Everything You Should Know About Intel SGX Performance on Virtualized Systems

ABSTRACT

Intel SGX has attracted much attention from academia and is already powering commercial applications. Cloud providers have also started implementing SGX in their cloud offerings. Research efforts on Intel SGX so far have mainly concentrated on its security and programmability. However, no work has studied in detail the performance degradation caused by SGX in virtualized systems. Such settings are particularly important, considering that virtualization is the *de facto* building block of cloud infrastructure, yet often comes with a performance impact. This paper presents for the first time a detailed performance analysis of Intel SGX in a virtualized system in comparison with a bare-metal system. Based on our findings, we identify several optimization strategies that would improve the performance of Intel SGX on such systems.

ACM Reference Format:

. 2019. Everything You Should Know About Intel SGX Performance on Virtualized Systems. In *Proceedings of ACM SIGMETRICS conference (SIGMETRICS 2019).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

With the increasing industry interest in cloud computing technologies and platforms, there have been numerous concerns about the secrecy and integrity of private data stored on these platforms. The trustworthiness of a cloud platform depends on numerous elements, where even one weakness can lead to collapse of its security model. First of all, customers need to trust the hypervisor used by the cloud provider, which most often is not open-source nor auditable, and might carry vulnerabilities or backdoors that will compromise its isolation guarantees. In addition, almost all of the system's hardware, software and configuration components are subject to vulnerabilities and misconfiguration, *e.g.*, guest OS, network configuration, access control, etc. Furthermore, more often than not, the human element is the weakest link: penetration testers consider social engineering as one of the most effective methods to infiltrate a secure environment, and insider threats might abuse their privileges to commit data theft, fraud or sabotage [2, 45].

Trusted computing systems [30, 40] attempt to resolve these problems by guaranteeing integrity and/or secrecy of the underlying platform in order to secure user data and prevent unauthorized accesses from the platform owner or malicious attackers. To ensure these guarantees, secure environments often offer a few common features. Firstly, *platform secrecy* prevents attackers from learning the system's operational state (CPU, memory, storage, network, etc.). Secondly, the environment can be further protected by *integrity assurances*, which deter malicious alterations of the platform's expected behavior by code modification, injection or malware attacks. Software running inside the secured environment can then prove its integrity and that of its environment to an external actor through an *attestation mechanism*, and protect its data both at rest and in transit using data protection facilities provided by the platform.

Some trusted computing solutions focus on protecting the hardware platform. For example, the *trusted platform module* (TPM) [17, 20, 21] is mainly used for platform integrity purposes by measuring critical system software (firmware, boot loader, kernel, etc.) through the use of platform configuration registers (PCRs), and binding specific cryptographic secrets to a set of PCRs, *e.g.*, for use in disk encryption. However, the TPM alone is not a complete solution to trustworthy computing, as the user still needs to trust many other components, such as the OS, storage devices or communication buses within the computer [32, 41].

To address these limitations, Intel introduced the *software guard extensions* (SGX) [6], which provide a trusted computing system that enforces a security boundary within the processor package. With SGX, users no longer need to trust the system software nor any other hardware components. In addition, SGX is a self-contained solution that does not depend on external devices or configurations, and does not interfere with virtualization.

Aside from Intel SGX, other vendors have also created virtualization-friendly trusted computing solutions. AMD proposed *secure encrypted virtualization* (SEV) [31], a memory encryption-based technology that promises to protect virtual machines (VMs) from hypervisor attacks and hardware snooping. Microsoft introduced the concept of *shielded VMs* [36], which are special VMs running on strictly controlled environments called *guarded fabrics*, and are protected from snooping and alteration by the host administrator.

This paper focuses exclusively on Intel SGX, which has attracted much attention from academia [7, 11, 43] and is already powering commercial applications [28, 39]. Cloud providers have started implementing SGX in their cloud offerings, beginning with Microsoft's Azure confidential computing or IBM Cloud [24], and new cloud programming frameworks already have support for SGX [19].

Research efforts on Intel SGX so far have mainly concentrated on its security and programmability [13, 34], notably for simplifying the use of its APIs and for supporting the execution of legacy code within SGX-protected enclaves. HotCalls [44] follows a different path by focusing on SGX performance impacts on native systems. Yet, no work has studied in detail the performance degradation caused by SGX in virtualized systems. Such settings are particularly important considering that virtualization is the *de facto* building

SIGMETRICS 2019, June 24-28, Phoenix, Arizona, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

block of cloud infrastructure, and often comes with a performance impact [33].

In this paper, we present for the first time a detailed analysis of the performance of Intel SGX in a virtualized system in comparison with a bare-metal system. Based on our findings, we identify several optimization strategies that would improve the performance of Intel SGX on such systems.

In summary, our paper presents the following findings:

(1) Hypervisors do not need to intercept SGX instructions in order to enable SGX for VMs; the only current exception is ECREATE that must be intercepted to virtualize SGX launch control [15].
(2) SGX overhead on VMs when running memory-heavy benchmarks stems mostly from address translation costs when using nested paging. With shadow paging, virtualized SGX has nearly identical performance to SGX on bare-metal (less than 1.3% overhead on average for enclave calls; 1%–3% for encryption). Comparatively, nested paging is 7.4% slower than bare-metal at enclave calls, and up to 10% slower at encryption.
(3) Conversely, when running benchmarks involving many context switches (*e.g.*, HTTP benchmarks), shadow paging performs worse than nested paging (13.8% vs. 4.1% for nested paging at low request rates).
(4) We propose a method for dynamically detecting the characteristics of a given workload to identify whether it is suitable with nested paging or shadow paging, therefore allowing the automatic selection of an appropriate memory virtualization technique.
(5) SGX imposes a heavy performance penalty upon switching between the application and the enclave, ranging from 10,000 to 18,000 cycles per call depending on the call mechanism used. This penalty affects server applications using SGX, as discussed in [8, 44].
(6) Swapping the enclave page cache (EPC) pages is expensive, costing up to hundreds of thousands of cycles per swapping operation. As SGX measures the contents of the enclave upon initialization, including any statically-declared arrays, such initialization will trigger enclave swapping if the enclave memory size is larger than the available EPC size. Additionally, virtualization causes an overhead of 28% to 65% when performing an EPC eviction operation, which increases depending on the number of threads running inside the enclave.

This paper is organized as follows: Section 2 briefly explains how Intel SGX works on bare metal and virtualized environments. We present an overview of our evaluation methodologies in Section 3, followed by the evaluation results and lessons learned from our experiments in Sections 4 and 5. Section 6 proposes an optimization for improving application performance in virtualized SGX environments. We present the evaluation of these optimizations in Section 7. Finally, we discuss some relevant works in Section 8. Section 9 presents the conclusion of our work.

2 BACKGROUND

In this section, we present the structure and operation of an SGX enclave, and provide details on the operation of SGX enclaves on virtualized platforms.

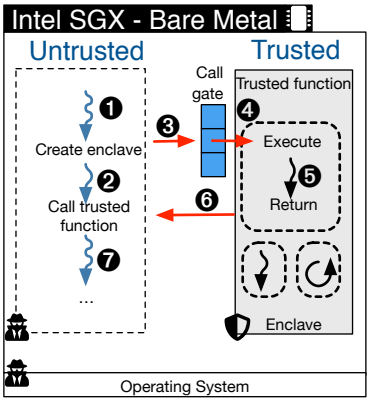

Figure 1: Intel SGX workflow

2.1 Intel SGX on bare-metal systems

Intel software guard extensions (SGX) [16] provides a mean for application developers to secure their code and data. It is available in Intel CPUs since the Skylake microarchitecture [38]. With Intel SGX, applications are protected from snooping and modification, including attacks from other processes, the underlying OS, the hypervisor, or even the BIOS. To benefit from SGX protection, applications create secure isolated regions called *enclaves*. Enclaves contain their own memory and data, which are encrypted and authenticated by the CPU, and can interact with outside programs through predefined entry points in the enclave binary. Application developers can rely on the provided Software Developer Kit (SDK) provided by Intel to easily create SGX-enabled applications across different operating systems [29].

One of SGX's notable features is a small trusted computing base (TCB): the TCB of an SGX enclave includes only the CPU package itself, and any application code running inside the enclave. Moreover, it is easy to identify and isolate parts of the program that should run inside the enclave [34].

To protect the enclave from unauthorized accesses and modifications, SGX utilizes an on-die *memory encryption engine* (MEE) [22] and in-CPU logic. As a result, attacks such as reads and writes to the enclave memory region or snooping of memory buses outside the CPU are blocked by SGX.

Intel SGX includes local and remote attestation as well as secret sealing capabilities. Attestation lets enclaves prove their identities to other programs: local attestation works with enclaves on the same machine, while remote attestation works with an external service. Secret sealing lets SGX enclaves securely store protected data outside the enclave, to reload when necessary.

For SGX enclaves to function, aside from having the necessary CPU and BIOS support for SGX, users must install a dedicated Intel SGX driver, which manages enclave creation and memory management; as well as the Intel SGX platform software, which provides architectural enclaves for enclave launch management, remote attestation and other platform services.

At boot time, the BIOS verifies whether SGX is enabled. It then reserves a region of physical memory for the CPU, designated as the *enclave page cache* (EPC). In the current iteration of SGX, the

maximum EPC size is limited to 128 MB, of which only 93MB are usable by applications; the rest is allocated for metadata and cannot be used by enclaves.

Figure 1 illustrates the typical operation of an SGX enclave, starting from its initialization to the execution of the demanded trusted function and the return of control to the untrusted code. For the sake of clarity, we omit details such as the enclave measurement and attestation, as well as the detailed enclave initialization process.

The initialization of an enclave involves four main steps:

(1) Creation of an SGX Enclave Control Structure (SECS) via the `ECREATE` instruction.
(2) Copying of code and data pages into the enclave using the `EADD` instruction.
(3) Creation of cryptographic measurement of enclave memory contents using the `EEXTEND` instruction.
(4) Verification of enclave measurement against signature from application developer and activation of the enclave using the `EINIT` instruction.

After the initialization is complete, applications can enter the enclave using the `EENTER` instruction. Once inside the enclave, there are two ways to exit the enclave: either by using the `EEXIT` instruction to return from the enclave, allowing the application to enter the enclave again using `EENTER`; or by exiting the enclave through the asynchronous exit (AEX) mechanism, from which the `ERESUME` instruction can be used to resume enclave execution.

SGX enclaves must be hosted on encrypted, reserved EPC memory. SGX allows enclaves to use EPC memory beyond the available capacity by swapping pages from EPC to normal memory. Before being written out from the EPC, the page is encrypted and sealed by the CPU, and kept track of for future restoration in special EPC pages called a *version array*. Its corresponding TLB entries are also flushed as part of the eviction process. When the page needs to be swapped in, it is verified by the CPU, decrypted and copied back into the EPC. This swapping support eases the problem of memory management inside SGX enclaves by making more memory available to enclaves than what is provided by the hardware.

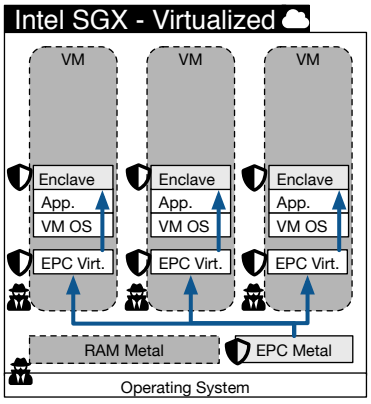

Figure 2: Intel SGX on a virtualized environment

2.2 Intel SGX in a virtualized system

One of SGX's main use cases is cloud computing, where the customer can start enclaves and perform secure computations without revealing sensitive data to the untrusted platform provider.

SGX is designed as a separate execution mode that can be entered from unprivileged mode (ring > 0) using special instructions. Hypervisors can expose SGX support to VMs by unmasking the CPUID flags relevant to SGX operations. For each VM started with SGX support, the hypervisor allocates a section of the EPC to use as virtual EPC; this EPC region can be allocated exclusively for one VM (on SGX v1) or oversubscribed to several VMs (SGX v2), and is treated the same way a bare-metal machine treats its EPC region. Each VM is then supplied with its own SGX runtime software (driver and platform software) to provide enclave functionalities. Upon enclave startup, the enclave memory is allocated from the virtual EPC and the enclave functions as normal. Figure 2 shows a possible deployment of SGX-enabled VMs, where the hardware EPC is partitioned and distributed to several VMs.

Intel provides two hypervisor implementations with SGX support based on the KVM and Xen hypervisors [27]. At the time of this writing, the KVM support for SGX is more up-to-date; hence, we perform our experimental evaluation exclusively on KVM.

The next section presents the methodology we adopted to systematically evaluate SGX in virtualized systems.

3 EVALUATION METHODOLOGY

To facilitate programming SGX applications, Intel provides the Intel SGX SDK on both Windows [25] and Linux [26]. The SDK allows developers to write both parts of an SGX application, the untrusted application and trusted enclave, using the same development toolchain. In the application, developers can use two different call mechanisms provided by the SDK: `ecall`, which is a call from the untrusted application to the enclave; and `ocall`, which is a call from the enclave to the untrusted application.

Both `ecalls` and `ocalls` support transferring a buffer between the application and the enclave, where the SDK will handle the necessary buffer transfer logic automatically. Enclave calls are defined through an enclave definition language (EDL) file, which lists all possible enclave calls and buffer transfers allowed by the enclave. The SGX SDK provides a tool called `edger8r` (pronounced "edgerator") for automatically generating code to support these calls on both the application side and the enclave side.

In order to investigate the performance of Intel SGX, we aim to measure the main SDK functions on both bare-metal and VMs. This includes:

- Performance of plain `ecall` and `ocall`.
- Performance of transferring buffers into/out/in&out of the enclave using buffer marshaling features included in the SDK, for both `ecall` and `ocall`.
- Performance of reading and writing to encrypted memory from inside the enclave.
- Performance of reading and writing to unencrypted memory from both inside and outside the enclave.
- Performance of evicting pages from EPC.
- Performance of initializing and destroying enclaves.

We are the first to measure the impact of swapping EPC page as done in the Intel SGX driver, as well as enclave initialization/destroy time. In addition to these specific evaluations, we have also experimented with two benchmarks involving functions commonly used in enclaves: encryption benchmarks, and an HTTP server benchmark.

For call latency, page eviction and init/destroy experiments, we measure performance in terms of CPU cycles using the `RDTSCP` instruction. To ensure consistency, we run and record each experiment multiple times, with a period of warm-up before the experiment results are collected. For example, the plain `ocall` experiment is done in 10 runs, each consisting of 10,000 warm-up iterations and 100,000 experiment iterations. The exact number of runs is noted in our results. We also considered the impact of CPU cache on run time by running experiments in which the cache is cleared before every execution. After these experiments are complete, their performance results are collected and analyzed by calculating the average, standard deviation, minimum, maximum and median values, and these results are discussed whenever relevant.

In order to make our work reproducible, all micro- and macrobenchmarks have been made (anonymously) available at [1]. These programs can also be used to perform other evaluations of SGX, avoiding the need to rewrite these tricky benchmarks. We discuss the evaluation results in the section below.

4 EVALUATION RESULTS

In this section, we first present our experimental setup and then discuss in depth the various results that we obtained during our comparative study.

4.1 Experimental setup

We performed our experiments using two machines. The first one is a Dell Latitude 5280 equipped with an Intel Core i5-7200U, with 2 cores and 16 GB of RAM. The second one is a Dell PowerEdge R330 with a Xeon E3-1270 v6, with 4 cores and 64 GB of RAM. Unless indicated otherwise, we carried out experiments on the first machine. All experiments are run on Ubuntu 16.04, using the kvm-sgx kernel release sgx-v4.14.28-r1 and qemu-sgx release sgx-v2.10.1-r1 from Intel, and a reserved EPC size of 128 MB. For experiments on VMs, unless indicated otherwise, we use a VM configuration with 2 virtual CPUs, 6 GB of RAM and 32 MB of EPC capacity. During all experiments, we disabled automatic CPU frequency scaling, Turbo Boost and hyper-threading to avoid inconsistent performance behaviour. The benchmark software was built with GNU GCC 5.4.0 and compiler optimizations were disabled using the -O0 flag.

4.2 `ecall` performance

We begin by measuring the performance of `ecall` in various conditions. The `ecall`s are SDK functionalities that let applications call into enclave code through a simple function call interface. They also allow enclave developers to specify buffers to marshal between the application and the enclave. There are four main buffer passing modes:

(1) Transfer from the application into the enclave, designated by the `in` EDL keyword. In this mode, the buffer is copied from the application into the enclave memory.
(2) Transfer from the enclave into the application, designated by the `out` EDL keyword. In this mode, the output buffer is allocated inside the enclave, and copied back to the application memory once the function returns.
(3) Transfer in the enclave then back, using both `in` and `out` keywords. The buffer is first copied into the enclave, processed by the `ecall` function, then copied back when the function returns. This prevents leaking secrets (*e.g.*, encryption keys) against memory-snooping attackers during data processing inside the buffer.
(4) Unvalidated pointer passing. In this scenario, the enclave receives an arbitrary pointer and does not use the buffer marshaling features provided by the SDK. The enclave must manually verify the validity of this pointer before reading or writing to it. This pointer can point to an address in unencrypted memory, allowing zero-copy data transfer from/to the enclave.

In our experiments, we consider the following different scenarios:

- `ecall` to an empty enclave function (warm/cold CPU cache);
- `ecall` to an enclave function, passing a 2 KB buffer "in" (warm/cold cache);
- `ecall` to an enclave function, passing a 2 KB buffer "out" (warm/cold cache); and
- `ecall` to an enclave function, passing a 2 KB buffer "in&out" (warm/cold cache).

In order to evaluate enclave function calls from a cold CPU cache, we clear the cache by reading and writing to a buffer with size larger than the CPU cache before every benchmark run, being sure to prevent compiler optimizations from eliminating the reads and writes. After clearing the cache, we insert a memory fence (using the `MFENCE` instruction) to ensure that the cache is completely flushed to RAM.

SGX must make access checks during each address translation, just before committing the translation to TLB [16]. This may be one of the causes of performance overhead. Therefore, we would like to evaluate SGX with different memory virtualization techniques. Most hypervisors support a software-based technique called shadow paging and a hardware-based one called nested paging (known as EPT in Intel CPUs) [37]. The address translation in a virtualized environment is about translating a guest virtual address (gVA) to a host physical address (hPA). With shadow paging, for each process in the VM, the hypervisor maintains a shadow page table mapping directly gVA to hPA. The shadow page tables (instead of the guest page tables) are then used by the native page table walk mechanism to do the address translation. The hypervisor must intervene each time the guest VM update its page tables in order to update the shadow page tables accordingly. This may causes a significant overhead on context switches and greatly reduces VM performance.

On each modern x86 CPU core, there are the page table pointer to the guest page table and the page table pointer to the nested page table. Under nested paging, both page tables are used: first the

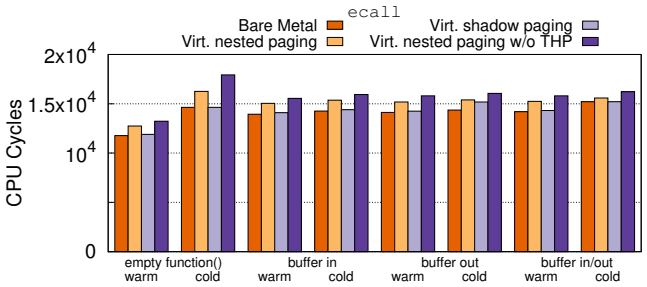

Figure 3: Time taken by one `ecall` (lower is better)

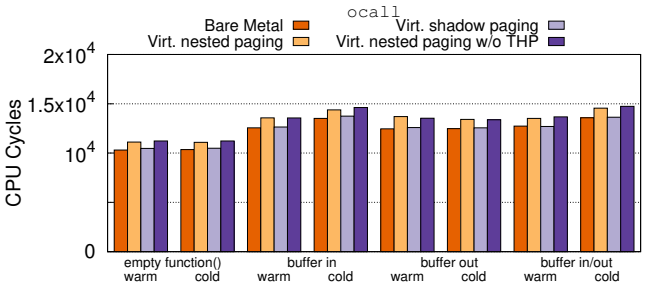

Figure 4: Time taken by one `ocall` (lower is better)

guest page table maps the gVA to the guest physical address (gPA) and then the nested page table maps the gPA to the hPA (this is usually referred as 2D translation). As a result, nested paging is more expensive than shadow paging for a TLB miss as it requires more memory references. For instance, with 4-level page tables, shadow paging requires only 4 references while nested paging requires 24 references to the two page tables. In applications triggering many TLB misses (*e.g.*, memory-heavy applications), this can lead to degredated performance [10].

The Linux kernel supports *transparent huge pages* (THP), which allow the allocation of large 2 MB memory pages when an application requests large allocations without having to use the `hugetlbfs` interface. The kernel supports three THP modes: `always`, `madvise` and `never`. The `always` option makes THP available to all allocations; `madvise` constrains THP to allocations marked with the `MADV_HUGEPAGE` flag; and `never` disables the feature altogether. KVM guests uses THP by default to eliminate one level of address translation on the host, therefore reducing the performance impact caused by 2D address translations on TLB misses.

To account for these features, we evaluated SGX enclave calls under four conditions: in a bare-metal system; in a VM with nested paging and THP enabled; in a VM with nested paging disabled and using shadow paging instead; and in a VM with nested paging enabled but THP disabled (denoted by *Virt. nested paging w/o THP*). We also ran these functions with warm and cold cache, where the buffer to be transferred is flushed with the `CLFLUSH` instruction before every run, except for the empty function benchmark where we read and write a large buffer to completely flush the CPU cache. Each benchmark was evaluated 1 million times, and the median cycle counts are listed below.

Figure 3 shows the time taken by one `ecall` under various conditions. We can see that VMs with shadow paging perform much closer to bare-metal systems, with only a 1.3% average overhead over all benchmarks, while VMs with nested paging enabled perform the tasks noticeably slower, at 7.4% overhead for the same scenarios. In addition, we can see a small performance penalty with THP turned off: the same experiments with THP disabled runs 12.5% slower on average than bare-metal, and 4.7% slower than VMs with THP enabled. This phenomenon can be explained by large pages not being used on the host adding one level of page table during address translation. From these results, we can conclude that `ecall` performance on VMs is impacted mostly by address translation overhead.

For the empty function benchmark with a cold CPU cache, we observe an overhead of 23%–28% for bare-metal and virtualized with shadow or nested paging, however VMs with THP disabled lagged behind in performance at a noticeably higher 35% overhead. Additionally, when comparing function calls involving buffer transfers with a warm and cold cache, we note only a small overhead of 2.1%–2.4% (when transferring buffers "in") and 1.4%–1.7% (when transferring buffers "out") except for shadow paging when transferring buffers "out" with a larger slowdown of 6.5%. However, this slowdown is magnified in the case of transferring buffers "in&out" on bare-metal and shadow paging (7.0% and 6.2% compared to 2.2% and 2.6% for nested paging with THP enabled/disabled, respectively).

4.3 `ocall` performance

Similar to `ecalls`, `ocalls` allow interfacing with SGX, but in the reverse direction: enclaves can call into application functions through predefined interfaces. This lets enclaves easily perform I/O calls (*e.g.*, printing to standard output) without having to rely on the application calling it. The `ocalls` support the same four buffer passing modes as `ecalls`, except "in" now means transferring from the enclave into the application, and vice versa. We experimented with the same four scenarios for `ocalls`: to an empty function, and to an application function, passing a 2 KB buffer "in", "out" and "in&out". Experiments with each of the four scenarios were performed with both a warm cache and cold cache.

Figure 4 shows these results. We can see that `ocalls` are generally faster than `ecalls`; with a warm cache, `ocalls` are 12.4% faster than `ecalls` with an empty function, and 10.6% faster when transferring a 2 KB buffer. On average, the performance breakdowns follow the same order as in `ecall` benchmarks: bare-metal, shadow paging (0.9% slower), nested paging with THP enabled (7.5% slower), and finally nested paging with THP disabled (8.2% slower).

When comparing performance between warm and cold cache, we observe little differences in performance in the empty function and buffer "out" benchmarks. However, for the buffer "in" and "in&out" benchmark, we observe a performance overhead of 6.0%–8.6% depending on the virtualization method when the cache is cold.

4.4 Memory accesses

To measure the overhead of SGX memory encryption, we compared the performance of memory reads and writes from the enclave

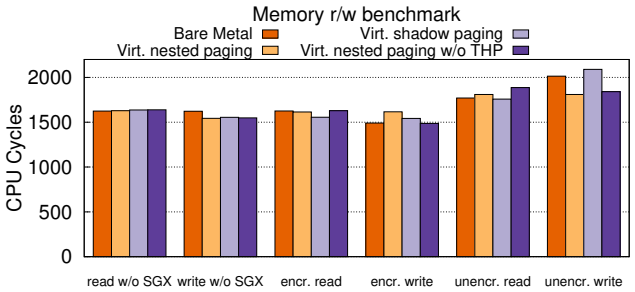

Figure 5: Time taken to read/write 2 KB buffer (lower is better)

with normal reads and writes from outside the application. The experiments included three scenarios:

(1) reading and writing to unencrypted memory from the application (without enclave involvement);
(2) reading and writing to encrypted memory from the enclave; and
(3) reading and writing to unencrypted memory from the enclave.

Since reading and writing to memory from within the enclave requires an enclave call, we subtracted the median empty call overhead obtained from the previous experiments from the displayed run times (this is because in the version of SGX used in the benchmarks, we can't read the TSC from inside the enclave). Figure 5 presents our results. Note that in some cases (i.g. write w/o SGX), the performance in bare metal seems lower than the performance in virtualization environment. However the performance differences are very small. We could say that the results are in the margin of error. While we see that memory reads and writes into encrypted memory from the enclave is similar to performance without SGX (less than 2%), reads and writes into unencrypted memory from enclave is slightly slower (approximately 17%). Similarly, comparing virtualized enclaves against bare-metal on average, we observe less than 1% difference in performance for all virtualization methods; however, individual benchmarks show more variability, with virtualized benchmark run times between 11% and 8% of bare-metal.

4.5 Swapping

The Linux SGX driver allows overcommitting of EPC to enclaves; *i.e.*, applications can consume more EPC than what is physically available on the machine. EPC swapping is supported by the EPC page eviction feature on CPUs with SGX; as a result, swapping confers an overhead during both page writeout and readback. In order to evict an enclave page from the EPC, it is first marked as "blocked" using the `EBLOCK` instruction; afterwards, no new TLB entries pointing to the page can be created. After `EBLOCK`, `ETRACK` is executed to keep track of TLB entries concerning the page. The OS must then cause all processors executing inside the enclave to exit using interprocessor interrupts to flush these TLB entries. Finally, once all processors have exited from the enclave, the OS can use `EWB` to complete the eviction process and write the sealed memory page to RAM.

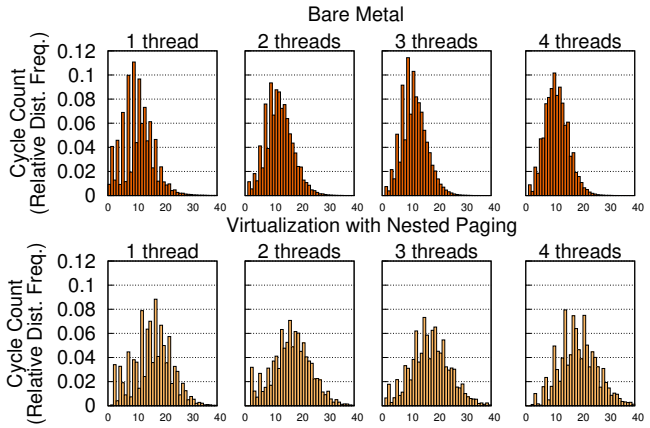

Figure 6: Frequency distribution of time taken by one eviction operation (rounded to 10,000 cycles)

It is important to note that the page eviction process requires interrupting all CPUs executing inside the enclave that owns the page to be swapped, in order to flush TLB entries associated with that page before it can be finally evicted from EPC. This process might potentially take a long time to finish and affect enclave operations.

We instrumented the code of the Linux SGX driver to measure the time taken to evict one page from the EPC, counting from before `EBLOCK` is called to after `EWB` has completed. We experimented with our Xeon machine. For VMs running on this machine, we allocated 8 vCPUs and 8 GB of RAM to the VM, while varying the number of enclave threads from 1 to 4. Each enclave thread continuously accessing random addresses in a large memory buffer inside the enclave, therefore forcing memory pages to be continually evicted from the EPC. In this experiment, we allocated the maximum EPC capacity available to VMs, and repeated the accesses 100 million times for each execution.

Figure 6 shows the distribution of execution time for each eviction operation on bare-metal and VMs depending on the number of enclave threads. We see that on bare-metal, eviction operations take almost the same time regardless of whether 1, 2, 3 or 4 enclave threads were used. However, while we observe a single peak in the histogram with 4 enclave threads, this is not the case with just 1 enclave thread: the cycle count tends to concentrate around several value ranges, rather than accumulating in one single peak. This behavior potentially originates from the handling of hardware timers, where the CPU has to stop the enclave threads whenever it receives a periodic interrupt.

However, when running the same experiment on VMs, we observe an increase in the number of cycles taken by an eviction operation as the number of enclave thread increases. In particular, we see a 12% increase in median eviction time from 1 enclave thread to 4 enclave threads. As a result, we see a virtualization overhead of 28% to 65% in our eviction experiments on VMs compared to bare-metal, depending on the number of enclave threads. This may be due to the Inter-Processor Interrupts (IPIs) which are slower

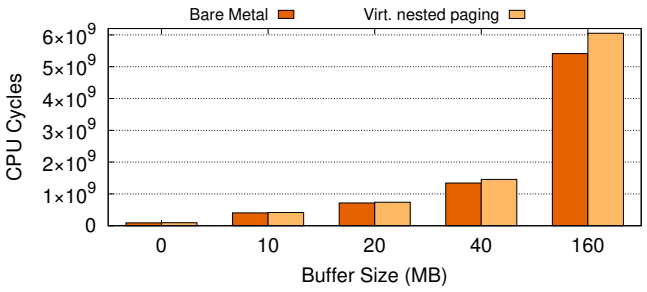

Figure 7: Median enclave startup time (lower is better)

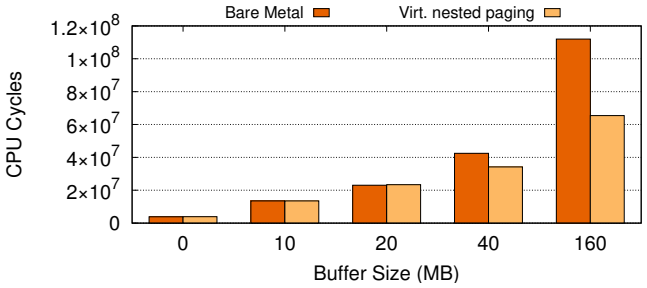

Figure 8: Median enclave shutdown time (lower is better)

in VMs compared with in bare-metal. Additionally, we see a similar behavior of concentrating frequency peaks when running 4 enclave threads on virtual machine, compared to what we see in the bare-metal experiment.

4.6 Enclave initialization and destruction

Before an enclave is ready for use, its memory contents are measured by the CPU to produce a cryptographic hash. We wish to measure the overhead caused by this measurement on enclave startup and shutdown time. To do so, we added a large static array into the enclave code and measured the enclave startup time vs. array size. Each experiment was run 10,000 times, and the total time for enclave startup and shutdown is measured using the `RDTSCP` instruction.

Figures 7 and 8 respectively show the median startup and shutdown time for bare-metal and virtualized enclaves. Note that the bare-metal machine utilizes a 128 MB EPC size, of which approximately 93 MB is available to applications, whereas the VM uses a 32 MB virtual EPC. While we see that enclave startup and shutdown times for enclaves with buffer size of 20 MB or lower are approximately equal, from 40 MB buffer size and above we see that the bare-metal machine takes less time to initialize the enclave, but more time for shutdown. This is explained by the larger EPC size of the bare-metal host: during startup, all pages allocated to the enclave must be touched during its cryptographic measurement. Larger EPC means that the enclave requires less swapping to touch all pages, and therefore faster startup; however, upon enclave shutdown, larger EPC also means more pages in the enclave's working set must be deallocated, leading to a slower shutdown.

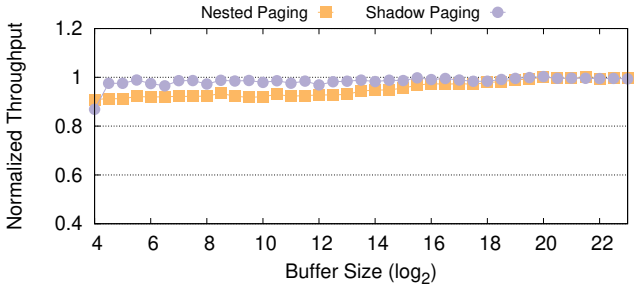

Figure 9: SGX-SSL encryption throughput normalized to bare-metal enclaves (higher is better)

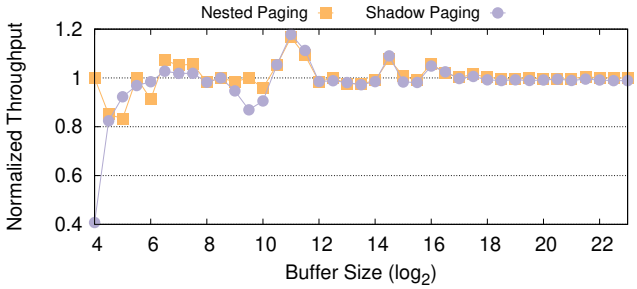

Figure 10: OpenSSL encryption throughput normalized to bare-metal (higher is better)

After the individual evaluation of SGX operations, we further evaluated SGX using macro-benchmarks involving cryptography.

4.7 SGX-SSL benchmarks

We ran benchmarks of the SGX-SSL library as provided in [23] on both bare-metal and VM. The benchmark measures the performance of encrypting memory buffers of various sizes from 16 bytes to 16 MB using AES-128-GCM. The encryption performance is measured in average throughput (MB/s). We executed the experiments 15 times, normalized the results in comparison to SGX-SSL on bare-metal enclaves, and recorded the performance results in Figure 9.

In general, virtualized enclaves have a small (less than 10%) performance penalty over bare-metal enclaves. At small buffer sizes (below 256 KB), virtualized enclaves with nested paging enabled perform worse than enclaves with shadow paging (up to 7.6% slower). This slowdown disappears with larger buffer sizes, where both nested paging and shadow paging perform close to bare-metal performance (within 1%).

To complete our experiments, we also recorded the encryption performance without SGX to see whether SGX introduces any differences. Figure 10 shows the results for the standard OpenSSL library which is the base library of the SGX-SSL library. We can see that the performance of shadowing paging and nested paging are quite identical without SGX. In other words, in the SGX-SSL benchmarks, SGX causes more performance overhead to nested paging.

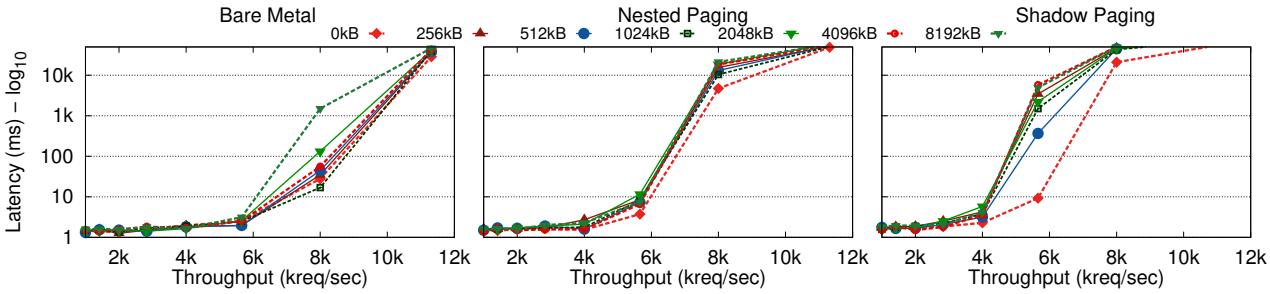

Figure 11: Nginx throughput/latency for increasing requests per second and different response sizes

4.8 HTTP benchmarks

The TaLoS library [8] provides an OpenSSL-based API that handles TLS connections inside an SGX enclave in order to protect connection secrets (private keys, session secrets, etc.) from snooping. In this experiment, we measure the HTTP request throughput and latency of the Nginx web server when using TaLoS for TLS connection termination. The web server is configured to use TLS for encryption and to serve various payload sizes from 0 to 8 KB. We used a separate machine connected through a 1 Gbit Ethernet connection and the *wrk2* utility [42] for benchmarking. The experiments were performed on a bare-metal machine, VM with nested paging, and VM with shadow paging. Benchmarks on VMs were ran with the virtual network card set to bridge mode. Each benchmark was run continuously for 5 minutes, and the throughput and median latency was recorded by *wrk2*.

Figure 11 shows the performance of bare-metal systems, as well as VMs with nested and shadow paging. The benchmark consists of issuing requests at increasingly high constant rates (x-axis) until the response latency spikes (y-axis, log scale). Unlike our micro-benchmarks and SSL benchmarks, the HTTP benchmarks show worse performance with shadow paging compared to nested paging: at 1,000 requests per second, shadow paging has an average latency impact of 13.8%, while in comparison the impact is only 4.1% with nested paging The difference widens beginning at a request rate of 4,000 requests per second, where shadow paging doubles the request latency while nested paging only shows a 8.9% impact. This is explained by the benchmark being I/O-heavy and therefore involving more context switches, which have a high penalty when shadow paging is used. Nonetheless, in our evaluation nested paging still displays a reduction in throughput ranging from 4.1% to over 56× depending on request rate, which is potentially caused by KVM's network interface virtualization.

We discuss the main lessons learned from these results in the next section.

5 LESSONS LEARNED

After evaluating the performance and also the source code of Intel SGX under various conditions, we learned the following lessons concerning SGX virtualization:

(i) Hypervisors do not need to intercept SGX instructions in order to enable SGX for VMs; the only current exception is intercepting `ECREATE` to virtualize SGX Launch Control [15]. As a result, SGX on VMs has an acceptable overhead compared to SGX on bare-metal platforms.

(ii) SGX overhead on VMs when running memory-heavy benchmarks consists mostly of address translation overhead when using nested paging. With shadow paging, virtualized SGX has nearly identical performance to SGX on bare-metal.

(iii) Conversely, when running benchmarks involving many context switches (*e.g.*, HTTP benchmarks), shadow paging performs worse than nested paging.

Secondly, the following lessons apply to Intel SGX in general (*i.e.*, on both bare-metal and virtualized platforms):

(iv) SGX imposes a heavy performance penalty on switching between the application and the enclave. This penalty affects server applications using SGX, *e.g.*, as described in [8, 44]. [44] additionally suggests improvements to the SGX call mechanism.

(v) Swapping EPC pages is expensive, costing hundreds of thousands of cycles per swapping operation. As SGX measures the contents of the enclave on start, including any statically-initialized arrays, this will trigger enclave swapping if enclave memory size is larger than available EPC size. Virtualization causes an additional overhead, which increases depending on the number of threads running inside the enclave.

Lesson (ii) indicates that nested paging contributes to enclave slowdown on VMs, as shown in our enclave micro-benchmarks. Address translation for SGX enclaves can be optimized by using shadow paging for EPC to reduce translation overhead and nested paging for general usage, leading to an agile address translator as described in [18].

Aside from this optimization, we propose further optimizations that apply to SGX in all systems:

(1) Lesson (iv) can be addressed by using mechanisms such HotCalls [44] that provide a fast call interface between the application and enclave code. Efficient call interfaces help reduce the significant overhead of `ecalls` and `ocalls`, and help ease the porting of applications to Intel SGX.

(2) Lesson (v) identifies swapping of EPC pages to RAM as major source of overhead, hence minimizing enclave size can help reduce swapping at enclave startup and during enclave operation, and consequently increase enclave performance.

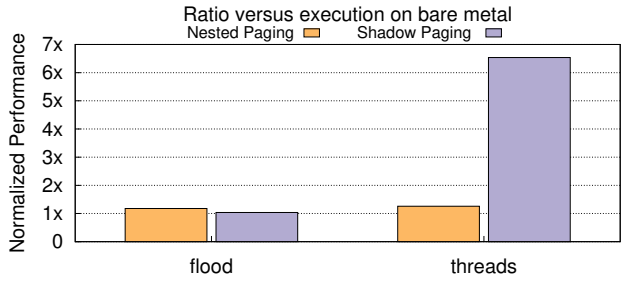

Figure 12: Normalized benchmark run time (lower is better)

(3) Along with minimizing enclaves, a smarter choice of pages during EPC page eviction helps reduce EPC swapping overhead when free EPC is low.

Next, we detail our optimization for SGX on VMs.

6 OPTIMIZATIONS

As shown in the previous section, both shadow paging and nested paging impose an overhead on applications running on VMs. With shadow paging, TLB misses are not slower than TLB misses in bare-metal, however, each modification of the page table requires trapping into the hypervisor to update the shadow page table. Conversely, with nested paging, modifications of the page table do not require trapping into the hypervisor, but each TLB miss require a more expensive 2D page walk [9]. As a result, a VM that generates numerous TLB misses should use shadow paging for memory virtualization, while in other cases nested paging can be used to facilitate quicker page table edits. Alternatively, an agile solution as proposed by [18] can be used to get the best of both worlds. In either case, the hypervisor should be able to dynamically identify the VM's workload type to make a suitable choice between the two methods.

In this section, we propose a non-intrusive method for measuring the characteristics of a given workload to identify whether it is suitable with nested paging or shadow paging, therefore allowing the automatic selection of an appropriate memory virtualization technique.

First, we identified the factors that influence the performance of shadow and nested paging. As seen in the previous results, shadow paging performed well at SGX calls and cryptography benchmarks, which are CPU- and memory-heavy benchmarks, but is poor for I/O-heavy applications like HTTP servers. Conversely, nested paging is faster at I/O-heavy applications, but performed slower in memory-heavy benchmarks. Following the aforementioned difference in virtualization mechanisms, we chose two criteria as the indicators of application characteristics: the total number of TLB misses and the number of context switches. While context switches are not the only cause of performance impact caused by shadow paging (unlike TLB misses, which directly slow down nested paging due to 2D page walks), we chose the number of context switches as an indicator for two reasons: firstly, page table updates for a user-mode process involve a context switch; and secondly, context switches can be easily profiled using the performance monitoring systems.

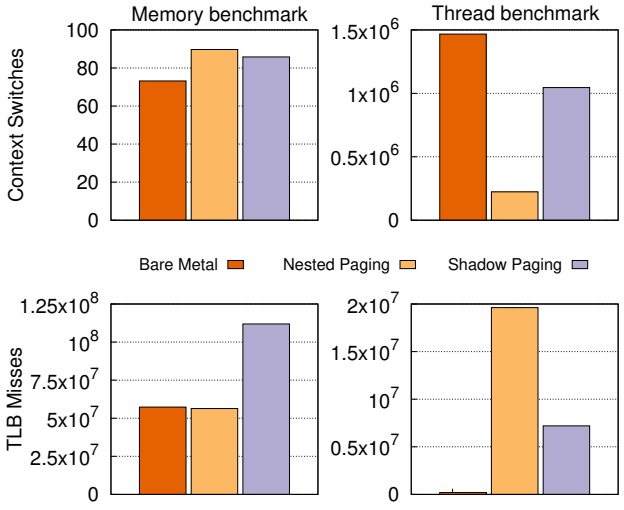

Figure 13: Context switch and TLB miss events per second

To profile the costs of shadow and nested paging, we experimented with two workloads on bare-metal and VMs that represent their respective "worst case" performance. The first workload involves memory operations on random addresses in a large buffer in order to force the occurrence of many TLB misses, which would significantly slow down nested paging. For the second workload, we used the *sysbench* [5] threading benchmark with 20 threads continuously yielding to each other, intended to slow down shadow paging by requiring hypervisor traps for every context switch invoked.

Figure 12 shows the relative benchmark performance of both memory virtualization methods compared to bare-metal. While shadow paging wins out in the memory benchmark, with a slowdown of only 3% compared to nested paging's 17%, it conversely shows a significant 6.5× slowdown in the threads benchmark compared to only 25% for nested paging. In general, these results are consistent with the goals of our benchmarks stated above.

Next, we performed benchmarks to identify the characteristics of these workloads based on the criteria established above. We used the *perf* utility (executed in the host) to measure the number of TLB misses and context switches performed by the benchmarks.

Figure 13 shows the average number of context switches and TLB misses happening every second during the execution of our benchmarks. In the memory benchmark, while we see similar TLB miss numbers for bare-metal and shadow paging, nested paging causes double the number of TLB misses compared to bare-metal, suggesting that nested paging itself is the source of these extra TLB misses in the memory benchmark. Conversely, while the threads benchmark on bare-metal causes only a small number of TLB misses per second, both memory virtualization methods have far higher event counts (37×–100× with nested and shadow paging, respectively). Finally, for the threads benchmark, we see a strong inverse correlation between the performance and number of context switches per second, suggesting that the benchmark is spending most of its time in context switches as intended.

With the observations presented above, we can now establish the baseline performance of shadow and nested paging. We assume that

the runtime contribution of context switches in the memory benchmark is minimal. Let c_0 be the total number of context switches in the threads benchmark, and let t_0, t_e and t_s be its run time on bare-metal, nested paging and shadow paging, respectively.

Similarly, let m_0 be the number of TLB misses in the memory benchmark, and u_0, u_e and u_s its run time. Note that since the benchmark run times u_0, u_e, u_s include the time taken by instructions and memory accesses in general, we run the same memory benchmark on bare-metal with transparent huge pages enabled to establish the run time of the benchmark while minimizing the time spent on TLB misses. We denote this run time u_0'. Therefore, the total time spent on TLB misses can be calculated by subtracting the two:

$$v_0 = u_0 - u_0'$$
$$v_e = u_e - u_0'$$
$$v_s = u_s - u_0'$$

Once the baseline has been established, we measure the workload's characteristics using performance counters to determine whether it is TLB-heavy or context switch-heavy. Assuming that over a period of time, the program generates c context switches and m TLB misses, then the optimal memory virtualization method can be chosen using the following algorithm:

(1) Extrapolate the performance reduction of nested paging:
$$r_e = \frac{c}{c_0} t_e + \frac{m}{m_0} v_e$$
(2) Extrapolate the performance reduction of shadow paging:
$$r_s = \frac{c}{c_0} t_s + \frac{m}{m_0} v_s$$
(3) Calculate the performance ratio of shadow vs. nested paging:
$$k = \frac{r_e}{r_s}$$
(4) Using k, select the preferred memory virtualization method: nested paging if $k \leq 1$, shadow paging if otherwise.

7 OPTIMIZATION EVALUATION

In the previous section, we presented an algorithm for selecting the optimal method of memory virtualization for various applications. In order to validate our algorithm, we applied it on various benchmarks:

- Randomly reading/writing memory-mapped files.
- Randomly reading/writing files using synchronous I/O.
- Randomly reading/writing files using synchronous I/O with periodic `fsync`.
- Random memory reads.
- MySQL OLTP read benchmark with a table containing 10 million rows.
- Nginx web server serving an empty HTTP response.

Aside from the Nginx web server benchmark, all of the above benchmarks were performed using the *sysbench* tool. Our files benchmarks worked on a test file set of 128 files, each 16 MB in size, for a total of 2 GB.

Table 1 shows the results of our algorithms on these benchmarks. The k column shows the predicted ratio of the performance impacts of nested paging vs. shadow paging. In short, a ratio of k means that when considering the combined time taken by context switches and TLB misses, shadow paging will be k times faster than nested paging. As a result, $k > 1$ denotes a TLB-heavy workload where shadow paging is preferred and vice versa for $k \leq 1$, as denoted in the "Method" column.

Benchmark	k	**Method**
File read/write, `mmap` (2 threads)	1.116	Shadow
File read/write (2 threads)	0.628	Nested
File read/write (2 threads, periodic `fsync`)	0.220	Nested
Random memory reads	1.138	Shadow
MySQL OLTP read benchmark	0.509	Nested
Nginx web server, empty responses	0.350	Nested

Table 1: Evaluation results of our optimization on various workloads

Our results predicted that nested paging is preferable to shadow paging except for two benchmarks: file read/write with `mmap` and random memory reads. However, since the predicted k ratio for these benchmarks are close to 1 (1.116 and 1.138, respectively), our algorithm only predicts a small performance improvement for these TLB-heavy benchmarks. In contrast, the remaining benchmarks show a bigger advantage for nested paging. For example, the Nginx benchmark predicts a k value of 0.350, or a slowdown of 2.9× for shadow paging. These predictions largely agree with the results of our benchmarks presented above: workloads involving many TLB misses tend to favor shadow paging, while workloads that are IO-bound or with several context switches (*e.g.*, Nginx) favor nested paging.

8 RELATED WORK

We note that there are no previous works that investigate the performance of SGX on virtualized systems. Having said that, we classify the related work on Intel SGX into four categories as detailed below.

Scrutinizing SGX security. Costan and Devadas [16] present in great depth the Intel SGX architecture. The authors gave an overview its design and features, and compared them to that of previous trusted computing platforms. [16] also includes a security analysis of SGX in relation to physical attackers, privileged software, memory mapping, peripherals, cache timing and side channels [35]. The authors demonstrated several SGX-related security properties: *(i)* TLB checks done by SGX are consistent with its security guarantees; *(ii)* EPCM entries of allocated pages follow the enclave's requirements; *(iii)* during enclave execution, EPCM and TLB contents belong to enclave-exclusive pages are consistent; and *(iv)* a page's EPCM entry is only modified when the TLB mapping corresponding to that page is not present. Finally, [16] proposed a method for tracking TLB flush during page eviction, noting that the actual used process is not described in the Intel SDM [4]. Evaluations of SGX's design have also lead to new attacks. AsyncShock [43] manipulates the scheduling of threads used to execute enclave code to exploit synchronization bugs of multi-threaded code. More recently, SgxPectre [14] exploits known limits of speculative execution of x86 architectures to leak secrets from the enclaves.

Using SGX for security-critical systems. We put into this category SCONE [7], a framework to run containerized applications completely inside SGX enclaves. This framework opens the possibility to securely deploy container-based systems, which are increasingly common in cloud-based applications. SecureKeeper [11] runs the ZooKeeper coordination service inside the shielded boundaries of the Intel SGX environment, hence supporting the distribution of sensitive information. In case of distributed systems with privacy concerns related to the network traffic, one can rely on TaLoS [8] to establish TLS endpoints directly inside the enclaves.

Performance analysis. HotCalls [44] observed that `ecall`s and `ocall`s as provided by the SGX SDK have a high overhead, ranging from approximately 8,000 to 17,000 cycles depending on cache state. The authors proposed a new asynchronous design using a shared buffer which reduces this overhead down to 620–1,400 cycles. Similar to our work, the authors identified three potential sources of overhead caused by SGX: *(i)* switching in and out of the enclave, *(ii)* passing buffers between application and enclave, and *(iii)* overhead of reading and writing encrypted memory. A recent upgrade to the Linux SGX SDK introduces a similar feature [3].

Proposals for improvements to SGX. Finally [46] and [12] present the future features that Intel will integrate into the next SGX revisions, namely dynamic memory allocation (in SGX v2) and oversubscription, respectively.

9 CONCLUSION

This paper presented for the first time a detailed performance analysis of Intel SGX on virtualized systems (in our case, KVM), the latter being the building block of cloud platforms. Knowing that cloud computing is one of the main SGX use case and that virtualization comes with an overhead, it is useful to have the performance analysis of SGX in VMs. To this end, we base our performance on an extensive set of micro- and macro-benchmarks.

We built a set of micro-benchmarks to carefully capture the impact of virtualization (and its different implementations) for every SGX function (`ecall`, `ocall`, transferring buffers into/out/in&out, read/write to (un)encrypted memory in/outside enclave, enclave creation/destruction) and feature (swapping), comparing the performance achieved on a bare-metal system.

Our macro-benchmarks (encryption benchmark and a HTTP server benchmark) capture functions which are commonly used in enclaves. All the benchmarks are made available as open-source at [1] to foster experimental reproducibility and to allow researchers to exploit our outcomes.

We provided a comprehensive explanation for every obtained results, and we summarized our main findings and lessons learned. Based on these findings, we identified several optimization strategies that would improve the performance of Intel SGX on virtualized systems. We implement and evaluate one of these optimizations, while the other optimizations are subject of our future work. We believe this study to bring valuable insights for developers and users of SGX, as well as for designers of future TEE micro-architectures.

REFERENCES

[1] [n. d.]. SGX benchmark source code. https://github.com/sgxbench/sgxbench/releases.
[2] 2011. The RSA Hack: How They Did It. https://bits.blogs.nytimes.com/2011/04/02/the-rsa-hack-how-they-did-it/.
[3] 2018. Intel Linux SGX SDK v2.2 — Switchless Calls. https://download.01.org/intel-sgx/linux-2.2/docs/Intel_SGX_Developer_Reference_Linux_2.2_Open_Source.pdf.
[4] 2018. Intel Software Development Manual. https://software.intel.com/en-us/articles/intel-sdm.
[5] 2018. SysBench. https://github.com/akopytov/sysbench.
[6] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. 2013. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd international workshop on hardware and architectural support for security and privacy*, Vol. 13. ACM New York, NY, USA.
[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark Stillwell, et al. 2016. SCONE: Secure Linux Containers with Intel SGX.. In *OSDI*, Vol. 16. 689–703.
[8] Pierre-Louis Aublin, Florian Kelbert, Dan OâĂŹKeeffe, Divya Muthukumaran, Christian Priebe, Joshua Lind, Robert Krahn, Christof Fetzer, David Eyers, and Peter Pietzuch. 2017. *TaLoS: Secure and transparent TLS termination inside SGX enclaves.* Technical Report. Imperial College London.
[9] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. 2008. Accelerating two-dimensional page walks for virtualized systems. In *ACM SIGARCH Computer Architecture News*, Vol. 36. ACM, 26–35.
[10] Nikhil Bhatia. 2009. Performance evaluation of Intel EPT hardware assist. *VMware, Inc* (2009).
[11] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference*. ACM, 14.
[12] Somnath Chakrabarti, Rebekah Leslie-Hurd, Mona Vij, Frank McKeen, Carlos Rozas, Dror Caspi, Ilya Alexandrovich, and Ittai Anati. 2017. Intel Software Guard Extensions (Intel SGX) Architecture for Oversubscription of Secure Memory in a Virtualized Environment. In *Proceedings of the Hardware and Architectural Support for Security and Privacy*. ACM, 7.
[13] Chia che Tsai, Donald E. Porter, and Mona Vij. 2017. Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*. USENIX Association, Santa Clara, CA, 645–658. https://www.usenix.org/conference/atc17/technical-sessions/presentation/tsai
[14] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. 2018. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. *arXiv preprint arXiv:1802.09085* (2018).
[15] Sean Christopherson. 2017. KVM: vmx: add support for SGX Launch Control. https://github.com/intel/kvm-sgx/commit/e9a065d3c1773ad72bfb28b6dad4c433f392eda8.
[16] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptology ePrint Archive* 2016 (2016), 86.
[17] Edward W Felten. 2003. Understanding trusted computing: will its benefits outweigh its drawbacks? *IEEE Security & Privacy* 99, 3 (2003), 60–62.
[18] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile paging: exceeding the best of nested and shadow paging. In *ACM SIGARCH Computer Architecture News*, Vol. 44. IEEE Press, 707–718.
[19] Google. 2018. Asylo: an open-source framework for confidential computing. https://cloudplatform.googleblog.com/2018/05/Introducing-Asylo-an-open-source-framework-for-confidential-computing.html.
[20] David Grawrock. 2009. *Dynamics of a Trusted Platform: A building block approach.* Intel Press.
[21] Trusted Computing Group. 2007. Design Principles Specification Version 1.2 Level 2 Revision 103 Part 1.
[22] Shay Gueron. 2016. Memory Encryption for General-Purpose Processors. *IEEE Security & Privacy* 6 (2016), 54–62.
[23] Danny Harnik and Eliad Tsfadia. 2017. Impressions of Intel SGX performance. https://medium.com/@danny_harnik/22442093595a.
[24] IBM. 2018. Data-in-use protection on IBM Cloud using Intel SGX. https://www.ibm.com/blogs/bluemix/2018/05/data-use-protection-ibm-cloud-using-intel-sgx/.
[25] Intel Corporation. [n. d.]. Intel Software Guard Extensions SDK. https://software.intel.com/en-us/sgx-sdk.
[26] Intel Corporation. [n. d.]. Intel Software Guard Extensions SDK for Linux. https://01.org/intel-software-guard-extensions.
[27] Intel Corporation. [n. d.]. SGX Virtualization. https://01.org/intel-software-guard-extensions/sgx-virtualization.
[28] Intel Corporation. 2017. Intel and NeuLion Bring Secure, 4K UHD Sports Streaming to Computers. https://newsroom.intel.com/news/intel-neulion-bring-secure-4k-uhd-sports-streaming-computers/.
[29] Intel Corporation. 2017. *Intel Software Guard Extensions SDK for Linux OS.* https://download.01.org/intel-sgx/linux-2.0/docs/Intel_SGX_Installation_Guide_Linux_2.0_Open_Source.pdf

[30] International Organization for Standardization. 2015. ISO/IEC 11889-1:2015.

[31] David Kaplan, Jeremy Powell, and Tom Woller. 2016. AMD memory encryption. *White paper* (2016).

[32] Klaus Kursawe, Dries Schellekens, and Bart Preneel. 2005. Analyzing trusted platform communication. In *ECRYPT Workshop, CRASH-CRyptographic Advances in Secure Hardware*.

[33] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. 2017. Performance Overhead Comparison between Hypervisor and Container based Virtualization. In *Advanced Information Networking and Applications (AINA), 2017 IEEE 31st International Conference on*. IEEE, 955–962.

[34] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer, and Peter Pietzuch. 2017. Glamdring: Automatic Application Partitioning for Intel SGX. In *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Berkeley, CA, USA, 285–298. http://dl.acm.org/citation.cfm?id=3154690.3154718

[35] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[36] Ryan Puffer and Liza Poggemeyer. 2016. Guarded fabric and shielded VMs overview. https://docs.microsoft.com/en-us/windows-server/virtualization/guarded-fabric-shielded-vm/guarded-fabric-and-shielded-vms.

[37] Marco Righini. 2010. Enabling Intel virtualization technology features and benefits. *Intel White Paper. Retrieved January* 15 (2010), 2012.

[38] Efraim Rotem and Senior Principal Engineer. 2015. Intel Architecture, Code Name Skylake Deep Dive: A New Architecture to Manage Power Performance and Energy Efficiency. In *Intel Developer Forum*.

[39] Mark Russinovich. 2018. Azure confidential computing. https://azure.microsoft.com/en-us/blog/azure-confidential-computing/.

[40] Samsung Electronics Co., Ltd. 2017. Samsung Knox Security Solution. https://www.samsungknox.com/docs/SamsungKnoxSecuritySolution.pdf.

[41] Evan R Sparks and Evan R Sparks. 2007. A security assessment of Trusted Platform Modules - computer science technical report TR2007-597. (2007).

[42] Gil Tene. 2018. WRK2 Http Benchmarking Took. https://github.com/giltene/wrk2.

[43] Nico Weichbrodt, Anil Kurmus, Peter Pietzuch, and Rüdiger Kapitza. 2016. AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves. In *European Symposium on Research in Computer Security*. Springer, 440–457.

[44] Ofir Weisse, Valeria Bertacco, and Todd Austin. 2017. Regaining Lost Cycles with HotCalls: A Fast Interface for SGX Secure Enclaves. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. ACM, 81–93.

[45] Wired. 2009. Google Hack Attack Was Ultra Sophisticated. https://www.wired.com/2010/01/operation-aurora/.

[46] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 2016. Intel Software Guard Extensions (Intel SGX) Software Support for Dynamic Memory Allocation inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 11.