Intel® Software Guard Extensions (Intel® SGX) Software Support for Dynamic Memory Allocation inside an Enclave

Bin (Cedric) Xing, Mark Shanahan, Rebekah Leslie-Hurd
Intel Corporation

{cedric.xing, mark.shanahan, rebekah.leslie-hurd }@intel.com

ABSTRACT

Intel® Software Guard Extensions (Intel® SGX) SGX2 extends the Intel® Software Guard Extensions (SGX) instruction set and enables software developers to dynamically manage memory within the SGX environment. This paper reviews the current SGX Software Run-Time Environment and proposes additions to the framework which will allow developers to benefit from features enabled by SGX2 such as dynamic heap management, stack expansion, and thread context creation.

1 INTRODUCTION

Intel® Software Guard Extensions (SGX) Technology allows software to create an execution environment with confidentiality and integrity protection. The instructions used to create this protected area, also known as an 'enclave', are described in [1]. This initial set of instructions require that all enclave memory be loaded and verified during enclave build time and before code is permitted to execute within the enclave. As described in [2], this imposes some limitations on enclave developers.

All enclave memory must be committed at enclave build time. This requires the developer to predict and use maximum heap and stack sizes in the enclave build. Likewise, additional code modules cannot be dynamically loaded into the enclave environment after enclave build. This increases enclave build time and limits the enclave's ability to adapt to changing workloads.

Additionally, page protections cannot be changed for an enclave memory. Executable code containing relocations must be loaded as Read, Write, and Execute (RWX) and remain that way for the life of the enclave. This also limits the capabilities of garbage collectors and dynamic translators or just-in-time (JIT) compilers with the enclave.

An extension to the SGX instruction set is designed to overcome these limitations. SGX2 Extensions give software the ability to dynamically add and remove pages from an enclave and to manage the attributes of enclave pages.

In this paper, we review a programming model for SGX enclaves and how it can be adapted to take advantage of SGX2 extensions. Specifically, we discuss the enclave layout and the loading of enclaves and why dynamic memory management is of specific importance in these areas. We will also review SGX Run-Time Processes for calling into and out of an enclave and processing enclave generated exceptions with respect to how these may be adapted to take advantage of the new instructions.

This paper is divided into several sections. In Section 2, we will first present an overview of the SGX Run-Time environment including the enclave layout, loading process, and control and communication paths. In Section 3, we will review the SGX2 instructions and how they are used. Section 4 provides a programming model for dynamic memory management within an enclave and, finally, Section 5 describes the implementation of several features enabled by dynamic memory management.

Throughout this paper, we will refer to the initial set of SGX instructions as SGX1 and the new extensions allowing dynamic memory management as SGX2.

2 SGX Run-Time Environment Overview

An introduction to a programming model and an SGX Run-Time Environment are provided in [3] and [4]. This model supports development and execution of enclaves on platforms supporting SGX. We will provide a brief overview of several concepts involved in programming and executing enclaves using SGX1. These include:

- Enclave Layout and Signing
- Run-time Environment
- Communication and Control Paths

SGX2 Extensions will allow us to build off of these concepts to produce a richer programming model.

2.1 Enclave Layout and Signing

An SGX Enclave executes within the memory space of an application in a special area of physical memory known as Enclave Page Cache (EPC). The Intel® Software Guard Extensions SDK (Intel® SGX SDK) uses a paradigm of an enclave as a dynamically loaded module, such as a Dynamic Link Library (.dll) in the Windows* OS or a Shared Object (.so) in the Linux* OS. The developer can program an Enclave Library using common programming toolchains for Windows* OS and Linux* OS that have been enhanced with Intel SGX SDK tools and libraries.

Merely running the Enclave Library code within the SGX environment is not sufficient to protect user data. The enclave must also host other data segments (also referred to as enclave runtime components in later sections) related to program execution. These are shown in Figure 1

- *Enclave Heap*: if the enclave needs to dynamically allocate memory buffers, then these buffers should be allocated on a

HASP '16, June 18 2016, ,
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-4769-3/16/06…$15.00
DOI: http://dx.doi.org/10.1145/2948618.2954330

heap within the enclave.

- *Thread Context:* Each thread that enters the enclave environment needs several constructs to operate within the enclave. These constructs are collectively referred to as a Thread Context. The enclave may contain more than one Thread Context.

The Thread Context consists of several components. These are:

- *Thread Control Structure* (TCS): the TCS is an SGX defined data structure which contains metadata used by the HW to transition into and out of the enclave.
- *Thread Local Storage:* Thread Local Storage is a set of SW defined objects used by the thread and accessible to the code. This includes a Thread Environment Block used by compilers and run-time libraries, thread local variables, and the SGX State Save Area (SSA). The SSA is used to store register context when the thread asynchronously exits the enclave due to an event such as an exception or an interrupt.
- *Thread Stack:* Enclave threads should protect stack variables by using a stack located within the enclave.

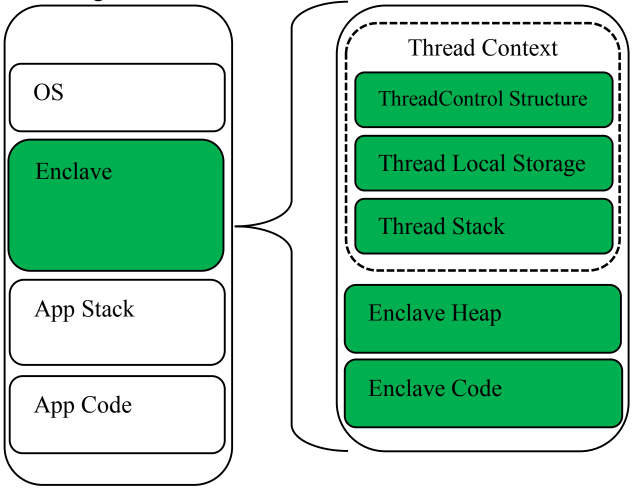

Figure 1: Detailed Enclave Layout

One drawback related to the layout of the enclave in SGX1 relates to these additional data segments. The number of these sections and their size must be statically assigned by the developer. SGX1 does not allow pages to be added to the enclave after it has been initialized and prepared to execute. Thus, the developer must define a static heap size, static number of thread contexts each with a static heap size for the enclave.

In SGX1, all pages comprising the enclave image must be loaded before the enclave can execute. This is required because all pages in the enclave image are measured to ensure the integrity of the enclave. The measurement of the enclave in the SGX environment and certain other attributes define the identity of the enclave. Entities communicating with the enclave will want attestation to this identity before sharing a secret with the enclave.

The Intel SGX SDK provides the SGX Signing Tool (hereto referred to as Signing Tool) which provides two basic functions:

- It adds metadata information to a special section in the enclave binary file. The metadata section contains enclave information and defines the extra segments including heap size, stack size, and number of threads in the enclave layout.
- It constructs an image of the enclave and measures its contents, then uses the measurement and other information to construct an SGX SIGSTRUCT[1]. The SIGSTRUCT is then digitally signed and added to the metadata information in the enclave binary file.

2.2 Run-Time Environment

Figure 2 below shows the three primary components responsible for managing SGX enclaves. The components execute in three different environments – the *Kernel Module (Driver)* executes in Ring 0, the *Untrusted Run-Time System (uRTS)* runs in untrusted Ring 3 Application space, and the *Trusted Run-Time System (tRTS)* runs inside the enclave.

These elements work together to perform two primary functions:

1. Enclave Management: this involves enclave loading, unloading, and page-swapping.
2. Call Management: manages calls into and out of the enclave including calls to handle enclave exceptions.

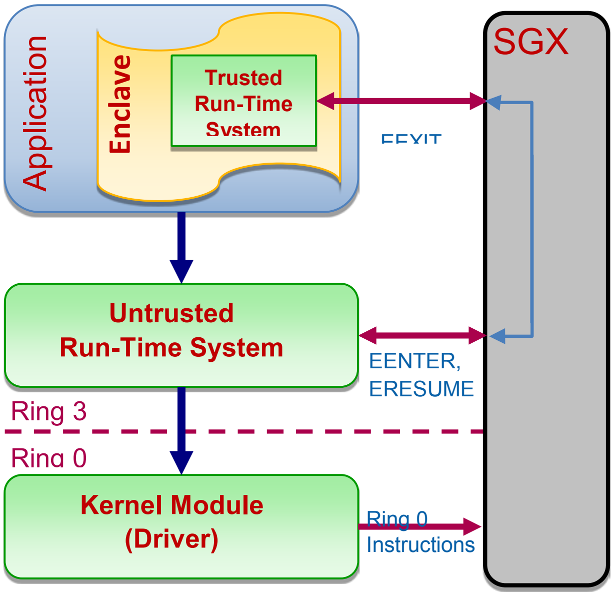

Figure 2: Intel® SGX Run-Time Environment

Enclave Management

To load an enclave, the *Untrusted Run-Time System (uRTS)* parses the enclave library file and creates an image of the enclave. The image includes both the loadable program segments from the enclave executable file (.dll or .so), the heap, and one or more thread contexts constructed from the metadata. The uRTS then sends a series of input/output control (IOCTL) messages containing enclave information and the enclave image to the *Kernel Module (SGX Driver)*. The SGX driver uses SGX Ring0 instructions to create an enclave, add each page to EPC, measure pages as they are added to EPC, and finally to initialize the enclave for execution.

The loading of the enclave is completed by calling into the *Trusted Run-Time System (tRTS)* within the enclave. The tRTS is a static library built into the enclave image. Functions that a normal program loader can perform on the program module during loading, such as relocating address references, must be performed by the enclave on itself (if address relocations were performed on the enclave image before it was loaded into EPC, then the measurement

[1] SIGSTRUCT is an SGX defined structure described in [1].

of the enclave would become dependent on the enclave load address. The measurement is crucial to establishing the identity of the enclave). The tRTS performs these functions and prepares the enclave to execute.

Unloading of the enclave is done through an uRTS function which calls the driver remove all enclave pages.

The driver uses SGX instructions to swap pages from the EPC to a backing store in regular memory. It then reloads pages back to EPC when a page faults (#PF) are generated from an access to swapped out pages.

Call Management

A call into the enclave, referred to as an ECall, is made through the uRTS. The uRTS selects a thread context to employ, configures a structured exception handler (Windows* OS) or signal handler (Linux* OS), and then issues an ENCLU[EENTER] instruction. This passes the processing into a tRTS function within the enclave. The tRTS function prepares the thread to execute securely within the enclave before execution is passed to a specific user defined function. The function return path exits the enclave via the ENCLU[EEXIT] instruction and passes back through the uRTS.

Exceptions that occur within the enclave which are not handled by the kernel, will be caught by the uRTS handler. The uRTS will pass exceptions into the enclave where they may be handled by the tRTS or be passed to user defined handlers within the enclave.

Calls out of the enclave (referred to as OCalls) originate in the tRTS and are received by the uRTS and then routed to the correct routine in the application. An OCall is essentially the reverse of an ECall.

3 Communication and Control Paths

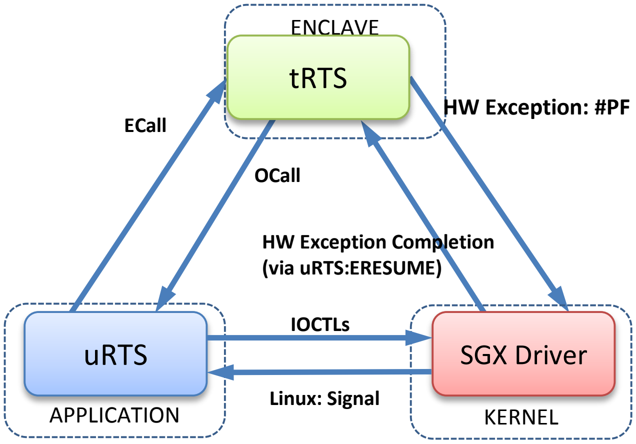

Figure 3: Communication and Control Paths

The protected environment of Intel SGX adds complexity to the management of enclave memory. The Kernel may not directly manage or modify data within the enclave. Likewise, the enclave may not directly call OS functions. To explain how the three components of the SGX Run-Time Environment, as shown in Figure 3, may work together to manage enclave memory, we must first explain the unique control and communication paths between them.

As previously explained, the uRTS and tRTS may communicate via ECalls and OCalls.

The uRTS and Driver communicate via:

- Input/Output Control (IOCTL) Messages from the uRTS to the Driver.
- Linux Signals: the SGX Driver may generate a signal which can be caught by a signal handler in the uRTS. The uRTS can then make an ECall into the enclave to effectively handle the signal. The Windows* OS does not provide a documented interface for a driver to inject an exception into an application to be subsequently caught by a structured exception handler.

The tRTS and the SGX Driver have an indirect method of communicating. In the Linux* OS, the enclave may generate a HW Exception such as a #PF which will be routed to the SGX Driver. The driver may use the fault type and fault information, such as the faulting address in the case of a #PF, to handle the exception. If the exception is handled, the driver can return processing to the enclave by continuing execution at the faulting instruction. When this is done, the execution flow actually jumps into the uRTS which then issues an ENCLU[ERESUME] instruction to resume processing at the faulting instruction within the enclave.

If the driver cannot complete the processing of the HW Exception, it may generate a Signal to the application, which will be caught by the uRTS and handled. It may also be subsequently passed to the tRTS for handling via an ECall. The tRTS can use information stored in the State Save Area (SSA) to either process the exception itself or to call a custom handler in the enclave (for information on installing a custom handler, see [5].)

The design of processes to dynamically manage enclave memory must consider the strengths and limitations of the aforementioned communication and control paths.

4 SGX2 Dynamic Memory Management

SGX2 introduces several instructions for the management of enclave memory. The full set of instructions and basic flows for their use are described in [1] and [2]. A subset of the instructions are needed for dynamic heap management, stack expansion, and thread context creation. These instructions are detailed in Table 1: SGX2 Instruction for Dynamic Memory Mgmt..

Leaf function	Description
ENCLS[EAUG]	Add a read/write accessible regular page to an initialized enclave. The page is added in a ‘pending’ state and may not be used until the enclave issues EACCEPT on the page.
ENCLS[EMODT]	Modifies the type of an existing EPC page
ENCLU[EACCEPT]	Accepts a page or page type modifications into the running enclave
ENCLU[EMODPE]	Extends access permissions of an existing EPC page
ENCLS[EMODPR]	Restricts access permissions of an existing EPC page
ENCLU[EACCEPT COPY]	Copies existing EPC page content into a PENDING page and accepts the page into the running enclave

Table 1: SGX2 Instruction for Dynamic Memory Mgmt.

A simple programming model to explicitly add a new page into an enclave is detailed in [1]. It is summarized here (note: SGX driver and Untrusted Run-Time are substituted for the OS):

1. The enclave needs a page of memory. It must keep a record of its virtual address space and then record that the page has been

reserved. Then it must make an OCall to the uRTS with the page address. The uRTS will, in turn, send a command to the driver.

2. The SGX Driver should:
   a. Ensure that the address range of the page is available for use. This can be accomplished by the driver keeping information on enclave memory allocation or by the driver working through OS memory management APIs.
   b. Select a free page within EPC (If needed, evict a page from EPC).
   c. Create a PTE and map it to the page within EPC
   d. Issue an EAUG instruction with the address of the page
   e. The page is now added to the enclave, but is in a PENDING state. A regular access to a page in this state will generate a #PF. The page must be accepted by the enclave before it can be used.
3. After the driver completes, it will complete the command and return to the uRTS, the uRTS will then complete the OCall to return processing back to the enclave. The enclave must then:
   a. Note in its internal virtual memory space record that the page has been committed to the enclave. This information should be used to ensure that a page cannot be added multiple times to the enclave.
   b. Issue an ENCLU[EACCEPT] instruction for the page to remove it from the PENDING state and allow it to be used within the enclave..

5 SGX2 Programming Model for Dynamic Memory Allocation

This section describes flows that can be used by SGX enclaves to expand or create various runtime components (e.g. expand heaps, expand stacks, or create thread contexts).

5.1 Enclave Memory Layout for SGX2

Section 3 has summarized the flow of committing new EPC pages to an existing enclave. In general, virtual address space needs to be reserved first before any EPC pages can be added.

Unlike a regular application where the virtual address space is managed solely by the OS kernel, an enclave's virtual address range is part of its measurement and must be allocated at enclave load time.

Figure 4 shows an example memory layout of an SGX2 enclave. When compared to Figure 1 the additions are the gray areas which represent reserved virtual address ranges for expandable components. For example, the enclave heap is comprised of the green bottom half and the gray top half. The bottom half is the current heap (with pages committed) while the top half marks an area for heap expansion. A similar method applies to stacks, except that they are expanding downwards (because ESP/RSP moves towards lower addresses on "push" operations). There is also reserved virtual address space, depicted in the gray boxes towards the top of the enclave virtual range, to allow more thread contexts to be committed at runtime.

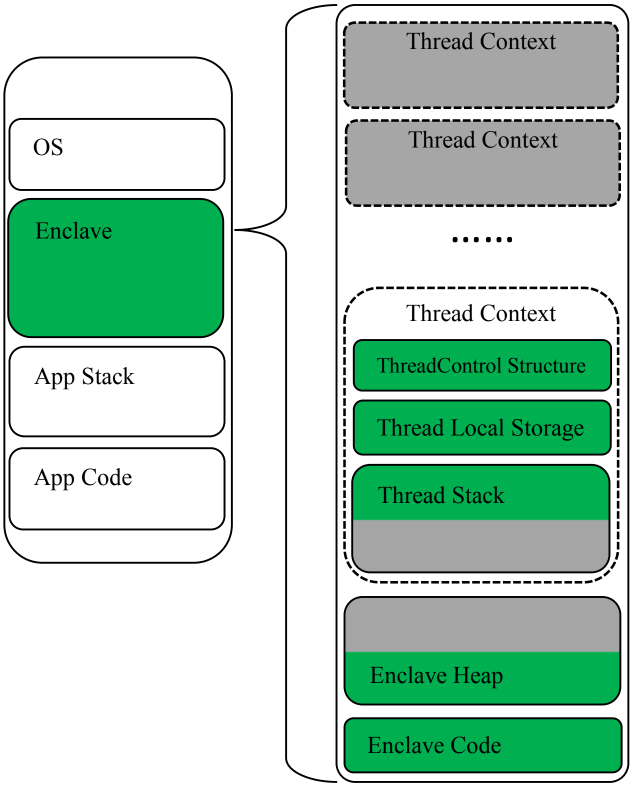

Figure 4: Enclave Layout with EDMM Support

5.2 #PF Based Page Allocation

From Section 4, we see that page allocation can is essentially a 3-step process:

1. The enclave requests additional EPC pages to be mapped at specified addresses.
2. The uRTS (with help from the SGX driver) commits (i.e. allocates, maps and invokes ENCLS[EAUG] on) EPC pages at the requested addresses.
3. The enclave accepts the newly committed pages by invoking ENCLU[EACCEPT].

As described in section 4 the aforementioned steps can be encapsulated into an OCall for the enclave to invoke when needed. However, OCalls have a high overhead. It is costly to perform a call out to the uRTS (via ENCLU[EEXIT]), send an IOCTL to the driver, and then return back to the enclave. Moreover, it is difficult to expand an already-overflowed stack using an OCall as the OCall requires stack space to execute.

A better approach is to use a page fault (#PF) to trigger the page allocation process. This is similar to how today's operating systems expand heaps and stacks for regular applications. That is:

1. The enclave accesses a yet-to-be-committed page as if that page existed.
2. A #PF results, and causes the SGX driver to commit a new page at the faulting address.
3. The enclave is resumed (by ENCLU[ERESUME]) and the faulting instruction is retried, as if no #PF had ever occurred.

The above process may seem complete, but newly committed pages in SGX2 must be accepted by the enclave before they may be accessed. The faulting instruction that is retried in step #3 will fault again. However, there is one exception. If the faulting instruction is an ENCLU[EACCEPT], then it will complete the process. In fact, ENCLU[EACCEPT] can be used to drive the page allocation process as described below.

1. The enclave determines the address at which a new EPC page needs to be committed, and executes ENCLU[EACCEPT] on that address.
2. A #PF results, and causes the SGX driver to commit a new EPC page at the faulting address.
3. The enclave is resumed to retry ENCLU[EACCEPT], which will succeed this time.

The #PF based approach makes the trusted code efficient and extremely simple (i.e. as simple as only one instruction – ENCLU[EACCEPT]).

This method also makes it feasible to grow a stack, except that the SGX driver must generate an exception to the application after committing a page. The enclave's (trusted) exception handler can be designed to avoid the use of the thread's stack, unlike an OCall approach.

One drawback in the aforementioned process is that only one page is allocated for each #PF. This can be a performance drag should the enclave try to allocate a range of pages. The concept of a "Dynamic Region", described next, is targeted to improve the efficiency by reducing the number of #PFs necessary for batch allocations.

5.3 Dynamic Region

Given the overhead of processing an exception and subsequently returning to the enclave via ENCLU[ERESUME], it would be desirable to have the enclave be able to request a range of continuous pages and then have the driver commit the entire range of pages at one time.

The SGX driver generally needs to know the following in addition to the fault address, for both correctness and efficiency.

- Whether or not the fault address is within an expandable enclave component (e.g. heap, stack, etc.). – A #PF may result from many reasons, and the driver needs to tell whether a #PF is a page allocation request or not.
- The usage of that expandable component, such as heap, stack, executable code, etc. – By exploiting the usage, the SGX driver may be able to "predict" which other pages will be requested in the near future, therefore avoid additional #PFs.

The **Dynamic Region** is a data structure which provides the SGX driver supplemental information regarding #PFs. The SGX driver maintains an array of dynamic region structures per enclave. Each dynamic region structure contains the following information.

- Range of the region – Defines the address range of the region. #PFs that fall within the range should be handled in accordance with the information defined in the dynamic region structure.
- Growing direction – This is a one-bit flag, either **up** or **down**.
  - A **growing-up** region is a range in which the enclave is expected not to request a page at an address unless pages between the lower bound and that address have all been allocated. A heap is usually a growing-up region.
  - In contrast, a **growing-down** region is a range in which the enclave is expected not to request a page at an address unless pages between that address and the upper bound have all been allocated. Stacks on x86/x64 architectures are growing-down regions, for example.
- Allocation Alignment – This is a mask to specify the alignment for chunk allocations. The driver will add pages from the #PF address in the grow direction until the page address aligns with the Allocation Alignment (e.g. the AND of the page address and the Allocation Alignment mask is zero.) A mask of -1 (i.e. all 1's) means no alignment, hence chunks are considered to start/end at lower/upper bound of a growing-up/down region. In contrast, a mask of 0 (i.e. all 0's) means no chunks, hence the region is "discrete" regardless growing direction and only a single page is committed per #PF.

The SGX driver uses the following algorithm to determine the appropriate action to take for each #PF:

1. Locate the dynamic region that contains the fault address. If no dynamic region is found, the current #PF is **not** a page allocation request and the process should complete by generating an exception to the application; otherwise
2. Commit the page containing the #PF.
3. For a growing-up region, keep committing pages towards lower addresses until:
   a. An existing page is reached; or
   b. The lower bound of the region is reached; or
   c. The logical AND of the current address and the region's alignment mask yields 0 (zero).
4. For a growing-down region, keep committing pages towards higher address until:
   a. An existing page is reached; or
   b. The upper bound of the region is reached; or
   c. The logical AND of the next address and the region's alignment mask yields 0 (zero).

5.4 Implicit EPC Allocation

So far we have discussed EPC allocations initiated explicitly by an enclave, such as when a heap manager expands a heap when it cannot satisfy a malloc() request. In contrast, implicit EPC allocations are initiated without an enclave's awareness. An implicit EPC allocation must be used when a thread exceeds its committed stack and the stack must expand beyond the lower bound of the committed area. The major challenge in implicit allocations is when and how to accept newly committed pages given the faulting instruction is **not** ENCLU[EACCEPT]. The solution lies in enclave exception handling.

Section 3 describes how HW exceptions such as #PF may be ultimately passed to the enclave for processing. Again from high level, the process of an implicit EPC allocation could be described as below.

1. The enclave tries to access a non-existing page "accidentally", and triggers a #PF.
2. The SGX driver intercepts the #PF, and commits a page (or pages depending on the dynamic region's Grow Direction and Allocation Alignment).
3. The SGX driver notices at this point that the allocation is implicit (more details follow), and injects an exception (This is OS dependent. A SIGBUS is used in Linux) to the faulting applications.
4. The application or more precisely, the uRTS, handles the exception (or signal on Linux* OS) by making an ECall to the enclave. This ECall is made using the same enclave Thread Context (identified by a TCS) that faulted.
5. The enclave's exception handler supplied by the tRTS (The tRTS gets the first chance at an exception within the enclave) handles the exception by accepting the newly committed pages, and then exits back to the application's exception/signal handler.
6. The enclave is resumed (by ENCLU[ERESUME] instruction), and tries the faulting instruction, which will succeed this time.

One may wonder how the SGX driver is able to tell implicit allocations from explicit ones in step #3 above. This is described in detailed in the following subsection.

Distinguishing Explicit and Implicit Allocations

The trick to distinguish the explicit from implicit allocations lies in the logical assumption that software, if coded correctly, never reads uninitialized memory buffers. In practice, a page being requested must not exist at the time of the request, hence it could not contain any initialized memory buffers, and thus its first access (as an implicit allocation request) by software must be a Write access; otherwise, the access would be due to a software bug. In contrast, an explicit allocation is triggered by ENCLU[EACCEPT], which results in a Read access per [1].

Below summarizes how implicit allocations are identified by the SGX driver upon receiving a #PFs within a dynamic region:

- If the access attempted was a Read Access (per error code associated with the #PF, see [1]), then it is considered an explicit allocation – No exception/signal will be injected.
- Otherwise, it is considered an implicit allocation – An exception/signal will be injected to the faulting application.

One thing worth mentioning, is that it's possible for a buggy enclave to read a non-existent page within a heap or stack. This access will then be misidentified as an explicit EPC allocation by the SGX driver. The driver may be induced to commit pages (via ENCLS[EAUG]) into the enclave; however, the pages will not be accessible because they will not have been accepted by the enclave. When the faulting instruction is retried, it will #PF again. Given that the page has already been committed this time, the fault will be considered as an access violation within the enclave.

Enclave Exception Handling

The enclave must be able to invoke ENCLU[EACCEPT] in its exception handler to accept newly committed pages before the pages can be successfully accessed..

The SGX architecture has limitations around nested exception handling. Specifically, SGX uses SSA (State Save Area) frames to save processor context on asynchronous exits (i.e. enclave exits due to interrupts or exceptions), and the number of SSA frames associated with a particular Thread Context is fixed. Each call into the enclave via EENTER must reserve on SSA frame to handle a subsequent interrupt or exception. Thus, nested exceptions can be handled only to the depth specified by the number of SSA frames. The latest version of Intel SGX SDK supporting SGX1 allocates 2 (two) SSA frames per Thread Context, meaning nested exceptions cannot be supported. That said, no implicit EPC allocations could be supported in the enclave exception handler because there would not be an SSA frame available for entering the enclave again to accept a page requested during the processing of an initial enclave exception. Therefore, caution must be taken not to trigger any implicit allocations in the context of the enclave exception handling. In practice, the tRTS exception handler can make sure the available stack space exceeds a certain (probably configurable) threshold by expanding the stack if needed before dispatching custom exception handling routines. Alternatively, the implementation may add one more SSA dedicated to handling implicit memory allocations within the exception handling context.

6 Implementation of SGX2 Dynamic Memory Management

This section describes three most desired enclave dynamic memory management features implemented by Intel SGX SDK.

6.1 Enclave Signing and Loading

Before going into details of the dynamic memory management features, let's go over how runtime components (e.g. heap, stacks, etc.) are created/reserved and loaded/initialized in an enclave.

The SGX architecture requires enclaves to be digitally signed, and their signatures will be verified at enclave load time (i.e. by ENCLS[EINIT] instruction). Figure 1 and Figure 4 show examples of enclave layouts, for SGX and SGX2 respectively.

Enclave Signing

As stated in Section 2.1, the Signing Tool provided with the Intel SGX SDK can be used to generate digital signatures for enclaves. To sign an enclave, the signing tool must construct an image of the enclave and measure the image. As shown in Figure 4, the image is composed of both the executable (or loadable) components from the enclave .dll or .so file and the runtime components (e.g. heap, stack, TCS, etc.). The format of these components and their characteristics such as their range (for heap and stack components) must be calculated by the signing tool. The Signing Tool accepts an XML file, the "enclave configuration" file, for developers to specify parameters (e.g. size, count) of certain components. Loosely speaking, the signing tool uses the configuration along with the enclave executable image to calculate the locations, attributes (and sometimes contents) of all of the enclave pages, then stores the resultant information(referred to as enclave "metadata") into a dedicated section (i.e. ".sgxmeta") embedded inside the executable file, measures the page contents/attributes of the resulted enclave in the same way as if it were done by the hardware, and finally signs (using a user-supplied private key) and stores the measurement along with the enclave metadata.

For SGX2, this schema of the enclave configuration has been extended to accommodate parameters relating to reserved address space for expandable/creatable components. Newly added parameters include, but are not limited to

- Min/Max heap sizes
- Min/Max stack sizes
- Min/Max number of threads

More information regarding enclave configurations is available in [5].

Enclave Loading

Enclave loading is driven by the uRTS, which loads the enclave executable file and extracts the location/size information for each of the enclave runtime components (and sometimes patches the executable image), and invokes (via IOCTL interface) the SGX driver for the actual loading of pages into EPC.

As mentioned in Section 5, the additions (compared to SGX1) to the loading process is the dynamic regions that need to be conveyed to the SGX driver. Given that the uRTS loader understands the locations and usages of the runtime components, the dynamic regions are trivial to establish. The SGX driver provides an IOCTL interface to accept dynamic regions from the uRTS. Please note that dynamic regions are tied to the enclave layout so usually don't change across the lifespan of the enclave.

6.2 Dynamic Heap Allocation

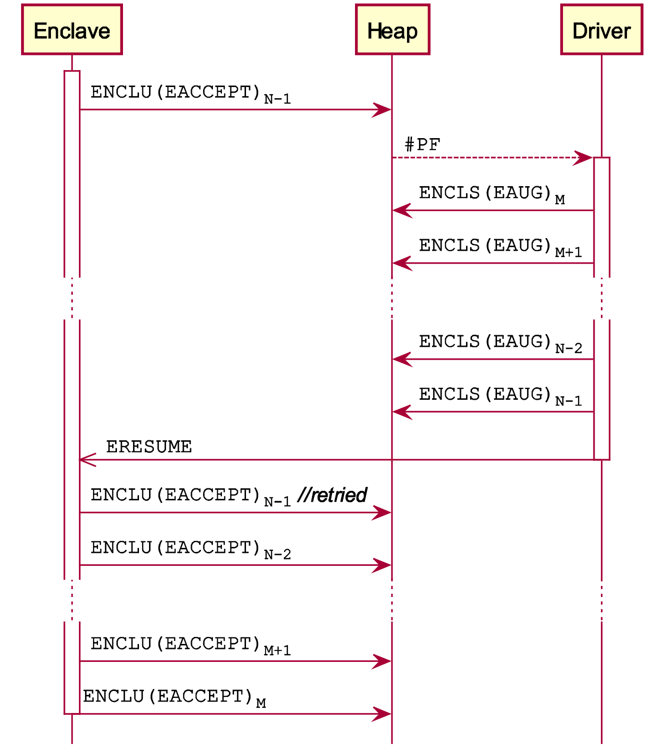

Figure 5: Range Allocation by a Single #PF

The tRTS implementation of malloc()/free() was based on *Doug Lea's Malloc* or *dlmalloc* for short (see [6]). dlmalloc inherently uses sbrk() to allocate and free pages. sbrk() is expected to take a size delta (positive number for allocation and negative number for deallocation) as input and return the new end address of the heap. The tRTS version of sbrk() simply tracks the boundary of the allocated portion of the heap, and pushes the boundary higher/lower on allocation/deallocation respectively. With SGX2, sbrk() is changed slightly to also maintain the boundary between the committed and uncommitted portions of the heap and to execute ENCLU[EACCEPT] for new pages whenever the allocated boundary moves beyond the committed boundary. The following summarizes the behavior of sbrk() in a tRTS implementation with SGX2 support, assuming the virtual range of the heap has been declared as growing-up dynamic region with its alignment mask set to -1.

- Two global variables – heap_allocated and heap_committed.
    - heap_allocated tracks the end address of the heap. It is initialized to the lower bound of the heap range.
    - heap_committed tracks the end of the committed region of the heap. It is also initialized to the lower bound of the heap range.
- sbrk() invoked with a positive delta is an allocation request. heap_allocated is adjusted up by the specified delta. If it passes beyond heap_committed:
    1. Set heap_committed to heap_allocated; and
    2. Accept all pages between the old and new values of heap_committed. Please note that the page at the highest address should be accepted first to reduce the number of #PFs (due to the fact that the region is a grow-up region and the SGX driver will commit pages up to the #PF address).
- sbrk() is invoked by a negative delta – This is a deallocation request. heap_allocated is adjusted down by the specified amount. The excess of the committed heap (i.e. pages between heap_allocated and heap_committed) could be freed at this point. In a future paper we hope to discuss efficient algorithms for trimming heaps.

Figure 5 depicts the sequence taken by sbrk(), as the enclave is trying to expand its heap from M to N (where N >M) pages. Please note that the subscripts denote the distance of the target page from the beginning of the heap, divided by page size. That is, the enclave starts with an M-page heap and is trying to expand it to N pages in size by requesting pages M through N-1. For optimal performance, the enclave does ENCLU[EACCEPT]$_{N-1}$ first. As described in Section 5.2, this results in a #PF which causes the SGX driver to commit pages up to the N-1 Page. It doesn't matter in what order the rest of the pages are committed or accepted.

6.3 Stack Expansion

Expanding a stack is trickier than expanding a heap in the sense that the enclave has to know when the stack needs expansion. There are two approaches in practice:

1. Handling #PFs resulting from a stack overflow.
2. Probing the stack using ENCLU[EACCEPT]

The first approach can be considered a default approach and must be implemented to ensure that the run-time is prepared to grow the stack at arbitrary points in the enclave when a variable is "pushed" onto the stack. The second method takes advantage of a compiler feature which probes the stack whenever function is called with local variables in excess of a specific size (usually one page). In this case, the compiler will insert a function that probes the stack to ensure that memory is committed for the stack. For example, the Microsoft* Visual C Compiler calls the __chkstk() helper routine to probe the stack. The run-time can make the stack expansion process more efficient by inserting an ENCLU[EACCEPT] in the trusted stack probe.

The following summarizes how stack is expanded in a tRTS implementation with SGX2 support, assuming the virtual range of the stack has been declared as a growing-down dynamic region with its alignment mask set to -1.

- One thread local variable – stack_committed is maintained per thread. Please note that it is accessed by its own thread to avoid race conditions. It is initialized to the minimal stack size specified in the enclave configuration XML file.
- Probing the stack – The stack probing routine (e.g. __chkstk()) compares ESP/RSP with stack_committed and accepts (using ENCLU[EACCEPT]) all pages between them if ESP/RSP passes below stack_committed, and then updates stack_committed to ESP/RSP.
- Enclave exception handler – As discussed in Section 5.4, the exception handler subtracts a certain amount from ESP/RSP as the reserved stack for exception handling, and compares ESP/RSP with stack_committed, and accepts all pages between them if ESP/RSP is below stack_committed, and then updates stack_committed to ESP/RSP.

6.4 Thread Creation

Creating a new thread context involves creating all of its components, initiating its TCS page and converting it to the page type of PT_TCS.

What comprise of a thread context are summarized below, in the order of their addresses from high to low.

- SSA – State Save Area for asynchronous exits. There are typically two SSA frames, one page each per current SGX architecture.
- TLS – Thread Local Storage, which includes both compiler specific per-thread control information and ISV defined TLS variable.
- TCS – Thread Control Structure containing per-thread control information needed by the processor to start an enclave thread. More information available in [1]
- Stack.

Figure 6: Thread Context Components and Initialization depicts the composition of a thread context. Given all of the components are consecutive in virtual memory and they are all needed before the TCS could be entered, they could be considered as a whole a single stack, whose bottom (highest address) would then be the SSA$_1$ page.

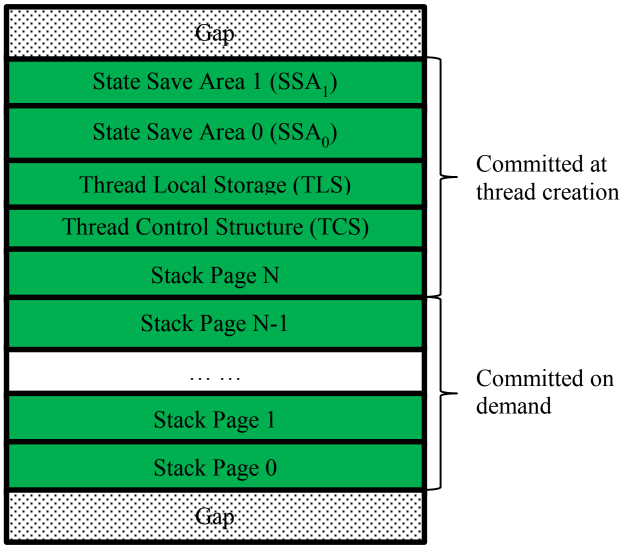
Gap
State Save Area 1 (SSA$_1$)
State Save Area 0 (SSA$_0$)
Thread Local Storage (TLS)
Thread Control Structure (TCS)
Stack Page N
Stack Page N-1
… …
Stack Page 1
Stack Page 0
Gap
Committed at thread creation
Committed on demand

Figure 6: Thread Context Components and Initialization

Therefore, the creation process could be described below:

1. The whole thread context should be declared to the SGX driver as a growing down region at enclave load time.
2. The enclave accepts "Stack Page N" (see Figure 6). This causes the driver to commit all pages between "Stack Page N" and the upper bound of this thread context.
3. The enclave accepts the rest of the pages (i.e. the TCS, TLS and SSA pages) in any order deemed convenient by the implementation.
4. The enclave initializes the content of the TCS page.
5. The enclave makes an OCall requesting the SGX driver to convert the TCS page to the type of PT_TCS.
    a. The uRTS passes the address of the TCS page and the requested page type to the SGX driver via IOCTL.
    b. The SGX driver converts the page type by ENCLS[EMODT] and issues ENCLS[ETRACK], followed by broadcasting an IPI to flush out stale TLB entries for all logical processors.
6. The enclave accepts the page type change using ENCLU[EACCEPT]. The new thread context is ready for use hereon.

One more thing worth pointing out is that, unlike expanding heaps or stacks, it's typically the uRTS that determines when additional thread contexts are needed (hence created). In practice, the uRTS maintains a pool of free threads to be assigned to ECall requests, and initiates new thread creation only when the pool has "dried up".

7 Future Considerations

We have seen how three new SGX2 instructions listed in Table 1 can be used for heap expansion, stack expansion, and adding thread contexts to a running enclave. This is just a small subset of the programming features that can be enabled with SGX2. Additional features that may be explored include, but are not limited to:

- Heap Contraction with the implementation of page trimming made possible by the addition of the ENCLS[EMODT] and ENCLU[EACCEPT] instructions.
- Modification of Page Access Permissions using the ENCLS[EMODPR], ENCLU[EMODPE] and ENCLU[EACCEPT] instructions
- The dynamic loading of code either by dynamically loading and linking to libraries or by employing Just-in-Time compilers within the enclave. Code loading is made possible with the addition of the ENCLS[EAUG] and ENCLU[EACCEPTCOPY] instructions.

In the future, we hope to detail run-time support for these additional features.

8 Summary

SGX2 Instructions increase the flexibility of the SGX programming environment by allowing the programmer to dynamically manage memory within the enclave space. We have shown examples of a few run-time enabled features such as heap expansion, stack expansion, and the creation of thread contexts. These features allow enclave developers to design their enclaves to more efficiently adapt to varying programming workloads.

9 ACKNOWLEDGEMENTS

The authors of this paper wish to acknowledge the contributions of many hardware and software architects and designers who have worked in developing this innovative technology.

Intel is a trademark of Intel Corporation in the U.S. and/or other countries.

10 REFERENCES

[1] Intel Corp., "Intel® 64 and IA-32 Architectures Software Developer's Manual," April 2016. [Online]. Available: http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developers-manual.pdf. [Accessed 7 May 2016].

[2] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd and C. Rozas, "SGX Instructions to Support Dynamic Memory Allocation Inside an Enclave," in *HASP*, Seoul, South Korea, 2016.

[3] M. Hoekstra, R. Lal, P. Pappachan, C. Rozas, V. Phegade and J. Del Cuvillo, "Using Innovative Instructions to Create Trustworthy Software Solutions," in *ISCA-HASP*, Tel-Aviv, 2013.

[4] F. McKeen, I. Alexandrovich, A. Berenzon, C. Rozas, H. Shafi, V. Shanbhogue and U. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *ISCA-HASP*, Tel-Aviv, 2013.

[5] Intel Corporation, "Intel(R) Software Guard Extensions Evaluation SDK for Windows* OS User's Guide," 2016.

[Online]. Available: https://software.intel.com/sites/default/files/managed/d5/e7/Intel-SGX-SDK-Users-Guide-for-Windows-OS.pdf.

[6] D. Lea, "A Memory Allocator," 4 April 2000. [Online]. Available: http://gee.cs.oswego.edu/dl/html/malloc.html.

Intel disclaims all express and implied warranties, including without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement, as well as any warranty arising from course of performance, course of dealing, or usage in trade.

This document contains information on products, services and/or processes in development. All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest forecast, schedule, specifications and roadmaps.

The products and services described may contain defects or errors known as errata which may cause deviations from published specifications. Current characterized errata are available on request.

Intel technologies features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at Intel.com, or from the OEM or retailer.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting www.intel.com/design/literature.htm.

Optimization Notice

Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804