in the Enclave Page Cache Map (EPCM). The EPCM is an array that holds one entry for each EPC page and Table 2.1 shows the information used for EPCM to keep track the ownership of each EPC page.

Field	**Bits**	**Description**
VALID	1	0 for un-allocated EPC pages
PT	0	page type (PT_REG, PT_SECS, etc.)
ENCLAVESECS		identifies the enclave owning the page

Table 2.1: EPCM fields and descriptions (physical memory related)

After allocating an EPC page, the VALID bit of corresponding EPCM entry will be set to 1, and next time when a allocation happens, system software will safely skip the EPC page with the VALID bit set to 1 to guarantee there is no shared memory region for any EPC page. As for the page type (PT), most EPC pages have the type of regular (PT_REG), while some EPC pages have the special type of (PT_SECS), indicating these EPC pages contain SGX Enclave Control Structures (SECS). Each SECS is stored in a dedicated EPC page that does not store any enclave data, but store only the metadata of the enclave.

The SECS is a per-enclave structure, which means each enclave has a corresponding EPC page to store the SECS. The SECS is very important and goes through the entire life cycle of an enclave: we create an EPC page for SECS before allocating enclave data EPC pages; and we destroy the EPC page with SECS after we de-allocate the enclave data EPC pages. This is similar to a `malloc-free` pair in C programming language. The EPC pages that stored enclave data will record their corresponding EPC page with SECS, and these information will be put in the ENCALVESECS filed.

SGX Enclave Memory Organization

All SGX instuctions take virtual addresses as input, and we have to fill the gap between physical addresses to virtual addresses. Figure 2.2 shows the SGX virtual memory organizaiton. For simplicity, we only consider single-process application with only one enclave.

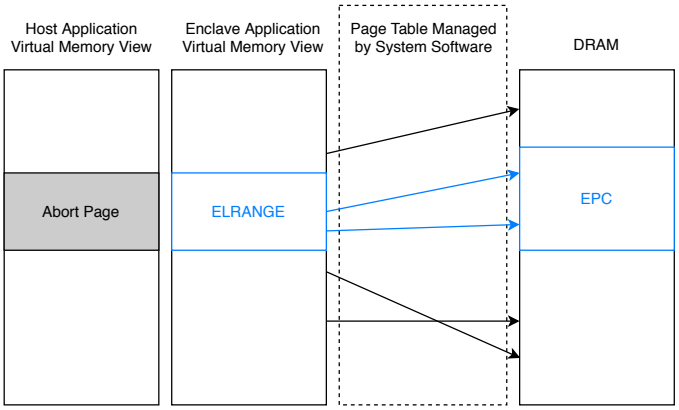


Figure 2.2: SGX virtual memory organization

An EPC page can only be accessed within a dedicated memory region called Enclave Linear Address Range (ELRANGE). An enclave image is loaded in the ELRANGE like a dynamic library, and maps sensitive code and data into ELRANGE. When the host application accesses ELRANGE memory, it will experience an abort semantics and switch to enclave application's view to access the sensitive code and data.

Table 2.2 shows virtual memory related fields of EPCM. ADDRESS shows the virtual address used to access the EPC page, and R, W, and X represent the EPC can or cannot be read, written, and executed, respectively. Although the hardware mechanism protects EPC pages from non-enclave access, the access privilege inside enclave must be checked.

Field	Bits	Description
ADDRESS	48	the virtual address used to access this page
R	1	allow reads by enclave code
W	1	allow writes by enclave code
X	1	allow execution of code inside the page, inside enclave

Table 2.2: EPCM fields and descriptions (virtual memory related)

2.3.2 Life Cycle of an Enclave

Creation

An enclave is created when an ECREATE instruction is issued. The creation of an enclave makes a free EPC page into a valid page that stores the SECS for the newborn enclave. SECS also determines the base address and size of the ELRNAGE, and stores these values in BASEADDR and SIZE fields in the SECS, respectively.

Loading

The ECREATE instruction marks a newborn SECS as uninitialized (but have already been created). An enclave must have its own data and code. When a SECS is in uninitialized state, the EADD instruction is used to load initial data and code to the enclave.

Initialization

After the loading process is done, the system software must invoke a privileged enclave provided by Intel, i.e., the Launch Enclave (LE) to mark the enclave to initialized state. The EINIT instruction is used during this process.

Teardown

After an enclave finishes its job, the system software issues the EREMOVE instruction to deallocate corresponding EPC pages. EREMOVE marks corresponding VALID fields of EPCM to 0, indicating they are not occupied by any enclave. After all these EPC pages have been deallocated, the final step of destroying an enclave can be performed: the SECS of an enclave will also be removed and destroyed.

2.4 Intel SGX Programming

We have a discussion on the mechanism of Intel SGX. Naturally, the next step is to discuss how to utilize the security benefits of SGX and how to build SGX programs. Unlike traditional programming models, Intel SGX programming has some unique requirements, on both hardware and software environments. Building SGX applications needs the support of SGX SDK and a tool designed by Intel, and running SGX applications in a trusted execution environment requires the SGX Driver and a dedicated machine. In this section, we will have a thorough overview on hardware and software environment of Intel SGX, and then have a demonstration on how to write and build a simple SGX application.

2.4.1 Hardware Requirements

Intel SGX was first introduced in 2015 along with the release of 6th generation (Skylake) Intel CPUs. Skylake and newer generations is a must for enabling SGX. However, we must notice that not all PCs with a new Intel CPU support SGX, e.g., to the best of my knowledge, none of Apple's PC fully supports SGX by now. We still need support from the motherboard and BIOS. Since a motherboard BIOS is most likely provided by third-party vendors and they do not typically provide such a detail specification, it is difficult for users to determine if there is support for SGX before they purchase any hardware. To resolve this, unofficial lists of hardware and tools that can determine SGX support is a great starting point[4]. In most situations, the SGX features are not enabled by default, so we should enable them manually. The BIOS will expand Processor Reserved Memory (PRM) at a typical size of 128MB, making it exclusive to SGX related code and applications[16].

2.4.2 Software Requirements

Intel provides SGX software support for both Windows and Linux operating systems. In this section we will only focus on SGX software stack in Linux operating systems (e.g., Ubuntu 16.04). The Linux SGX software stack is the collection of the Intel SGX driver, the Intel

SGX Software Development Kit (SDK), and the Intel SGX Platform Software (PSW). Intel holds repositories [10, 12, 11] on GitHub for public access.

SGX Platform Software

The SGX PSW is the prerequisite of running a SGX program. It provides the platform services, run-time attestation components as well as the SGX driver. PSW contains the utilities that support three major functionalities: launch service, Enclave Page ID based attestation service, and algorithm agnostic attestation service. Among them, launch service is of the most important, since every SGX application should start with loading and initializing enclaves. To make SGX applications run properly, the Architectural Enclave Service Manager Daemon (AESMD) service should also to turned on so that applications can utilize Architectural Enclaves (AE) such as Launch Enclave (LE) to load and run an enclave[16].

SGX Software Development Kit

The Intel SGX SDK provides a collection of APIs, libraries and tools that can assist software developers to build SGX application in C/C++ running in a SGX enabled environment. The SGX SDK is comprised of four major components: Untrusted Run-Time System (uRTS), Trusted Run-time System (tRTS), Edge Routines as well as third party libraries[13].

The Untrused Run-Time System (uRTS) runs outside of the enclaves, and that is why they are been called "untrusted". The uRTS is responsible for loading and managing enclave, as well as sending enclave calls (ECALLs) to and receiving outside calls (OCALLs) from an enclave.

The Trusted Run-Time System (tRTS) runs insides of the enclaves, and by definition, they are "trusted". The tRTS is responsible for receiving calls (ECALLs) from outside and making responses (OCALLs) to outside untrusted environment. The tRTS also contains carefully designed SGX-enabled C/C++ standard libraries.

Some low-level functions are served as glue for trusted and untrusted communication, and they may execute inside an enclave (trusted edge routines) or outside an enclave (untrused

edge routines). The edge routine functions are complicated and full of low-level details, so writing edge routines directly can be difficult and inefficient. Fortunately, a tool `edger8r` which developed by Intel resolves this issue, and developers just need to focus on higher level API and code and split them into trusted and untrusted parts properly[15].

Intel also built the Intel Software Guard Extensions SSL (Intel SGX SSL) cryptographic library[14] on top of OpenSSL Open Source project[18] to provide cryptographic services for SGX applications. This is also an great example of porting an non-trivial application to the SGX environment.

2.4.3 A Simple Tutorial

Trusted Code

Unlike traditional programming model, developers must to split their code into trusted and untrusted parts. Code and data that developers want to keep them secure must be explicitly put into a trusted world, that is, an enclave. An enclave is self-contained and will be compiled to a binary called an enclave image. Whenever running a SGX application, the AESMD service will invoke the Architectural Enclaves, load the application enclaves into EPC pages in the protected memory and then execute.

As mentioned above, developers can only access an enclave by defining and using the access-points. Theses access-points are defined using a Encalve Definition Language (EDL). An EDL file shows a collection of access-points for an enclave as well as the included trusted libraries and other data structures. An EDL file first includes trusted libraries that a developer want to use in this enclave if any, followed by prototypes of trusted ECALL functions and then untrusted OCALL functions.

Listing 2.1 shows that an EDL file which contains five ECALL functions and one OCALL functions. The ECALLs should be put into the trusted section with the prefix `public`, and the OCALL is put into the untrusted section. The function `ecall_print_secret` and `ecall_print_updated_secret` print an enclave's secret and updated secret, respectively. The function `ecall_return_value` is a demo of how a return value can be retrieved with an

ECALL. The function `ecall_array_increment` discuss the data marshalling as well as the user-specified boundary. The function `ecall_user_check` demonstrates how to use a user-checked function pointers. The functions above are all ECALLs and they are implemented in a C file reserved for enclave implementation, typically named `enclave.c`. The OCALL function `ocall_print_string`, however, must be implemented in untrusted application code, typically named `app.c`.

```
1  enclave {
2     trusted { // Your ECALLs here
3          public void ecall_print_secret();
4
5          public void ecall_print_updated_secret();
6
7          public int ecall_return_value();
8
9          public void ecall_array_increment(
10           [in, out, count=len] int *array,
11           int len);
12
13         public void ecall_user_check([user_check] int *ptr);
14     };
15     untrusted { // Your OCALLs here
16         void ocall_print_string([in, string] const char *str);
17     };
18 };
```

Listing 2.1: Enclave definitions

Listing 2.2 shows the implementation of trusted functions, and many of them are self-explained. But we still should notice that:

- Not all of them are ECALLs, such as the function `update_secret` and `printf`.
- The ECALL `ecall_array_increment` is a dual data marshalling function with keywords `in` and `out`, which means that information in array can go either direction, and developers are expected to provide the size of marshalling data when calling this ECALL function.

- The function `printf` is not the function in standard library `<stdio.h>`. In fact, trusted version of `<stdio.h>` does not support the traditional `printf` function: we need to build an OCALL like `ocall_print_string`, to print a string. The function `printf` here is only a wrapper for easy use.
- The keyword `user_check` means we interpret a pointer as its original address, i.e., the address in the untrusted memory.

Bring an untrusted memory address into an enclave can be a problem, because we need to differentiate it from a trusted memory address, which can be misleading.

```
1  char *secret = "I am the SGX secret\n";
2
3  void ecall_print_secret() {
4      printf("[trusted] %s", secret);
5  }
6
7  void update_secret() {
8      secret = "I am the updated SGX secret\n";
9  }
10
11 void ecall_print_updated_secret() {
12     update_secret();
13     printf("[trusted] %s", secret);
14 }
15
16 int ecall_return_value() {
17     return 42;
18 }
19
20 void ecall_array_increment(int *array, int len) {
21     printf("[trusted] %p\n", array);
22     for (int index = 0; index < len; index++)
23         array[index]++;
24 }
25
```

```
26 void ecall_user_check(int *ptr) {
27     printf("[trusted] %p\n", ptr);
28 }
29
30 void printf(const char *fmt, ...)
31 {
32     char buf[BUFSIZ] = {'\0'};
33     va_list ap;
34     va_start(ap, fmt);
35     vsnprintf(buf, BUFSIZ, fmt, ap);
36     va_end(ap);
37     ocall_print_string(buf); // invoking the OCALL function
38 }
```

Listing 2.2: Enclave implementations

Application Code

Listing 2.3 shows the entry code of a SGX application (printing statements and other unimportant statements are ignored). It is the untrusted code that runs in the traditional unsafe environment, therefore, it cannot process any sensitive data directly by definition. However, we still need it to initialize and load enclaves, and to execute ECALLs to process sensitive information in enclaves.

```
1  int main(int argc, char *argv[])
2  {
3      if(initialize_enclave() < 0){
4          return -1;
5      }
6
7      ecall_print_secret(global_eid);
8      ecall_print_updated_secret(global_eid);
9
10     int ret_val = 0;
11     ecall_return_value(global_eid, &ret_val);
```

```
12
13      int array[10];
14      for (int i = 0; i < 10; i++)
15          array[i] = 1;
16      ecall_array_increment(global_eid, array, 5);
17
18
19      int *ptr = (int *)malloc(sizeof(int));
20      memset(ptr, 0, sizeof(int));
21      ecall_user_check(global_eid, ptr);
22
23      sgx_destroy_enclave(global_eid);
24      return 0;
25  }
```

Listing 2.3: Entry of application code (printing statements excluded)

Listing 2.4 shows a sample output generated by the above application. The prefix in each line indicates that data in this line is either trusted or untrusted execution environment. As we can see in line 4 and 5, the very same function pointer's address is interpreted in different ways by untrusted and trusted environment, which means they are isolated from each other by default. But when a `user_check` keyword is provided in the ECALL `ecall_user_check`, it is the developer's responsibility to remember that the pointer is from untrusted environment. And in this case, trusted and untrusted environment have the same view for this user-checked pointer, so that their addresses are the same. E.g, in this case, see line 7 and line 8.

```
1  [trusted] I am the SGX secret
2  [trusted] I am the updated SGX secret
3  [untrusted] 42
4  [untrusted] 0x7fff8124a540
5  [trusted] 0x7fbd8922c010
6  [untrusted] 2 2 2 2 2 1 1 1 1 1
7  [untrusted] 0xf2b0b0
8  [trusted] 0xf2b0b0
```

Listing 2.4: Sample outputs

Some utility functions and OCALLs are also implemented in the main application. The utility function `initialize_enclave` is widely used for creating enclaves and the OCALL `ocall_print_string` is used for printing debug information.

```
1  int initialize_enclave(void)
2  {
3      /* Step 1: try to retrieve the launch token saved by last transaction
4       * if there is no token, then create a new one. */
5
6      // ...
7
8      /* Step 2: call sgx_create_enclave to initialize an enclave instance */
9      ret = sgx_create_enclave(
10       ENCLAVE_FILENAME, // enclave image
11       SGX_DEBUG_FLAG, // 1 for debug on, 0 otherwise
12       &token, // token file
13       &updated, // 1 for token file is updated, 0 otherwise
14       &global_eid, // enclave id
15       NULL // optional attributes
16     );
17
18     // ...
19
20     /* Step 3: save the launch token if it is updated */
21
22     // ...
23     return 0;
24 }
```

Listing 2.5: Enclave initialization implementation

```
1  void ocall_print_string(const char *str)
2  {
3      printf("%s", str);
4  }
```

Listing 2.6: OCALL print function implementation

Supplementary Tools

Figure 2.3 shows the process of generating binary and enclave image. As mentioned above, the low-level edge routines are the actual functions executing when an `ECALL` or an `OCALL` invoked. These details are quite distracting and make it inefficient for developing SGX applications. Fortunately, a tool named `Edger8r` included in the SGX SDK is responsible for parsing the EDL files and generating the trusted and untrusted routines between the application and the enclave. These auto-generated files have default names. `Enclave_u.c/.h` are generated in the application side, and `Enclave_t.c/.h` are generated in the enclave side. To utilize those routines, developers must explicitly include these SDK generated code as well as Run-Time System library `<sgx_urts.h>` and `<sgx_trts.h>` accordingly.

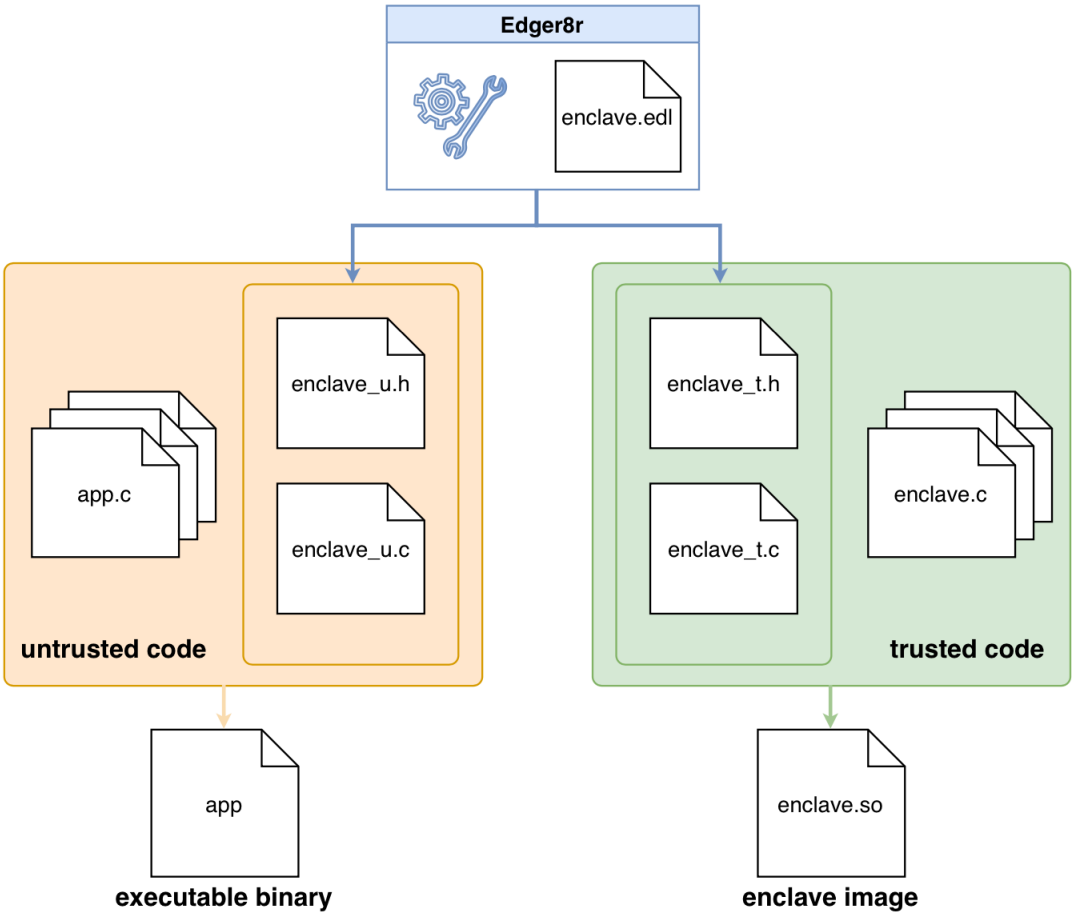


Figure 2.3: Process of generating executable binary and enclave image

Chapter 3

Performance Analysis of Native SGX and SGX Platforms

In this chapter, we will have discussion on performance of native Intel SGX and SGX platforms. We will find where the performance overhead comes from and classify the factors into three categories: switching delay, computation related and memory access related. We will also setup experiments and evaluate how these factors can affect the performance and usability of Intel SGX.

3.1 Performance Overhead of Native SGX

This sections presents three factors that affect the performance of native SGX applications and SGX platforms. In general, we will review them individually and carefully, and answer why and how these factors could have possibly negative impact on SGX applications.

3.1.1 Switching Delay

The switching delay comes from the switching from normal execution environment to trusted execution environment, or vice versa. Many operations can introduce the switching delay, e.g, an ECALL or OCALL, or even a system call. An ECALL or OCALL will obviously slow down the execution because of switching between trusted environment and untrusted environment[42, 19]. As for a system call, it can not be executed inside a SGX environment

directly, and needs to be executed outside. Therefore, a system call is comprised in an OCALL for native SGX applications. Same as an OCALL, switching to the normal execution environment to perform a system call will inevitable introduce a delay. Switching delay is the major source of overhead for some I/O intensive applications, to reduce the overhead, developers need to invoke ECALLs, OCALLs wisely.

3.1.2 Computation Related

The computation related overhead is small and negligible. For most modern processors and compilers, they will try to place most variables of a function (in a program) in registers for best access and execution time. Since a SGX program utilizes the CPU in the same way that a normal program do, if necessary optimization is presented and the workload is not memory intensive, the performance overhead for a computation intensive workload is negligible.

3.1.3 Memory Access Related

The memory access related overhead is the major source of overhead for a memory access intensive SGX program. Since SGX programs are executed in a special region with limited memory, they will experience the memory paging delay if the EPC size is not enough for them to execute. Although paging is common even in a normal execution, SGX programs are more likely to suffer and suffer more from this[36]. Prior work shows that if the memory need for a SGX application is at a very high level, it might not even start to execute properly[19].

3.2 Analysis of Switching Delay

The switching delay of the Intel SGX happens every time when a SGX program tries to cross the boundary between trusted and untrusted environment, i.e., whenever an ECALL or OCALL happens. In this section, we will have discussion on how it can affect the performance of SGX applications and evaluation of it.

3.2.1 Methodology

To evaluate the switching delay, we use two groups of functions. In general, functions in group 1 do not have switching delay and functions in group 2 contain switching delay. For group 1 which listed in Listing 3.1, it contains three functions which execute in the normal execution environment. They can be executed in the normal environment directly, which means invoking them does not require switching the environment, and therefore will not introduce any switching delay. For group 2 which listed in Listing 3.2 and Listing 3.3, it contains one OCALL and four ECALLs. To execute these functions, we must start from untrusted part of the application and then switch to the SGX environment, therefore the switching delay will be introduced. And an nested `ecall_ocall` function will even experience such delay twice.

We write some simple and representative functions for normal and SGX environment. The functions `normal_empty` and `ecall_empty` are the simplest functions that can reflect the influence of the switching delay; the function `ecall_ocall` is simplest ECALL-OCALL pair; the functions `normal` and `ecall_print` can be used to show how print statements can slow-down the execution; and the functions `normal_malloc` and `ecall_malloc` can be used to show whether a simple memory allocation can affect the performance.

Listing 3.1: Normal functions

```
1  void normal_empty()
2  {
3      ; // do nothing
4  }
5
6  void normal_print()
7  {
8      printf("print something.\n");
9      // print directly
10 }
11
12 void normal_malloc()
13 {
14     void *ptr =
15         malloc((size_t)1024);
16     free(ptr);
17 }
```

Listing 3.2: OCALL function

```
1  void ocall_empty()
2  {
3      ; // do nothing
4  }
```

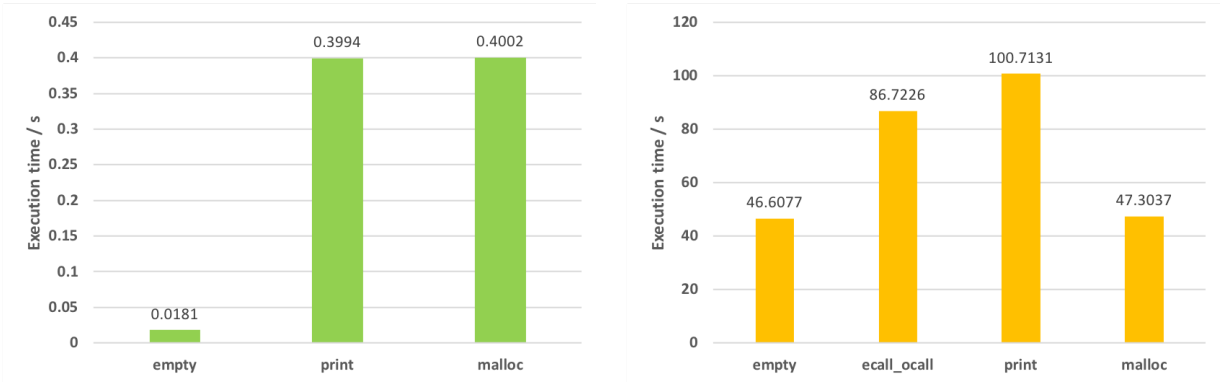
Listing 3.3: ECALL functions

```
1  void ecall_empty()
2  {
3      ; // do nothing
4  }
5
6  void ecall_ocall()
7  {
8      ocall_empty(); // nested call
9  }
10
11 void ecall_print()
12 {
13     printf("print something.\n");
14     // do an ocall and print
15 }
16
17 void ecall_malloc()
18 {
19     void *ptr =
20         malloc((size_t)1024);
21     free(ptr);
22 }
```

3.2.2 Evaluation

For each experiment, we execute each function for one million times, and 50 runs for each experiment are reported so that we can calculate the average. Figure 3.1 shows the results.

From the figure we can see that the switching delay is absolutely significant, e.g., an empty ECALL `ecall_empty` is more than 2500x times slower than the `normal_empty` function in the normal environment. The `ecall_ocall function` takes approximately double time of



(a) Group 1: normal functions

(b) Group 2: SGX functions

Figure 3.1: Performance of normal and SGX functions

`ecall_empty` and this is consistent with the fact that an ECALL which contains an OCALL would experience the switching delay twice.

For the performance of `normal_print` and `ecall_print` functions we can see printing statements will heavily affected the execution performance. In the normal environment, because invoking a `printf` function involves system calls which can take a relatively longer time, the performance is affected. And in the SGX environment, a print statement will introduce an ECALL dealy and an OCALL switching delay plus preparing time for the string, as well as the real printing time in the normal environment. In general, using a print statement in the SGX environment is more than 250x times slower than using a `printf` directly in the outside.

As for the `ecall_malloc`, its execution time is only slightly higher than the simplest ECALL `ecall_empty`. This means the switching delay is the major source of performance overhead compared to libc functions in the enclave and system calls outside.

In general, the surprisingly high performance overhead introduced by the switching delay of Intel SGX needs to be pay attention with. Developers must build the SGX applications wisely and try not to abuse the ECALL, OCALL or any other functions that will introduce the significant switching delay.

3.3 Analysis of Computation Intensive Workload

This section presents the analysis of computation intensive workload. We first goes through the computation intensive workload which we choose: the stress-ng and its SGX version stress-sgx, and then perform the experiment. Finally, we show a performance analysis of computation intensive workload running in a SGX environment.

3.3.1 CPU Stress Test

A great example of computation intensive workload is the CPU stress test application. The stress-ng is a tool that performs stress test to a computer system in various ways. It is designed to test various functionalities and performance of a computer system and its subsystems. The stress-ng supports multiple stressors (stress mechanisms) and multiple instances of stressors can run at the same time. However, the stress-ng is not designed for running in the SGX environment. In order to run the stress test in the SGX environment and put enclaves in a high load, the ported SGX version of stress-ng, the stress-sgx has been introduced in the prior work[41].

3.3.2 Evaluation

All experiments use Intel NUC7i5BNH with an i5-7260U processor with 4 cores at 2.20 GHz with 4 MB cache and 12 GB main memory, and the host OS is Ubuntu 16.04 LTS 64-bit. Since the machine has four cores, we create four stressor instances for both stress-ng and stress-sgx, using various CPU stressors and keep them running for 60 seconds. We make them execute ten times so that we can calculate the average performance statistics.

The parameter that we focus is the number of "bogo" operations. The "bogo" operations (or "bogo" operations per real time second) is the measure of a stress test created by stress-ng or stress-sgx. The size of a "bogo" operations depends on the type of stressor, and are only comparable between stress tests that using the same stressor. Although the stress test is not meant to be the scientifically accurate benchmarking metric, it can always give us some rough sense.

The statistics of the "bogo" operations and performance is listed in Table 3.1, ranging from lowest performance overhead to highest. The overhead here is defined and discussed by equation 3.1 and equation 3.2, respectively.

$$Overhead = \frac{bogoops(normal)}{bogops(sgx)} = \frac{bogoops(normal)/second}{bogoops(sgx)/second} \tag{3.1}$$

$$Overhead \begin{cases} < 1 & \text{SGX is better than normal,} \\ = 1 & \text{identical performance,} \\ > 1 & \text{normal is better than SGX} \end{cases} \tag{3.2}$$

Basically, if the overhead is smaller than one, the cpu stressor shows a better performance in a SGX environment, and vice versa. The Table 3.1 tells us that the most cpu stressors performance overhead are close to one, which means there are no significant slowdown when running in an SGX environment compared to stress-ng in a normal environment. What is more, some of them even shows better performance, e.g, `hyperbolic`, `explog`, and `fft`. The overall overhead is 1.298, which is not a huge number. However, for some CPU stressors, the slowdown is not negligible. Some stressor's overhead are larger than 1.5, e.g., `hanoi`, `hamming`, and the `callfunc` and `ackermann` stressors even shows a significant slowdown in an SGX environment at the overhead of 2.877 and 4.082, respectively.

All these stressors with high overhead (i.e., poor performance in an SGX environment) contain large amount of function calls. Some of them are classic recursive problems that contain a lot of function calls, e.g., `hanoi`, `hamming`, `ackermann`, and `callfunc`. Besides, the `callfunc` stressor is designed to invoke function calls repeatedly. In summary, we can conclude that although invoking function calls always takes time, SGX environment seems suffer more from doing so and will have poor performance.

cpu stressor	sgx/ops	native/ops	overhead
fft	147079.2	96264.9	0.655
explog	51560.3	38381.5	0.744
hyperbolic	356118.7	265163.4	0.745
factorial	2772294.7	2164025.9	0.781
idct	8328986.6	7624782.8	0.915
loop	5209792.5	4806023.9	0.922
omega	849213929.9	828142954.4	0.975
bitops	532234.6	527669.3	0.991
ln2	13314384.7	13735355.6	1.032
parity	2249085.4	2446941.1	1.088
crc16	29104.5	33136.5	1.139
gray	2784047.3	3184057.2	1.144
gcd	345060.1	396898.7	1.150
nsqrt	26856.5	31013	1.155
correlate	8306.4	9836.4	1.184
jenkin	7315521.6	8718239.7	1.192
euler	2531513286	3035166219	1.199
gamma	65047.8	80445.9	1.237
djb2a	17038784	21493209.7	1.261
fibonacci	2589914734	3344835222	1.291
fnv1a	16940672.3	22030862.2	1.300
jmp	11658819.1	16435452.3	1.410
hanoi	20571.2	31955.6	1.553
hamming	54579.5	94874.2	1.738
callfunc	17509766.4	50381666.7	2.877
ackermann	965.8	3942.7	4.082

Table 3.1: Statistics of CPU stressors

3.4 Analysis of Memory Access Intensive Workload

This section presents the analysis of memory access intensive workload. We first go through the memory intensive workload: the stress-ng and stress-sgx are also used for memory stress tests, and then perform the experiments. Finally, we show a performance analysis of memory access intensive workload running in an SGX environment.

3.4.1 Memory Stress Test

A great example of memory intensive workload is the memory stress test application. Again, we use stress-ng and stress-sgx application because they can also be used for memory stress test.

3.4.2 Evaluation

All experiments use Intel NUC7i5BNH with an i5-7260U processor with 4 cores at 2.20 GHz with 4 MB cache and 12 GB main memory, and the host OS is Ubuntu 16.04 LTS 64-bit. Since the machine has four cores, we create four stressor instances for both stress-ng and stress-sgx, using various memory stressors and different memory load. We keep them running for 60 seconds, and make them execute ten times so that we can calculate the average performance statistics.

Again, the parameter that we focus is the number of "bogo" operations and "bogo" operations per real time second. The "bogo" operations (or "bogo" operations per real time second) is the measure of a stress test created by stress-ng or stress-sgx. The size of a "bogo" operations depends on the type of stressor, and are only comparable between stress tests that using the same stressor. Although the stress test is not meant to be the scientifically accurate benchmarking metric, it can always give us some rough sense.

Figure 3.2 summarizes the results of memory stressors and there are two ways to interpret it. The first measurement is that the percentage of each disk that filled with non-white color represents the relative performance in a SGX environment, compared to the performance of the stressors running on the native host machine. This means 50% full indicates that the memory stressor in SGX can run at the half speed, compared to the speed of running in the native environment. And a 100% full disk means the memory stressor is not affected by the SGX environment. The second measurement is the color of the disk itself that represents the relative performance. From high to low performance, the color is ranged from dark green, light green, yellow, orange to red. It is not meant to be an accurate indicator, but in general, a "greener" disk has higher relative performance than yellow or red ones.

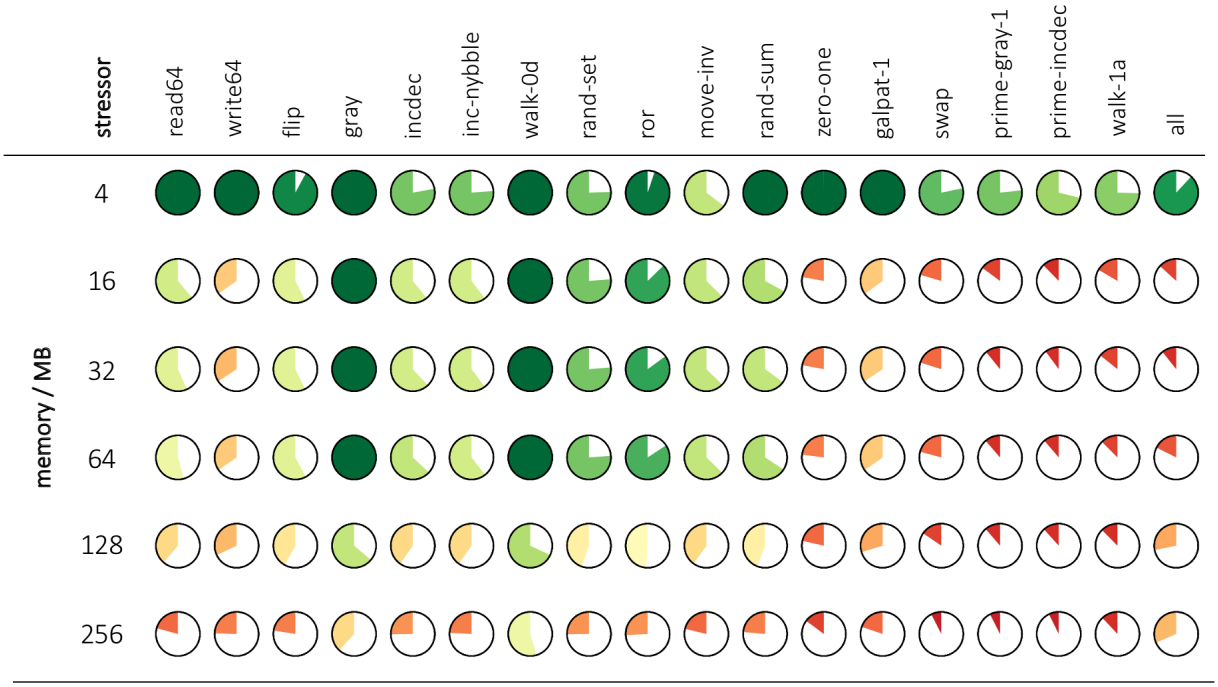


Figure 3.2: Relative performance of memory stressors in an SGX environment

From the figure we can conclude that the performance is not heavily affected when the memory load is small, e.g., 4 MB. However, when the memory load grows to a amount larger than the size of cache, the slowdown of execution is noticeable. In general, larger memory load will make the performance even worse, and that is why the upper section of the figure is much "greener" than the lower section.

What is more, the execution is also negatively affected by the random memory access. We arrange stressors from sequential access to random access, except the `all` stressor which take each stressor's pattern in a round-robin manner that reflects the general situation. The figure clearly tells us that the left half of stressors only have "red" performance when the memory load rise to 256 MB, and they are much "greener" than the right half. Especially for the `gray` and `walk-0d`, since they can enjoy "dark green" performance even if the memory load is 64 MB. The right half of stressors, on the other hand, are suffered from the random memory access, and the commonly presented "red" disks can prove our thought. Stressors such as `swap`, `prime-gray-1`, `prime-incdec` and `walk-1a` are negatively affected even at a small memory load, e.g., 16 MB.

3.5 Analysis of SGX Platforms

This section presents the usability problems of Intel SGX and how it can be solved by SGX platforms. We will have discussion on two of the Intel SGX platforms: Secure Container Linux Containers with Intel SGX (SCONE), and a library OS for unmodified application (Graphene-SGX). We will have performance evaluation on both platforms.

3.5.1 SGX Platforms

Intel SGX solves some important security problems, but it also brings some usability problems. For example, developers need to rewrite traditional programs to SGX capable programs. This requires developers to have a deep understanding of the programs as well as the security and SGX knowledge. Although Intel offers a trusted C library in SGX SDK, it is not complete and many functions are removed for security concerns. The fact is that they have to build it from scratch, since not so many trusted version of applications or libraries are available. Researchers basically try to solve the usability problems in the following two ways: the first method is to set up special SGX platforms so that unmodified programs can execute with little effort, and the second method is to port the traditional applications to SGX applications. The second method seems exciting and some work has been published. The Glamdring is a framework that can automatically analysis the program and port it to a SGX program[26]. However, it is not open-sourced and released currently. Thus, utilizing a SGX platform to protect the unmodified application with the help of Intel SGX is the best option currently.

SCONE SCONE is a secure container mechanism designed for Dockers to protect process inside with the help of the Intel SGX[2]. The design goals of the SCONE including small trusted computing base (TCB) and low performance overhead. SCONE offers a trusted C library and a control interface that can supports user-level multi-threading and system calls. We will have discussion on SCONE in detail in the next chapter.

Graphene-SGX Graphene-SGX is a library OS tries to solve the usability problem using "shim" layers and it has some improvements like integrity support for dynamically-loaded libraries[9]. Graphene-SGX supports a wide range of unmodified applications, including Apache, GCC, and the R interpreter, and the performance overhead for single-process programs is typically less than 2 times.

3.5.2 Methodology

All experiments use Intel NUC7i5BNH with an i5-7260U processor with 4 cores at 2.20 GHz with 4 MB cache and 12 GB main memory, and the host OS is Ubuntu 16.04 LTS 64-bit. We evaluate two applications, Lis, Libsvm; and one benchmark suite, DIBS. We compare the performance of three variants for each application: (i) one built with GNU C library (glibc) and executes in the host OS; (ii) one build with musl C library and runs inside a SCONE Docker container; and (iii) one built with GNU C library (glibc) and run in the Graphene-SGX library OS. We use glibc to build these applications or benchmark in the first and third setting is because the glibc is the standard C library for most Linux distributions. We run each experiment for ten times so that we can calculate the average.

3.5.3 Application Benchmarks

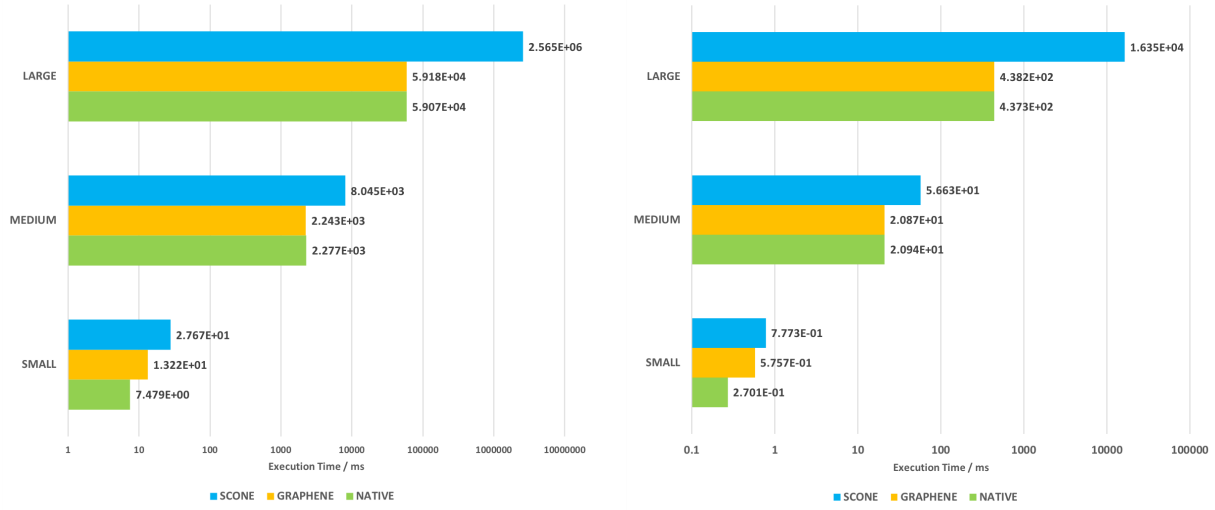
Lis Library of Iterative Solvers for linear systems (Lis) is a mathematical library for solving discrete linear equations and eigenvalue problems[33, 25, 24, 38]. Solving mathematical problems involves complex operations and can possibly create a high memory load. Since more and more complicated real world applications will be put into SGX environment, it is necessary to evaluate the performance of math libraries.

We evaluate the performance of `linear_solver` and `eigen_solver` in each three different settings. The size of problems (i.e., the size of matrix) are ranged from small (100x100), medium (1000x1000) to large (5000x5000). Figure 3.3 shows the results.

As for Graphene, when the matrix sizes are medium and small, the execution time for both problems is nearly identical to the native OS. However, if the size of the problem is small,

the overhead of booting up the Graphene library OS is not negligible. But in general, the overhead of running lis library in Graphene library OS is very small.

As for SCONE, no matter how large is the matrix, the performance overhead of running lis library in SCONE is significant. When the size is small (100x100), SCONE are 3.7x and 2.8x times slower in solving linear problem and eigenvalue problem than native OS, respectively; when the size is medium (1000x1000), SCONE are 3.5x and 2.69x times slower. However, the performance is even worse when the problem size is large (5000x5000), as these two numbers surprisingly increase to 42.4 and 37.4, respectively.



(a) Linear Solver

(b) Eigen Solver

Figure 3.3: Lis performance

Libsvm Libsvm is a simple and efficient library for solving SVM classification and regression problems[8]. Previously, machine learning communities spent a lot of efforts on efficiency and usability, but the privacy problem in machine learning now attracts a lot of attention. Many ways of protecting data privacy has been introduced and deployed such as differential privacy. The traditional procedure is that users who do not have enough computation resources have to upload their local data to a remote server which they may or may not trust, and then wait until receive any results. Therefore, the security problem is almost inevitable: users do not know if their data is carefully transmitted, stored and protected. However, things can change in the next generations of machine learning libraries or any applications

		train/s			predict/s		
model	size	native	Graphene	SCONE	native	Graphene	SCONE
w8a	3.4 MB	16.184	15.693	31.92	11.187	11.021	16.24
a3a.t	2.1 MB	37.802	36.246	77.61	23.672	22.73	32.79
w8a.t	1022 KB	1.513	1.933	5.938	1.186	1.664	4.849
w4a	504 KB	0.411	0.927	4.013	0.321	0.841	3.83
a1a	113 KB	0.131	0.67	3.584	0.101	0.634	3.558

Table 3.2: Libsvm performance

that use these libraries. They might let users to decide whether to upload local data for processing. The ideal situation is that small amount of data can be trained and processed locally, and data privacy can be guaranteed with the help of the Intel SGX or other hardware assisted security technologies.

Table 3.2 shows the results of libsvm. We select five models ranged from 113 KB to 3.4 MB and perform train and predict operations in three different environments. For each train or predict, we repeat the experiment for ten times to calculate the average.

As we can see from Table 3.2, when the size of matrix is small, e.g., a1a is 113KB and w4a is 504KB, the slowdown for train and predict are both significant. This is because the "warm up" time for Graphene and SCONE to start a program takes up a large percentage of execution time, which make it seem that performance overhead is as large as 5x times for Graphene and even 30x times for SCONE. When the size of matrix grows larger, the advantage of Graphene can be noticed easily. Graphene nearly enjoys the same execution speed of native OS, while SCONE has a overhead of more than 2x times.

DIBS Data Integration Benchmark Suite (DIBS) is a suite of applications that select some representative data integration workloads[7]. With the development of Intel SGX, its libraries, and SGX applications and platforms, more and more amount and types of data will be put into that small enclaves. Thus, the performance of these operations is becoming more and more important in the near future.

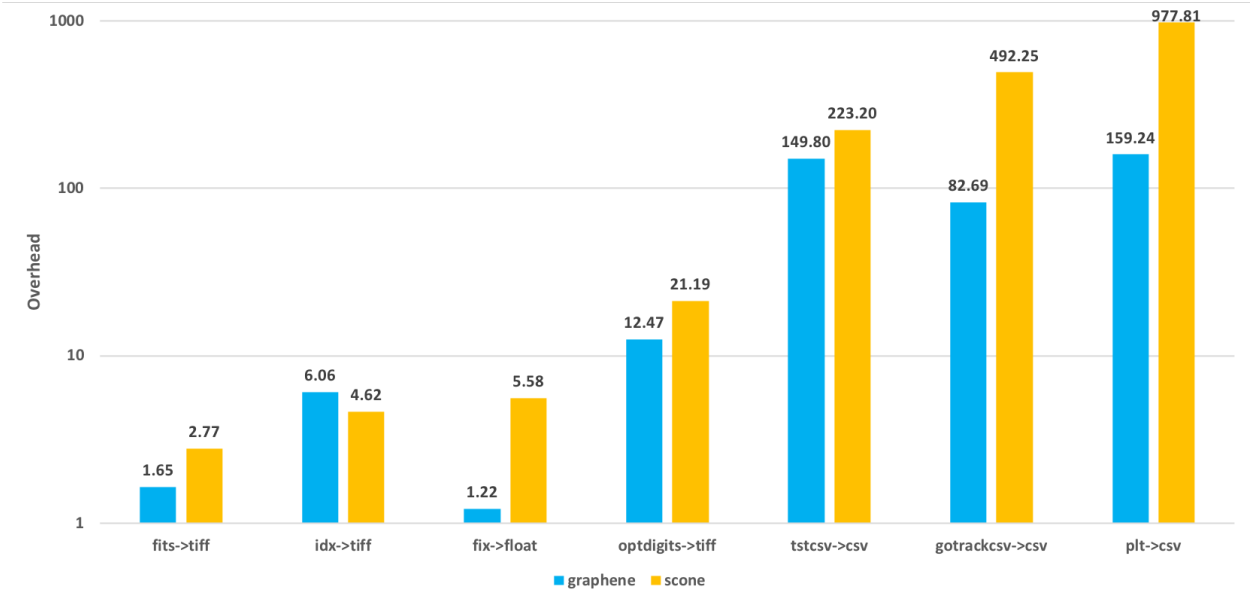


Figure 3.4: DIBS performance

We evaluate seven applications of DIBS in all three settings and Figure 3.4 shows the performance overhead in Graphene and SCONE compared to native OS. We know that one of the most important measurement of data related benchmarks is the data throughput. Since the tested files are unchanged, the performance overhead also reflects the slowdown of throughput. From the Figure 3.4 we can see that the slowdown of applications involving converting to tiff files have much smaller than those applications dealing with csv files. We may conclude this as the csv file format is not a good fit for the SGX environment, but this is incorrect.

The major cause of this is about the implementation of applications. We examine the source code of `tstcsv->csv`, `gotrackcsv->csv` and `plt->csv` and find that they are heavily relied on some libc functions, e.g., `strtok`, `strlen`, while those with low performance overhead do not have such reliance. These libc functions are even used for thousands of millions of times due to the size of data. So, they would generate thousands of millions of system calls. Since system calls cannot be executed within a SGX environment, it must be switched to normal environment and talks to the Linux kernel directly, therefor introducing a significant switching delay. SCONE tries to solve this using an asynchronous system call interface with a scheduling algorithm, but it still has a significant delay when such a large number

of system calls presents. Graphene-SGX does a relatively better job because it is a library OS that provides a shim layer and substitutes these system calls with its own implementations. However, no matter what kind of mechanism that a SGX platform utilized to reduce the switching delay, it still exists and sometimes can be a huge problem. This is a great reminder for SGX application developers that only invoke a system call if truly necessary when developing data integration applications.

3.6 Summary

This chapter discuss the performance of native Intel SGX and SGX platforms. We classify the causes into three categorizes: switching delay, computation related and memory access related. In general, the switching delay and memory access delay are noticeable and sometimes significant. We further evaluate performance of two applications and a library running in SCONE and Graphene-SGX, respectively. We then some discussions on what needed to be pay attention when building SGX applications.

Chapter 4

Securing the OpenCV using Intel SGX

This chapter will have discussion on securing real-world applications with the help of hardware-assisted security technologies such as Intel Software Guard Extensions (SGX). We choose Open Source Computer Vision Library (OpenCV) as an example, use the Secure Linux Containers with Intel SGX (SCONE) platform to secure the application entirely. We will also have a thorough analysis on the performance of OpenCV in trusted execution environment.

4.1 Introduction

This section will have discussion on the background of OpenCV and the motivation of protecting it against various types of attacks.

4.1.1 OpenCV

Open Source Computer Vision Library (OpenCV) is one of the most widely utilized computer vision library written in C++ programming language[5]. More than 2500 algorithms can be found in the OpenCV library, which includes a comprehensive list of classic and modern computer vision algorithms, and also, machine learning algorithms. All these algorithms can be applied to a variety of tasks, e.g., facial recognition, object identification and classification, image and video processing, and many more. OpenCV also has portability. Although written

in C++, it provides Python, Java and MATLAB interfaces which supports running on Windows, Linux, Mac OS and even Android. OpenCV aims to be a real-time vision library, so it also takes advantage of Matrix Math Extensions (MMX) and Stream SIMD Extensions (SSE) instruction sets when possible. OpenCV community is actively working on interfaces that support Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL)[34, 40].

4.1.2 Motivation

Nowadays, with the increasing size of image, video or other types of data that people want to process, local computational resources has become a bottleneck. People prefer to submit their data to servers with sufficient computation power. However, this can lead to a brand new problem: do we trust people behind those servers and do we trust their security mechanisms? Despite network security issues, data which uploaded by users stored in a server may or may not in an encrypted way, since encrypting data, especially a huge amount of data takes time and resources. This means sensitive data will eventually be transparent to administrators and can be accessed easily at some time, no matter what kinds of security features have been applied, because machine learning and image processing algorithms need honest input to process.

Because software and network vulnerabilities can always been identified in attack vectors, researchers and companies have tried to find solution to such problem in system level as well as hardware level. The hardware-assisted security features such as the Intel SGX and ARM TrustZone can resolve this issues in varying degrees. In terms of OpenCV, the Intel SGX is the better solutions over ARM TrustZone. It is not only because writing ARM TrustZone compatible applications is even more difficult than working with Intel SGX, but OpenCV is mostly used in desktop operating systems with a chip manufactured by Intel.

Although previous works shows that porting some non-trivial applications or libraries proved to be possible, working on writing SGX application can be tedious and challenging, especially for such a large and complicated library. Developers must have deep understanding of the whole application to partition the trusted and untrusted code. They also need to verify the SGX-enabled library have the identical behaviors to the original code. Currently, there is

no open source solution on partitioning applications or libraries automatically. Although we can borrow some thoughts and insights from Glamdring[26], this is still an open question. Therefore, using a SGX platform seems to be a simple and practical way. Secure Linux Containers with Intel SGX (SCONE) is a secure container mechanism for Docker that utilizes the Intel SGX to protect processes inside Docker containers being attacked from outside environment, and it supports unmodified programs running in containers as long as it can be built by its SCONE compilers. According to these features, using SCONE to secure the OpenCV library is feasible and perfectly meet our needs.

4.2 Technical Background

In this section we will first present the technologies that we use. We will discuss them one after another and then explain why these technologies can be used to secure the OpenCV, and potentially, other applications.

4.2.1 Intel SGX

As we mentioned in the previous section, the OpenCV is mostly deployed in desktop operating systems. Thus, to find a solution which can be applied to desktop operating systems is a must. Considering the market share of Intel's processors, the hardware-assisted security technology proposed by Intel - the Intel SGX can be a great candidate, obviously. However, there is still a long way from choosing it to actually using it. The SGX programming model requires developers to modify applications, partition them to trusted and untrusted part, this can be a great amount of work. Currently, the usability of Intel SGX, especially for a large-scale software is low, we may need to find a way of putting the whole application into the enclave, even it can enlarge the size of trusted computing base (TCB) greatly.

4.2.2 Linux Container

The concept of container is not a new thing. To resolve infamous software dependencies problem, several implementations of containers include LXC (Linux Containers), LXD (new generations of LXC), and Docker have been proposed. Most implementations use some the Linux kernel features such as namespaces and cgroup to provide OS-level virtualization, isolation and resource management. Among these implementations, the Docker is the de facato standard of the container. The introduction of the container not only resolves the software dependencies issue, but it also provides some security guarantees - processes running inside a container do not have chances of talking to outside environment directly unless we explicitly change the configurations, and they do not have unlimited computational resources by default. The container is also very light-weighted compared to traditional virtual machine (VM), since it does not need to boot up an entire operating system to start the application.

4.2.3 Approach: running in the SCONE

The Secure Linux Containers with Intel SGX (SCONE) is a secure container mechanism for Docker that makes the use of the Intel SGX. The benefits of using SCONE include small amount of work to put application in trusted execution environment, small TCB size, and acceptable performance overhead.

Figure 4.1 shows an overview of the SCONE design.

External interface In order to run unmodified applications inside the containers, the containers must have C standard library (libc) interface. And to reduce the TCB size of the SCONE, it uses Linux Kernel Library (LKL) and the musl libc library. However, system calls which invoked by libc is prohibited inside of an enclave, so the SCONE exposes an external interface to the host OS. To expose such things can be a problem, because host OS can initiate various attacks. To protect applications in the container, especially for those using unencrypted way of sending information, SCONE supports some shields. These shields include file system shield, network shield, and console shield.

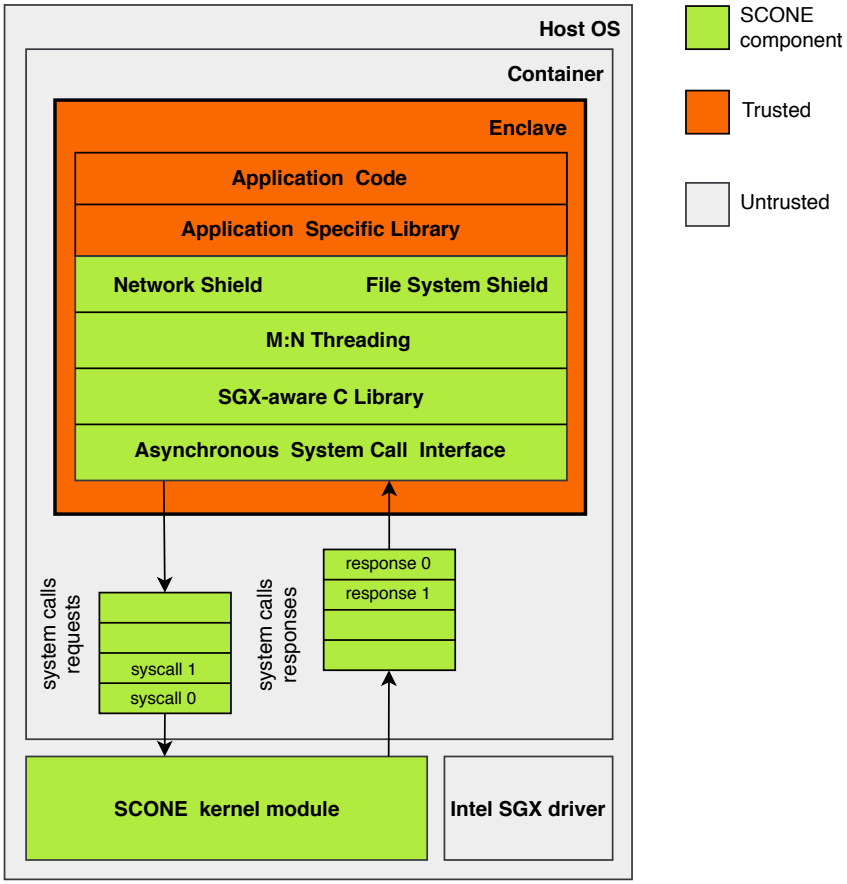


Figure 4.1: SCONE architecture

M:N threading and asynchronous system calls SCONE offers M:N threading mechanism to reduce the unnecessary transition from enclave to host OS, and vice versa. The scheduler of SCONE maps M application threads to N OS threads. Notice that these threads are all in the enclave since the entire application has been put in the trusted execution environment provided by the Intel SGX. The number of application threads (M) is a variable that can be a large number, while the number of OS threads (N) is typically set to the number of hardware threads that a machine can support. To avoid unnecessary transitions, the idea of SCONE scheduler is to maintain the "liveliness" of OS threads and forbid application threads talk to the Linux kernel individually and directly. OS threads in the enclave are man-in-the-middle and they served as an agency. They are extremely busy since they

are responsible for retrieving system call requests in the requests queue one after another. These requests are initiated by application threads, waiting to be forwarded to the SCONE kernel modules by OS threads. Whenever a response is available, it would be placed in a response queue so that an OS thread can retrieve and forward it to the target application thread. Both requests and responses queues are implemented in the way of shared memory.

Docker integration SCONE integrates with Docker, and to support full functionalities of SCONE, host OS needs a SCONE kernel module and a customized Docker engine. What is more, each container requires a startup configuration file (SCF) to start. The SCF is not included in the Docker image because SGX does not protect the confidentiality of code inside an enclave. Instead, it is sent to the enclave over the TLS secured channel after the enclave has been verified. The SCF contains some very important information, e.g, keys that used to encrypt the I/O stream, hash of some other protected files. Only after all of the security checks are passed that a SCONE container can start up.

4.3 Evaluation

This section is split into two parts. In the first part, we present the experiment setup of the SCONE version of the OpenCV as well as the Alpine version of the OpenCV. In the second part, we present the performance overview of the SCONE OpenCV and performance analysis of each OpenCV module.

4.3.1 Experiment Setup

To run a SCONE version of OpenCV, we must first launch a SCONE Docker container with a special musl libc compiler and build it from source. Since the image of SCONE is based on Alpine Linux, we may take some extra amount of work. One issue of the Alpine Linux is that many applications or libraries do not have official release or build instructions, which means we have to build them from scratch with great care. If these applications or libraries

happened to be one of the dependency of the target application, it will take a lot of time to finish these tedious work.

We chose OpenCV version 3.4.1, the host OS is Ubuntu 16.04 LTS and the Docker OS is alpine 3.7.3 for both SCONE version and Alpine version. All experiments use Intel NUC7i5BNH with an i5-7260U processor with 4 cores at 2.20 GHz with 4 MB cache and 12 GB main memory.

4.3.2 Performance Overview

Selected modules and their description are listed below:

- `core`: It defines basic data structures, including the dense multi-dimensional array Mat and many other basic functions which used by all other modules.
- `calib3d`: It includes basic multiple-view geometry algorithms, camera calibration and stereo correspondence algorithms. It also contains the elements of 3D reconstruction.
- `dnn`: It is the deep neural networks module of OpenCV.
- `features2d`: It is used for feature detection and description, descriptor matching, drawing function of keypoints and matches, and object categorization.
- `objdetect`: It is used to detect objects and instances of the predefined classes.
- `stitching`: It is used for rotation estimation, auto-calibration, image warping, seam estimation, exposure compensation and image blenders.

Among these modules, the `core` module is the most important one, because it contains some fundamental data structures and functions. Table 4.1 shows statistics of 39 over 127 unit tests of `core` module ranging from low to high overhead. The performance overhead is defined by equation 5.1.

$$Overhead = \frac{time(SCONE)}{time(Alpine)} \tag{4.1}$$

Unit Test	SCONE/ms	Alpine/ms	Overhead
OCL_UsageFlagsBoolFixture_UsageFlags_AllocHostMem	1119	1682	0.67
OCL_CopyToFixture_CopyToWithMaskUninit	4285	4057	1.06
Size_MatType_Mat_Clone_Roi	663	576	1.15
OCL_SqrtFixture_Sqrt	5539	4022	1.38
KMeans	81685	55992	1.46
OCL_MinFixture_Min	9557	6201	1.54
OCL_RepeatFixture_Repeat	5580	3322	1.68
OCL_PolarToCartFixture_PolarToCart	15945	9427	1.69
OCL_BitwiseNotFixture_Bitwise_not	9669	5425	1.78
Size_MatType_Flag_dct	134970	72429	1.86
OCL_LogFixture_Log	7276	3899	1.87
OCL_InRangeFixture_InRange	15345	8085	1.90
OCL_ScaleAddFixture_ScaleAdd	14598	7659	1.91
Size_MatType_bitwise_or	2665	1336	1.99
Size_MatType_ROp_reduceR	99518	49277	2.02
OCL_MixChannelsFixture_MixChannels	3500	1702	2.06
OCL_PSNRFixture_PSNR	1613	778	2.07
OCL_NormFixture_NormRel	11096	5198	2.13
Size_MatType_meanStdDev_mask	5248	2290	2.29
Size_MatType_mean	5196	2216	2.34
Size_MatType_CmpType_compareScalar	28549	12115	2.36
OCL_CompareFixture_CompareScalar	8184	3419	2.39
Size_MatType_Mat_Transform	21528	8956	2.40
Size_MatType_bitwise_and	3451	1427	2.42
OCL_UMatTest_CustomPtr	3319	1215	2.73
Size_MatType_max	4490	1639	2.74
Size_MatType_NormType_norm_mask	9608	3461	2.78
VectorLength_phase64f	1649	573	2.88
OCL_MeanStdDevFixture_MeanStdDev	7939	2632	3.02
Size_MatType_NormType_normalize	5299	1494	3.55
Size_MatType_minMaxLoc	6868	1805	3.80
Size_MatType_NormType_norm2	6434	1637	3.93
Size_MatType_CvRound_Float	2006	439	4.57
Size_MatType_NormType_norm	2688	555	4.84
Size_Mat_StrType_fs_text	7883337	1438876	5.48
Size_MatType_sum	1042	169	6.17
PerfHamming_norm	551	86	6.41
MatType_Length_dot	842	131	6.43
Size_Mat_StrType_fs_base64	2162298	257216	8.41

Table 4.1: Selected statistics of unit tests of `core` module

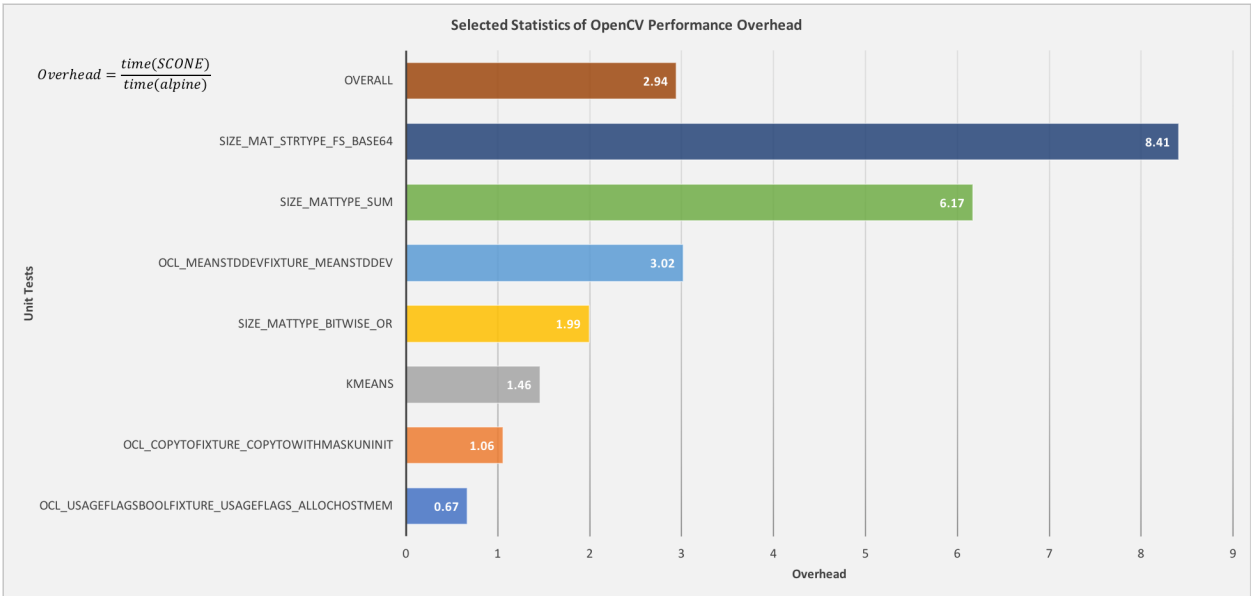


Figure 4.2: Selected statistics of unit tests of `core` module

As we can see in Table 4.1, the overhead of `OCL` series of unit tests are generally smaller than `Size_MatType` series of unit tests. The reason is because `Size_MatType` series unit tests usually involves more complicated operations (e.g., matrix arithmetic) than `OCL` unit tests, which can lead to even higher performance overhead in a SGX-enable environment.

Figure 4.2 shows some representative unit tests and their performance overhead. The lowest overhead is even less than 1, which means SGX environment has even better performance. However, this is the only case over 137 unit tests. Most performance overhead of unit tests are less than three times, which is relatively acceptable, and the overall number of 2.94 just reflects this fact. There are still some unit tests involves some complex asthmatics and have nearly intolerant performance. Some are five times, six times slower and `SIZE_MAT_STRTYPE_FS_BASE64` even shows the highest performance overhead at 8.41.

If we only have a look at the `core` module, the general overhead 2.94 is acceptable but not satisfiable. However, other modules shows better performance. Figure 4.3 shows statistics of other modules. The highest performance overhead is 1.40 in `calib3d` module and the lowest overhead is only at 1.11 in `objdetect` module, which is actually a great number. The reason why performance overhead of other modules is much lower than the `core` module is because

other modules only involves small amount of complicated operations and these operations are mostly resulted in using some data structures or functions from the `core` module.

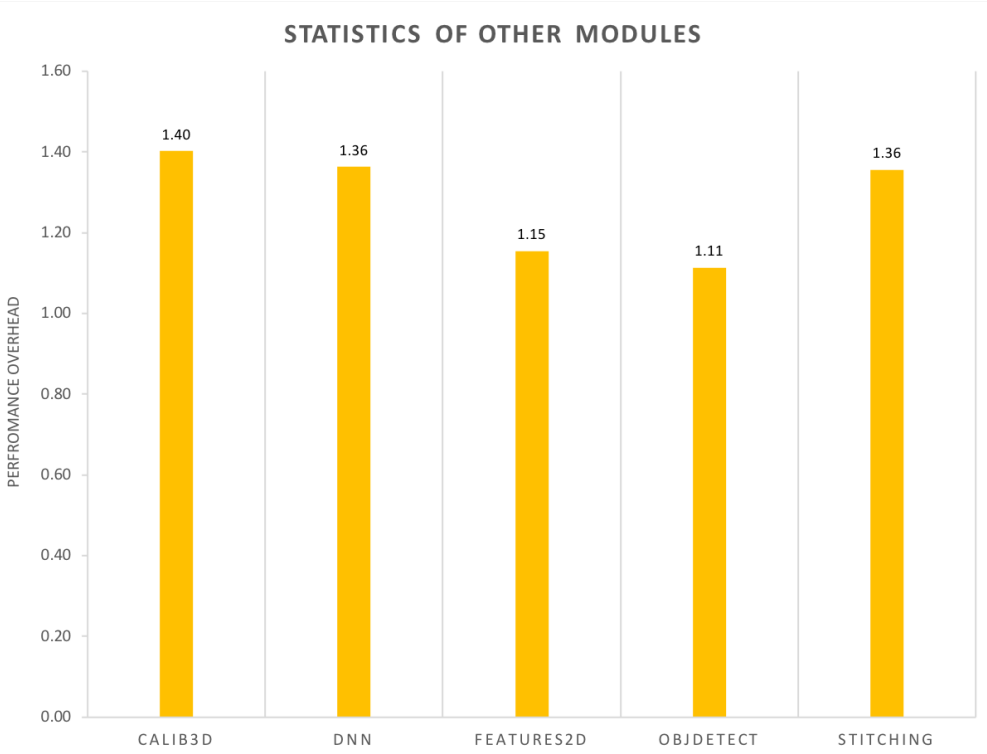


Figure 4.3: Statistics of other selected modules

4.4 Summary

In this chapter, we show that securing a unmodified real-world application such as OpenCV and execute in a SGX environment is possible. The platform that we used is Secure Linux Containers with Intel SGX (SCONE). We also evaluated the performance overhead of selected OpenCV modules running in a SGX environment. We found the overhead of `core` module is relatively higher at 2.94, while the overhead of other utility modules are ranged from 1.11 to 1.40, which are small number.

References

[1] Ross Anderson. Cryptography and competition policy: issues with'trusted computing'. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 3–10, 2003.

[2] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 689–703, Savannah, GA, November 2016. USENIX Association.

[3] Will Arthur and David Challener. *A Practical Guide to TPM 2.0: Using the Trusted Platform Module in the New Age of Security*. Apress, 2015.

[4] ayeks. Sgx hardware list github repo. `https://github.com/ayeks/SGX-hardware`.

[5] Gary Bradski and Adrian Kaehler. Opencv. *Dr. Dobb's journal of software tools*, 3, 2000.

[6] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, Vancouver, BC, August 2017. USENIX Association.

[7] Anthony M Cabrera, Clayton J Faber, Kyle Cepeda, Robert Derber, Cooper Epstein, Jason Zheng, Ron K Cytron, and Roger D Chamberlain. Dibs: A data integration benchmark suite. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, pages 25–28, 2018.

[8] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM transactions on intelligent systems and technology (TIST)*, 2(3):1–27, 2011.

[9] Chia che Tsai, Donald E. Porter, and Mona Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 645–658, Santa Clara, CA, July 2017. USENIX Association.

[10] Intel Corporation. Intel sgx linux driver github repo. `https://github.com/intel/linux-sgx-driver`.

[11] Intel Corporation. Intel sgx platform software github repo. https://github.com/intel/linux-sgx/tree/master/psw.

[12] Intel Corporation. Intel sgx software development kit github repo. https://github.com/intel/linux-sgx/tree/master/sdk.

[13] Intel Corporation. *Intel® Software Guard Extensions (Intel® SGX) Developer Guide*.

[14] Intel Corporation. Intel® software guard extensions ssl. https://github.com/intel/intel-sgx-ssl.

[15] Intel Corporation. Intel sgx sdk edger8r github repo. https://github.com/intel/linux-sgx/tree/master/sdk/edger8r/linux, 2020.

[16] Victor Costan and Srinivas Devadas. Intel sgx explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[17] International Organization for Standardization/International Electrotechnical Commission et al. Information technology—trusted platform module—part 1: Overview. *International Standard, ISO/IEC*, pages 11889–1.

[18] OpenSSL Software Foundation. Openssl. https://www.openssl.org.

[19] Christian Göttel, Rafael Pires, Isabelly Rocha, Sébastien Vaucher, Pascal Felber, Marcelo Pasin, and Valerio Schiavoni. Security, performance and energy trade-offs of hardware-assisted memory protection mechanisms. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 133–142. IEEE, 2018.

[20] Trusted Computing Group. https://trustedcomputinggroup.org.

[21] Inc Hex Five Security. The first tee for risc-v. https://hex-five.com/multizone-security-sdk.

[22] Prerit Jain, Soham Jayesh Desai, Ming-Wei Shih, Taesoo Kim, Seong Min Kim, Jae-Hyuk Lee, Changho Choi, Youjung Shin, Brent Byunghoon Kang, and Dongsu Han. Opensgx: An open platform for sgx research. In *NDSS*, 2016.

[23] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[24] Hisashi Kotakemori, Hidehiko Hasegawa, Tamito Kajiyama, Akira Nukada, Reiji Suda, and Akira Nishida. Performance evaluation of parallel sparse matrix–vector products on sgi altix3700. In *International Workshop on OpenMP*, pages 153–163. Springer, 2005.

[25] Hisashi Kotakemori, Hidehiko Hasegawa, and Akira Nishida. Performance evaluation of a parallel iterative method library using openmp. In *Eighth International Conference on High-Performance Computing in Asia-Pacific Region (HPCASIA'05)*, pages 5–pp. IEEE, 2005.

[26] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, et al. Glamdring: Automatic application partitioning for intel {SGX}. In *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*, pages 285–298, 2017.

[27] Arm Ltd. Arm security technology. *White paper*.

[28] Bishop Matt et al. *Introduction to computer security*. Pearson Education India, 2006.

[29] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos 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*, pages 1–9. 2016.

[30] Chris Mitchell. *Trusted computing*, volume 6. Iet, 2005.

[31] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How sgx amplifies the power of cache attacks. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 69–90. Springer, 2017.

[32] David Moore, Colleen Shannon, Douglas J Brown, Geoffrey M Voelker, and Stefan Savage. Inferring internet denial-of-service activity. *ACM Transactions on Computer Systems (TOCS)*, 24(2):115–139, 2006.

[33] Akira Nishida. Experience in developing an open source scalable software infrastructure in japan. In *International Conference on Computational Science and Its Applications*, pages 448–462. Springer, 2010.

[34] CUDA Nvidia. Compute unified device architecture programming guide. 2007.

[35] Gavin O'Gorman and Geoff McDonald. *Ransomware: A growing menace*. Symantec Corporation, 2012.

[36] Meni Orenbach, Pavel Lifshits, Marina Minkin, and Mark Silberstein. Eleos: Exitless os services for sgx enclaves. In *Proceedings of the Twelfth European Conference on Computer Systems*, pages 238–253, 2017.

[37] Graeme Proudler, LQ Chen, and Chris Dalton. *Trusted Computing Platforms*. Springer, 2014.

[38] Akihiro Pujii, Akira Nishida, and Yoshio Oyanagi. Evaluation of parallel aggregate creation orders: Smoothed aggregation algebraic multigrid method. In *High Performance Computational Science and Engineering*, pages 99–122. Springer, 2005.

[39] Mohamed Sabt, Mohammed Achemlal, and Abdelmadjid Bouabdallah. Trusted execution environment: what it is, and what it is not. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 1, pages 57–64. IEEE, 2015.

[40] John E Stone, David Gohara, and Guochun Shi. Opencl: A parallel programming standard for heterogeneous computing systems. *Computing in science & engineering*, 12(3):66–73, 2010.

[41] Sébastien Vaucher, Valerio Schiavoni, and Pascal Felber. Stress-sgx: Load and stress your enclaves for fun and profit. *arXiv preprint arXiv:1906.11204*, 2019.

[42] C. Zhao, D. Saifuding, H. Tian, Y. Zhang, and C. Xing. On the performance of intel sgx. In *2016 13th Web Information Systems and Applications Conference (WISA)*, pages 184–187, 2016.