# Port or Shim? Stress Testing Application Performance on Intel SGX

Aisha Hasan
Carnegie Mellon University in Qatar
Doha, Qatar
az-hasan@hotmail.com

Ryan Riley
Carnegie Mellon University in Qatar
Doha, Qatar
rileyrd@cmu.edu

Dmitry Ponomarev
Binghamton University
Binghamton, USA
dponomar@binghamton.edu

***Abstract*—Intel's newer processors come equipped with Software Guard Extensions (SGX) technology, allowing developers to write sections of code that run in a protected area of memory known as an enclave. In this work, we compare performance of two scenarios for running existing code on SGX. In one, a developer manually ports the code to SGX. In the other, a shim-layer and library OS are used to run the code unmodified on SGX. Our initial results demonstrate that when running an existing benchmarking tool under SGX, in addition to being much faster for development, code running in the library OS also tends to run at the same speed or faster than code that is manually ported. After obtaining this result, we then go on to design a series of microbenchmarks to characterize exactly what types of workloads would benefit from manual porting. We find that if the application to be ported has a small sensitive working set (less than the 6MB available cache size of the CPU), infrequently needs to enter the enclave (less than 110,000 times per second), and spends most of its time working on data outside of the enclave, then it may indeed perform better if it is manually ported as opposed to run in a shim.**

***Index Terms*—Security, SGX, Benchmarking, Workloads**

## I. Introduction

Intel's Software Guard Extensions (SGX) [1], [2] is a hardware-based design that offers developers a unique opportunity for securing their data even in the presence of compromised system software layers, such as operating systems or hypervisors. The key abstraction provided by SGX is a secure *enclave*, which is embedded within the virtual address space of an application. When stored inside an enclave, data can be protected from ex-filtration by even the operating system or a hypervisor itself, as the SGX hardware performs permission checks on every memory access and only allows code executed inside an enclave to access this data. Only the microprocessor and on-chip caches are trusted in the SGX model; all data is encrypted and integrity-protected when it leaves the last-level cache and is written to memory. On a memory access, the data is decrypted and integrity-checked before being deposited into the caches. SGX has been implemented in most recent Intel high-end processors, starting with Skylake, and trusted computing infrastructures are actively being developed around SGX, including domains such as anti-virus scanners, copyright protection, cryptography, and much more [3]. SGX represents the most significant investment by a major microprocessor design company into security in several decades.

To support a widespread deployment of SGX, it is important to consider the challenges of software development for SGX-based systems, taking into account the aspects of performance and ease of software design. In general, there are two approaches that developers can use if they wish to port their existing applications to work with SGX. The standard model of SGX development involves porting or re-building a program in C or C++ using the Intel SGX SDK [4], which typically involves a significant amount of effort. The program needs to be partitioned into secure and insecure portions, with only security-sensitive code executed inside an enclave. This *porting* approach involves non-trivial programming and sometimes performance overhead. To address these limitations, another approach is to run applications with little or no modifications using a library OS shim layer. Examples of this approach include Haven [5], SCONE [6] and Graphene-SGX [7]. These designs differ in the amount of code that is brought inside an enclave, and they significantly simplify programming interfaces for enclaves. We refer to this approach as *shimming* in the rest of the paper.

Porting and shimming have a variety of trade-offs related to the required developer expertise, the security of the resulting enclave, the size of the software trusted computing base, and the performance of the resulting code. The goal of this paper is to provide both anecdotal experiences and hard, measured data to quantitatively answer, with regards to performance: Porting or Shimming? We started out by asking an existing developer with some systems experience, but no SGX experience, to manually port a portion of version 3 of the lmbench [8] benchmarking suite to SGX. We then took the same benchmarks and run them in unmodified form under Graphene-SGX and compared performance. The results of this comparison showed that despite the tremendous difference in time investment, on the order of months, the two approaches produced virtually the *same performance*.

Given the significant training time required for a software engineer to become proficient in building SGX enclaves, this result caused us to ask a number of questions. From a performance perspective, is there ever a time when manually porting an application is worth the effort? What types of application workloads would benefit from porting as opposed to shimming? In order to answer these questions, we designed and implemented a series of new benchmarks to stress-test

various SGX capabilities. The goal of this study is to flesh out what type of workloads would benefit from porting vs. shimming, and to discuss how these results can inform software developers.

We designed our benchmarks to explicitly exercise performance overheads inherent in SGX. The first overhead is associated with longer memory access times on last-level cache misses for enclave accesses. Due to the overhead of encryption and integrity checks, every cache miss inside an enclave takes about 100 cycles more compared to a cache miss during normal execution, as shown in [9]. Therefore, if the working set of data consistently manipulated by a program exceeds the capacity of the last-level cache, but only a fraction of this data is sensitive and needs protection, it can be advantageous, in principle, to use the porting approach so that expensive cache misses are minimized. Indeed, if the entire application is embedded inside of an enclave due to shimming, then every cache miss will encounter longer latencies. On the other hand, if the application fits inside the last-level cache and does not experience a large number of misses, then putting the entire application inside an enclave will not incur additional latency.

Another overhead of enclave-based execution is the latency to enter and exit the enclave for operations such as system calls [10]. Since many execution cycles are lost in the course of these operations, fine-grain entering and exiting of the enclaves carries performance overhead.

In general, the overheads associated with enclave calls and memory accesses demonstrate complementary trends. Increasing the amount of code and data accessed in enclave mode exacerbates the lost cycles due to memory accesses (for memory-bound workloads), but reduces the overhead of enclave transitions. On the other hand, minimizing the amount of code and data inside an enclave improves memory performance, but may require additional enclave transitions in order to access the code and data inside the enclave.

The benchmarks that we construct in this study stress-test all of this behavior. Specifically, we study loop-like programs that access progressively larger amount of data, starting with a few MB (that comfortably fits in the cache) to the workloads that access tens of MBs of data, thus significantly exceeding the cache size and exhibiting poor cache performance. Synergistically, we also consider various models and frequencies for entering and exiting enclaves.

The main contributions and the key results of this paper are:

- We develop a series of stress-test benchmarks to exercise various SGX-related overhead and use these benchmarks to understand if manually porting applications to SGX can be better than using a shim layer in terms of performance.
- We demonstrate that if the entire application needs to be protected, then shimming provides nearly the same performance with a fraction of the development effort.
- We show that if only a fraction of the application needs to be protected, then porting can result in performance improvements and we systematically describe conditions under which this improvement can occur. This result should be useful for future SGX application developers and help them decide whether to port their applications or shim them.

## II. Background and Related Work

This section provides background information on Intel SGX, the Graphene-SGX framework, and benchmarking with LMBench that is required to understand the paper.

### A. Intel's Software Guard Extensions (SGX)

This section provides an overview of Intel's SGX focusing on the aspects of its operation necessary to understand this work. Further details on SGX can be found in [11]. Readers familiar with SGX can skip to the last paragraph of this section for a few lesser-known details that are relevant to this paper.

Intel's SGX is a set of CPU instructions and hardware modifications designed to protect running code and data from compromised or malicious system software (such as the operating system, virtualization layer, etc.) The key abstraction provided by SGX is the secure *enclave*, a piece of code and data embedded in a process and protected by the hardware. The general model is that a process running on the operating system needs to be partitioned into trusted and untrusted components. The trusted components become one or more enclaves and the untrusted components are simply part of standard memory in the process itself. This means that an enclave exists in the context of a process.

At boot time, an SGX-enabled processor reserves a fixed amount of physical memory for the Enclave Page Cache (EPC). The EPC is a contiguous portion of physical memory, typically 128 MB in size. All protected enclave data is stored in the EPC. The contents are protected by the Memory Encryption Engine (MEE), a hardware component of the processor's memory controller that ensures all writes to EPC memory are encrypted and all reads from EPC memory are decrypted and integrity checked. The MEE only operates on reads and writes going to the main memory located outside the CPU. The processor's caches (located on the CPU die) cache EPC contents unencrypted. The MEE ensures that even an attacker with physical access to the system's memory and data bus is unable to learn the contents of enclave memory.

A running process may have more than one enclave, and enclaves themselves are created and managed by the untrusted system software, such as the OS. In order to establish an enclave, the OS first creates an enclave using a new instruction and then loads the enclave's code and data into unallocated pages from the EPC. Next, those pages are assigned to the enclave using another instruction. After all of the enclave's pages have been loaded and assigned, the OS requests that the hardware disable the loading process and mark the enclave as initialized. After loading is disabled, the hardware tracks which EPC pages are assigned to the enclave and ensures that the contents of those pages can't be modified except by code running from within that enclave.

As may be seen, the OS has the ability to modify the enclave's contents prior to loading. In order to mitigate this threat, SGX also supports the ability for enclaves to *attest* themselves to external services, such as a cloud-based service. After loading is complete, the hardware hashes the enclave memory and associated meta-data in order to produce an *enclave measurement*. This measurement can then be signed by the hardware's private key and made available for attestation of the integrity of the loaded enclave. As part of the attestation process, an outside attestor can establish a secret key with the enclave, thus allowing the enclave to establish a secure channel (over the network using something like TLS) with the external service. In this way, secrets can be provided to the enclave only after the integrity of the enclave has been verified.

While a process is running outside the enclave, the hardware prevents access to enclave memory. If the process needs to execute code that is located within its enclave, then the OS uses an enclave enter instruction to transition control-flow to fixed entry points within the enclave. This operation is called an *ECALL*. While running within the enclave, access is permitted to both the enclave memory and regular memory of the process. (However, code can only be executed from enclave memory). If the OS needs to stop enclave execution for any reason (such as a context switch) then an interrupt can be used to pause enclave execution and pass control back to the non-enclave code. During this transition, the enclave's registers are saved to enclave memory and then wiped. The enclave can then be resumed at a later time.

At times, the running enclave may need to call functionality from outside the enclave. The most common reason for this is system calls. In this situation, the enclave initiates an *OCALL*: A call from inside to outside the enclave. OCALLs are, essentially, functions located in the non-enclave portion of the process' address space. In order to make the call the enclave copies any relevant arguments to non-enclave memory (in order to ensure the OCALL code can access them) and initiates the OCALL. Control is then passed outside of the enclave, the call is processed by the untrusted code, and control is returned back to the enclave. After the OCALL returns, the enclave collects the result from the non-enclave memory.

Memory paging is supported in the sense that pages from the EPC can be evicted to main memory. The OS manages this process, but the hardware ensures that evicted page contents are cryptographically protected prior to eviction. This includes ensuring the confidentiality and integrity of the contents as well as a few tricks to prevent replay attacks.

In addition to this general overview, there are a few details that are particularly relevant to this work. First, while the amount of memory reserved for enclaves in the enclave page cache (EPC) is limited to 128MB, in practice only about 90MB of that is usable. The other memory is used by various parts of the SGX SDK to manage the execution of a running enclave. Second, while the MEE protects the contents of EPC pages that are in main memory, those contents are decrypted prior to being brought into the cache. This means that the MEE introduces almost no overhead if the data being accessed is cached, but there is overhead while reading from memory and filling the cache [9].

### B. Graphene and Shim Layers

Graphene [12] is an open-source library OS that refactors a traditional OS kernel into an application library. Graphene-SGX [7] is a follow-up work that places Graphene inside of an enclave as shim layer to aid in the execution of applications. The TCB of Graphene-SGX is similar to that of standard SGX, except that in addition to trusting the CPU and the application being protected, we must also trust the Graphene library OS. In exchange for this slightly expanded TCB, the library OS can manage any system calls from the hosted application, meaning that the application does not need to be SGX-aware. It is important to note that the entire application is brought into the enclave, and Graphene makes no attempt to partition the application into enclave and non-enclave segments. The system is considered to provide a low overhead and it eases the challenges of developing applications for SGX.

Although our experiments in this work use Graphene-SGX as the shim layer, there are other systems as well. Haven[5] implements shielded execution of unmodified server applications on an untrusted cloud. SCONE[6], instead of protecting individual applications, protects entire containers. Panoply [13] attempts to combine the ease of a tool like Graphene with smaller TCBs by bringing standard POSIX abstractions to SGX by means of units of code and data - microns - that are isolated in SGX containers. SGXKernel [14] is a *switchless* library OS that eliminates the need for enclave transitions by using in-enclave multi-threading and asynchronous cross-enclave communication[14]. While Graphene-SGX performs better when requests can be handled internally, SGXKernel performs better on calls that require communicating with the OS.

### C. Benchmarking SGX

Some previous works targeted at building tools for SGX also did some SGX benchmarking. Haven [5], a system for running unmodified binaries in an enclave, evaluated performance using a simulated version of SGX and modeled memory overheads manually. Graphene-SGX [7] performed extensive testing of their performance, which indirectly evaluates SGX. SCONE [6] used a number of micro-benchmarks on SGX and found that memory accesses are slowed down by the overhead of accessing EPC pages, memory access penalties are negligible when data blocks fit into the L3 cache size (about 8MB), and applications that make a significant number of system calls incur high costs due to the overhead of enclave entry-exit transitions. In Opaque [15], a database system using SGX, the authors found that system performance was limited by the size of data blocks that could be completely contained within the EPC. sgx-perf [16] is a profiling tool designed to help developers find the primary sources of slowdown in their enclaves. They found that the primary sources of enclave slowdown are enclave transitions and paging.

Several studies are focused on performance analysis of SGX. The work of [17] compared Intel SGX to AMD Memory Encryption Technology, using their own benchmarks measuring floating point workloads inside an enclave, and marshaling data in and out of the enclave. Their results show that AMD SEV is faster than Intel SGX when buffer sizes are larger. Our work is specific to SGX, application to AMD processors is left for future work. Ngoc et al. [18] investigated the performance of SGX on virtualized systems. They noticed that performance varied noticeably depending on whether shadow paging or nested paging is used: For memory intensive workloads shadow paging is faster, but when a high number of context switches occur nested paging is faster. They propose dynamically detecting the workload and changing the paging methodology. They also noted that when SGX is virtualized, the VM gets access to a smaller EPC, which further impacts performance.

### D. LMBench

To measure performance of operations on native SGX and compare it against Graphene, we chose to use the LMBench benchmark suite (version 3). LMBench is a group of micro-benchmarks that measure latency and bandwidth of various aspects of a system, including read, write, and copy operations to a buffer, as well as the latency of system calls and communication protocols such as UDP and HTTP. We chose to use LMBench over other benchmarking suites, as it is open source and is widely used.

### E. Hardware and Software Configuration

All benchmarks throughout this study were evaluated on a Dell Optiplex 7440 machine containing an Intel®Core™i7-6700 CPU running at 3.4GHz (8MB L3 cache) with 16 gigabytes of RAM. The BIOS was configured to enable SGX with a 128MB EPC. The operating system is Ubuntu 16.04.06 LTS running Linux kernel version 4.4.0-142. We used the Intel®SGX Linux SDK version 2.7.1.

SGX allows for different configurations of enclave sizes, including sizes that go above the EPC limit. When an enclave requires more memory than the EPC can provide, a paging feature is enabled that allows for EPC pages to be encrypted and paged-out. This, obviously, has significant performance implications. This work is not primarily concerned with evaluating performance of the paging implementation in the Intel SGX SDK, so we ensured that data stored within the enclave is kept below 90MB in order to prevent paging. For benchmarks that required data buffers, we use buffer sizes of 20MB, 40MB, 60MB, and 80MB, which keep the workloads within the boundaries of the usable EPC of 90MB. All results represent an average of at least 10 runs for each benchmark.

## III. Initial Analysis of Shimming vs Porting

Our initial goal was to compare the performance of a subset of the LMBench micro-benchmarks in two scenarios: 1) running them on Graphene-SGX, and 2) manually porting them to SGX. As an initial investigation, we ported lmbench3 benchmarks related to memory performance and system calls: `bw_mem` (read, write, read-write, bzero, copy, bcopy), `lat_rand`, and `lat_syscall`.

The manual port ended up being a multi-month effort requiring the developer to do the following: 1) Follow the SGX tutorial series [1] provided by Intel; 2) Significantly modify LMbench to remove its reliance on fork/exec for execution. Initially overwhelmed by the prospect of rewriting those portions using threading model provided by the SGX SDK, the multi-process/multi-threading ability was simply removed entirely; 3) Troubleshoot and debug a number of issues related to memory usage, including tracking down paging activity.

In contrast, the Graphene-SGX port required around a week of effort.

### A. Benchmark Versions

We generated and evaluated four versions of the LMbench benchmark:

1) *Forkless*: LMBench, by default, supports running multiple instances of benchmarks in parallel. It accomplishes this by having a parent process spawn, and manage, worker children using `fork`. Given that our goal was, primarily, to test SGX performance on a single core, we did not need this functionality. In addition, SGX enclaves do not support `fork`. As such, we modified LMbench to remove its multi-processing features.
2) *SGX*: We then took our forkless implementation and ported it to SGX. This involved actually separating the code into enclave and non-enclave portions. We took the general approach of favoring inclusion inside the enclave. The exception to this is stepping out to execute code that is forbidden within the enclave, such as making system calls. To support this, we provided some basic "glue" code to pass arguments back and forth properly as part of the OCALL interface.
3) *NoSGX*: The glue code in the SGX version of the benchmarks causes some slowdown independent of actually running on SGX. (The code is required to run on SGX, but the slowdown it causes is independent of the SGX hardware.) In order to better distinguish the performance penalty of the SGX hardware vs. the software modifications required, we also produced a version of the benchmark that maintains all of the same glue code, but does not actually run on SGX. It maintains the extra function calls and argument copying, but without the overhead of OCALLs or the memory encryption engine. It has the same code pathways as SGX, without involving enclave interfaces.
4) *Graphene*: Lastly, in order to compare our manual port performance with that of a library OS/shim, we ran our forkless implementation on Graphene.

### B. Performance Results

In this section we will go over the performance results of the our four versions of LMbench.

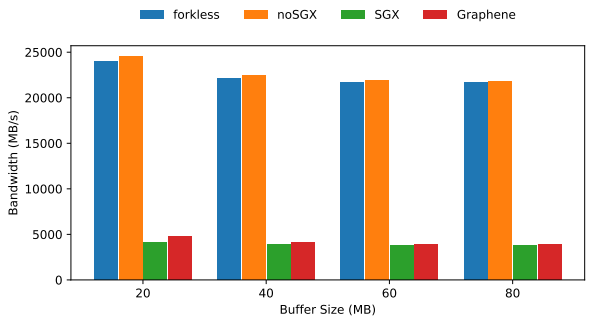



Fig. 1: bw_mem: read

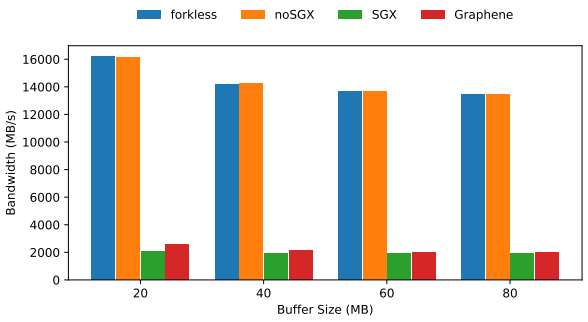



Fig. 3: bw_mem: read-write

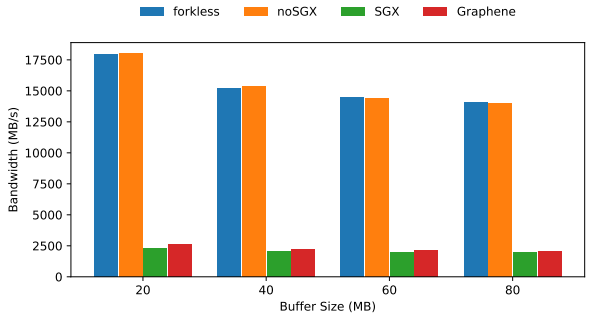



Fig. 2: bw_mem: write

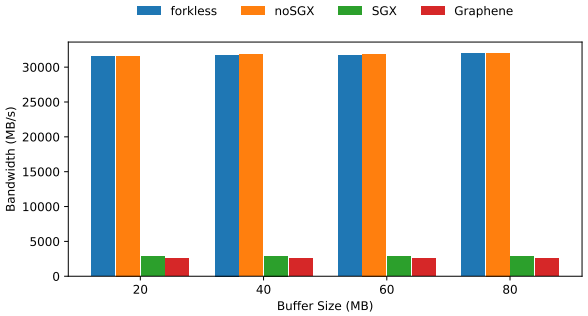



Fig. 4: bw_mem: bzero

*a) Memory Bandwidth:* The memory bandwidth benchmarks are represented in Figures 1-6. The results shown are for buffer sizes of 20MB, 40MB, 60MB and 80MB, and performance is measured in terms of bandwidth (higher is better). Fig. 1 measures the performance of reads from a buffer. A series of reads is performed, with every fourth `int` being read. Fig. 2 measures the performance of writes, with every fourth `int` written to with a value of `1`. Fig. 3 combines `read` and `write`, resulting in every fourth `int` read and written to. Fig. 4 shows the results for `bzero`, which measures how quickly the values of a buffer can be set to zero using `bzero`. Figs. 5 and 6 show the performance of two different methods for copying data between buffers. Since two buffers are now allocated, the buffer sizes are reduced in order to ensure the enclave does not grow beyond 90MB and triggers paging.

*b) CPU Usage:* Fig. 7 is a CPU-intensive workload showing time taken to generate random numbers using two different techniques (lower is better). Again, *forkless* and *NoSGX* have similar run times, with *SGX* and *Graphene* running much slower. It is interesting to note that for *SGX*, `drand48()` is much slower than `lrand48()`, with about a 30% drop, while for *Graphene* the numbers remain roughly the same.

*c) System Calls:* Fig. 8 shows the results for benchmarks that make a lot of system calls (lower is better). The performance is roughly the same across all variants, showing that a workload heavy in enclave transitions performs about the same in both the manual port and the shim. There are two points of difference worthy of discussion. First, the Graphene-SGX version of the `getppid` benchmark was faster. This is because that particular system call is handled by the Graphene library OS, and so no ECALL to the real OS is required. All of the other system calls could not be handled solely by the library OS and so required an ECALL to the actual OS in order to be fulfilled. Second, `stat` is slower on the Graphene-SGX version. This is due to Graphene duplicating some of the file path management code, making that overall code path for `stat` much longer than usual. A similar issue was noted by Graphene [12] in §6.4.

### C. Summary and Discussion

For almost all of the benchmarks, the same pattern emerges: The manual SGX port and the Graphene-SGX port have virtually the same performance. While this may seem intuitive in hindsight, at the time it was not. Our hypothesis when we started was that manual porting would result in better performance. Instead, we found that there was almost no performance benefit for manual porting. Given how much easier it is to simply run an existing implementation using shimming, this opens up an interesting question: Is it ever worthwhile, from a performance perspective, to manually port? Or does shimming perform just as well for a fraction of the effort? We explore this question in the next section.

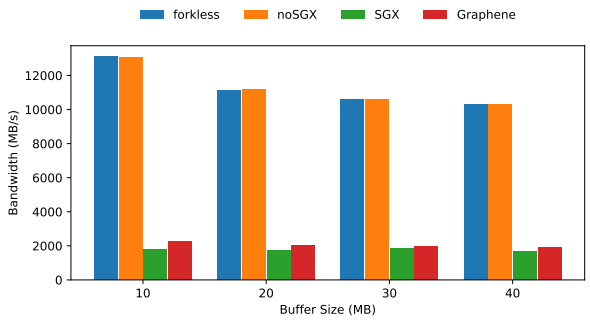



Fig. 5: bw_mem: copy

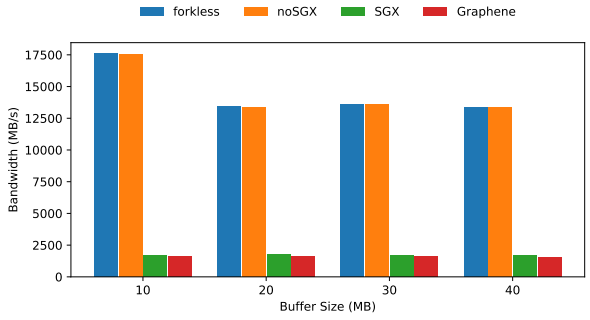



Fig. 6: bw_mem: bcopy

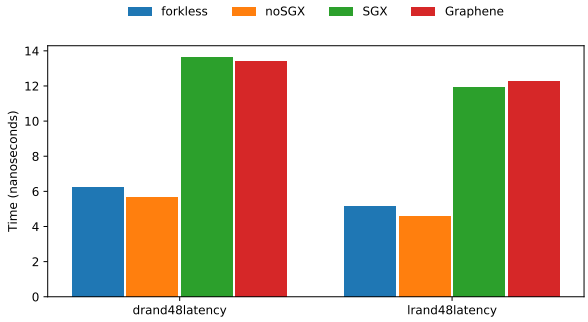



Fig. 7: lat_rand

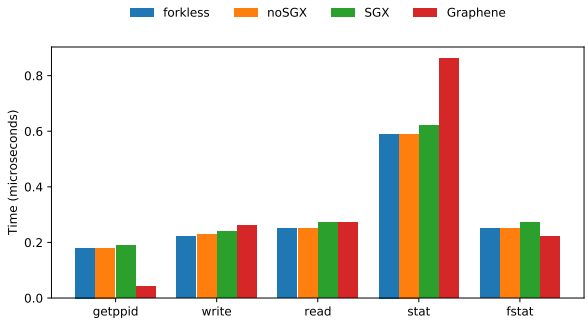



Fig. 8: lat_syscall

## IV. Determining When Porting is Better Than Shimming

Based on the results of the previous section, we now turn to benchmarking the specific performance attributes that contribute to enclave slowdown and demonstrate the types of workloads that can benefit from porting instead of shimming. Previous work [9], [10], [16] has identified two leading causes of performance degradation caused by the SGX hardware: (1) The memory read and write performance caused by the memory encryption engine (MEE). (2) The transition time needed to enter and exit the enclave (ECALLs and OCALLs). We will now investigate these two overheads further. We note that while paging has also been identified as a major source of slowdown, multiple other works [6], [16] have looked at it and came to the same conclusion: It should be avoided at all costs. We did not feel the need to re-investigate that point here. It may be worth investigating in the future if the overhead can be brought down to reasonable levels [19].

### A. Memory Performance

In order to profile enclave memory performance we designed a set of benchmarks that perform accesses to all of the bytes of a fixed size buffer in a loop. We will then vary the total size of the buffer. The goal of these benchmarks is to determine how the size of the enclave working set impacts performance. We hypothesize, based on previous work [6], [9], that when the working set is smaller than the cache size, the performance impact of the MEE will be minimal.

We start by assuming that all accesses occur inside the enclave. Our read benchmark is straight-forward and can be found in Algorithm 1. As can be seen, we allocate a buffer, fill it with 1s, and then iterate over it a chosen number of times. Note that we intentionally do not attempt to flush or otherwise manage the cache. We also create a version of the benchmark

**Algorithm 1** Read Benchmark

```
1: // This runs inside the enclave
2: procedure IN_ALLOCATEBUFFER(num_ints)
3:     in_buf ← malloc(num_ints * sizeof(int))
4:     for j ← 0, num_ints do
5:         in_buf[j] ← 1
6:     end for
7: end procedure

8: // This runs inside the enclave
9: procedure IN_SUMBUFFERREAD(loops, num_ints)
10:    sum ← 0
11:    for i ← 0, loops do
12:        for j ← 0, num_ints do
13:            sum ← sum + in_buf[j]
14:        end for
15:    end for
16: end procedure

17: // Execution starts here, and this runs inside the enclave
18: procedure IN_READBENCHMARK(loops, num_ints)
19:    in_AllocateBuffer(num_ints)
20:    sum ← 0
21:    StartTimer()
22:    in_SumBuffer(loops, num_ints)
23:    StopTimer()
24: end procedure
```

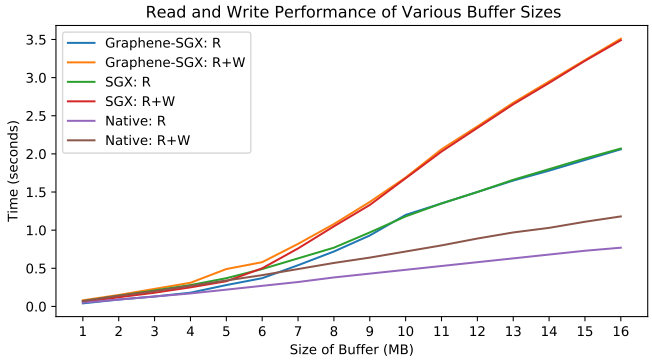



Fig. 9: Comparing sequential reads and writes for SGX, Graphene-SGX, and native

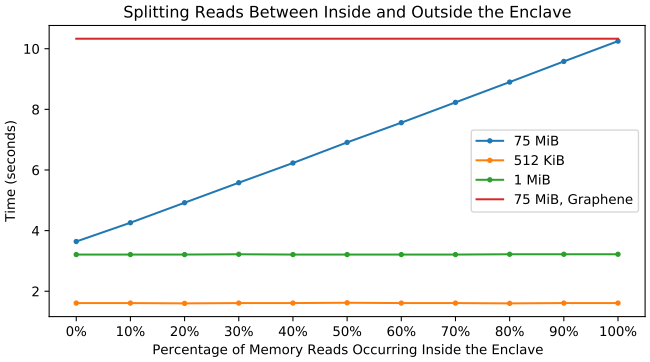



Fig. 10: Comparing performance of reads with different proportions of data split across application and enclave.

that performs both reads and writes. It differs from the read benchmark only in that the `in_SumBufferRead` procedure has been modified to include the addition of a memory write after line 13.

We created three versions of this benchmark: (1) an SGX version that runs the code entirely within an enclave; (2) a native version that does not involve an enclave at all; (3) the native version using Graphene. The size of the buffer is varied and the total time to complete 500 loops is measured. The performance results can be found in Fig. 9.

There are a number of observations to be made. First, for both the read benchmark and the read+write benchmark, our SGX version and Graphene both perform almost identically. This is yet another example of not seeing an advantage to manual porting. Second, similar to prior work [6], [9] it can be seen that the penalty for writes is much larger than for reads. Third, there is a noticeable change in the graph that occurs between 6 and 8MB. Prior to this point, the lines are all tightly grouped together. After this, the slope of the benchmarks using an enclave (SGX and Graphene) increases and the plots diverge quickly.

The deviation at 6MB deserves further attention. SCONE [6] noticed the same phenomenon and drew the same conclusions we have: this is where the capacity of the last-level cache is exceeded. The particular CPU we used for evaluation has an 8MB L3 cache. Shimizu et. al. [9] investigated the cache miss performance more thoroughly and found that enclave cache miss penalties are on the order of 1.45X when compared to non-enclave miss penalties. Given that our benchmark repeatedly accesses the same buffer, if that buffer can fit into the cache, then the memory encryption engine is not used and there is almost no performance penalty from running inside an enclave. Once the buffer no longer fits entirely in the cache, however, the reads and writes involve main memory accesses, which activates the memory encryption engine. Our results differ from SCONE [6] in one important way: They found that sequential reads and writes had no additional performance penalty over native execution as long as the working set fits into the EPC. They attributed this to pre-fetching. Our benchmark does not show this same effect, instead we do show a performance difference for sizes larger than the cache but still smaller than the EPC. We have no definitive explanation for this difference, but we hypothesize it could be related to the multitude of microcode updates related to Spectre [20] and Foreshadow (L1TF) [3].

**Algorithm 2** Read Benchmark with Split Memory Access

```
1: // This runs outside the enclave
2: procedure OUT_ALLOCATEBUFFER(num_ints)
3:     o_buf ← malloc(num_ints * sizeof(int))
4:     for j ← 0, num_ints do
5:         o_buf[j] ← 1
6:     end for
7: end procedure

8: // This runs outside the enclave
9: procedure OUT_SUMBUFFERREAD(loops, num_ints)
10:    sum ← 0
11:    for i ← 0, loops do
12:        for j ← 0, num_ints do
13:            sum ← sum + o_buf[j]
14:        end for
15:    end for
16: end procedure

17: // Execution starts here, and this runs outside the enclave
18: procedure SPLITREADBENCH(loops_out, loops_in, num_ints)
19:    in_AllocateBuffer(num_ints)
20:    out_AllocateBuffer(num_ints)
21:    StartTimer()
22:    out_SumBufferRead(loops_out, num_ints)
23:    in_SumBufferRead(loops_in, num_ints)
24:    StopTimer()
25: end procedure
```

We now move on to evaluating the performance when some memory accesses occur inside the enclave and others occur outside. In order to do this, we modified our read benchmark to allow us to specify the percentage of loops that operate on a buffer outside vs. inside the enclave. This benchmark allocates two buffers: one inside the enclave and one outside. It then splits our memory-read loops between them. It first performs a set of reads outside the enclave, and then moves into the enclave to perform another set. The detailed algorithm can be found in Algorithm 2.

We executed the benchmark using three different buffer sizes (512 KiB, 1 MiB, and 75 MiB) covering sizes that fit

entirely within the cache and sizes that do not. We performed 500 loops, but split them between inside and outside. We varied the split by 10 percentage points for each test. Our results can be found in Fig. 10. As can be seen, for small buffer sizes that fit entirely within the CPU's cache (512 KiB and 1 MiB), the split between inside and outside has no effect on the total runtime of the benchmark. However, when the buffer exceeds the cache size, we see a linear increase in runtime as we run more and more of the loops inside the enclave. This is consistent with Fig. 1, and further demonstrates the penalty incurred by the MEE when the CPU cache is exceeded. We have also included a line showing the Graphene-SGX performance of this benchmark. Graphene-SGX does not vary at all because it pulls the entire application inside the enclave, meaning that none of the reads actually occur outside. Given this, the x-axis on this chart is not meaningful for Graphene-SGX. The line shows that, as expected, Graphene-SGX performs the same as our SGX version when all of the reads occur inside the enclave.

*Conclusions*

If the working set of an enclave can fit entirely in the cache, then the performance overhead due to the MEE is minimal. This applies equally to a manually ported SGX enclave or when executing using a shim layer such as Graphene. However, this opens up an opportunity for finding a workload that *will* benefit from manual porting. There is a key difference between running in Graphene-SGX and doing it manually: with Graphene-SGX, the entire application must be inside the enclave. With a manual port, a developer decides what goes in and what stays out based on security requirements. If an application can be partitioned in such a way as to ensure that the working set of data inside the enclave is smaller than the cache size, then manual porting may be faster. Doing so would require the developer to transition in and out of the enclave as needed in order to operate on sensitive data. Given this, the next logical step is to better evaluate the impact of enclave entrances and exits.

### B. Enclave Exit and Entrance Performance

Entering and exiting the enclave using ECALLs and OCALLs incurs significant overhead. [10] and [16] both observed ECALLs and OCALLs requiring between 5,850 and 14,000 cycles depending the cache state and CPU microcode revision. This delay has the potential to dominate enclave performance when compared to the slowdown from the MEE. In order to confirm this experimentally, we modified our benchmark to incorporate a variable number of ECALLs. Our new benchmark is shown in Algorithm 3. This is a modified version of our read benchmark in Algorithm 1, except that now the work of summing the data is shared between code inside and outside the enclave. The main idea is that the buffer is stored and initialized inside the enclave, and the nested summing loops run outside. The inner loop (line 13) uses an ECALL to get the sum of a specified number of integers each time. If the number of integers per read is very large (such as 262,144), then very few ECALLs will be issued during this

**Algorithm 3** Read Benchmark with Variable ECALLs

```
1: // This runs inside the enclave
2: procedure IN_GETPARTIALSUM(idx, howm)
3:     sum ← 0
4:     for i ← idx, howm do
5:         sum ← sum + in_buf[i]
6:     end for
7:     return sum
8: end procedure

9: // This runs outside the enclave
10: procedure OUT_SUMBUFFERPARTIALREAD(loops, num_ints, ints_per_read)
11:     sum ← 0
12:     for i ← 0, loops do
13:         for j ← 0; j < num_ints; j ← j + ints_per_read do
14:             sum ← sum + in_GetPartialSum(j, ints_per_read)
15:         end for
16:     end for
17: end procedure

18: // Execution starts here, and this runs outside the enclave
19: procedure ECALLBENCHMARK(loops, num_ints, ints_per_read)
20:     in_AllocateBuffer(num_ints)
21:     sum ← 0
22:     StartTimer()
23:     out_SumBufferPartialRead(loops, num_ints, ints_per_read)
24:     StopTimer()
25: end procedure
```

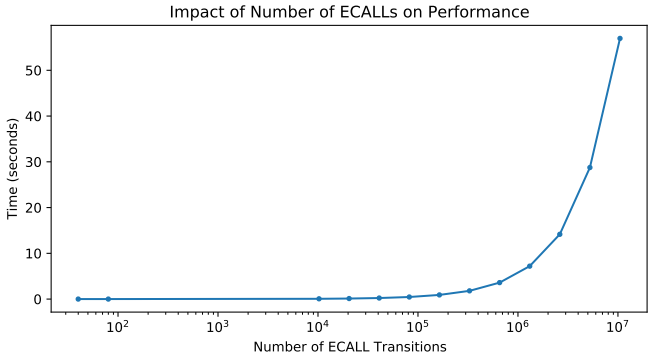



Fig. 11: Benchmark Completion Time vs. Number of ECALLs

benchmark. If it is very small (such as 1), then a large number of ECALLs will occur.

We executed the benchmark, specifying a 4 MiB buffer (so it fits in the cache) and 10 loops. We varied the number of integers to read per ECALL to include all powers of two starting from one (causing 10,485,760 ECALLs) to 262,144 (causing 40 ECALLs). The results are presented in Fig. 11 showing time for benchmark completion vs. number of ECALLs. A log-scale is used on the x-axis in order to emphasize the long tail that begins the graph. The results show a significant variation in total time to execute depending on the number of ECALLs. The farthest left data point (at 40 ECALLs) requires 5.7 ms to execute, while on the other extreme the same amount of memory accessed with 10485760 ECALLs requires almost 57 seconds. This variability is much larger than what is seen in Fig. 1, reinforcing the idea that the number of enclave transitions has the ability to dominate the execution time of the benchmark far more than the overhead from the MEE.

*Conclusions*

**Algorithm 4** Read Benchmark with Split Memory Access and Variable Enclave Transitions

```
1: // Execution starts here, and this runs outside the enclave
2: procedure SplitEcallBenchmark(loops_out, loops_in, num_ints_out,
   num_ints_in, ints_per_read)
3:     in_AllocateBuffer(num_ints_in)
4:     out_AllocateBuffer(num_ints_out)
5:     sum ← 0
6:     StartTimer()
7:     out_SumBufferPartialRead(loops_out, num_ints_in, ints_per_read)
8:     out_SumBufferRead(loops_out, num_ints_out)
9:     StopTimer()
10: end procedure
```

The significant impact of enclave transitions on the performance of our benchmark makes manual partitioning more challenging in terms of performance. Any separation necessitates transitioning in and out of the enclave, thus impacting performance. Trading better MEE performance for more enclave transitions is not an attractive choice, since the impact for enclave transitions is more significant.

### C. Combining the Effects of Memory and Enclave Transition Performance

Given our previous results, an application workload that would benefit from manual porting instead of running in a shim layer would have the following characteristics:

1) The working set of data inside the enclave is smaller than the cache size, but the total working set of the application is larger than the cache size. This would benefit manual porting, since the shim would pull the entire application into the enclave, causing expensive cache misses.
2) The application spends a substantial amount of time working on the larger dataset outside the enclave. This would benefit manual porting because the memory reads and writes that occur outside the enclave would be faster than the in-enclave reads performed by the shim.
3) The application has a relatively small number of enclave transitions.

Putting this together, a porting-friendly workload is an application with a larger-than-cache working set (say, 70 MiB) that could be split into a 2 MiB sensitive portion and a 68 MiB insensitive portion, with a small number of transitions in and out of enclave. We now establish how many enclave transitions are permissible to maintain performance benefits.

We designed a benchmark that combines the relevant properties of our previous benchmarks. The pseudo-code can be found in Algorithm 4. There are several variables for such a benchmark: 1) The number of loops performed inside and outside (to control the overhead from the MEE). 2) The size of the inside and outside buffers (to control whether or not the working sets of each fits into the cache). 3) The number of integers summed per enclave transition (to control the number of enclave transitions). ALL memory accesses outside the enclave are performed first, and then enclave accesses are performed while inducing enclave transitions.

As a first test, we run the benchmark with 64 MiB outside buffer, 1000 outside loops, a 2 MiB internal buffer, 10 internal loops, and we varied the number of integers to read per ECALL to include all powers of 2 from 1 (causing 5,242,880 ECALLs) to 262,144 (causing 20 ECALLs). Much like our standard split benchmark, we also show the result for Graphene-SGX, assuming that everything runs inside the enclave and no ECALLs are required. Fig. 12a shows the results. As can be seen, Graphene-SGX is unaffected by the enclave transitions (because they do not actually occur), while the manual port performance degrades linearly. There are a number of interesting observations to be made about this graph. First, the cross-over point between the manual port and Graphene is at around 2,100,000 ECALLs. To the left of this point, our manual port is faster, to the right of this point, the shim is faster. This illustrates that this benchmark does indeed help us answer the question of how many transitions are acceptable for the manual port to be faster. Another point to notice is that on the far left, where there are almost no code transitions, the total benchmark time is about 6 seconds. This actually represents the time required to perform the memory reads outside the enclave (line 8 in Algorithm 4). If we were to alter the number of outside loops, this point would shift either up or down, taking the entire line with it. To confirm this, we repeated the benchmark, but this time doubled the amount of outside-enclave work to 2000 loops. The results can be seen in Fig. 12b. As expected, when doubling the outside work the entire blue line shifts up by about 6 seconds for each data point. The Graphene line, however, instead almost doubles. The new cross-over point is closer to 4,000,000 ECALLs.

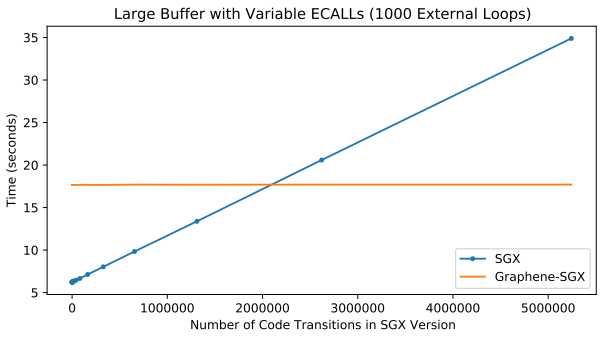
We performed further experiments for 3000, 4000, 5000, 6000, and 7000 external loops and found an interesting phenomena: For all of the tests, the ratio at the cross-over point is around 110,000 transitions/second. This rate may be a good indicator of which workload would benefit from porting: Workloads meeting the three criteria at the beginning of this section and that cause less than 110,000 transitions/second would be faster with a manual port than a shim.

## V. Key Take-Aways

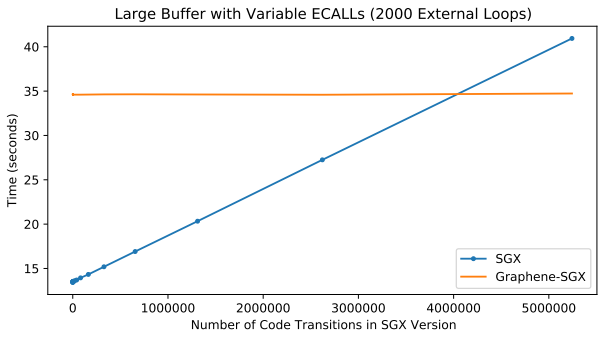
Based on the results presented in previous sections, what advice do we have for application developers? If your goal is to protect your entire application with SGX, then from a performance perspective it is almost always better to shim rather than port. Shimming takes significantly less developer time and can achieve the same basic performance for most workloads. (However, note our discussion in Sec. VI-A below for a discussion of non-performance factors.) However, if you only have a small piece of your application that you want to protect with SGX, then porting may end up being beneficial.

Let's look again at the workload that benefits from porting instead of shimming:

1) The working set inside the enclave is smaller than the processor's level 3 cache size (8 MB, 6MB of which is usable), while the working set outside the enclave is larger than the cache size.
2) The program spends a significant amount of time working on the larger data set outside the enclave.




(a) Large Buffer with Variable Code Transitions, 1000 external Loops




(b) Large Buffer with Variable Code Transitions, 2000 external Loops

Fig. 12: Determining the Point Where Manual Porting is Faster than Graphene for Two Workloads

3) The program has an average ECALL/second rate of around 110,000 or less.

In order to put the 110,000 rate into perspective, we refer to Weisse et al. [10] and their work profiling some large server applications. Unoptimized enclave ports of Memcache, OpenVPN, and Lighttpd were created and their per second ECALL rates were found to be 200K, 275K, and 270K, respectively. We suspect it would be difficult to re-architect those applications to cut the number of ECALLs in half without placing most of the application into the enclave (which would increase the working set size beyond that of the cache). Our results indicate that, from a performance perspective, it is not worthwhile to manually port these applications.

In practice, what sort of application would exhibit this type of workload? One that does the vast majority of its work outside the enclave and only enters the enclave occasionally to work on small amounts of data. For example, one could imagine a webserver that wants to use SGX to protect its private key from compromise. In this scenario, the private key could be loaded into an enclave which is only used when establishing new connections and performing the TLS key exchange. The vast majority of the work (socket communication, file reading/write, web application execution, and even the TLS transport encryption itself) would occur outside the enclave with only infrequent work happening inside. Such a scenario would benefit from manual partition and porting as opposed to running the entire webserver in a shim. We expect that there are actually quite a few applications/workloads like this: At the core, only a small piece of the application's functionality truly needs to be protected.

This implies that application developers need to carefully consider two issues: How much of their application is truly sensitive and how frequently that code needs to run. The smaller the working set of sensitive data, and the less frequently it runs, the more it will benefit from porting. The larger the working set and/or the more frequently it runs, the less benefit it will receive from porting. Don't be too quick to assume that minimizing the working set of sensitive data is easy: Reasoning about what portions of an application are sensitive or not is an open research problem [21], and outside of very simple examples (such as the private key above) is very difficult to solve.

## VI. Limitations

In this section we will discuss the limitations of this work.

### A. Factors Other Than Performance

While this work sought to investigate port vs shim from a performance perspective, there are other factors that should be considered when making the decision:

1) A large trusted computing base (TCB) is usually undesirable. By using a shim layer, the software TCB will include the SGX SDK, the library OS, and the entirety of the application. A bug in any of that code could potentially be used to compromise the enclave. In manual porting, there is no library OS and only security sensitive portions of the program need to be included.
2) Partitioning is hard. Determining which portions of your existing code base are security-sensitive and which are not is a difficult problem. A mistake could mean that sensitive data is revealed. Systems like Glamdring [21] seek to make the process of partitioning existing code bases easier and less error prone, but there is a certain level of confidence in knowing that everything is inside the enclave.
3) Shim layers may be better optimized than a manual port. A shim layer used by many people and maintained by expert developers seems more likely to receive performance enhancements. For example, when an advance like hotcalls [10] is made available for SGX [22], if the shim layer is upgraded to support it then the code can get that boost for free.

### B. Shim-Specific Results

The benchmark results in this work are all obtained using Graphene-SGX, and as such are specific to it. While we believe that the general conclusions are applicable to a variety of shim systems, we have not experimentally verified it. Further validation of these results could be done by repeating the

experiments using other shim layers such as SCONE [6]. In addition, the exact configuration of Graphene-SGX may have subtle impacts on both the shape and raw measurement values of the results. We used Graphene-SGX in its default configuration, but settings such as exitless calls[1], an implementation of switchless calls [22], will impact performance as well.

## VII. Conclusion

In this work, we benchmark various aspects of Intel's SGX in order to answer the question: Port or Shim? Our results show that if the goal is to protect the entire application with SGX, then from a performance perspective it is almost always better to shim rather than port: the development time is much lower and the performance of the resulting enclave is on-par with manual porting. Further investigating the issue, we experimentally determined the characteristics of a workload that would instead benefit from porting. We discovered that if an application only has a small sensitive working set (less than the 6MB available cache size of the CPU), infrequently needs to enter the enclave (less than 110,000 times per second), and spends most of its time working on data outside of the enclave then it may indeed perform better if it is manually partitioned and ported. Using these results, a developer should be able to ascertain whether their particular application would benefit from manual porting as opposed to shimming. Although this work is primarily concerned with performance, we also discuss some of the non-performance based issues that should be considered when deciding whether to port or shim.

Finally, the source code of our benchmarks can be found at https://github.com/vsecurity-research/sgx-bench

## Acknowledgement

This publication was made possible by the NPRP award NPRP 8-1474-2-626 from the Qatar National Research Fund (a member of The Qatar Foundation). The statements made herein are solely the responsibility of the authors.

## References

[1] J. P. Mechalas and B. J. Odom, "Intel Software Guard Extensions Tutorial Series: Part 1, Intel SGX Foundation," 2018, accessed 2020-04-15. [Online]. Available: https://software.intel.com/en-us/articles/intel-software-guard-extensions-tutorial-part-1-foundation

[2] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. New York, NY, USA: ACM, 2013, pp. 10:1–10:1.

[3] J. V. Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution," in *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, p. 991–1008.

[4] Intel, "Intel®Software Guard Extensions for Linux OS," 2019, accessed 2019-11-25. [Online]. Available: https://github.com/intel/linux-sgx

[5] A. Baumann, M. Peinado, and G. Hunt, "Shielding Applications From an Untrusted Cloud with Haven," *ACM Transactions on Computer Systems (TOCS)*, vol. 33, no. 3, pp. 1–26, 2015.

[1] https://github.com/oscarlab/graphene/blob/2093d64ce22c430d599da1dd9ba0ae5802be56d4/Documentation/perf-practices.rst#exitless-feature

[6] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. OKeeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE: Secure Linux Containers with Intel SGX," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 689–703.

[7] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. Santa Clara, CA: USENIX Association, Jul. 2017, pp. 645–658.

[8] L. McVoy and C. Staelin, "LMbench - Tools for Performance Analysis," 1998, accessed 2019-11-25. [Online]. Available: http://www.bitmover.com/lmbench/

[9] A. Shimizu, D. Townley, M. Joshi, and D. Ponomarev, "EA-PLRU: Enclave-Aware Cache Replacement," in *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '19. New York, NY, USA: ACM, 2019, pp. 5:1—-5:8.

[10] O. Weisse, V. Bertacco, and T. Austin, "Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ser. ISCA '17. New York, NY, USA: ACM, 2017, pp. 81–93.

[11] V. Costan and S. Devadas, "Intel SGX Explained," *IACR Cryptology ePrint Archive*, vol. 2016, no. 086, pp. 1–118, 2016. [Online]. Available: https://pdfs.semanticscholar.org/2d7f/3f4ca3fbb15ae04533456e5031e0d0dc845a.pdf

[12] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter, "Cooperation and Security Isolation of Library OSes for Multi-Process Applications," in *Proceedings of the Ninth European Conference on Computer Systems*. New York, NY, USA: Association for Computing Machinery, 2014.

[13] S. Shinde, D. Le Tien, S. Tople, and P. Saxena, "Panoply: Low-TCB Linux Applications With SGX Enclaves," in *Network and Distributed System Security Symposium (NDSS)*, 2017.

[14] H. Tian, Y. Zhang, C. Xing, and S. Yan, "SGXKernel: A Library Operating System Optimized for Intel SGX," in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 35–44.

[15] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica, "Opaque: An Oblivious and Encrypted Distributed Analytics Platform," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2017, pp. 283–298.

[16] N. Weichbrodt, P.-L. Aublin, and R. Kapitza, "sgx-perf: A Performance Analysis Tool for Intel SGX Enclaves," in *Proceedings of the 19th International Middleware Conference*. New York, NY, USA: ACM, 2018, pp. 201–213.

[17] S. Mofrad, F. Zhang, S. Lu, and W. Shi, "A Comparison Study of Intel SGX and AMD Memory Encryption Technology," in *Proceedings of the 7th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2018, pp. 1–8.

[18] T. Dinh Ngoc, B. Bui, S. Bitchebe, A. Tchana, V. Schiavoni, P. Felber, and D. Hagimont, "Everything You Should Know About Intel SGX Performance on Virtualized Systems," *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, vol. 3, no. 1, pp. 1–21, 2019.

[19] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 665–678.

[20] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre Attacks: Exploiting Speculative Execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA: IEEE, may 2019, pp. 1–19.

[21] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch, "Glamdring: Automatic Application Partitioning for Intel SGX," in *2017 USENIX Annual Technical Conference (USENIXATC 17)*, 2017, pp. 285–298.

[22] H. Tian, Q. Zhang, S. Yan, A. Rudnitsky, L. Shacham, R. Yariv, and N. Milshten, "Switchless Calls Made Practical in Intel SGX," in *Proceedings of the 3rd Workshop on System Software for Trusted Execution*, 2018, pp. 22–27.