Exception Handling in Intel® Software Guard Extensions (Intel® SGX) Applications

Scope
This article describes how HW exceptions are handled in Intel® Software Guard Extensions (SGX) enclaves. The article assumes knowledge of Intel SGX. For general information on Intel SGX, visit the Intel SGX portal at https://software.intel.com/en-us/sgx.

Introduction
Intel® SGX enabled applications are divided into two logical components, as shown in Figure 1. These components are:

1. **Trusted component**—The part of the application code that accesses the secret. This component of the application code is also referred to as an “enclave.” An application can have more than one enclave.
2. **Untrusted component**—The remainder of the application, including all of its modules, which does not access the secret.

**Figure 1. Intel SGX application architecture diagram**

Intel provides special hardware instructions to create and support enclaves. Intel SGX enclaves use the same OS and hardware as other applications on the system. This allows applications that harness Intel SGX enclaves to make use of capabilities and features provided by the OS, such as multi-threading and exception handling.

Exception handling is the process of responding to the occurrence, during computation, of exceptions: anomalous or exceptional conditions requiring special processing, which can often change the normal flow of program execution. In general, an exception is handled (that is, resolved) by saving the current state of execution in a predefined place and then switching the execution to a specific subroutine known as an exception handler. If
an exception is resolved, then the handler can resume the execution from the point of the exception using the saved context information.

Typical exception handling
Let's look at that transition to and from of a typical exception handler before discussing the differences with respect to enclaves. Figure 2 shows the content of a typical exception handler stack that contains, EFLAGS, CS, and EIP register contents while handling the exception.

![Figure 2. Handler stack with execution context](image)

When an exception occurs, the processor does the following before calling an exception handler:

- Pushes the current contents of the EFLAGS, CS, and EIP registers onto the stack
- Pushes an error code onto the stack
- Loads the segment selector for the new code segment and the new instruction pointer into the CS and EIP registers, respectively
- Begins execution of the handler's procedure

A return from an exception is initiated with the IRET instruction. When executing a return from an exception handler, the processor performs the following actions:

- Restores the CS and EIP registers to their values prior to the exception
- Restores the EFLAGS register
- Increments the stack pointer appropriately
- Resumes execution of the interrupted procedure

Intel SGX exception handling
Although Intel SGX applications support exception handling, exception handling in these applications differs from that in conventional applications in several ways. The purpose of
these differences is to maintain the trusted nature of the enclave. Let's look at a basic flow of Intel SGX exception handling to highlight the differences:

1. A HW exception is encountered in the enclave. This initiates an Asynchronous Exit (AEX) from the enclave. An AEX involves the following steps:
   a. If the exception is an enclave-supported exception, then information on the exception is put into the EXITINFO area of the current State Save Area (SSA). (The SSA is an area allocated within the enclave specifically to hold thread state information when an exception or interrupt occurs while processing inside the enclave. See the SSA section for more information about the SSA.)
   b. Enclave thread state is stored away into the SSA.
   c. A Synthetic State is loaded into the context of the thread. This puts the Asynchronous Exit Pointer (AEP) into the Instruction Pointer (IP). The AEP is an address that is provided when calling into the enclave. Intel provides a library, the Untrusted Run-Time System (uRTS), which handles making calls into SGX Enclaves. The uRTS selects an AEP address that is prepared to handle the exception and a subsequent return back to the enclave. This makes it look like the exception occurred in the uRTS code outside the enclave. Limited information about the exception is provided to the untrusted code.
   d. The exception handling flow now occurs, which is the same flow as for the typical exception handling case, but this time starting with Synthetic State information.

2. If the OS (Windows or Linux) cannot handle the exception, then it generates one of the following:
   a. Structured Exception in Microsoft* Windows*
   b. Signal in Linux*

3. To get the first chance at handling the exception, the uRTS installs one of the following:
   a. Structured Exception Handler in Microsoft Windows
   b. Signal Handler in Linux

The handler calls into the enclave to ask the enclave to handle the exception. Inside the enclave, Intel provides a library, the Trusted Run-Time System (tRTS), which is linked into the enclave image by the enclave developer. The tRTS receives this call and first makes sure that the enclave is in an "exception state". (For security purposes, the tRTS makes sure that there was an AEX first, and then checks the SSA frame exit info structure to see if there was an SGX supported exception; Table 1 lists supported exceptions). The tRTS may handle some exceptions itself. For example, in SGX2, Linux support will be added to expand the stack on a PF# exception. (For a PF# exception with SGX2 and Linux, the uRTS and tRTS will work together to add a page to the stack.) Generally, however, the tRTS starts
calling developer-installed exception handlers, looking for one that says it handled the exception.

Table 1: Intel SGX-supported exception vectors

<table>
<thead>
<tr>
<th>Name</th>
<th>Vector #</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>#DE</td>
<td>0</td>
<td>Divider exception.</td>
</tr>
<tr>
<td>#DB</td>
<td>1</td>
<td>Debug exception.</td>
</tr>
<tr>
<td>#BP</td>
<td>3</td>
<td>Breakpoint exception.</td>
</tr>
<tr>
<td>#BR</td>
<td>5</td>
<td>Bound range exceeded exception.</td>
</tr>
<tr>
<td>#UD</td>
<td>6</td>
<td>Invalid opcode execution.</td>
</tr>
<tr>
<td>#GP</td>
<td>13</td>
<td>General protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.</td>
</tr>
<tr>
<td>#PF</td>
<td>14</td>
<td>Page protection exception. Only reported if SECS.MISCSELECT.EXINFO = 1.</td>
</tr>
<tr>
<td>#MF</td>
<td>16</td>
<td>X87 FPU floating-point error.</td>
</tr>
<tr>
<td>#AC</td>
<td>17</td>
<td>Alignment check exceptions.</td>
</tr>
<tr>
<td>#XM</td>
<td>19</td>
<td>SIMD floating-point exceptions.</td>
</tr>
</tbody>
</table>

Intel SGX thread-related control structures

Each enclave contains one SGX Enclave Control Structure (SECS), one or more Thread Control Structure (TCS), with each TCS containing one or more State Save Area (SSA) frames. Technical details about the SECS, TCS, and SSA are provided in Section 38 of the Intel® 64 and IA-32 Architectures Software Developer's Manual: Volume 3D.

SECS

The SECS is a data structure that contains meta-data about the enclave which is used by the Intel SGX hardware and cannot be directly accessed by software. This data includes information like enclave size and base address, attributes, and measurement.

TCS

Each executing thread is associated with a TCS. The TCS is a data structure that contains meta-data used by the hardware to save and restore thread specific information when entering/exiting the enclave. This information also includes information about SSAs, for example, the base address Offset of the SSA (OSSA), the slot index for the Current SSA (CSSA), and the total Number of SSAs (NSSA).

SSA

The SSA is a data structure for storing the execution context. The SSA is composed of one or more frames each of which store the architectural state of an enclave thread before an AEX occurs. The architectural state includes general purpose register state, XSAVE area
state, and a miscellaneous region for extended information. An SSA is allocated for each TCS during the creation of an enclave.

On entry into the enclave, the TCS must point to an SSA, which can contain multiple frames. The EENTER instruction makes sure that there is an SSA frame ready to accept thread context information in the event of an exception or interrupt. The SSA is then populated as shown in Figure 3, when the exception occurs.

Figure 3. Stack values with an SSA frame

When a “thread context” is created during the creation of the enclave, it must populate the TCS page with certain parameters such as the location of the SSA (as an offset from the enclave base address) for the thread. The TCS keeps track of the Current SSA frame (CSSA) by incrementing CSSA on each AEX and decrementing it on each ERESUME. EENTER and ERESUME also verify that SSA Frame pages are present in memory and ready to be populated on an AEX. The CSSA value is provided to the enclave by the EENTER instruction.
Figure 4. Obtaining the SSA Frame

The SSA frame contains various fields to store current execution context:

- General-purpose registers (GPRs) and EXITINFO are saved to a fixed-size GPR area at the top of the SSA frame (in virtual machine control structure [VMCS] format). (See Table 38.8 in the Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 3D for details.)
- Intel® Advanced Vector Extensions (Intel® AVX), x87, Streaming Single-Instruction Multiple Data (SIMD) Extensions (SSE), and other states are saved into a variable-size XSAVE area at the bottom of an SSA frame (in XSAVE format).
- The XFRM (XCR0 format) field in SECS controls the size of the XSAVE area and is contained in the upper bits of the SECS.ATTRIBUTES field defined by the ECREATE instruction.
- MISC area fields supported by the CPU are enumerated by CPUID.SE_LEAF.0.EBX.

Since the TCS can contain more than one SSA frame, the address of a specific SSA frame and its various fields can be calculated as follows:

The GPRs are stored in the GPR region. The address of the GPR region is then:

$$SSA \text{ GPR} = (\text{EnclaveBaseAddress} + \text{TCS. OSSA}) + (\text{TCS.CSSA} \times \text{SSAFramesize}) - \text{GPRSize}$$

The SSAFRAMESIZE field in SECS defines the size of each SSA frame in pages. The size of an SSA Frame must be large enough to hold the regions in the SSA frame.
Extended information is stored in the XSAVE region. The fields that will be populated in the XSAVE region are provided in SECS.ATTRIBUTES.XFRM. If a field is not populated in the XSAVE region, then the registers will not be used inside the enclave. The Async Exit will not clear these registers and thus they can be read directly by a debugger. The formula for calculating the base of the XSAVE region is then:

$$\text{SSA XSAVE} = (\text{EnclaveBaseAddress} + \text{TCS.OSSA}) + ((\text{TCS.CSSA} - 1) \times \text{SSAFrameSize})$$

The base of the MISC region is located directly below the GPR region:

$$\text{SSA MISC} = \text{SSA GPR} - \text{MISCSize}$$

**CSSA**
The CSSA in a TCS is the slot index pointing to the Current SSA frame. Like single-threaded applications, multi-threaded applications use the SSA frame to store current execution context. Each TCS inside an enclave has its own architectural state. Thus, during an AEX event, the architectural state of each thread is saved before an exit occurs.

When invoking an enclave thread, an EENTER instruction checks whether an SSA frame is available. When an AEX event occurs, the enclave thread gets suspended and resumed back to the execution by the ERESUME instruction.

When an AEX event occurs, the CSSA is incremented to point to the current CSSA frame. When the ERESUME occurs, the CSSA is decremented.

**EXITINFO field in SSA**
EXITINFO contains the information used to report the exit reason to the software inside the enclave. EXITINFO is a 4-byte field as defined in Table 2.

<table>
<thead>
<tr>
<th>Field</th>
<th>Bit Position</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VECTOR</td>
<td>7:0</td>
<td>Exception vector number reported inside the enclave. Must be one of the numbers in Table 1.</td>
</tr>
<tr>
<td>EXIT_TYPE</td>
<td>10:8</td>
<td>001b: Hardware exceptions&lt;br&gt;110b: Software exceptions&lt;br&gt;Other values: Reserved</td>
</tr>
<tr>
<td>RESERVED</td>
<td>30:11</td>
<td>Reserved as zero</td>
</tr>
<tr>
<td>VALID</td>
<td>31</td>
<td>0: Unsupported exceptions&lt;br&gt;1: Supported exceptions. Includes two categories:&lt;br&gt;Unconditionally supported exceptions: #DE, #DB, #BP, #BR, #UD, #MF, #AC, #XM&lt;br&gt;Conditionally supported exceptions: #PF, #GP if SEC.MISCSELECT.EXINFO = 1</td>
</tr>
</tbody>
</table>

The VALID bit is set only for exceptions which are supported inside the enclave. If an exception is not supported inside the enclave, then VECTOR and EXIT_TYPE are cleared.
Exception Handling in Enclaves

When an AEX event occurs, the sequence of operation shown in Figure 4 occurs. (The word trampoline refers to the fact that exception processing bounces through the application to get to the OS for exceptions and back through the application to resume.)

1. During an AEX event, after storing the CPU-state information in the SSA frame, the Instruction Pointer (RIP) is modified to point to the trampoline area in the untrusted component of the application. This is called the Synthetic context or state. The RIP eventually is included in the exception information context. At the same time, synthetic information about the exception is loaded into registers during the AEX event (actual CPU-state information from the AEX event remains stored inside the enclave).

2. After executing the exception handler, the IRET instruction will return the flow control to the trampoline area that resides in the untrusted component of the application. Because the OS, by definition, runs in an untrusted environment, the trampoline is represented as an execution point for the OS where exceptions occur.

3. The OS will see the exception and call its exception handling module. If the exception is something that it cannot handle, then it will generate an exception to the application. The actual HW exception such as #PF, #UD, or #GP will be translated into a Window application exception such as STATUS_ACCESS_VIOLATION or Linux Signal such as SIGSEGV. Only after the exception is fully handled will the OS issue IRET.

4. After the control reaches the trampoline area, it will execute the ERESUME instruction stored there. Control will then be transferred back to the enclave.
5. The register state is then restored from the SSA, and execution resumes from the earlier, interrupted location.

Page Fault Handling in Intel SGX
Program faults such as division by zero, breakpoint instructions, undefined instructions, and floating point/SIMD exceptions are reported in the EXITINFO in the SSA.GPR area.

In SGX2, Page faults (#PF) and general-protection faults (#GP) may also be reported to an enclave by configuring MISCELECT. (Details on MISCELECT are provided in Section 38.7 of the Intel® 64 and IA-32 Architectures Software Developer’s Manual: Volume 3D.) When a #PF occurs, the page that the #PF occurred on is written to CR2 so that the OS can process it. The lower 12 bits are cleared so that it only points to the base of the 4kb page (the OS only needs the page in which the fault occurred). If MISCELECT is configured to record #PF inside the enclave, then the #PF code and address will be recorded in the SSA EXITINFO structure.

In SGX2, the Enclave Dynamic Memory Management (EDMM) feature will give SGX support for adding pages to running enclaves and for changing enclave page attributes such as access permissions. Page Fault information within the EXITINFO structure of the SSA will help the enclave run-time respond to Page Faults.

Intel SW support vs developer’s responsibilities

This list below summarizes what Intel SW takes care of vs. what developers need to do in their applications with regard to exception handling in Intel SGX-enabled applications.

Developer

- Define the Number of enclave thread contexts, stack size, heap size, etc., in the enclave.config.xml file (this is an input to the enclave signing tool provided in the SDK). The enclave signing tool inserts this information into the enclave file where the enclave loader tRTS sgx_create_enclave() function reads it.
- Define Calls into the enclave (ECALLs) and out of the enclave (OCALLs) in the Enclave Definition Language (EDL) file. The SGX SDK Edger8r Tool parses the EDL file (filename.edl) and creates a proxy/stub, which is then compiled into the application.
- Define Exception Handling Functions within the enclave and register the handlers in your enclave code by calling the tRTS register_exception_handler() function. See the Intel® Software Guard Extensions SDK Developer Reference for details.
- Note: Exception handing includes transitions between the enclave, the application, and the OS. These transitions introduce some overhead due to the AEX and ERESUME instructions that accompany them. Therefore, to the extent possible, developers should
minimize exceptions occurring within the enclaves. (Nested exceptions involving multiple threads necessarily lead to additional overhead.)

**Intel SW**

- Configures TCS and SSA for each thread running within the enclave and loads these when loading the enclave.
- Makes calls into and out of the enclave for the application. This includes selection of a TCS for each call. The developer just needs to compile in the Edger8r produced proxy/stub code.
- Catches exceptions in the uRTS, then calls into a tRTS function within the enclave. The tRTS will then call the Developer's registered exception handling functions.

**Summary**

Intel SGX supports the handling of several general exceptions such as floating-point-operation exceptions and undefined-instruction exceptions within an enclave. (In SGX2, support for handling PF# and GP# exceptions will be added.)

To effectively handle these exceptions originating within an enclave while protecting CPU state information specific to an enclave, Intel SGX relies on methodologies and data structures that hide the inner details of the enclave. Intel provides special support within the Intel SGX Untrusted Run-Time (uRTS) and SGX Trusted Run-Time (tRTS) libraries to help developers use these methodologies and data structures. This helps eliminate some of the overhead for developers in processing exceptions within enclaves.

Developers should understand the how exception handing works with Intel SGX applications and take advantage of the support provided by the Intel SGX SDK to ease their development efforts in this area.

**References**


A "Mission Critical Application" is any application in which failure of the Intel Product could result, directly or indirectly, in personal injury or death. SHOULD YOU PURCHASE OR USE INTEL'S PRODUCTS FOR ANY SUCH MISSION CRITICAL APPLICATION, YOU SHALL INDEMNIFY AND HOLD INTEL® AND ITS SUBSIDIARIES, SUBCONTRACTORS AND AFFILIATES, AND THE DIRECTORS, OFFICERS, AND EMPLOYEES OF EACH, HARMLESS AGAINST ALL CLAIMS COSTS, DAMAGES, AND EXPENSES AND REASONABLE ATTORNEYS' FEES ARISING OUT OF, DIRECTLY OR INDIRECTLY, ANY CLAIM OF PRODUCT LIABILITY, PERSONAL INJURY, OR DEATH ARISING IN ANY WAY OUT OF SUCH MISSION CRITICAL APPLICATION, WHETHER OR NOT INTEL® OR ITS SUBCONTRACTOR WAS NEGLIGENT IN THE DESIGN, MANUFACTURE, OR WARNING OF THE INTEL® PRODUCT OR ANY OF ITS PARTS.

Intel may make changes to specifications and product descriptions at any time, without notice. Designers must not rely on the absence or characteristics of any features or instructions marked "reserved" or "undefined". Intel reserves these for future definition and shall have no responsibility whatsoever for conflicts or incompatibilities arising from future changes to them. The information here is subject to change without notice. Do not finalize a design with this information.

No computer system can provide absolute security under all conditions. Built-in security features available on select Intel® processors may require additional software, hardware, services and/or an Internet connection. Results may vary depending upon configuration. Consult your system manufacturer for more details.

Intel®, the Intel® Logo, Intel® Inside, Intel® Core™, Intel® Atom™, and Intel® Xeon® are trademarks of Intel Corporation in the U.S. and/or other countries. Other names and brands may be claimed as the property of others.

* Other names and brands may be claimed as properties of others.

Copyright © 2017 Intel® Corporation