Department of Computer Science

University of Hong Kong

Final Year Project

Analyzing and Improving the Performance of SGX

Author: Fung Yuk Leung

Date: March 31, 2019

Issue: 2.0

Distributions: Fung Yuk Leung (FYP student)
Dr. Cui H.M. (FYP supervisor)

Change History

Issue	Date	Description	Author
1.0	15-March-2019	First issue	Fung Yuk Leung
2.0	6-April-2019	1. Changed format 2. More literature reviews 3. Reordered content 4. Improved Abstract, Introduction, Results and Improvements	Fung Yuk Leung

ABSTRACT

Software Guard Extensions (SGX) is an Intel product that provides strong security guaranty (including confidentiality and integrity) for confidential data. It makes use of a processor-hardened container, called *enclave*, to isolate protected data from external environment. However, it results in performance degradation (lager latency) at the same time because it requires extra running time to initiate and maintain the *enclave*. This project used the method of control experiment, to analyze the performance of SGX in micro/basic operations (such as *ecall* and *ocall*) using the latest version of SGX Software Development Kit (SDK). This project investigated totally three kinds of overheads that are caused by applying SGX, which are switching, data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves*. With the experiment environment of Intel Core I7-6700k 4GHz with 4 hyper-threaded cores and an operating system of Ubuntu 14.04 LTS, these three kinds of overheads consume 13473, 2537, 755 instruction cycles respectively. These obtained statistics can help identify the performance overhead caused by SGX. Basing on these findings, this project then applied two improvements on *SecureKeeper* (*ZooKeeper* with SGX enabled) with the aim of reducing the above three kinds of overheads. Specifically, this project applied a heuristic function to predict required sizes of buffer in calling *enclave* functions. The overhead of data transferring between *enclaves* and untrusted environment was reduced by this improvement. Additionally, this project made use of smaller size of *enclave* to reduce the overhead of memory access inside *enclaves*. However, the overhead of switching cannot be reduced because the frequency of switching in *SecureKeeper* is already minimized. As a result, the above improvements improved the overall performance (latency) by 3.5%. This can give inspirations to programmers on how to improve the performance of SGX with optimal design and implementation of program/software.

ACKNOWLEDGMENT

This project is supported/partially supported by Dr. Cui H.M and his PhD student, Mr. Jianyu Jiang. I thank them for providing insight and expertise to help me set up scientific experiments so that I can get concrete statistics to analyze the performance of Software Guard Extensions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABBREVIATIONS

SGX:	Software Guard Extensions
SDK:	Software Development Kit

1. INTRODUCTION

1.1 Background

Cloud computing (*e.g.,* Google Cloud Platform) is the on-demand delivery of hardware, software and other IT resources through a cloud services platform via the internet. It is highly popularized and commercialized nowadays because of its advantage of lower cost of computation and storage. In cloud computing, users send their data to service providers, who process these data on behalf of users and then return corresponding results. A problem arises in such a model, which is the leakage of data due to irresponsible or untrusted service providers.

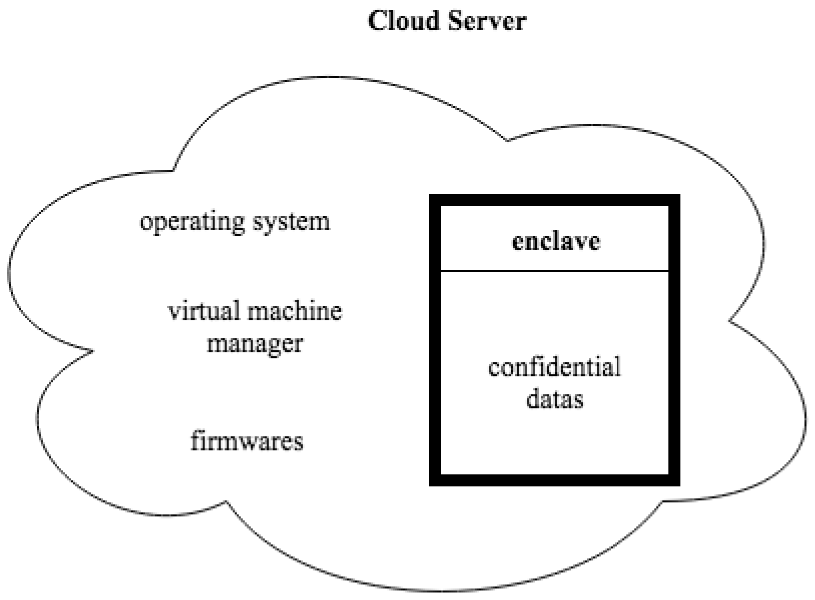


Figure 1. SGX creates a trusted zone, called *enclave*, to isolate confidential data from untrusted environment, including operating system, virtual machine manager and firmwares.

Software Guard Extensions (SGX), is invented to solve this problem. On the cloud server side, SGX creates a processor-hardened container, called *enclave*, to isolate confidential data from untrusted environment (see Figure 1) [2]. Additionally, the manufacturer of SGX works as a trusted intermediary agent between users and service providers [7]. As Figure 2 shows, it measures the identity (integrity) of the executing *enclave* in the execution environments using *software attestation* (a technique for attesting authority), and then sends the corresponding results to users so that they can know whether they are communicating with a safe and trusted device.

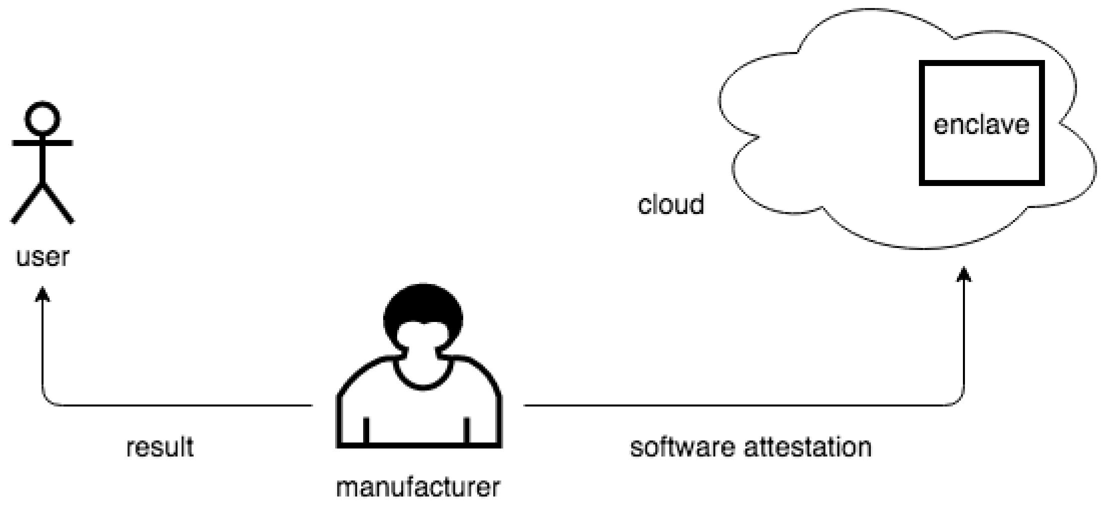


Figure 2. The manufacturer of SGX works as a trusted intermediary agent between users and service providers.

.

1.2 Major concern and existing studies

While providing strong security guarantee, SGX leads to a concern about performance. SGX is completely implemented with *microcodes* (the lowest specified level of processor and machine instructions sets) [7]. Running these *microcodes* incurs higher overhead and consumes more *instruction cycles* (the basic operational process of a computer system) than non-*enclave* execution. Thus, SGX results in lower performance than regular programs [6]. Specifically, while SGX creates an *enclave* to protect confidential data, it divides a program into two parts. One part containing secret data and some important functions is placed inside the *enclave*, another part containing regular functions is placed outside the *enclave*. However, functions in a same program interact with each other regularly and hence they need to cross the border of *enclave*. Crossing the border involves flush of Translation Lookaside Buffer (TLB) and data transferring between *enclaves* and the external environment, which requires extra running time and hence it results in performance degradation.

Although there are some existing studies analyzing performance of SGX, they are contradictory with each other. For example, [4] investigated that programs using SGX ran 55% slower in *mcf* (a kind of memory access pattern) while [5] claimed that they ran just 12% slower. This gap implies that more researches are needed to prove which one is accurate. In addition, existing studies (such as [4] and [6]) measured the performance of SGX with old versions of Software Development Kit (SDK). However, Intel launched a latest version, 2.3.101, on October 18th, 2018. With new version of SDK, the performance of SGX might improve. Given the above two reasons, there is a demand for new researches on latest version.

1.3 Scope and contributions of the project

This project focuses on the performance (especially the latency) of SGX in micro/basic operations. Specifically, it aims at investigating the performance overhead of programs/software applying SGX. The second focus of this project is to improve the performance of SGX. There are two ways to achieve this aim. The first one is to modify the design and implementation of SGX, which results in a better version of SGX. The second one is to optimize the design and implementation of the target program/software so that it is well fitted with SGX. However, this project concentrates on only the second method. Specifically, it tries to suggest any feasible improvement on the target program/software to reduce those overheads identified in the first focus, so that the performance of SGX is enhanced.

This project makes two contributions. Firstly, it investigated totally three kinds of overheads that are caused by applying SGX, which are switching, data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves*. Concrete statistics are provided to describe the effect of these overheads. Comparing to existing studies, these statistics provide latest analysis on latest version of SGX SDK. Also, they help solve the contradiction between existing studies as mentioned before. Secondly, this project used *SecureKeeper* (*ZooKeeper* with SGX enabled; *ZooKeeper* is an open-source software for maintaining configuration information and providing distributed synchronization and group services) as an example, to show methods of reducing the previously mentioned overheads in order to improve the performance of SGX.

1.4 Outline of this report

This report lists the two objectives of this project in Section 2. Then it explains how to carry out experiments for performance analysis in Section 4. The difficulties of these experiments are described in Section 5. Section 6 of this report is the results and findings of the experiments in Section 4. Next, Section 7 mentions the limitation of the findings in Section 6. Based on these obtained findings, this report suggests two improvements to increase the performance of SGX in Section 8.

2. OBJECTIVES

The first objective of this project is to investigate concrete statistics to describe the effect of overheads that are caused by applying SGX, by means of investigating the performance (latency) of SGX in micro/basic operations.

The second objective of this project is to suggest any feasible improvement on the target program/software (*SecureKeeper*) to make it be well fitted with SGX, so that the overhead identified in the first objective is reduced and hence the performance of SGX can be heightened.

3. LITERATURE REVIEW

3.1 Overheads caused by SGX

[4] and [7] pointed out three kinds of overheads that are caused by applying SGX, which are switching, data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves*.

3.1.1 switching

Switching to/switching out of SGX mode is triggered when dealing with confidential data inside the *enclave* (other modes have no right to process data inside the *enclave*). It results in an overhead because it requires the computer to flush TCB during switching, which consumes some running time [7].

3.1.2 data transferring between *enclaves* and untrusted environment

Passing data into/extract data out of SGX *enclaves* is triggered by the data interactions between the *enclave* and the external environment. Data cannot pass the border of *enclave* directly. Instead, the transportation of data is done by creating a new copy of the data on another side of *enclaves* [4]&[7]. For example, when some data is passed from the external environment to an *enclave*, SGX first allocates new memory space inside the *enclave* and then copy the data into it, which costs extra time.

3.1.3 memory access inside *enclaves*

SGX maintains an integrity tree to ensure confidentiality of data, integrity of data, and anti-roll-back protections [4]. Accessing (read/write) memory inside the *enclave* needs to go through/modify the tree, which requires some running time.

With the inspiration of this, this project conducted experiments in Section 6 to investigate how these three kinds of overheads affect the performance of SGX and hence the latency of the whole program/software.

3.2 Performance analysis in actual applications

[6] suggested that the throughput of *ZooKeeper* applying SGX dropped by 11% comparing to the one of original version of *ZooKeeper*, while providing stronger security guarantee. [4] applied SGX to three applications, which are *OpenVPN*, *Memcached* and *Lighttpd* (OpenVPN is an open-source software for encryption in Virtual Private Network; *Memcached* is a key-value database; *Lighttpd* is a light-weight web server). It investigated that the latencies of these software with SGX enabled increased by 220%, 370% and 420% respectively. With this inspiration, this project analyzed and compared the latencies of *SecureKeeper* (*ZooKeeper* with SGX enabled) and improved *SecureKeeper* (which is implemented by this project) to prove that those improvements suggested by this project are effective to enhance the performance of SGX.

3.3 Improvements

As mentioned before, there are two ways to improve the performance of SGX. One is to modify the design and implementation of SGX, which results in a better version of SGX. Following this principle, [4] constructed an new architecture for SGX, which consists of a *requester*, a *responder* and a communication channel using un-encrypted shared memory. They worked together to buffer and poll for new messages, which are function calls that cross the border of an *enclave*. As a result, software context switch was avoided and hence it led to a 13-17x speed up over the original interface. Although this approach is out of the scope of this project (as explained in Section 1.3), it gives a general and useful idea to this project to enhance the performance of SGX, which is reducing the frequency of switching.

The second way of improving the performance of SGX is to optimize the design and implementation of the target program/software so that it is well fitted with SGX. [3] suggested several general and intuitive ideas for achieving this aim:

a. Use batch calls to replace short identical successive calls so that the frequency of switching decreases;
b. Merge short different successive calls into one so that the frequency of switching decreases;
c. Reduce memory usage to avoid SGX paging so that the overhead of memory access inside *enclaves* decreases.

In conclusion, [3] pointed out that the frequency of switching and memory usage inside the *enclave* should be minimized to optimally apply SGX and hence to improve the performance of SGX. These ideas give some guidance to this project to optimize the design and implementation of *SecureKeeper* in Section 8.

3.4 From *ZooKeeper* to *SecureKeeper*

ZooKeeper is an open-source software for maintaining configuration information and providing distributed synchronization and group services. It provides users with functions of creating, getting, setting, deleting data on cloud servers. Therefore, it can be regarded as a kind of cloud storage.

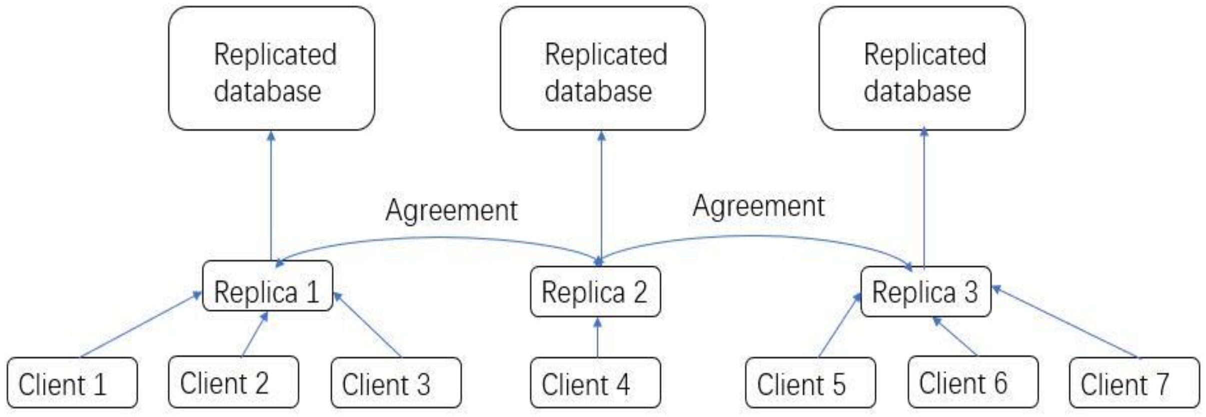


Figure 3. The architecture of *ZooKeeper*.

Figure 3 shows the architecture of *ZooKeeper*. Clients are directly connected to one replica, which is a processor handling requests from clients. At the same time, a replica is also connected to a replicated database so that it can access (read or write) data in the database and process responses from the database. Besides connecting with clients and replicated databases, replicas also connect with peer replicas and have agreements with each other so that they can cooperate and serve different clients at the same time while ensuring consistency and synchronization of data inside replicated databases.

Since replicas are processors of requests from clients and responses form databases, messages and data must be decrypted inside replicas so that they are understandable to replicas. However, these replicas do not provide sufficiently strong security guarantee for data. Thus, those decrypted messages and data might be exposed to hackers if they can control a replica in some threat models.

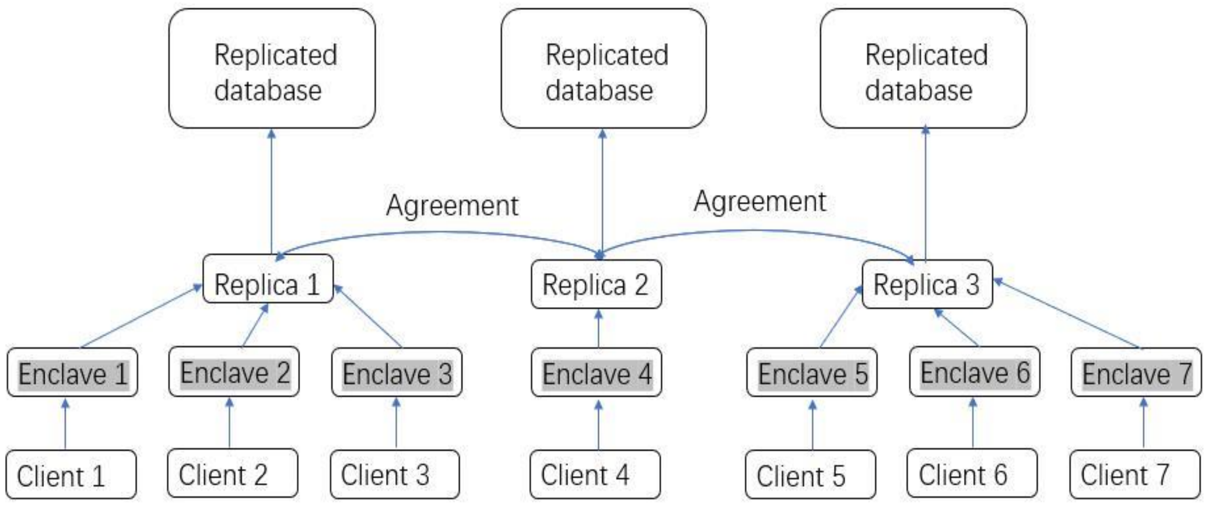


Figure 4. The architecture of *SecureKeeper*.

To solve the above problem and enhance the security level of *ZooKeeper*, [6] applied SGX technique to *ZooKeeper* in order to develop a new version of it, called *SecureKeeper*. As Figure 4 shows, [6] added an *enclave*, which is implemented using SGX, between each client and replica. The *enclave* can check the authorization of a client and block any communication if the client is regarded to be malicious. The most significant modification is that the main processing logic is moved from replicas to the *enclave*. As mentioned before, the *enclave* can provide confidentiality, integrity and anti-roll-back protections for sensitive data and it can prevent any illegal external accesses, even if they come from the operating system. It is regarded to be impossible to hack the *enclave* because it is processor-hardened. Thus, messages and data inside the *enclave* are guaranteed to be safe even though they are decrypted. In addition, the *enclave* also re-encrypts messages and data, after processing and before sending them to a replica or a client. In this case, hackers cannot know about the original messages or data even they have full control of replicas or clients or even replicated databases because they can only get encrypted messages or data, which are not understandable.

In fact, the actual architecture of *SecureKeeper* is more complicated than what Figures 4 shows. For example, there is another type of *enclave*, called central *enclave* inside a replica for handling accesses of sequential nodes. However, those improvements suggested by this project in Section 8 are based on the above architecture and they focus on only the *enclaves* between replicas and clients. Therefore, that extra information is irrelevant to this project and hence they are not shown or described in this report. The architecture shown in Figure 4 is sufficient to provide necessary background information for understanding those improvements.

4. METHODOLOGY

4.1 Control experiment

This project used the method of control experiment. There were one control group of programs and at least one experiment group of programs for each separate control experiment in this project. The details of each experiment, including what the independent variable is, the number of groups and the differences between control groups and experiment groups, are described in Section 6. By comparing the results of control group and experiment group, the influence of independent variable on the dependent variable is revealed.

4.2 Statistical method

In this project, each program in one experiment was run for 10, 000 times to reduce the effect of accidental factors so that residuals are minimized. One example of accidental factors is context switch to the operating system, which is discussed in the following section, Section 5. Since each program was run for 10, 000 times, there were 10, 000 different numbers. In this situation, their median was chosen to become the final result. Median instead of mean was selected because median can get rid of extreme values but mean is influenced by extremely small or extremely large values. Thus, median is more persuasive and convincing than mean.

4.3 Timer

For each execution of a program, the *real time stamp counter* (RDTSCP, an instruction to obtain the current cycle) was used as a timer. With one RDTSCP at the beginning and another one RDTSCP at the end of a program, the running time (latency, in unit of instruction cycles) of a program is obtained by the difference between these two RDTSCPs. RDTSCP is selected because it is the most precise way to measure the running time.

4.4 Experiment environment

All programs in this project were run in an environment with Intel Core I7-6700k 3.4GHz with 4 hyper-threaded cores, disabled dynamic frequency and voltage scaling, an operating system of Ubuntu 14.04 LTS, 16GB DDR4 RAM@2133MHz, 256KB L1-cache, 1MB L2-cahce and 8MB

L3-cache. I7-6700k was selected because only 6th (or later) generation of Intel Cores support SGX [2].

5. DIFFICULTIES

When executing programs in experiments, there is a certain porbability that context switch to the operating system (an unpredictable event triggered by accidents) happens, which contaminates the measurement. If this kind of event happens during experiments, the running time is longer than the actual one. For solution, the negative influence can be minimized by running each program for 10, 000 times. This is because context switch to the operating system happens rarely, so the majority of 10, 000 executions are not affected and hence the negative influence is reduced and becomes negligible.

There is a worse case that context switch to the operating system occurs when the computer is running in SGX mode. This result in asynchronous Exit (AEX), which is much more expensive than normal context switch. For solution, that particular execution must be discarded to ensure the accuracy of performance estimation.

6. EXPERIMENT DETAILS AND FINDINGS

According to Section 3, literature review, there are totally three kinds of overheads that are caused by applying SGX, which are switching, data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves*. Therefore, this project conducted one control experiment for each kind of overhead in order to investigate concrete statistics that describe its effect on the performance (latency) of programs/software applying SGX. For each separate control experiment, the details of experiment settings and findings are explained in each of the following sub-sections.

All experiments in this project involve *ecall* or *ocall*. *Eacll* is to call a function, which is defined inside an *enclave*, from untrusted environment (outside the *enclave*). On the contrary, *ocall* is to call a function, which is defined outside the *enclave*, from inner *enclave*. If data transferring between *enclaves* and untrusted environment is involved in *ecall* or *ocall* (in section 6.2), there are totally four options to pass the data, which are *user_check*, *in*, *out*, and *in&out*. *User_check* is a special case because there is no copying involved and hence there is no overhead for both *ecall* with option *user_check* and *ocall* with option *user_check* when passing the data. Therefore, this project did not analyze this option.

6.1 Switching

Figure 5 shows the work flow of experiment group in analyzing the overhead of switching to SGX mode. First, the timer starts. Then the program calls a function, which is defined inside an SGX *enclave*, through an *ecall*. At this moment, the computer switches to the SGX mode from user mode. The called function has no parameter (except the id of *enclave*), does nothing and returned immediately. Then the computer switches back to original mode. Finally, the timer ends. In this situation, the difference between the two timers, which is the result (27556 instruction cycles), is the total running time of the timer itself, a function call and two times of switching (switching into *enclave* and switching out of *enclave*). In theory, it is better if the timer ends inside the *enclave* instead of in the main function. In this case, the running time of switching is counted only once (only switching into the *enclave*). However, the timer is not allowed to run inside the *enclave*. Therefore, this project can only make the timer end in the main function and count the running time of switching twice (switching into *enclave* and switching out of *enclave*).

Figure 7 shows the content of enclave definition language (EDL) file of the experiment group. It defines the interface between the *enclave* and untrusted environment. As mentioned before, it has no parameter (except the id of *enclave*) and hence involves no data transferring. Thus, there is no need to specify an option (*user_check*, *in*, *out*, or *in&out*. *User_check*).

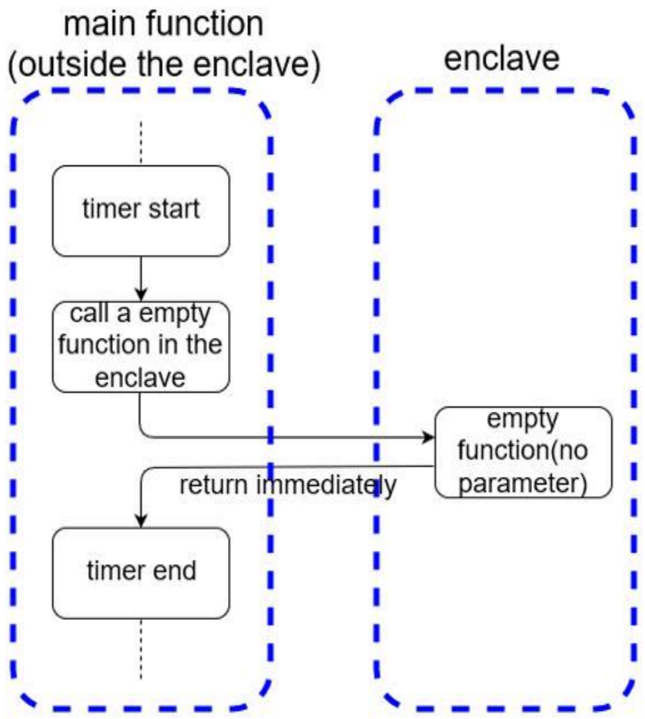


Figure 5. The program of experiment group in analyzing the overhead of switching to SGX mode.

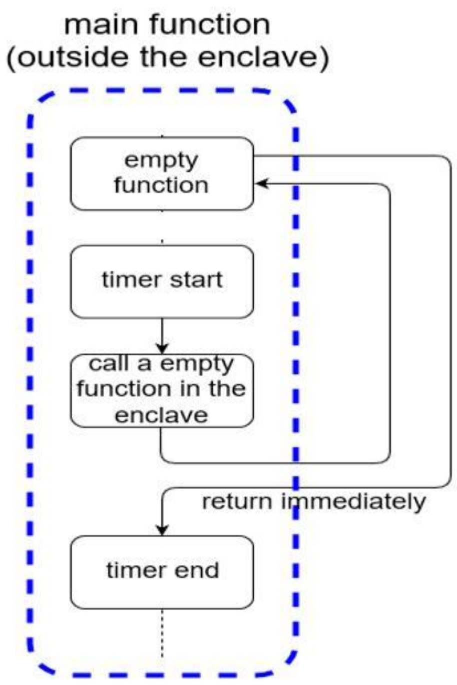
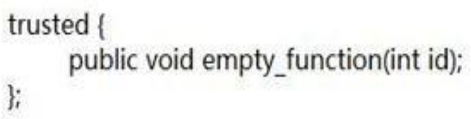


Figure 6. The program of control group in analyzing the overhead of switching to SGX mode.



```
trusted {
        public void empty_function(int id);
};
```

Figure 7. The *.edl* file of experiment group in analyzing the overhead of switching to SGX mode.

The result of experiment group includes not only the running time of two times of switching, but also the running time of the timer and a function call, which is not in interest. Thus, a control group is required. Figure 6 shows the work flow of the control group. It is the same as the one of experiment group except that the function called in control group is not defined inside the *enclave*. Thus, there is no switching to SGX mode or switching out of SGX mode when it is called. In this situation, the result of control group (610 instruction cycles) is the running time of the timer and a function call. Then the result of experiment group subtracted by the result of control group is the running time of two times of switching (27556 – 610 = 26946 instruction cycles) and hence the running time of one switching is 13473 instruction cycles.

6.2 Data transferring between *enclaves* and untrusted environment

6.2.1 data transferring using *ecall*

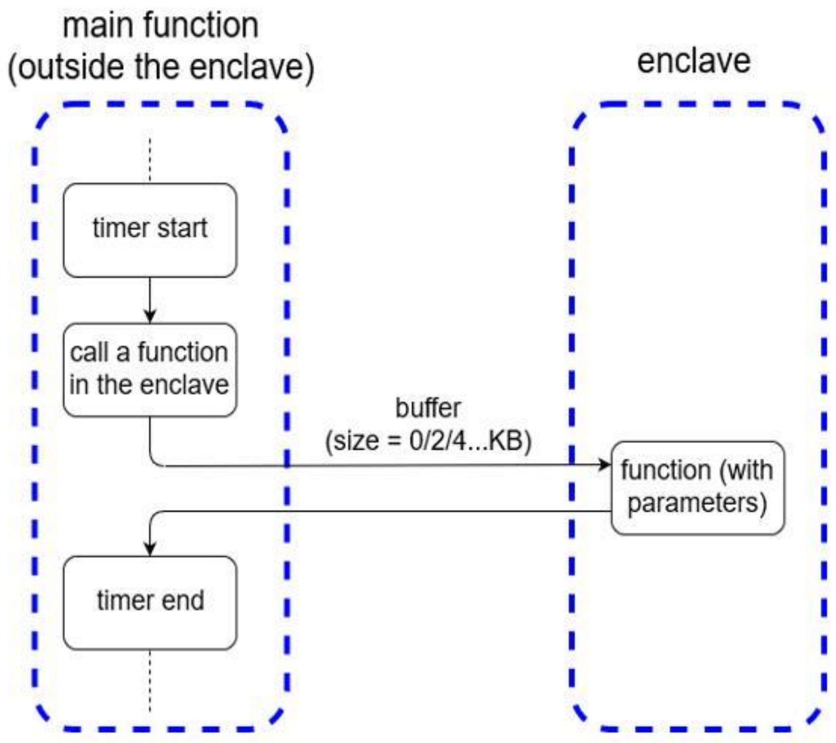
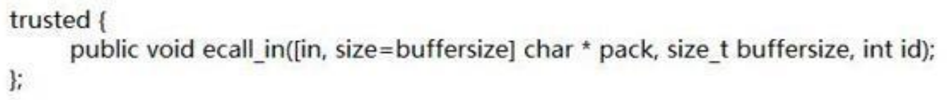


Figure 8. The experiment in analyzing the overhead of *ecall* with option *in.*

Figure 8 shows the work flow of experiment in analyzing the overhead of data transferring using *ecall* with option *in*. It is similar with the one in Figure 5 except that there is data transferring using *ecall* involved. There are several groups of programs in this experiment. Different groups have different sizes of data to transfer. For example, the first group passes a 0-KB buffer of data while the second group passes a 2-KB buffer of data into the *enclave*. Since the only difference between different groups is the size of data, the difference of results between different groups is caused by the difference of sizes of data. Thus, the total running time is a function of the size of data. With similar experiments, relationship between the total running time and the size of data in *ecall* with option *out* and *ecall* with option *in&out* are also obtained.



```
trusted {
        public void ecall_in([in, size=buffersize] char * pack, size_t buffersize, int id);
};
```

Figure 9. The *.edl* file of experiment group in analyzing the overhead of data transferring between enclaves and untrusted environment using *ecall* with option *in*.

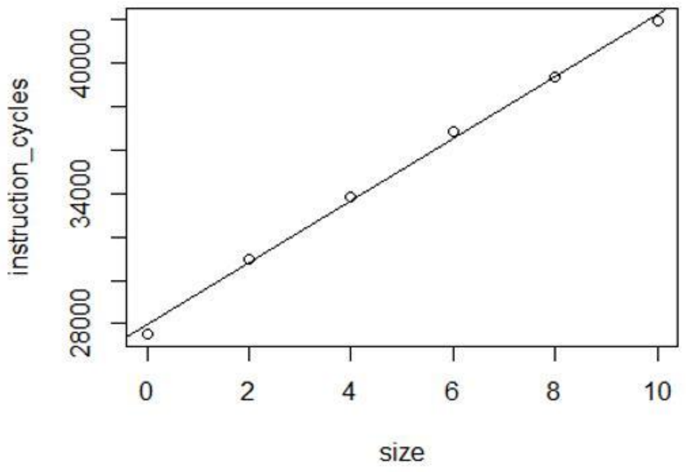
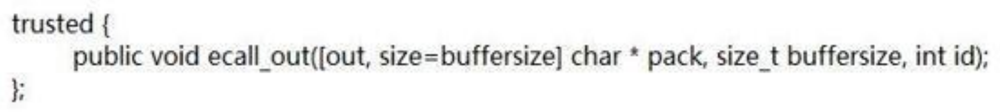


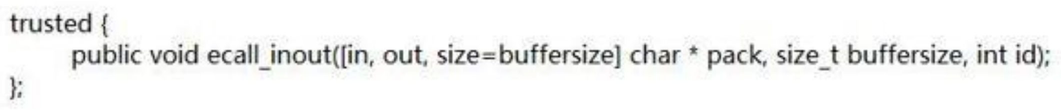
Figure 10. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ecall* with option *in*.

Figure 9 shows the content of EDL file of those groups in analyzing the overhead of data transferring using *ecall* with option *in*. It defines the interface between *enclaves* and trusted environment. The variable ‘pack’ is a buffer of data for tansferring, the variable ‘buffersize’ is the size of data. Data transferring using *ecall* is performed when the function ‘ecall_in’ is called. Different groups in this experiment use the same EDL file as the one in Figure 9, except that they set the variable ‘buffersize’ to be different values so that different groups have different size of data transferring. Figure 10 shows the obtained results, which are the latencies of different groups. The intercept and slope of the straightline in Figure 10 are 27937 and 1429 respectively. Thus, a mathematic function is obtained: total running time = 1429 * size of data + 27937 (in unit of instruction cycles), which implies that it consumes 1429 instruction cycles to transfer every 1-KB data using *ecall* with option *in*.



```
trusted {
        public void ecall_out([out, size=buffersize] char * pack, size_t buffersize, int id);
};
```

Figure 11. The *.edl* file of experiment group in analyzing the overhead of data transferring between enclaves and untrusted environment using *ecall* with option *out*.



```
trusted {
        public void ecall_inout([in, out, size=buffersize] char * pack, size_t buffersize, int id);
};
```

Figure 12. The *.edl* file of experiment group in analyzing the overhead of data transferring between enclaves and untrusted environment using *ecall* with option *in&out*.

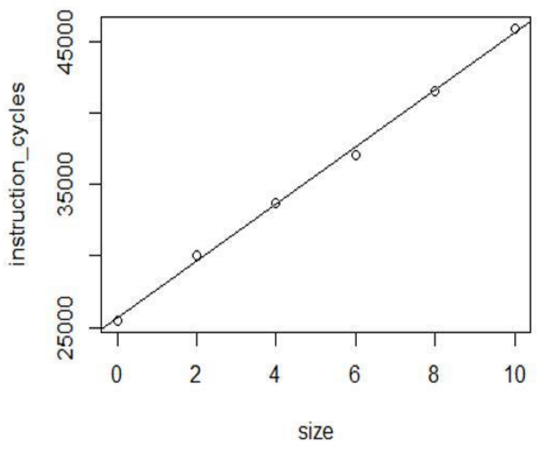


Figure 13. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ecall* with option *out*.

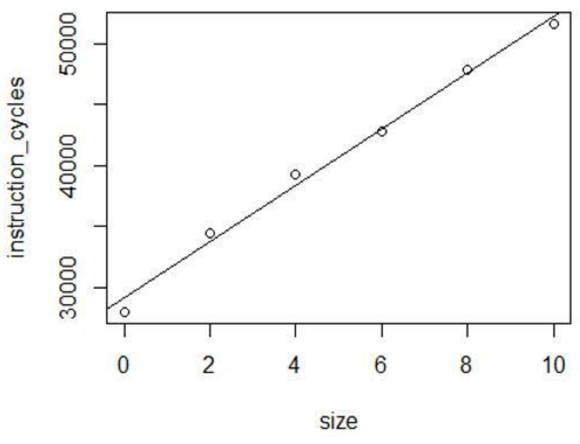


Figure 14. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ecall* with option *in&out*.

Experiments in analyzing overheads of data transferring using *ecall* with option *out* and *ecall* with option *in&out* have the same work flows as the one in Figure 8, except that their directions of data transferring are different. The contents of their EDL files are shown in Figure 11 and Figure 12 respectively. They are similar with the one in Figure 9 except that they use different options. Figure 13 and Figure 14 are the corresponding results of data transferring using *ecall* with option *out*, and *ecall* with option *in&out* respectively. Similar with the previous explanation, slopes of straightlines in Figure 13 and Figure 14 show that it consumes 1995 instruction cycles to transfer every 1-KB data using *ecall* with option *out*, and that it consumes 2309 instruction cycles to transfer every 1-KB data using *ecall* with option *in&out*.

6.2.1 data transferring using *ocall*

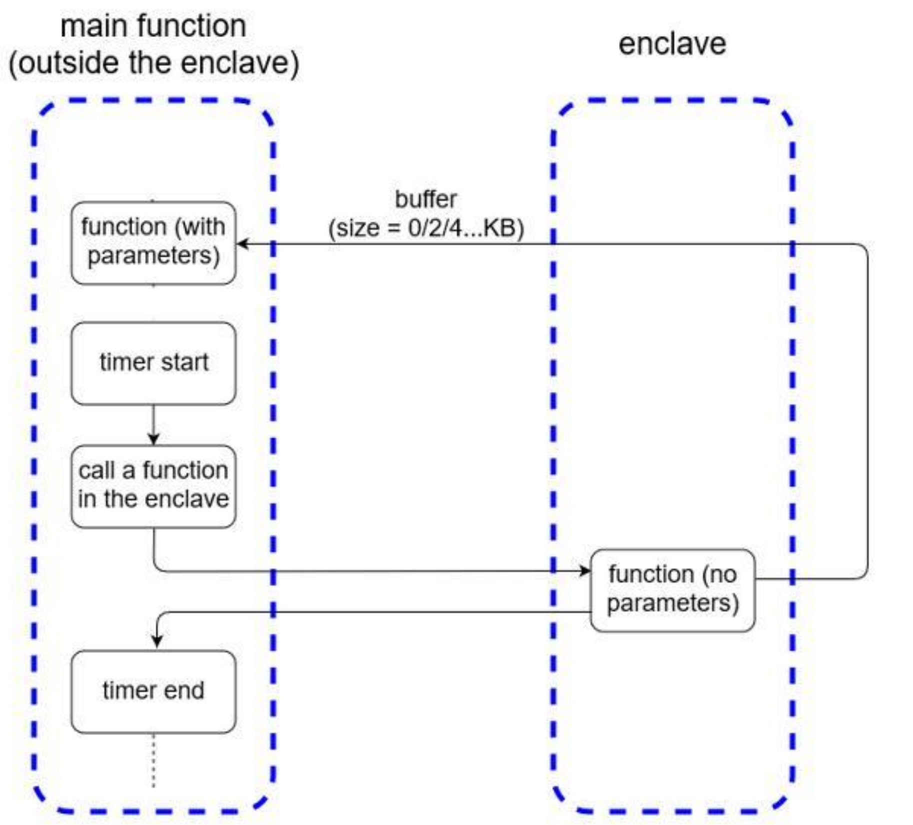
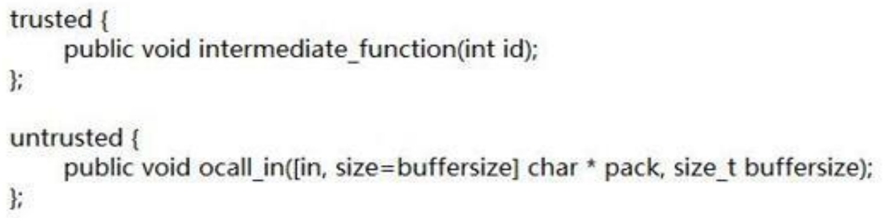


Figure 15. The experiment in analyzing the overhead of *ocall* with option *in*.

Figure 15 shows the work flow of experiment in analyzing the overhead of data transferring using *ocall* with option *in*. The difference between this one and the one in figure 5 is that in Figure 15, the function defined in the *enclave* calls and passes data to a function defined outside the *enclave* before it returns. Thus, there is data transferring using *ocall* involved. Similar with before, there are several groups of programs in this experiment. Different groups have different sizes of data to transfer. Since the only difference between different groups is the size of data, the difference of results between different groups is caused by the difference of sizes of data. Thus, the total running time is a function of the size of data. With similar experiments, relationship between the total running time and the size of data in *ocall* with option *out* and *ocall* with option *in&out* are also obtained.



```
trusted {
        public void intermediate_function(int id);
};

untrusted {
        public void ocall_in([in, size=buffersize] char * pack, size_t buffersize);
};
```

Figure 16. The *.edl* file of experiment group in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ocall* with option *in*.

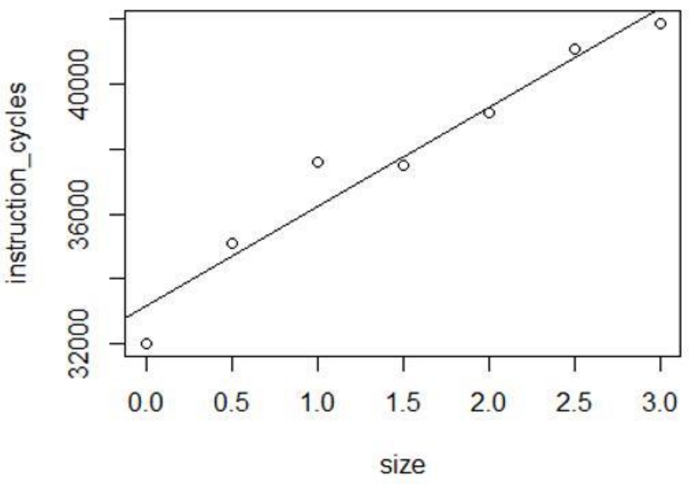
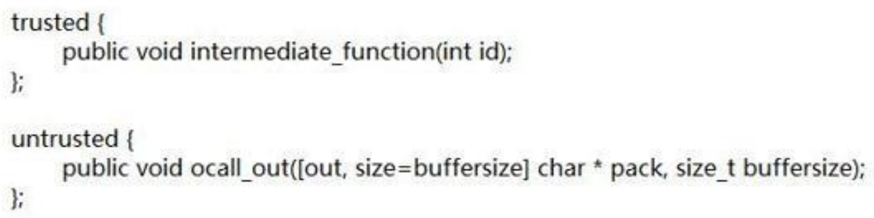


Figure 17. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ocall* with option *in*.

Figure 16 shows the content of EDL file of those groups in analyzing the overhead of data transferring using *ocall* with option *in*. These variables have the same meaning as those in *ecall*. During execution, the function ‘intermediate_function’ calls ‘ocall_in’ to perform data transferring using *ocall*. Different groups in this experiment use the same EDL file as the one in Figure 16, except that they set the variable ‘buffersize’ to be different values so that different groups have different size of data transferring. Figure 17 shows the obtained results, which are the latencies of different groups. The intercept and slope of the straightline in Figure 17 are 33160 and 3054 respectively. Thus, a mathematic function is obtained: total running time = 3054

* size of data + 33160 (in unit of instruction cycles), which implies that it consumes 3054 instruction cycles to transfer every 1-KB data using *ocall* with option *in*.

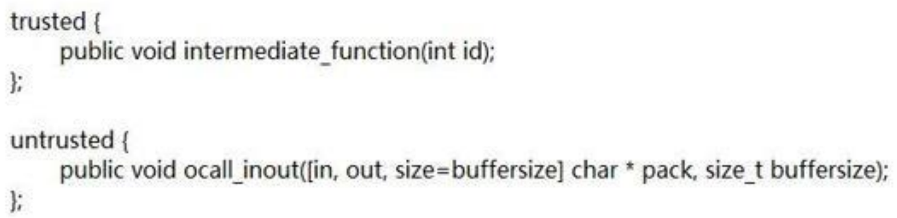
One important note is that the range of size of data in experiments of *ocall* is smaller than the one of *ecall*. This is because the buffer of *ocall* is allocated on the untrusted stack with similar mechanism with '*alloca'* but not '*malloc'* [7]. As the size of stack is limited, the size of buffer of *ocall* and its range are limited as well.



```
trusted {
      public void intermediate_function(int id);
};

untrusted {
      public void ocall_out([out, size=buffersize] char * pack, size_t buffersize);
};
```

Figure 18. The *.edl* file of experiment group in analyzing the overhead of data transferring between enclaves and untrusted environment using *ocall* with option *out*.



```
trusted {
      public void intermediate_function(int id);
};

untrusted {
      public void ocall_inout([in, out, size=buffersize] char * pack, size_t buffersize);
};
```

Figure 19. The *.edl* file of experiment group in analyzing the overhead of data transferring between enclaves and untrusted environment using *ocall* with option *in&out*.

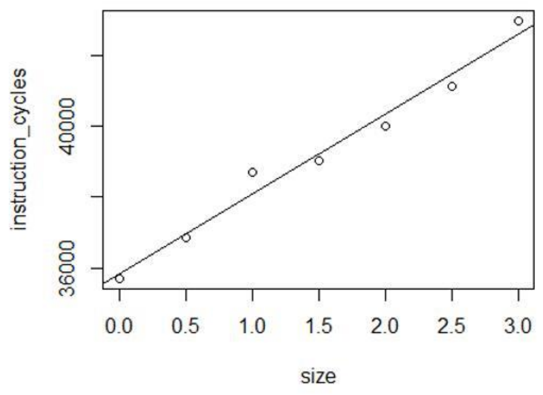


Figure 20. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ocall* with option *out*.

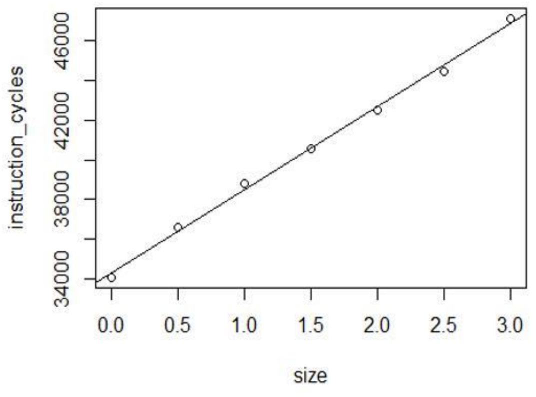


Figure 21. The results in analyzing the overhead of data transferring between *enclaves* and untrusted environment using *ocall* with option *in&out*.

Experiments in analyzing overheads of data transferring using *ocall* with option *out* and *ocall* with option *in&out* have the same work flows as the one in Figure 15, except that their directions of data transferring are different. The contents of their EDL files are shown in Figure 18 and Figure 19 respectively. They are similar with the one in Figure 16 except that they use different options. Figure 20 and Figure 21 are the corresponding results of data transferring using *ocall* with option *out*, and *ocall* with option *in&out* respectively. Similar with the previous explanation, slopes of Figure 20 and Figure 21 show that it consumes 2249 instruction cycles to transfer every 1-KB data using *ocall* with option *out*, and that it consumes 4188 instruction cycles to transfer every 1-KB data using *ocall* with option *in&out*.

6.3 Memory access inside *enclaves*

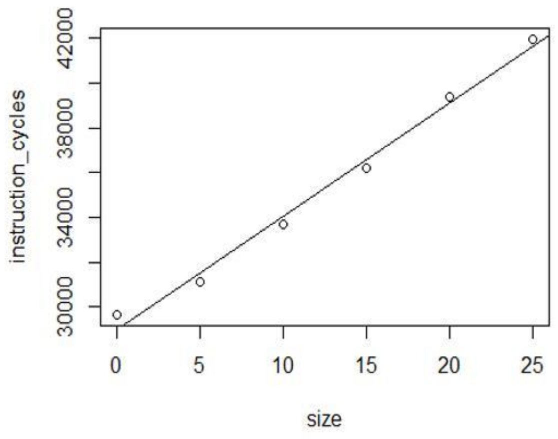


Figure 22. The results in analyzing the overhead of memory access inside the *enclave* using consecutive read.

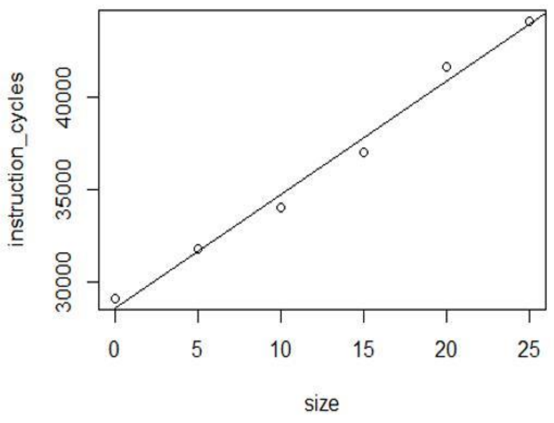


Figure 23. The results in analyzing the overhead of memory access inside the *enclave* using non-consecutive read.

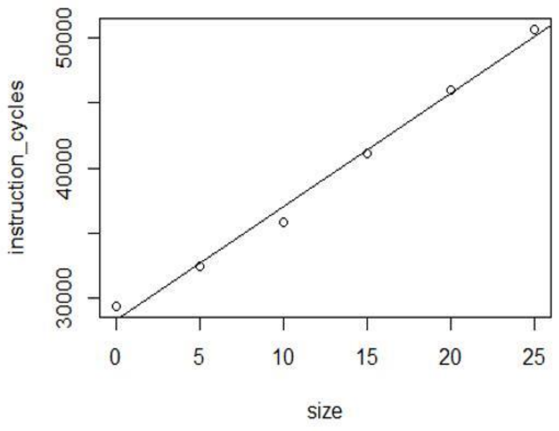


Figure 24. The results in analyzing the overhead of memory access inside the *enclave* using consecutive write.

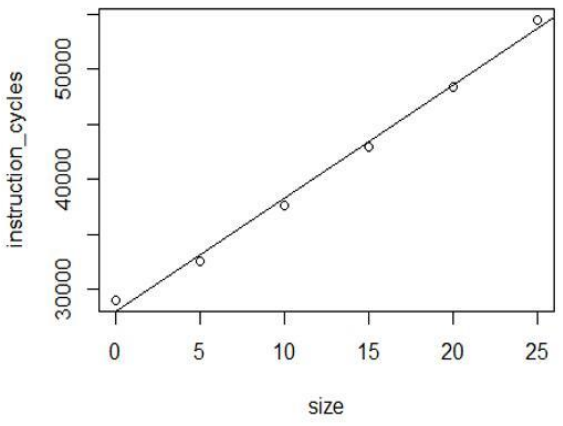


Figure 25. The results in analyzing the overhead of memory access inside the *enclave* using non-consecutive write.

There are four kinds of memory access inside enclaves, which are consecutive read, non-consecutive read, consecutive write and non-consecutive write. Figure 22 – Figure 25 show the corresponding results respectively. Similar with Section 6.2, they are obtained by measuring different running time of accessing different sizes of target memory. Through slopes of those straightlines in Figure 22 – Figure 25, it is shown that it consumes 506 instruction cycles to consecutively read every 1-KB data; it consumes 615 instruction cycles to non-consecutively read every 1-KB data; it consumes 867 instruction cycles to consecutively write every 1-KB data; and it consumes 1030 instruction cycles to non-consecutively write every 1-KB data

6.4 Conclusion

With the results and findings in Section 6.1, 6.2 and 6.3, it is derived that frequently invoking *ecall*/*ocall* is not efficient because it results in high frequency of switching; the size of data transferring between *enclaves* and untrusted environment should be minimized to reduce the related overhead; it is not efficient to access a large size of data/codes inside the *enclave* because the corresponding overhead is proportional to the size.

7. LIMITATION

Since the experiment environment in this project has smaller size of RAM and fewer CPUs than actual server, the above results might have a certain level of residual, however, it can still give a reasonable estimation of true value.

8. IMPROVEMENTS

With the above findings, this project suggested some improvements on *SecureKeeper* in order to reduce those overheads that are caused by applying SGX and discussed in Section 6. Specifically, this project focused on improving the design and implementation of *enclaves* between replicas and clients (as shown in Figure 4), so that *SecureKeeper* is optimally fitted with SGX and hence the performance (latency) of *SecureKeeper* is improved.

8.1 Lower frequency of switching

In *SecureKeeper*, since one *enclave* is connected with one client as well as one replica (as shown in Figure 4), it involves two times of *ecall*, which results in two times of switching, for dealing each request from a client: the client passes the request to the *enclave* through an *ecall*; then the replica passes the corresponding response from databases to the *enclave* through another *ecall*.

According to the findings in Section 6.1, it is optimal to minimize the frequency of switching for reducing the overhead. In theory, removing all databases and let the *enclave* act as a new database can eliminate the *ecall* called by replicas so that there is only one switching involved for dealing one client request. However, the size of *enclave* is limited (with maximum of 128MB) [7], which implies that *enclaves* cannot serve as databases. Therefore, this modification is not effective in practice. In other words, the frequency of switching in *SecureKeeper* is regarded to be minimized and hence there is no further improvement on this kind of overhead.

8.2 Heuristic function

In *SecureKeeper*, the main logic of processing requests and responses is implemented inside the *enclave* (as explained in Section 3.4). In this situation, the client program calls the main logic inside the *enclave* through an *ecall* with option *in&out* to raise a request. Option *in&out* is used because the main logic requires input arguments and returns responses. Before calling an an *ecall* with option *in&out*, the size of buffer (which transfers input arguments into the *enclave* and then transfers outputs out of the *enclave* later) need to be fixed and it cannot be changed later. However, the problem is that the encryption supporting message authentication code (MAC) in the main logic appends extra data to the input arguments after processing. Therefore, the corresponding output has larger length and size than input arguments. If the size of buffer of *ecall* is set to be the length of input arguments at the beginning, the buffer cannot hold the corresponding response because its size is smaller than the response.

To solve this problem, *SecureKeeper* sets the size of buffer to be a large number, which is much larger than the size of input arguments, to ensure that the buffer can store the output (according to the codes on *GitHub*). As a result, the buffer is not fully utilized in most cases, which results in memory waste while solving the problem. Furthermore, according to findings of in Section 6.2, the overhead of data transferring is proportion to the size of buffer. Thus, redundant part of buffer causes unnecessary overhead and hence lower performance.

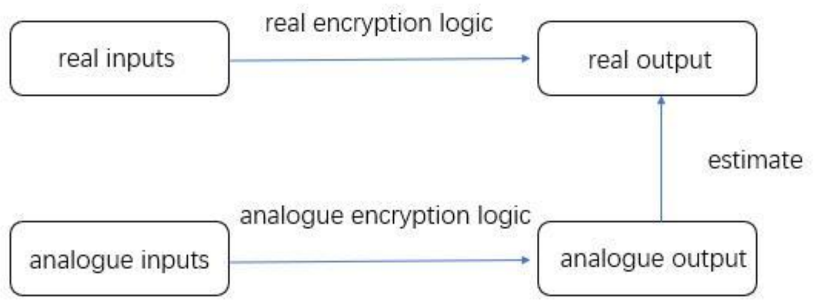
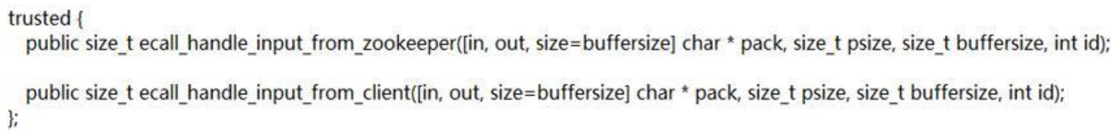


Figure 26. The principle of the heuristic function.

For improvement, this project applied a heuristic function to predict the required size of buffer so that there is less wasted memory and hence smaller overhead. As Figure 26 shows, the heuristic function makes use of an analogue encryption logic, which is similar with that one in the *enclave*, and analogue inputs, which are similar with actual inputs. It applies those analogue imputs to the analogue encryption logic to get an analogue output. Then the length of the analogue output is an estimation of the one of real output. Finally, the size of buffer is set to be the sum of the estimated length and a safety margin, which is a constant number. The reason why actual encryption logic and actual request are not used to get the real size is that they must be processed inside the *enclave* for safety reason. It will make them exposed inside an untrusted environment if they are processed before an *ecall*.

With the above heuristic function, required size of buffer is reduced, so the overhead of data transferring between *enclaves* and untrusted environment is also reduced. As a result, the performance of *SecureKeeper* improves.

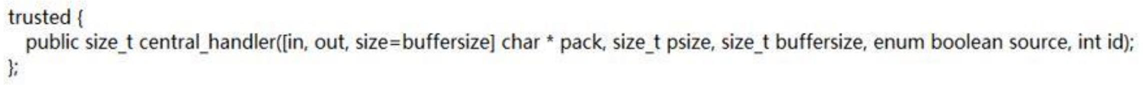
8.3 Smaller *enclave*



```
trusted {
  public size_t ecall_handle_input_from_zookeeper([in, out, size=buffersize] char * pack, size_t psize, size_t buffersize, int id);

  public size_t ecall_handle_input_from_client([in, out, size=buffersize] char * pack, size_t psize, size_t buffersize, int id);
};
```

Figure 27. The interface of SecureKeeper.



```
trusted {
  public size_t central_handler([in, out, size=buffersize] char * pack, size_t psize, size_t buffersize, enum boolean source, int id);
};
```

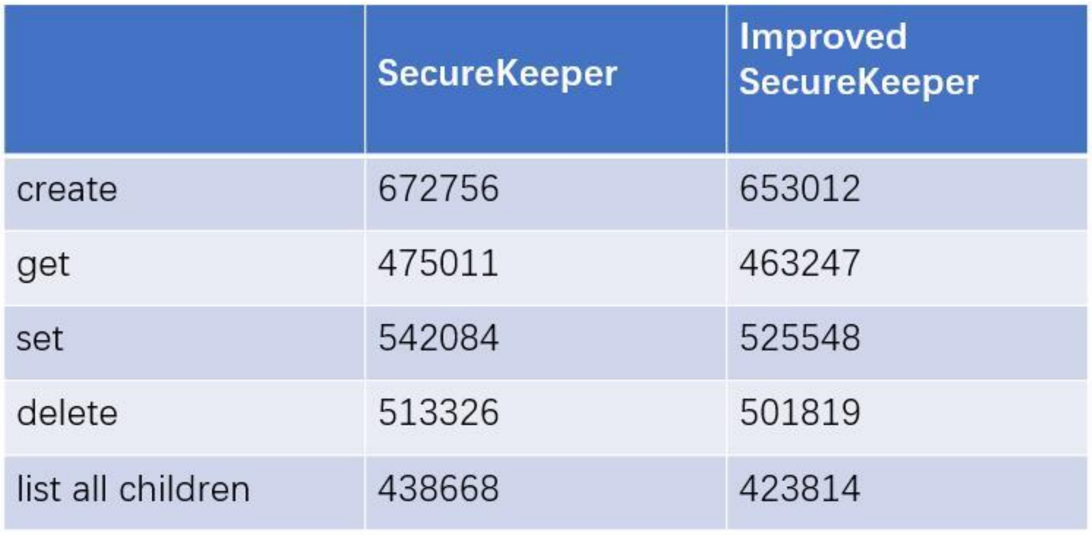
Figure 28. The interface of improved SecureKeeper.

Figure 27 shows the content of EDL file of *SecureKeeper*. It defines the interface between *enclaves* and untrusted environment in *SecureKeeper*. It contains two functions to handle *ecalls* from clients and replicas in *SecureKeeper* respectively. Variable ‘pack’, ‘id’ and ‘buffersize’ have same meaning as those in Section 6. The variable ‘psize’ is the original size of inputs and it is smaller than the variable ‘buffersize’ for solving the problem that is discussed in Section 8.2.

Figure 28 shows the interface of improved *SecureKeeper*. It contains only one function, which is the combination of those two in Figure 27. It has one more argument than the previous one. The variable ‘source’ is used to distinguish the source of an *ecall*, which is either from clients or replicas in *SecureKeeper*. Since the interface of improved *SecureKeeper* is relatively narrower than the one of *SecureKeeper* (this is because the one of improved *SecureKeeper* has only one function), the probability that its interface is under attack is smaller and hence it is safer. Furthermore, it results in smaller size of *enclave*. This because those functions in Figure 27 are similar with each other. Both of them involve serializing/deserializing, encryption/decryption and other common functionalities. Combining them into one can reduce some duplicated codes and hence the size of *enclave*.

According to findings of Section 6.3, the overhead of memory access inside *enclaves* is proportional to the size of memory. Therefore, improved *SecureKeeper* with smaller *enclave* has smaller overhead and hence higher performance because it accesses codes of only one function instead of two functions during running time. In addition, [6] proved that smaller *enclave* can allow more random read/write at the same time, which leads to higher overall performance. Thus, smaller *enclave* is effective to enhance the performance of *SecureKeeper*.

8.4 Latency of improved *SecureKeeper*



	SecureKeeper	Improved SecureKeeper
create	672756	653012
get	475011	463247
set	542084	525548
delete	513326	501819
list all children	438668	423814

Table 1. Latencies of *SecureKeeper* and approved *SecureKeeper* in different operations (in unit of instruction cycles).

Table 1 shows the latencies of *SecureKeeper* and improved *SecureKeeper* (applying improvements in Section 8.2 and 8.3) in different operations, including *create*, *get*, *set*, *delete* and *list all children*. It is shown that the performance (latency) of *SecureKeeper* improves by 3.5% on average with those two improvements. Although the effect is tiny considering only one request, it can save running time cumulatively because the software handles lots of requests in one day. As a result, the change is significant considering the overall performance in one day.

9. CONCLUSION

With control experiments, this project investigated totally three kinds of overheads that are caused by applying SGX, which are switching, data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves*. Frequency of *ecall*/*ocall*, size of buffer and size of memory are key factors of these overheads respectively. Additionally, this project applied a heuristic function and reduced sizes of *enclaves* in *SecureKeeper*. With these two improvements, overheads of data transferring between *enclaves* and untrusted environment, and memory access inside *enclaves* were reduced. The performance of *SecureKeeper* was enhanced as a result. Since the experiment environment in this project has smaller size of RAM and fewer CPUs than actual server due to limitation of laboratory, which results in residuals to true values, future researches equipped with real server can be done to obtain true values of performance analysis.

10. REFERENCES

[1] Frank McKeen. Intel Labs. Stanford University [Internet]. 2015. Available from: https://web.stanford.edu/class/ee380/Abstracts/150415-slides.pdf

[2] Intel [Internet]. USA: Intel. [cited 2018 Sep 26]. Available from: https://software.intel.com/en-us/sgx

[3] Nico Weichbrodt et al. Sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves. IBR TU[Internet]. Available from: https://www.ibr.cs.tu-bs.de/users/weichbr/papers/middleware2018.pdf

[4] Ofir Weisse, Valeria Bertacco, Todd Austin. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. University of Michigan [Internet]. Available from: http://www.ofirweisse.com/ISCA17_Ofir_Weisse.pdf

[5] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors. University of Haifa, Israel [Internet]. Available from: https://eprint.iacr.org/2016/204.pdf

[6] Stefan Brenner et al. SecureKeeper: Confidential ZooKeeper using Intel SGX. Institut für Betriebssysteme und Rechnerverbund [Internet]. 2016. Available from: https://www.ibr.cs.tu-bs.de/users/brenner/papers/2016-middleware-brenner-securekeeper.pdf

[7] Victor Costan, Srinivas Devadas. SGX Explained. International Association for Cryptologic Research [Internet]. 2016. Available from: https://eprint.iacr.org/2016/086.pdf