Tutorial: SGX Programming with OpenSGX

Seongmin Kim, KAIST

Contact: OpenSGX team (sgx@cc.gatech.edu)

Introduction

Intel Software Guard Extensions (Intel SGX), provided by Intel, is an extension to the x86 instruction set architecture which enables an application to run inside a protected container called an enclave. It offers isolated execution and memory protection of applications on the untrusted privilege software (e.g., OS and hypervisor). By reducing the Trusted Computing Base (TCB), Intel SGX provides narrow attack surface. An adversary who has control over all software components and hardware except the CPU package cannot compromise the data/code of the SGX applications. Finally, Intel SGX not only ensures the integrity of program's code and data, but also it guarantees the confidentiality of SGX program.

OpenSGX is a software SGX emulator implemented based on binary translation of qemu. It emulates Intel SGX hardware components at the instruction level. Additionally, OpenSGX is a complete platform for SGX development that includes emulated hardware and operating system components, an enclave program loader, an OpenSGX user library, and debugging and performance monitoring support. Using OpenSGX, it is possible to implement, debug, and evaluate a SGX application.

OpenSGX: Overview

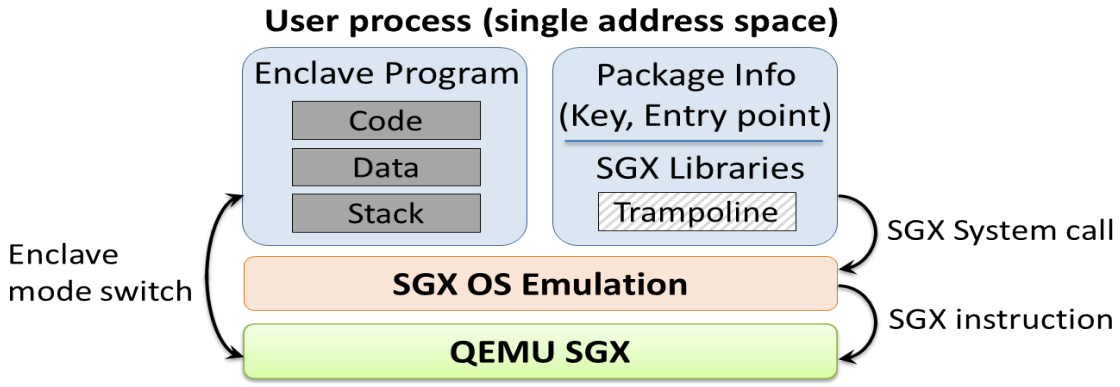

Figure 1: Overview of OpenSGX's design and memory state of an active enclave program. A packaged program including an enclave program, package information and SGX libraries runs as a single process in the same virtual address space.

To implement Intel SGX instructions and emulate its hardware components, we leverage QEMU. In particular, we implement OpenSGX on top of QEMU's user space emulation mode, while leveraging its binary translation feature. Figure 1 illustrates the overall design components. OpenSGX consists of six components that work together to provide a fully functional SGX development environment.

- **Emulated Intel SGX hardware**: hardware components including SGX instructions, SGX data structures, EPC and its access protection, and the SGX processor key as software within QEMU.
- **OS emulation:** system calls that the emulated enclave programs use to perform SGX operations (e.g., enclave provisioning and dynamic memory allocation)
- **Enclave program loader:** A loader which automatically takes care of the process by loading the enclave code and data sections into EPC and provisioning an enclave heap and stack regions appropriately.
- **OpenSGX user library (sgxlib):** a convenience library (sgxlib) for use inside and outside the enclave. It provides wrappers for all SGX user-level instructions as well as high-level application programming interfaces.
- **Debugger support:** extended gdb to map to the instruction being emulated. (Here, OpenSGX also exposes key SGX data structures (e.g., EPCM) through gdb commands.)
- **Performance monitoring:** performance counters/profiler which allows users to collect performance statistics about enclave programs.

Note that OpenSGX does not support binary-compatibility with Intel SGX. Currently, no specification or standardization exists for the binary-level interoperability. Although OpenSGX supports majority of instructions specified, we do not implement all instructions. For example, OpenSGX does not implement debugging instructions specified by Intel SGX, as our software layer can give rich environment for debugging (e.g., familiar GDB stub). OpenSGX is a software emulator and provides no security guarantees. Its security guarantees are not at the same-level as Intel SGX at all.

Prerequisites

- Tested platform: Ubuntu 14.04-15.04, Arch (64 bit)
- Needed package: libglib, zlib, libaio, autoconf, libtool, libssl-dev, libelf-dev (Ubuntu: apt-get install libglib2.0-dev zlib1g-dev libaio-dev autoconf libtool libssl-dev libelf-dev)

Configuration & Build

Get repo

$ git clone https://github.com/sslab-gatech/opensgx.git

Compile QEMU

$ cd opensgx/qemu

$ ./configure-arch

$ make –j $(nproc)

Compile sgx library

$ cd ..

$ make –C libsgx

Compile user-level code

$ make –C user

Source code organization

- **opensgx** : A toolchain for running OpenSGX applications

- **qemu/** : Hardware emulation for SGX instructions

  - qemu/target-i386/sgx-helper.c : helper functions for ENCLU/ENCLS instructions

  - qemu/target-i386/sgx.h : EPC, EPCM and data structures of SGX

- **libsgx/** : OpenSGX libraries

  - libsgx/musl-libc/ : Customized libc libraries for OpenSGX program

  - libsgx/polarssl/ : Light cryptographic libraries for OpenSGX program

  - libsgx/sgx-basic.c : OpenSGX wrapper functions for ENCLU instructions

  - libsgx/sgx-entry.c : Entry wrapper for enclave program

- **user/** : System emulation, system call interface, user-level APIs, and test case for OpenSGX

  - user/sgx-tool.c : Tool for creating keys and data structures (EINITTOKEN, SIGSTRUCT)

  - user/sgx-runtime.c : Runtime library to start an enclave

  - user/sgx-user.c : User-level APIs and system call interfaces

- user/sgx-kern.c : OS-level emulation wrappers for an enclave creation
- user/sgx-loader.c : Enclave loaders for running an enclave program
- user/sgx-trampoline.c : Trampoline wrappers for SGX library supports
- user/sgx.lds : Linker script for OpenSGX program
- user/test/ : Simple test cases

- **gdb/** : Python script for debugging EPCM, EPC and SECS

SGX hardware configurations

Originally, hardware components such as EPC, EPCM and processor keys are actually part of the processor. Also, software components such as SIGSTRUCT and EINITTOKEN should reside in the EPC as part of protected data structures. To emulate those components, OpenSGX provides a tool called "**sgx-tool**" for specifying hardware configurations, such as the size of EPC and the SGX processor key. It is located in the user/ directory.

Configuring hardware components

To compile, build, and execute an OpenSGX program, two keys are necessarily required. One is a processor key and the other is an enclave key. Both keys can be generated by using sgx-tool with –k option. By default, OpenSGX uses 128 bits for processor key pair and 3072 bits for enclave key pair using RSA.

Generate RSA key with give bits
$ sgx-tool -k [BITS]

Default setting
$ sgx-tool -k 3072 (For enclave key pair) > test.key
$ sgx-tool -k 128 (For device key pair) > device.key

The size of EPC and EPCM is also configurable. In the qemu/target-i386/ directory, there is a "sgx.h" header files. In line 26, NUM_EPC variable is defined (by default, 1500). It is the number of EPC pages which an enclave initially contains. If you want to change the size of an enclave, you can change this value and recompile qemu.

Configuring data structures

There is two core data structures to verify the enclave identity and enclave programmer's identity. These are SIGSTRUCT and EINITTOKEN. SIGSTRUCT contains information about the enclave from the enclave signer (OpenSGX programmer) and also has a hash value of enclave. It is used to verify the identity of enclave and signer while launching an enclave. EINITTOKEN is used while EINIT instruction to verify that the target enclave is permitted to launch. It contains a cryptographic MAC calculated by launch key (processor key) to check whether an enclave is actually running on the SGX-enabled platform or not. Here is the detail procedure of generating SIGSTRUCT and EINITTOKEN data structures.

Generate sigstruct format (with reserved fields filled with 0s)

$ sgx-tool -S

Measure binary (generate enclave hash)

$ sgx-tool -m path/to/binary --begin=STARTADDR --end=ENDADDR --entry=ENTRYADDR

(e.g., sgx-tool -m test/simple --begin=0x426000 --end=0x426018 --entry=0x426000)

Note: offset can be obtained from using "readelf -S" and find the offset of corresponding section.

Sign on sigstruct format with given key (after manually fill the fields)

$ sgx-tool -s path/to/sigstructfile --key=path/to/enclavekeyfile

(e.g., sgx-tool -s sgx-sigstruct.conf --key=sgx-test.conf)

Note, user should manually fill the sign information in the sigstruct file after sign.

Generate einittoken format (with reserved fields filled with 0s)

$ sgx-tool -E

Generate MAC over einittoken format with given key (after manually fill the fields)

$ sgx-tool -M path/to/einittokenfile --key=path/to/devicekeyfile

(e.g., sgx-tool -M sig-einittoken.conf --key=sgx-intel-processor.conf)

Note, user should manually fill the mac into einittoken file.

For the convenience of OpenSGX users, we provide a toolchain called "**opensgx**". Above procedure are automatically done by simple commands. An explanation for using opensgx toolchain is described in next section.

Compilation of OpenSGX binary

To compile an OpenSGX binary, it requires several prerequisite steps. First, an SGX application developer creates his/her own signing key for the signer identity. After that, you can compile your OpenSGX program and sign it using your key. In this phase, OpenSGX also calculates the identity of enclave program by calculating the hash value of the code and data section. Then, you can run your own OpenSGX application.

How to run OpenSGX program (user/demo/hello.c as an example)

Generating own signing key
$ ./opensgx –k
(Then, sign.key file is created)

Generate hello.sgx (compile an OpenSGX application)
$ ./opensgx –c user/demo/hello.c
(Then, hello.sgx is created in user/demo directory)

Generate hello.conf (Sign and calculate identity of an OpenSGX application)
$ ./opensgx –s user/demo/hello.sgx --key sign.key
(Then, hello.conf is created in user/demo directory)

Run an OpenSGX program
$ ./opensgx user/demo/hello.sgx user/demo/hello.conf

Implementing OpenSGX applications

Implementing an OpenSGX program is similar to that of normal C applications. Since operating system support, system call interfaces and user-level APIs for SGX program is supported by OpenSGX, user can simply user them to build and execute an OpenSGX binary. The only difference is that using enclave_main() instead of main() and using sgx_exit(NULL) instead of return. In the OpenSGX program, it is possible to use existing libc library functions or crypto library functions such as polarssl by linking archive files while compiling OpenSGX program. Since OpenSGX offers its own customized linker script and loader, it is possible to use other libraries in the enclave binary by modifying them. Here is a source code of simple hello world program.

Hello world example

```
#include “test.h”
void enclave_main()
{
	char *hello = "hello sgx!₩n";
	puts(hello);

	sgx_exit(NULL);
}
```

Trampoline & Stub

OpenSGX provides stricter form of communication protocols by using shared code and data memory, called trampoline and stub, respectively. The use of trampoline and stub defines a narrow interface to the enclave, which is readily tractable for enforcing the associated security properties. Figure 2 shows the example case of stub and trampoline. The enclave first sets up the input parameters in stub, and then invokes a predefined handler, trampoline, by exiting its enclave mode (i.e., by invoking EEXIT).

Using libc libraries

For using libc libraries in the OpenSGX application, trampoline and stub interface is used by modifying several codes of original glibc library. Here is a source code of modified read libc function which contains the trampoline and stub interface. The red colored lines are newly added part. Here, in the stub data structure, it sets the mapped shared region (STUB_ADDR), function types (fcode) and in/out arguments for the library function. Here, out arguments (out_arg) are input parameters for the function and in argument (in_arg) has a return value of the function. After out_arg values are set, it calls **“**sgx_exit(stub->trampoline)**“** to exit the enclave mode.

```
ssize_t read(int fd, void *buf, size_t count)
{
	sgx_stub_info *stub = (sgx_stub_info *)STUB_ADDR;
	int tmp_len;
	ssize_t rt;
	int i;

	rt = 0;
```

```
    for (i = 0; i < count / SGXLIB_MAX_ARG + 1; i++) {
        stub->fcode = FUNC_READ;
        stub->out_arg1 = fd;

        if (i == count / SGXLIB_MAX_ARG)
            tmp_len = (int)count % SGXLIB_MAX_ARG;
        else
            tmp_len = SGXLIB_MAX_ARG;

        stub->out_arg2 = tmp_len;
        sgx_exit(stub->trampoline);
        memcpy((uint8_t *)buf + i * SGXLIB_MAX_ARG, stub->in_data1, tmp_len);
        rt += stub->in_arg1;
    }

    return rt;
}
```

Once the host program (or OS) processes the enclave request, it stores the result or return values to stub, and enters the enclave mode again by invoking ERESUME. After transferring the program's control back to the known location inside the enclave, the enclave program can, finally, obtain the returned value (e.g., in_arg1 in stub).

```
static int sgx_read_tramp(int fd, void *buf, size_t count)
{
    return read(fd, buf, count);
}

void sgx_trampoline()
{
  switch (stub->fcode) {
  case FUNC_PUTS:
  ...
  case FUNC_READ:
        stub->in_arg1 = sgx_read_tramp(stub->out_arg1, stub->in_data1, (size_t)stub->out_arg2);
        break;
  ...
}
```

By adding the trampoline & stub codes into the OpenSGX user library and glibc library function, it is possible to use them inside the enclave program. The usage of them inside the enclave program is exactly same with implementing a normal C program as shown in Hello world example (See puts function).

Using 3rd party libraries

To use 3rd party libraries inside the enclave program, archive files (*.a) or object files (*.o) of them should be compiled and linked with SGX program. Currently, OpenSGX supports mbedtls (lightweight ssl library) and OpenSSL library. You can use functions of those libraries inside the enclave program without any changes. In the OpenSGX, we use mbedtls library to calculate the sha256 hash for mrenclave and get MAC (Message Authenticate Code) for report data structure. The only thing to use those libraries is modifying Makefile; you should link archive files or object files to the target SGX program during the compilation (See user/Makefile for the detail). Here is an example code of using OpenSSL library functions inside the enclave.

/user/test/openssl/simple-openssl.c

```
#include "../test.h"
#include <openssl/bn.h>
#include <openssl/rsa.h>

BIGNUM *bn = NULL;
RSA *rsa = NULL;

void enclave_main()
{
	// Bignum allocation for global variable
	bn = BN_new();
	// Bignum allocation for local variable
	BIGNUM *bn2 = BN_new();
	// RSA allocation for global variable
	rsa = RSA_new();
	// RSA allocation for local variable
	RSA *rsa2 = RSA_new();
	printf("%x₩n", (unsigned long)rsa2);
}
```

Argument vector

OpenSGX also supports argument vector to pass command inputs to the enclave program. To achieve this property, OpenSGX offers sgx-entry.S assembly code to pass the argc and argv which is located in /libsgx directory. Before calling the enclave_main, sgx-runtime saves argument vectors to the user-space register (RDI, RDX and etc) and pass them to the entry point (*enclave_start*). Here, "sgx-entry.S" file defines the *enclave_start* symbol. Here, it gets the value of registers and put them into the stack. You can see the details of argument vector handling procedure in /user/sgx-runtime.c and /libsgx/sgx-entry.S files. Here is an example code which uses argument vector.

/user/test/simple-arg.c

```
#include "test.h"

void enclave_main(int argc, char **argv)
{
	printf("argc = %d₩n", argc);
	puts(argv[0]);
	puts(argv[1]);
	puts(argv[2]);
}
```

The execution result of above simple-arg program with argument "test_arg" is as follows. Similar to general usage of argument vector, it is possible to pass argument to the enclave_main with argc and argv. Note that argv[0] contains "./../user/sgx-runtime" string since the test.sh script executes a sgx-runtime executable to load the SGX binary and call EENTER instruction.

```
smkim@ubuntu:~/src/SGX/opensgx/user$ ./test.sh test/simple-arg test_arg
make: `test/simple-arg' is up to date.
-----------------------------------------
kern in count      : 2
kern out count     : 2
-----------------------------------------
...
...
...
-----------------------------------------
Pre-allocated EPC SSA region         : 0x2000
```

```
Pre-allocated EPC Heap region      : 0x12c000
Later-Augmented EPC Heap region: 0x0
Total EPC Heap region      : 0x12c000
argc = 3                                    ← print argc
./../user/sgx-runtime                       ← print argv[0]
test/simple-arg                             ← print argv[1]
test_arg                                    ← print argv[2]
```

Separation of enclave process and non-enclave process

While implementing SGX program, the most important thing is reducing TCB (Trusted Computing Base). To achieve this goal, SGX programmer need to minimize the size of SGX program. For example, if the programmer try to implement and execute the existing application, they need to separate the core part (e.g., private key, creating a signature) of it and put it into the enclave. In this tutorial, we call the process which will be executed inside the enclave as an *enclave process* and the process which executes the rest of the program called *non-enclave process*. In this section, we provide how they communicate each other and briefly give a simple example of them.

The enclave process and non-enclave process communicate through a pipe protocol. When non-enclave process requests a result of core operations (e.g., request for creating a signature), it sends a request message to the enclave process. Then, it sends input data (e.g., data will be signed) to the enclave process. In the enclave process side, it receives the input data from non-enclave process and execute secret operation (e.g., create a signature) inside the enclave. Note that the secret of application (e.g., private key) will not be leaked and securely protected inside the enclave. After the secure operation is done, the result will be send back to the non-enclave process. This is an overall procedure of communication between the enclave process and the non-enclave process. The example of enclave process and non-enclave process is located in user/test/simple-pipe.c and user/non_enclave/simple-pipe.c.

Using multiple enclaves

Will be updated soon.

Remote attestation

Attestation related source codes

OpenSGX supports intra/remote attestation features. There are several functions related to attestation property in libsgx directory. You can use intra/remote attestation library functions to enclave program by including 'sgx-lib.h'. Below is attestation related function prototypes.

Part of /libsgx/include/sgx-lib.h

```
...
 The preface omitted
...
/* Attestation supporting funcitons */
extern int sgx_make_server(int port);
extern int sgx_connect_server(const char *target_ip, int target_port);
extern int sgx_get_report(int fd, report_t *report);
extern int sgx_read_sock(int fd, void *buf, int len);
extern int sgx_write_sock(int fd, void *buf, int len);
extern int sgx_match_mac(unsigned char *report_key, report_t *report);
extern int sgx_make_quote(const char* pers, report_t *report, unsigned char *rsa_N, unsigned char *rsa_E);

/* Intra attestation supporting functions */
extern int sgx_intra_attest_challenger(int target_port, char *conf);
extern int sgx_intra_attest_target(int port);

/* Remote attestation supporing functions */
extern int sgx_remote_attest_challenger(const char *target_ip, int target_port, const char *challenge);
extern int sgx_remote_attest_target(int challenger_port, int quote_port, char *conf);
extern int sgx_remote_attest_quote(int target_port);
```

Also you can see function codes in libsgx. Related files are listed below.

```
libsgx/sgx-attest.c
libsgx/sgx-intra-attest.c
libsgx/sgx-remote-attest.c
```

Mechanism of intra/remote attestation

OpenSGX produces two kinds of attestation properties. One is intra attestation, the other is remote attestation.

Intra attestation occurs between two enclaves, challenger and target. First, challenger enclave which wants to verify the target enclave makes socket connection with target enclave. Then challenger enclave obtains the identity of target enclave from configuration file (.conf file). Challenger enclave loads SIGSTRUCT of target enclave by configuration file of target enclave. Next, Challenger enclave creates REPORT by EREPORT instruction and sends it to target enclave via socket write. The target enclave gets the REPORT and extracts report key to compute MAC. Then the target enclave verifies the REPORT by MAC and sends its REPORT to challenger enclave. Finally, challenger enclave reciprocates verifying process using target enclave's REPORT for mutual verification.

Remote attestation is bit more difficult than intra attestation. Three enclaves are needed for remote attestation process, challenger, target, and quoting enclave. First, challenger enclave makes socket connection with target enclave, then sends the request to target enclave. After the target enclave receives the request from the challenger enclave, it starts intra attestation with the quoting enclave which resides in same SGX platform. After the both enclaves finish the mutual verification process, Target enclave send verification success message to quoting enclave. Then the quoting enclave makes QUOTE and RSA keys and sends them to target enclave. After the target enclave receives QUOTE and RSA keys from quoting enclave, it forwards them to the challenger enclave. Finally, the challenger enclave verifies the QUOTE using RSA key pairs to verify the target enclave. Over all procedure of remote attestation in OpenSGX is implemented based on the SGX specification by using an RSA key scheme as an alternative to EPID. We use a public signature scheme (RSA) as a proof-of-concept and leave the adoption of EPID as future work.

Testing intra/remote attestations

OpenSGX supports test codes related to intra/remote attestations. Because configuration file of target/quoting enclave is necessary for intra/remote attestation, you need to build the file in the demo directory firstly. You can simply test attestation codes by typing following commands.

Test intra attestation

Terminal #1

```
$ ./opensgx –k
$ ./opensgx –c user/demo/simple-intra-attest-target.c
$ ./opensgx –s use user/demo/simple-intra-attest-target.sgx --key sign.key
$ ./opensgx user/demo/simple-intra-attest-target.sgx user/demo/simple-intra-attest-target.conf
```

Terminal #2

```
$ cd user
$ ./test.sh test/simple-intra-attest-challenger
```

Test remote attestation

Terminal #1

```
$ ./opensgx –k
$ ./opensgx –c user/demo/simple-remote-attest-quote.c
$ ./opensgx –s use user/demo/simple-remote-attest-quote.sgx --key sign.key
$ ./opensgx user/demo/simple-remote-attest-quote.sgx user/demo/simple-remote-attest-quote.conf
```

Terminal #2

```
$ cd user
$ ./test.sh test/simple-remote-attest-target
```

Terminal #3

```
$ cd user
$ ./test.sh test/simple-remote-attest-challenger
```

Since above tests designed for testing in local machine environment, the remote attestation test makes socket connection locally. However, if you want to test remote attestation in remote OpenSGX-enabled environment, you can test it also by simply changing the IP address and port number in test code.

If the test successes, "Intra/Remote Attestation Success!" messages print out. In addition, for remote attestation test, you might have to wait for several seconds because generating RSA key and decrypting QUOTE takes some time.

Debugging support

Since OpenSGX is implemented based on user mode emulation of qemu, its binary translation can make debugging more difficult because a debugger can only observe translated instructions. Thus, OpenSGX extend *gdb* to map to the instruction being emulated. Here is an example procedure for debugging an OpenSGX application.

1. Run a target application in the background with debug option

```
$ ./opensgx -d 1234 user/demo/hello.sgx user/demo/hello.conf &
```

2. Attach remote gdb on the target port

```
$ gdb user/sgx-runtime
(gdb) target remote localhost:1234
Remote debugging using localhost:1234
[New Remote target]
Reading symbols from /lib64/ld-linux-x86-64.so.2...(no debugging symbols found)...done.
Loaded symbols for /lib64/ld-linux-x86-64.so.2
[Switching to Remote target]
0x0000004000802190 in ?? () from /lib64/ld-linux-x86-64.so.2
(gdb) b sgx-runtime.c:63
Breakpoint 1 at 0x401a80: file sgx-runtime.c, line 63.
(gdb) c
Continuing.

Breakpoint 1, 0x0000000000401a80 in main ()
```

3. Find a text section offset

```
$ readelf -s user/demo/hello.sgx | grep text
   [ 2] .text                 PROGBITS             0000000050000110   00000110
```

4. Add symbol file in gdb by specifying a text section offset

```
(gdb) add-symbol-file user/demo/hello.sgx 0x0000000050000110
add symbol table from file "user/demo/hello.sgx" at
      .text_addr = 0x50000110
(y or n) y
Reading symbols from /home/mingwei/gatech/opensgx_test/user/demo/hello.sgx...done.
```

5. Set a breakpoint on the enclave binary and start debugging

```
(gdb) b enclave_main
Breakpoint 2 at 0x50000110
(gdb) c
Continuing.
```

```
Breakpoint 2, 0x0000000050000110 in enclave_main ()
(gdb)
```

Performance monitoring

OpenSGX also supports performance monitoring features. Since it is a software emulator, it cannot provide the accurate performance measures. Instead, OpenSGX helps developers and researchers to speculate potential performance issues by providing its emulated performance statistics similar to that of the *perf* counter. It exposes a system call to query the OpenSGX emulator about statistics such as the number of context switch occurred, SGX instructions executed, and etc. Also, opensgx toolchain offers –i option to count the number of CPU cycles consumed by enclave program in a software manner (by using tcg-plugin of qemu).

```
$ cd user
$ ./test.sh –i test/simple-hello
```

Example: Given performance measurement of Hello world program

```
-----------------------------------------
kern in count      : 2
kern out count     : 2
-----------------------------------------
encls count        : 6071
ecreate count      : 1
eadd count         : 357
eextend count      : 5712
einit count        : 1
eaug count         : 0
-----------------------------------------
enclu count        : 2
eenter count       : 1
eresume count      : 0
eexit count        : 1
egetkey count      : 0
ereport count      : 0
eaccept count      : 0
-----------------------------------------
mode switch count : 2
```

```
tlb flush count     : 2
----------------------------------------
Pre-allocated EPC SSA region        : 0x2000
Pre-allocated EPC Heap region       : 0x64000
Later-Augmented EPC Heap region: 0x0
Total EPC Heap region       : 0x64000                    ← Statistics of SGX program
hello sgx!                                               ← Result of program

number of executed instructions on CPU #0 = 120950414    ← Consumed CPU cycles
```