OpenEnclave Design Overview

September 4, 2017

Table of Contents

1. Introduction 2
2. The Enclave Application 3
3. The Inter-Call Model 5
   OE_Enter() 7
   OE_Main() 7
   OE_Exit() 7
4. Thread Binding 9
   Operation 9
   Implementation 10
5. Enclave Creation 12
6. Enclave Page Layout 14
   The Signature Section (.oesig) 14
   Text Pages 15
   Data Pages 15
   Relocation Pages 16
   ECALL Pages 16
   Heap Pages 17
   Thread Context 17
   Guard Pages 18
   Per Thread Stack 18
   Thread Control Structure 18
   Set-Aside Areas 19
   Segment Page 19
   Thread-Specific Data Page 20
7. The Exception handling 20

Hardware exception 20
Hardware exception handling workflow 20
Vector exception handling functions 21
8. The Signing Tool 22
Example 22
9. The Debugger Extension 24
Ptrace Library 24
Python Extension 24
Stack Stitching 24
Split stack 25
10. The IDL Generator 25
Supported Data Types 25
Function and parameter constraints 26
Example 1 27
Example 2 27

1. Introduction

This document provides a brief design overview of the **OpenEnclave SDK** (**OE**). It describes the parts of the SDK and how they work together to create, invoke, and terminate enclaves. This document assumes the reader is familiar with enclaves.

OE aims to provide an abstract API for developing enclave applications whether they are based on the Intel® Software Guard Extensions (SGX) or the Microsoft® Virtual Secure Mode (VSM). This early version though only supports SGX, so the design discussion that follows refers exclusively to SGX.

2. The Enclave Application

OE helps developers build **enclave applications**. An enclave application is partitioned into an untrusted component (called a **host**) and a trusted component (called an **enclave**). An enclave is a secure container whose memory (text and data) is protected from access by outside entities, including the host, privileged users, and even the hardware. The enclave may run in an untrusted environment with the expectation that secrets will not be compromised.

On Linux, enclaves are packaged as shared objects that have been digitally signed. For example, an enclave called **turtle** might be redistributed with the following filename.

```
turtle.signed.so
```

A host may load and instantiate this enclave as shown by the following snippet.

```
OE_Enclave* enclave;
OE_Create("turtle.signed.so", 0, &enclave);
```

The **enclave** variable is a handle used to refer to the enclave throughout its lifetime. A host may wish to create several enclaves, either of the same type or different types.

Once an enclave is created, the host may invoke its functions, known as **enclave calls** or **ECALLs**. In the snippet below, the host calls the enclave's **Walk** function.

```
OE_CallEnclave(enclave, "Walk", args));
```

This enters the enclave and calls its Walk function. To service this request, the turtle enclave implements a Walk function with the following signature.

```
OE_ECALL void Walk(void* args);
```

Once an enclave is entered, the enclave may call functions within the host, called **outside calls** or **OCALLs**. For example, the enclave may call the host's **WhoAreYou** function as shown in the snippet below.

```
OE_CallHost(enclave, "WhoAreYou", args));
```

To service this request, the host implements a WhoAreYou function with the following signature.

```
OE_OCALL void WhoAreYou(void* args);
```

The **args** parameter for ECALLs and OCALLs is defined by the developer of the enclave application. It might be a pointer to a string or a C structure. Although **OE_CallEnclave** and **OE_CallHost** are not type safe, we will see later how to use the IDL generator (**oegen**) to produce type-safe, parameterized stubs for safely calling those functions.

The host and enclave may continue to exchange ECALLs and OCALLs. ECALLs and OCALLs may be arbitrarily nested as stack memory allows.

Eventually the host will terminate the enclave as shown in the snippet below.

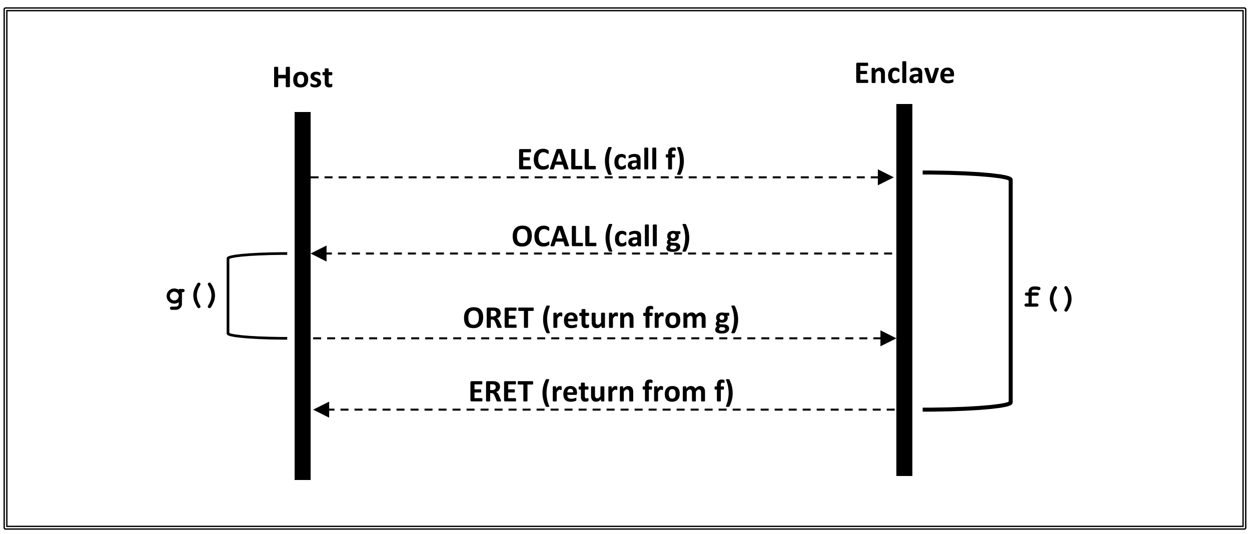
```
OE_TerminateEnclave(enclave);
```

3. The Inter-Call Model

OE defines an **inter-call model** whereby the host and enclave call each other's functions. These may be thought of as messages exchanged between the host and enclave. There are four kinds of messages, defined in the table below.

Message	Description	Sender	Receiver
ECALL	An enclave call	Host	Enclave
ERET	An enclave return	Enclave	Host
OCALL	An outside call	Enclave	Host
ORET	An outside return	Host	Enclave

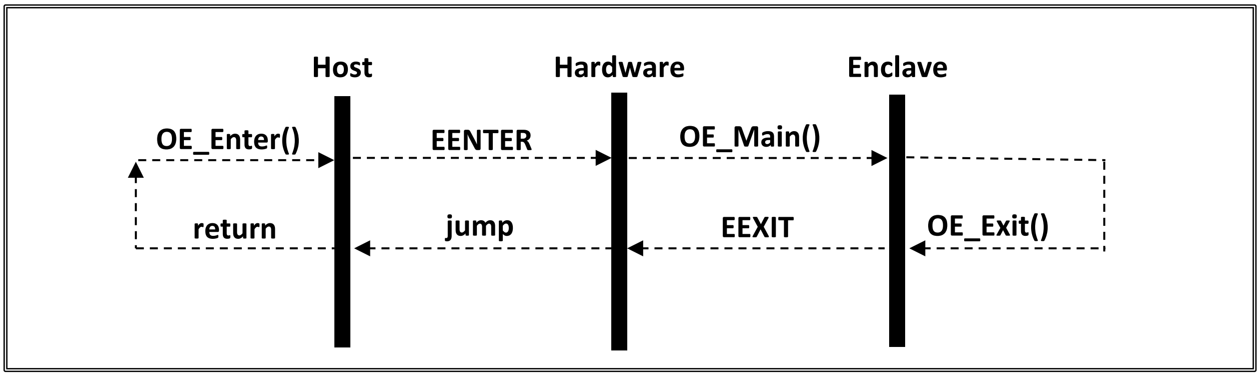
An enclave may perform an OCALL while servicing an ECALL, and a host can perform an ECALL while servicing an OCALL. So ECALLs and OCALLs may be nested arbitrarily. The diagram below illustrates one level of nesting.



ECALLs and OCALLs (and their respective returns) are software constructs. There is no direct analogue with SGX hardware instructions. Rather these calls are implemented using the **EENTER** and **EEXIT** instructions as defined in the table below.

Instruction	Description
EENTER	Executed by the host (**OE_Enter**) to enter the enclave (**OE_Main**)
EEXIT	Execute by the enclave (**OE_Exit**) to return to host (**OE_Enter**)

The interaction between the host and enclave during execution of these instructions is depicted below along with the OE functions that are involved.



ECALLs and OCALLs are layered on the EENTER and EEXIT instructions. The table below shows how the four message types are mapped onto the SGX instructions.

Message	SGX Instruction	Explanation
ECALL	**EENTER**	ECALL executes EENTER to enter enclave
ERET	**EEXIT**	ERET executes an EEXIT to exit enclave
OCALL	**EEXIT**	OCALL executes EEXIT to exit enclave
ORET	**EENTER**	ORET executes EENTER to reenter the enclave

When the host performs an ECALL, it executes an EENTER instruction to enter the enclave. The host blocks until the EENTER instruction returns (when the enclave executes EEXIT). The enclave exits either to perform an ERET or an OCALL. The reason for the exit is returned in a register. The host examines this register to determine what to do next: to process an ERET or to service an OCALL.

The table below identifies key OE functions that participate in ECALLs and OCALLs along with source code references for each.

Function	Description	Source Location
OE_Enter()	Host calls to execute EENTER to enter an enclave.	host/enter.S
OE_Main()	EENTER calls this enclave entry point.	enclave/main.S
OE_Exit()	Enclave calls to execute EEXIT to exit the enclave.	enclave/exit.S

The behavior of these functions is defined below.

OE_Enter()

OE_Enter() executes the EEXIT instruction with the following register assignments.

- RDI: The address of a Thread Control Structure (TCS) in the enclave
- RSI: Address of an Asynchronous Exception Procedure (AEP) in the host
- RDX:
  - High-word: code indicating whether ECALL or ORET
  - Low-word: function number
- RCX: ECALL or ORET argument

EENTER obtains the enclave entry point from the TCS, which is always the OE_Main() function. EENTER calls OE_Main(), passing it the return address (the instruction in the host immediately following the EENTER instruction execution). The calling thread executes in the enclave until the enclave calls OE_Exit(), which executes the EEXIT instruction, which jumps to the return address in the host.

OE_Main()

The EENTER instruction calls OE_Main() to enter the enclave with the following register assignments.

- RAX: index of current SSA (a non-zero value indicates an exception)
- RBX: address of a TCS
- RCX: return address (address of instruction in host immediately following EENTER)
- RDI:
  - High-word: code indicating whether ECALL or ORET
  - Low-word: function number
- RSI: ECALL or ORET argument

OE_Main() performs the following tasks:

- Saves the hosts registers
- Initializes the enclave stack frame
- Invokes __OE_HandleMain()

__OE_HandleMain() handles either an ECALL or an ORET, dispatching as necessary. __OE_HandleMain() never returns but instead calls OE_Exit() which executes an EEXIT instruction.

OE_Exit()

OE_Exit() is called to either perform an ERET or an OCALL. It performs the following:

- Clears the enclave registers
- Restores the host registers
- Executes the EEXIT instruction with the following register assignments:
    - RBX – return address
    - RCX – Asynchronous Exception Procedure (AEP)
    - RDI:
        - High-word: code indicating whether OCALL or ERET
        - Low-word: function number
    - RSI: OCALL or ERET argument

The host examines the high-word of RDI (the code) to determine what action to take and the low-word of RDI (the function number) to determine what function to call. The function argument is taken from RSI.

4. Thread Binding

This chapter discusses the binding between **host threads** and **enclave thread contexts**. When an enclave is created, it has *N* thread contexts. Each **thread context** consists of the following pages.

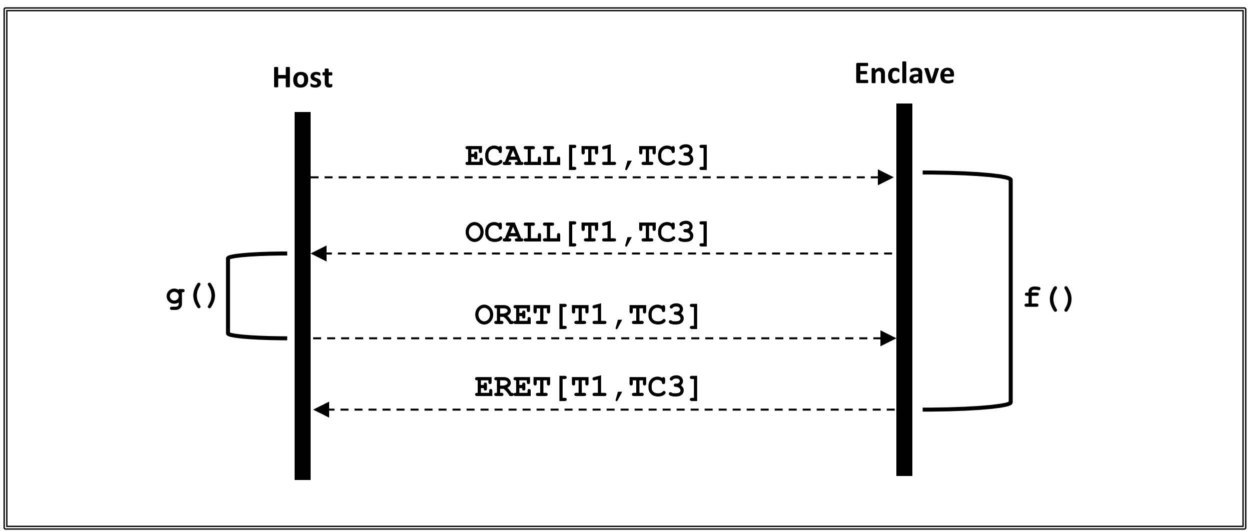
- **Thread control structure** (**TCS**) **pages**—used by SGX to maintain thread state
- **Stack pages**—dedicated to that thread context
- **Set aside areas** (**SSA**) **pages**—used by SGX to handle exceptions and interrupts
- **Segment page**—holds thread information (bound to the GS register)
- **Thread-specific data page**—holds word-sized TSD slots (or TLS slots)

For more information about thread context pages, see **Chapter 6** (**Enclave Page Layout**).

Operation

When a host thread performs an ECALL, the invocation is targeted at one of these thread contexts. The thread binds to the thread context for the duration of the ECALL. When the ECALL returns, this binding is dissolved.

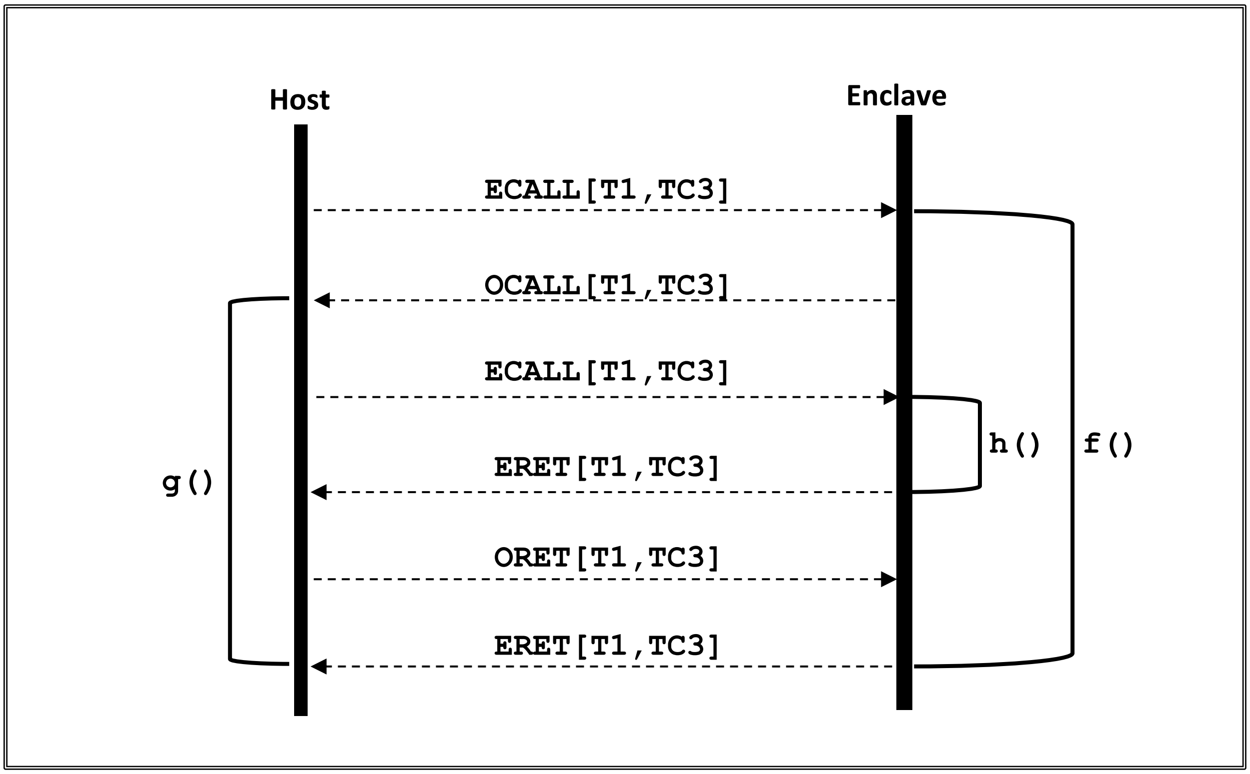
This is simple enough, but what happens when an ECALL performs an OCALL? The same thread that invoked the ECALL performs the OCALL. The thread is depicted as **T1** in the diagram below and thread context is is bound to is depicted as **TC3**. The binding is indicated with the following notation: **[T1,TC3]**.



Note that the binding between T1 and TC3 remains in effect until the ECALL returns, and then the binding is dissolved.

Now, what happens when an OCALL performs a nested ECALL? Is the same binding used for the nested ECALL as well? The answer depends on whether the same thread that performed the

OCALL is used to make the nested ECALL (note that the host can create a new thread to perform the call). If so, then OpenEnclave uses the same binding. This is depicted in the diagram below.



As with the previous example, the binding between T1 and TC3 is dissolved when the original ECALL returns.

Implementation

Enclaves are created by calling **OE_CreateEnclave(),** defined as follows.

```
OE_Result OE_CreateEnclave(
    const char* path,
    uint32_t flags,
    OE_Enclave** enclave);
```

The **enclave** output parameter is an opaque pointer to an internally defined structure. The **OE_Enclave** structure maintains the following information.

- The base address of the enclave
- The size of the enclave in bytes
- The hash of the enclave
- An array ECALL name-address structures (ECALL array)
- An array of thread bindings (between **host threads** and **enclave thread contexts**)

To call into an enclave, one either calls **OE_CallEnclave()** or the low-level function **OE_ECall()**. For example, **OE_CallEnclave()** is defined below.

```
OE_Result OE_CallEnclave(
    OE_Enclave* enclave,
    const char* func,
    void* args);
```

This function performs the following steps.

1. Searches the ECALL array for a function named **func**.
2. If found, it initializes the fields of an **OE_CallEnclaveArgs** structure as follows:
    a. **OE_CallEnclaveArgs**.**vaddr**—the virtual address of the enclave ECALL function
    b. **OE_CallEnclaveArgs**.**func**—the function number of the enclave ECALL function
    c. **OE_CallEnclaveArgs**.**args**—the **args** parameter to **OE_CallEnclave()**
3. Calls **OE_ECall()** with these arguments:
    a. **enclave**—same as the enclave argument passed to **OE_CallEnclave()**
    b. **func**—the predefined **OE_FUNC_CALL_ENCLAVE** constant
    c. **args**—the **OE_CallEnclaveArgs** structure initialized above
4. **OE_ECall()** function performs these steps.
    a. Finds an available enclave thread context.
    b. Enters the enclave, targeting the thread context found in the previous step
    c. Waits for the ECALL to return

If **OE_ECall()** finds an available thread context, it marks it as busy, else it returns **OE_OUT_OF_THREADS**. When the ECALL returns, **OE_ECall()** releases the thread context, clearing the busy flag.

If the enclave performs an OCALL before **OE_ECall()** returns, the host might perform a nested ECALL. If it does, steps 1 through 4 above are repeated using the same enclave thread context used in the original ECALL.

5. Enclave Creation

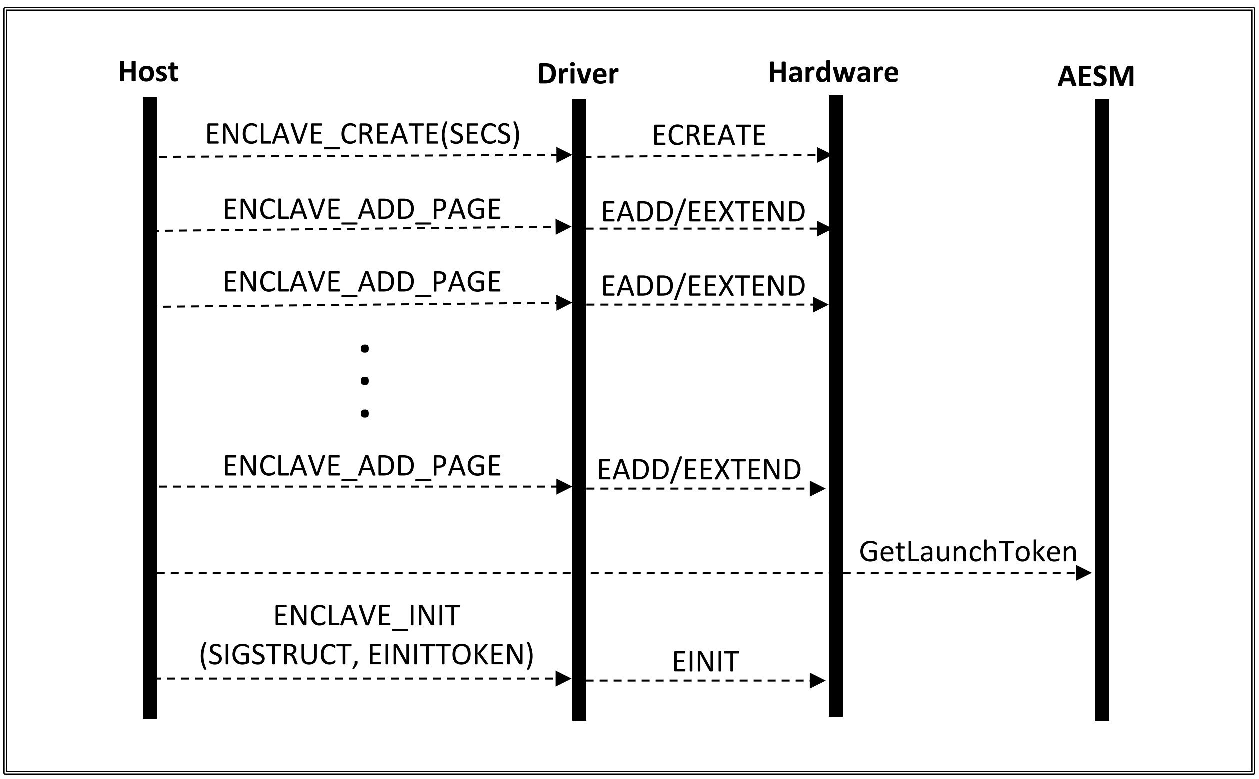
The host is responsible for creating enclaves (loading, building, and instantiating). This chapter describes this process in detail. The host does this by calling the following function.

```
OE_Result OE_CreateEnclave(
    const char* path,
    uint32_t flags,
    OE_Enclave** enclave);
```

During enclave creation, the host performs the following steps.

1. Loads the enclave image file into memory (using an ELF-64 loader)
2. Asks the Intel® SGX driver to create the enclave (execute ECREATE)
3. Asks the Intel® SGX driver to add pages to the enclave (execute EADD and EEXTEND)
4. Asks Intel® AESM service's launch enclave for a launch token
5. Asks the Intel® SGX driver to initialize the enclave (execute EINIT) passing the launch token

These last four steps are depicted in the diagram below.



The Intel® SGX driver is an **ioctl** driver that defines the requests defined in the following table.

OCTL Request	Description	SGX Instructions	SGX Structures
ENCLAVE_CREATE	Reserves enclave memory.	ECREATE	SECS
ENCLAVE_ADD_PAGE	Adds a page to the enclave and optionally extends it.	EADD, EEXTEND	
ENCLAVE_INIT	Finalizes the enclave measurement (MRENCLAVE) and initializes the enclave.	EINIT	SIGSTRUCT, EINITTOKEN

The SGX structures in the third column below are briefly described in the following table.

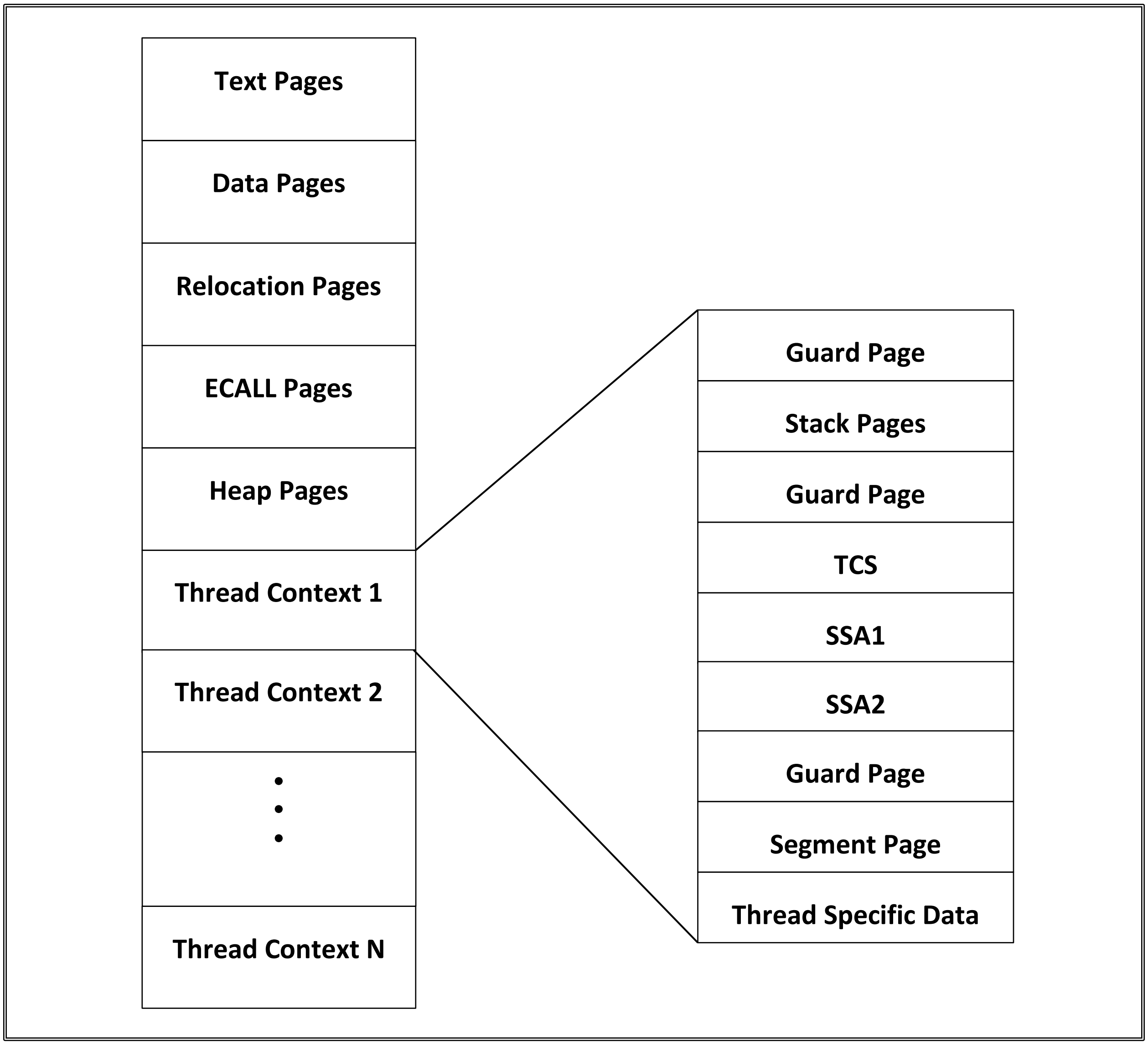
Structure	Description	SGX Instruction
SECS	Specifies the base, size, and flags of the enclave	ECREATE
SIGSTRUCT	Specifies the public key and enclave hash (MRENCLAVE)	EINIT
EINITTOKEN	An initialization token obtained from the Intel® launch enclave	EINIT

For more details about SGX instructions and structure, see the following document.

[Intel® 64 and IA-32 Architectures, Software Developer’s Manual, Volume 3D]

6. Enclave Page Layout

During enclave creation, the host asks the driver to add pages to the enclave, which results in an enclave image that resides in the enclave page cache (EPC). The diagram below depicts the layout of these pages.



The following sections describe each of these.

The Signature Section (.oesig)

The following sections refer to information obtained from the image file's signature section (.oesig), which the **oesign** utility injects into the image file during signing (see Chapter 7). This section defines the following structure.

```
typedef struct _OE_SignatureSection
{
    oe_uint64_t magic;
    OE_EnclaveSettings settings;
    SGX_SigStruct sigstruct;
}
OE_SignatureSection;
```

This structure defines **settings** (which specifies the number of heap pages, the number of stack pages, and the number of TCSs) and **sigstruct** (which contains the digital signature formed over certain pages of the image).

Text Pages

The text pages are copied from the text segment of the ELF-64 image file. The first few bytes are the ELF-64 header structure (Elf64_Ehdr). This structure contains the virtual address of the entry point (Elf64_Ehdr.e_entry), which refers to the **OE_Main()** function. During enclave creation, the following fields are cleared:

```
Elf64_Ehdr.e_shoff
Elf64_Ehdr.e_shnum
Elf64_Ehdr.e_shstrndx
```

Otherwise, the text segment is copied from the image file exactly as it is.

The host asks the driver to add and extend (measure) the text pages.

Data Pages

The **data pages** are copied from the data segment of the ELF-64 image file. OE initializes the following global variables during enclave creation.

Variable	Description
__oe_baseRelocPage	Page number of first relocation page relative to start of image
__oe_numRelocPages	Total number of relocation pages (obtained from the image file)
__oe_baseECallPage	Page number of first ECALL page relative to start of image
__oe_numECallPages	Total number of ECALL pages (obtained from the image file)
__oe_baseHeapPage	Page number of first heap page relative to start of image
__oe_numHeapPages	Total number of heap pages (obtained from .oesig section)
__oe_numPages	Total number of pages in the image
__oe_virtualBaseAddr	Virtual address of the __oe_virtualBaseAddr variable itself

The **relocation page** variables are discussed in the next section. The **heap page** variables are used to determine the boundaries of the heap (used by the heap allocator). The **__oe_virtualBaseAddr** is the virtual address of the variable itself. The real base address of the enclave can be obtained by subtracting this value from the address of the variable itself. For example:

```
(unsigned char*)&__oe_virtualBaseAddr - __oe_virtualBaseAddr;
```

The host asks the driver to add and extend (measure) the data pages.

Relocation Pages

The **relocation pages** contain symbol relocations that must be applied the first time the enclave is entered. The enclave must be self-relocating since applying relocations during enclave creation would obviously break the enclave measurement (MRENCLAVE). Recall from the table above, that the following variables tell the enclave where it can find the base relocation pages and how many there are.

```
__oe_baseRelocPage
__oe_numRelocPages
```

The content of these pages consists of zero or more structures defined below.

```
typedef struct _OE_Reloc
{
    oe_uint64_t offset;
    oe_uint64_t info;
    oe_int64_t addend;
}
OE_Reloc;
```

This structure has the same layout as the **Elf64_Rela** structure.

OE_Reloc field	**Description**
offset	Virtual offset to symbol to be relocated (zero marks last struct)
info	Ignored if not R_X86_64_RELATIVE
addend	Add to enclave's base address to relocate the symbol

The host asks the driver to add and extend (measure) the relocation pages.

ECALL Pages

The ECALL pages contain the virtual addresses of all ECALL functions in the enclave. The following structure defines the content of these pages.

```
typedef struct _OE_ECallPages
{
    uint64_t magic;
```

```
    uint64_t num_vaddrs;
    uint64_t vaddrs[];
}
OE_ECallPages;
```

When a host performs an OCALL, it passes a **function number** and the expected **virtual address** of the function. The function number is an index into the **OE_ECallPages.vaddrs** array. The enclave checks that this index is within bounds and if so obtains the virtual address from the array. Then the enclave checks that the virtual address passed by the host matches the virtual address obtained from the array. If all checks are valid, the enclave adds the virtual address to the enclave's base address to obtain and call the function.

Heap Pages

The heap pages are zero-filled pages. Recall that the number of heap pages is obtained from the signature structure (read from the enclave image file). The following global variables (in the data pages) tell the enclave where the heap pages are located and how many there are.

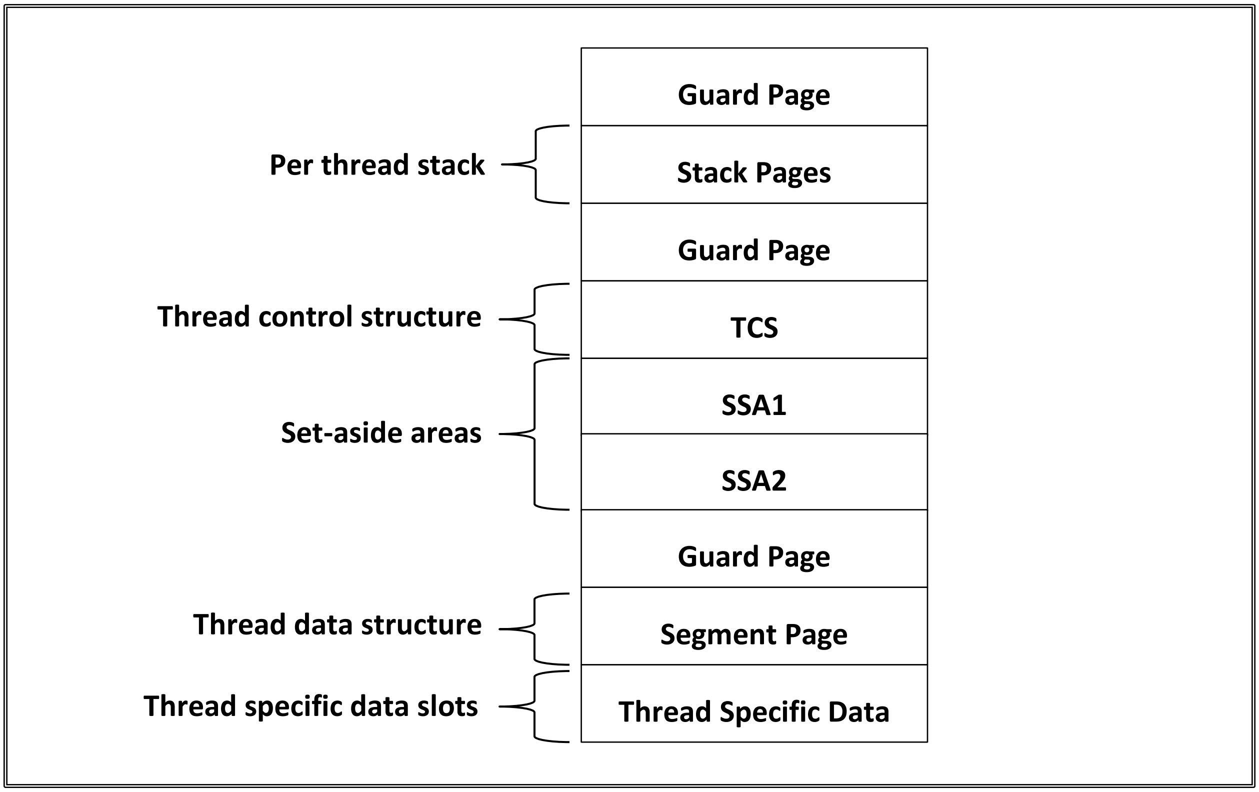
```
__oe_baseHeapPage
__oe_numHeapPages
```

The heap allocator relies on these variables as well as the base address of the enclave.

The host asks the driver to add **but not extend** the heap pages.

Thread Context

There is a thread context for each thread control structure (TCS). Each thread context has the following layout.



Guard Pages

Non-readable and non-writable guard pages are injected into the thread context at various places. These will cause a fault if the stack is underflowed or overflowed.

Per Thread Stack

Each TCS has its own stack pages. The number of stack pages per TCS is determined during enclave signing (using the **eisign** tool). The enclave creation code reads this information from the signature section (.oesig) of the enclave image file. These pages are zero-filled pages.

Thread Control Structure

The thread control structure or TCS is initialized during enclave creation but not accessible to the enclave at runtime. The SGX instructions read and write information to the TCS page during enclave entry and exit. The TCS structure is defined below. The initial value provided during enclave creation is shown in the second column.

TCS field	Initially	Description
oe_uint64_t state	0	0 indicates that TCS is available. EENTER sets to 1.
oe_uint64_t flags	0	

oe_uint64_t ossa	address	Virtual address of the first set-aside area slot
oe_uint32_t cssa	0	Current set-aside area slot
oe_uint32_t nssa	2	The number of set-aside areas
oe_uint64_t oentry	address	Virtual address of the entry point: OE_Main()
oe_uint64_t aep	0	EENTER stores asynchronous-exception handler here
oe_uint64_t fsbase	address	Virtual address of segment page
oe_uint64_t gsbase	address	Virtual address of segment page
oe_uint32_t fslimit	0xFFFFFFFF	
oe_uint32_t gslimit	0xFFFFFFFF	

Set-Aside Areas

The SGX hardware uses the set-aside areas (**SSAs**) when an exception (or fault) occurs. The hardware stores the current state in the next available SSA (**TCS.cssa** + 1) and then exits to the host, invoking the host's AEP (Asynchronous Exception Procedure). The host may opt to resume enclave execution by executing the ERESUME instruction. If so, then OE_Main() is invoked again, this time with **TCS.cssa** greater than zero (indicating an exception).

Note: currently OE enclaves do not support exception handling. Currently they are treated as fatal events.

Segment Page

The SGX EENTER instruction initializes the GS segment register (X86-64bit) to refer the segment page. The segment page is initially zero-filled. When the enclave is entered for the first time, it is initialized with the **thread data (OE_ThreadData)** structure, which defines the following key fields.

OE_ThreadData Fields	Description
self_addr	Self-pointer to thread data structure
last_sp	The last stack pointer (used for enclave reentry)
initialize	Whether the enclave has been initialized
depth	The depth of the ECALL stack (for ECALL nesting)
host registers	Fields for saving and restoring host registers
callsites	A list of structures for resuming after an OCALL
simulate	Whether simulation mode is active

The address of this structure serves as the current thread identifier, returned by **OE_ThreadSelf()**.

Thread-Specific Data Page

OE defines one thread-specific data page per thread, for a total of 512 8-bytes slots (4096 / 8). These slots are managed by the following functions.

Function	Description
OE_ThreadKeyCreate()	Creates a thread-specific data key
OE_ThreadKeyDelete()	Delete a thread-specific data key
OE_ThreadSetSpecific()	Sets thread-specific data for the given key
OE_ThreadGetSpecific()	Gets thread-specific data for the given key

Although the page itself is limited to 512, the implementation can be extended later to support more slots by using indirection to additional slot pages.

7. The Exception handling

Hardware exception

When a hardware exception happens inside enclave, the hardware will save the processor state inside the enclave (in the thread's current SSA frame), and set the processor state to a synthetic state to prevent any leakage of secrets, and exit the enclave mode. To the host side, it looks like an exception happens in the AEP. To enable hardware exception handling inside enclave, we use the two pass exception mechanism, see below:

Hardware exception handling workflow

Host	Hardware	Enclave
Enter enclave		
	EENTER	
		…
		Hardware exception happens
	AEX	
	Exit enclave mode	

Signal handler get the exception, call the first pass exception handler to solve it		
	EENTER	
		First pass exception handler is called. It saves the minimal exception context inside enclave, and return to host.
	EEXIT	
The exception is handled from host point of view, signal handler returns. The AEP will be executed again.		
	ERESUME	
		Second pass exception handler is called. It call the registered exception handlers one by one. If one handler returns EXCEPTION_CONTINUE_EXECUTION, it will continue execution on the context. Otherwise the enclave will be aborted because un-handled exception happened.

Vector exception handling functions

We provide two functions to help developers to handle the hardware exceptions inside enclave.

- Register a new vector exception handler.

Call this function to add a new vector exception handler. If success, the registered handler will be called when an exception happens inside enclave.

```
void* OE_AddVectoredExceptionHandler(
    uint64_t      isFirstHandler,
    POE_VECTORED_EXCEPTION_HANDLER vectoredHandler);
```

- Remove an existing exception handler.

Call this function to remove a registered exception handler. If success, the registered handler will not be called when an exception happens inside enclave

```
uint64_t OE_RemoveVectoredExceptionHandler(
    void* vectoredHandler);
```

8. The Signing Tool

OE provides a signing tool for signing enclaves. This tool injects a signature section into the ELF-64 image file called **.oesig**. The content of this section is determined by three sources.

- Fields defined in the signing configure file
- The enclave's measurement (MRENCLAVE)
- The enclave's private key

The signature section is defined as follows.

```
typedef struct _OE_SignatureSection
{
    oe_uint64_t magic;
    OE_EnclaveSettings settings;
    SGX_SigStruct sigstruct;
}
OE_SignatureSection;
```

The **settings** field is defined by the following structure.

```
typedef struct _OE_EnclaveSettings
{
    oe_uint64_t debug;
    oe_uint64_t numHeapPages;
    oe_uint64_t numStackPages;
    oe_uint64_t numTCS;
}
OE_EnclaveSettings;
```

All settings are derived directly from the signing configuration file.

The **sigstruct** field is an SGX-defined data structure, defined in **Section 38.13** of the following document (see SIGSTRUCT).

Intel® 64 and IA-32 Architectures, Software Developer's Manual, Volume 3D

The signing tool fills in this structure and passes it to the driver during enclave initialization.

Example

This example shows how to sign an enclave. Three input files are needed:

- An enclave image file

- A signing configuration file
- A private key

Suppose that the enclave image file has already been created with the following name.

```
myenclave.so
```

Next, we define a signing configuration file called **myenclave.conf** with the following contents.

```
NumHeapPages=1024
NumStackPages=1024
NumTCS=2
```

Then, we generate a self-signed key with the following command.

```
$ openssl genrsa -out myenclave.key -3 3072
```

Finally, we are ready to sign the enclave file.

```
$ oesign myenclave.so myenclave.conf myenclave.key
Created myenclave.signed.so
```

To verify that myenclave.signed.so contains the signature section, use the following command.

```
$ readelf -S myenclave.signed.so | grep oesig
  [25] .oesig            PROGBITS         0000000000000000  00022150
```

This signature section will be used during enclave creation.

9. The Debugger Extension

We can't use GDB directly to debug enclave application since it doesn't understand enclave yet. OpenEnclave includes a GDB plugin to help developers to debug enclaves that is developed using this SDK.

The debugger extension is consist of two parts: ptrace library and python extension. And to help the developer to debug the enclave call in and call out function, we support the stack stitching feature that will show one complete stack across enclave and host call stack.

Ptrace Library

The Ptrace library(oe_ptrace.so) library implements the customized ptrace and waitpid function to get and set enclave registers, and fix the enclave breakpoint. This library will be preloaded into the GDB by oe-gdb script.

Python Extension

The GDB python extension is used to load enclave symbol, enable enclave debugging, and stitch stacks etc. It will be loaded into GDB as a plugin. Please refer to https://sourceware.org/GDB/onlinedocs/GDB/Extending-GDB.html for GDB python extension model.

Stack Stitching

For stack stitching, the basic idea is to setup the stack frame that GDB can do stack walk.

Give a PC, and rbp, rsp value of current frame, how can GDB the previous frame? The default logic is to find the previous rbp at memory of current rbp value, and find the previous rip at memory of current rbp+8. Here is an example:

```
(GDB) info frame
Stack level 0, frame at 0x7fffffffdc20:
rip = 0x40b0d5 in HandlePrint
(/home/lei/repos/openenclave/host/ocalls.c:40); saved rip =
0x403c56
called by frame at 0x7fffffffdc70
source language c.
Arglist at 0x7fffffffdc10, args: argIn=140737488346504
Locals at 0x7fffffffdc10, Previous frame's sp is
0x7fffffffdc20
```

```
Saved registers:
rbp at 0x7fffffffdc10, rip at 0x7fffffffdc18
```

In the nutshell, the stack stitching is to setup the stack frame in a GDB expected way to GDB to find the "correct" previous stack frame.

Note: GDB doesn't have to use rbp to do stack walking. We use some cfi directives to change the rule for computing the CFA. Please refer to: https://sourceware.org/binutils/docs-2.16/as/CFI-directives.html.

Split stack

In normal cases, the stack is and should be one continuous memory block. But we have two stacks in a "thread", one is for host, another is for enclave. These two stacks are discontinuous, and our code stitches these two stacks together. When GDB does stack walking on these stack by default, it will fail because the stack looks like corrupted (sanity check fails). Fortunately, GCC supports a feature called split stacks to permit a discontinuous stack which is grown automatically as needed. Refer to: https://gcc.gnu.org/wiki/SplitStacks.
To support the split stack feature, GDB uses a marker to disable the sanity check. It hardcodes the __morestack function name, and will bypass the sanity check when the stack is not discontinuous on that function. Refer to GDB\binutils-GDB\GDB\frame.c in GDB source code. Following this GDB convention, we use the hardcode function name (__morestack) in stack switching. Without that, the GDB will tell developer the stack is corrupted when he issues a back trace command on a stitched stack.

10. The IDL Generator

OE provides an IDL generator called **oegen** for producing type-safe wrappers for performing ECALLs and OCALLs. Here are the major design decisions.

- The oegen utility is an add-on rather than an integral part of the OE intrinsics. This allows users to utilize alternative IDL technologies.
- The enclave stubs automatically copy parameters to and from host memory.
- All stubs are generated for the C language.
- The IDL language supports definition of structures and functions.
- The generated stubs perform limited buffer overrun checking.
- The utility generates general-purpose metadata that could be used for other purposes.
- The utility provides constraints modifiers on functions and parameters.

Supported Data Types

The oegen tool supports the following data types.

IDL type	Generated type
char	char
short	Short
int	Int
long	long
int8	oe_int8_t
uint8	oe_uint8_t
int16	oe_int16_t
uint16	oe_uint16_t
int32	oe_int32_t
uint32	oe_uint32_t
int64	oe_int64_t
uint64	oe_uint64_t
float	float
double	Double
bool	oe_bool
size_t	oe_size_t
ssize_t	oe_ssize_t
wchar_t	oe_wchar_t
void	Void
signed char	signed char
signed short	signed short
signed int	signed int
signed long	signed long
unsigned char	unsigned char
unsigned short	unsigned short
unsigned int	unsigned int
unsigned long	unsigned long

Function and parameter constraints

Constraint	Scope	Meaning
ecall	function	This function is an ECALL
ocall	function	This function is an OCALL
in	parameter	This is an input parameter
out	parameter	This is an output parameter
inout	parameter	This is an input and an output parameter
ref	parameter	This is a reference parameter (pointer to pointer)
unchecked	parameter	This parameter is unchecked (passed through as is)
count(arg)	parameter	The cardinality of this array is given by arg, where arg is a constant or the name of another parameter
one	parameter	The cardinality of this array is one
string	parameter	The parameter is a C string

Example 1

```
function [ocall] void WriteFile(
    [inout, one] struct File* file,
    [in, count=size] const void* data,
    [in] size_t size);
```

Example 2

```
struct Person
{
    [string] const char* first;
    [string] const char* last;
    uint16 age;
};

function [ecall] int CreatePerson(
    [in, one] struct Person* person);
```