Seeds of SEED: Characterizing Enclave-level Parallelism in Secure Multicore Processors

Brandon D'Agostino and Omer Khan
{*bdag, khan*}@*uconn.edu*

University of Connecticut, Storrs, CT USA

***Abstract*—Secure processor technologies incorporating some form of enclave-based isolation are being deployed in remote cloud computing environments. However, commercial enclave-based systems, such as Intel SGX, incur performance penalties due to architectural limitations arising from enclave interactions with the operating system (OS), encryption and attestation checks for data accesses to main memory, and limitations on the enclave memory size. Enclave software development frameworks like Graphene-SGX aim to improve these limitations with performance enhancements such as exitless calling that offset the latency of expensive enclave interactions with the OS. However, to the best of our knowledge, prior works have not presented a thorough characterization of enclave performance in the presence of increased enclave-level parallelism. In this paper, we characterize how enclave overheads trade off exploitable parallelism on an Intel SGX-enabled multicore CPU for a set of parallelized workloads. We develop a microbenchmark to study the effects of threading as a function of application characteristics, such as the intensity of memory operations and system calls to the OS. We extend our characterization to realistic parallelized enclave workloads from the database and web server domains. We find that application performance scaling with threading is tightly correlated to system call and memory-bound activities in applications. The real world applications stress these constraints, while the underlying system calling implementations deliver competing performance at different thread counts.**

I. Introduction

The proliferation of cloud computing services has provided a compelling alternative to costly on-site computing [1]. However, cloud computing gives rise to the problem of securely executing software on remote computing resources owned and operated by an untrusted third-party [2]. To combat this challenge, processors with enhanced security features are being increasingly deployed in cloud computing environments. AMD Secure Encrypted Virtualization (SEV) [3] provides virtual machine memory encryption that protects against co-resident virtual machines, compromised system software, and adversaries with physical memory access. ARM TrustZone [4] allows trusted applications and system software to securely operate alongside untrusted applications and system software by providing hardware-enforced parallel execution environments called worlds. Intel Software Guard Extensions (SGX) [2] allows application developers to create secure application environments called *enclaves* that reside in a region of encrypted memory, are protected from privileged system software, and can be remotely attested by trusted remote clients. SGX enclaves require the smallest Trusted Code Base (TCB) compared to AMD SEV and ARM TrustZone, which encompass an entire virtual machine instance and secure OS respectively. Moreover, SGX provides additional security features such as remote attestation and integrity verification for enclave applications. Consequently, the enclave abstraction popularized by SGX has become a staple of secure processor technologies [5], and thus processors incorporating some form of enclave-based execution have remained prevalent in both academia [6]–[8] and industry [9].

Enclave-based systems improve application security but introduce performance challenges due to overheads intrinsic to their architectural implementation. For example, SGX and similar systems incur overheads from (i) page swapping due to Enclave Page Cache (EPC) size limitations [10], [11], (ii) memory encryption, decryption, and integrity checks on each enclave memory access [12], [13], and (iii) context switching between secure enclave and insecure software due to state purging and security checks [11]–[13]. Prior works have characterized these overheads, and researchers have also developed open-source frameworks (such as Graphene-SGX) to efficiently port applications to SGX [14]. Hasan et al. [13] show that executing applications under Graphene-SGX results in performance comparable to a manual port. Overheads due to memory encryption and EPC size are hardware limitations that are addressed at the application level by writing enclave software that avoids expensive memory accesses [12], [13] and page swapping [10], [11]. HotCalls [12] and Eleos [11] address the context switching limitations and introduce a switchless, or *exitless*, system calling mechanism that uses an asynchronous requester-responder calling mechanism to avoid expensive enclave enters and exits while upholding SGX's isolation guarantees between the untrusted OS and secure enclave. Consequently, exitless calling has been added as an optional feature to the official Intel SGX SDK [15] and the popular Graphene-SGX [14] library OS. However, exitless calling requires additional responder threads and a polling mechanism, which introduce its own set of challenges.

Notably absent from the literature is a dedicated study of the effects of parallelism on enclave performance in representative parallel workloads. This is surprising given the highly parallel nature of the processors and applications typically deployed in cloud computing environments where both security and parallel performance is critical. Thus, the objective of this paper is to characterize how well-known enclave overheads behave in the presence of increased enclave parallelism. Particular emphasis is made to study the tradeoffs between synchronous exit-based and asynchronous exitless calling, as well as the effects of memory operation intensity on exploitable parallelism. The goal of this paper is to identify both performance scaling opportunities and limitations on parallel secure processors.

We develop a microbenchmark to systematically study known enclave overheads as a function of increased threading. We evaluate this microbenchmark with exit-based and exitless Graphene-SGX for compute and memory-bound configurations while varying the intensity of system call activity on a real Intel SGX-capable machine with hyper-threaded cores. The characterization reveals that enclave parallelism is limited by memory encryption and system call processing overheads. Exit-based calling involves less threads than exitless, thus scales to relatively higher thread counts. These insights extend to realistic parallelized workloads from the in-memory database *Memcached*, and two web servers, *Apache* and *Nginx*.

A key finding reveals that both exit-based and exitless setups are capable of achieving similar performance at their optimal thread counts. Finally, we conclude by seeding the development of methods that make optimal configuration decisions for parallel enclave workloads in existing and emerging secure processors as a promising area of future research.

II. Background

A. Intel SGX

This section describes the original implementation of SGX known as SGX1. Recently, some Intel processors have included SGX2, an incremental improvement over SGX1 that enables dynamic memory management within enclaves. As SGX2 is largely unchanged from SGX1, we do not further delineate between the two. In addition, no official list of hardware supporting SGX2 is available [16] as of this writing, which makes acquiring such hardware troublesome.

Starting with the sixth generation "Skylake" microarchitecture, several processor offerings from Intel have included Software Guard Extensions (SGX), a set of security-related extensions to the x86 instruction set that allows application developers to create secure execution environments—called enclaves—that protect security-critical application components from compromised or malicious system software (such as the OS, hypervisor, etc.). A single application may create several enclaves, each of which execute temporally on the system with user-level privileges in the virtual address space of the calling application.

Enclave code and data pages are stored in the Enclave Page Cache (EPC), a region of main memory located in Processor Reserved Memory (PRM) that is inaccessible by any software regardless of privilege level. The EPC size is limited by the size of the PRM, which is 128 MB on the latest SGX-enabled systems. If an enclave's size exceeds that of the EPC, its pages are securely swapped out to main memory by the SGX driver. As this process invokes the OS and involves additional hardware security checks, EPC page swapping is expensive [10], [11]. The EPC is encrypted and integrity checked by the on-chip Memory Encryption Engine (MEE) [17], and is only decrypted when brought into the CPU core. In other words, enclave code and data never leave the processor chip unencrypted. Consequently, memory loads and stores between the last-level cache (LLC) and EPC incur a 20–30% performance overhead over non-enclave memory operations [12].

An enclave is initialized by using special CPU instructions that set up enclave control structures and copy application code and data pages into the EPC. After initialization, the enclave is sealed so that no new memory pages can be added, and trusted CPU hardware generates a signed measurement hash that is used by a trusted remote client to authenticate the enclave's contents.

Special CPU instructions are also used to control context switches between the secure enclave and insecure OS, as explained in [2]. Context switches between the enclave and OS occur (i) at the beginning and end of enclave execution, (ii) during hardware interrupts, and (iii) when the enclave requires functionality outside of the enclave, as with external library or operating system calls. In the latter case, an enclave may call into the insecure environment by using synchronous, exit-based calling or asynchronous, exitless calling. Both exit-based and exitless calling are supported by the official Intel SGX SDK [15] as well as some other SGX application frameworks such as Graphene-SGX [14].

1) Exit-based (Synchronous) Calling: The EENTER instruction is used by untrusted application components to perform an *enclave call* (ecall) that makes a secure context switch into trusted enclave code. Conversely, the EEXIT instruction is used within an enclave to make an *outside call* (ocall), or to return control back to the untrusted application. Ecalls make several hardware security checks, flush the TLBs, and switch the processor core into enclave mode before transferring control to a predefined entry point within the enclave. Similarly, ocalls save the current enclave state, flush the core pipeline and TLBs, and switch the processor core out of enclave mode before returning control to the untrusted application. Consequently, ecalls/ocalls incur additional performance overheads, and have been shown to be 83–113x slower than typical OS system calls [12]. Although exit-based calling is expensive, it is necessary to guarantee isolation between secure enclaves and the untrusted OS during synchronous execution.

2) Exitless (Asynchronous) Calling: Prior works [11], [12] have proposed an alternative *exitless* calling model that mitigates the ecall/ocall overheads in the exit-based model without violating isolation guarantees between the secure enclave and OS. This is accomplished by introducing an asynchronous requester-responder calling mechanism between secure enclave requester threads and untrusted OS responder threads that communicate over a shared unencrypted memory buffer. Because these threads do not explicitly invoke each other, the enclave and OS threads are decoupled, and the enclave's isolation guarantees are upheld.

As described in [12], an enclave requester thread makes a system call request to a responder thread outside the enclave over a shared unencrypted memory buffer that is synchronized using a spinlock. The requester loads system call information into the shared buffer and sets a flag that indicates a pending request. After the requester issues a request, it continuously polls the shared buffer to check if the request has been processed, and if so, copies the results into the enclave and continues execution. Asynchronously, a responder thread continuously polls the shared buffer for pending requests and, once received, processes them by invoking the OS. The responder then copies the result of the system call into the shared buffer and sets a flag indicating the request has been processed. In the case of multiple enclave threads, a thread pool containing a responder thread for each enclave requester thread is utilized [11].

Exitless calling avoids the standard exit-based calling overheads at the expense of requiring additional responder threads to service enclave call requests. Tian et al. [15] show that exitless calls only improve performance over standard exit-based calling if the requested function is short and called frequently.

B. SGX Application Frameworks

Enclave-based systems present a developmental challenge in how to best adapt existing applications to take advantage of enclave security. The traditional approach of doing this on Intel SGX systems is by manually refactoring an application into insecure and secure enclave components. However, SGX levies additional restrictions on enclave code for security reasons, such as disallowing system calls and dynamic memory allocation inside of an enclave. Consequently, prior works [13], [14], [18] have argued that modifying any non-trivial application to circumnavigate the restrictions imposed by SGX requires a significant amount of development time and effort.

An alternative approach that involves much less development effort is that of using an application framework, such as SCONE [10], Graphene-SGX [14], or Occlum [19], that allow executing applications entirely within an enclave with very little, if any, modifications to the application. SCONE provides secure in-enclave Docker containers and a standard C library implementation that can be linked against containerized applications to transparently handle system calls. Other frameworks, such as Graphene-SGX and Occlum, bring a library OS into the enclave so that most system calls can be handled without incurring expensive ecalls/ocalls. In situations where an application cannot avoid interacting with the host OS (such as with file or network IO), the library OS transparently invokes the ocall to the OS on behalf of the calling application. Library OS-based application frameworks do not require modification or recompilation of the target application, and thus require less porting effort. Despite this, Hasan et al. [13] show that executing applications under a library OS (Graphene-SGX) results in performance comparable to a manual port.

Graphene-SGX and Occlum are two popular library OS application frameworks, with the former being more mature and supporting exitless calling as well as more application functionality. Therefore, we pick Graphene-SGX as our baseline framework for evaluation in this paper. Moreover, Graphene-SGX provides an improved implementation of exitless calling that utilizes a scheduling algorithm that quiesces responder threads if they have not received requests for some time. This reduces the overhead of additional responder threads at the expense of increased response latency when a request arrives while a responder thread is sleeping.

III. Effects of Parallelism on Enclave Performance

The objective of this paper is to characterize the effects of parallelism on enclave performance in representative parallel workloads, and to identify performance scaling opportunities and limitations on parallel secure processors. To recap, the overheads in current enclave-based systems such as Intel SGX are mainly due to (i) EPC size limitations, (ii) memory encryption, decryption, and integrity checks on enclave memory accesses, and (iii) ecalls/ocalls from system call requests originating from within the enclave. How then are these overheads expected to behave when enclave parallelism is increased?

In the case of paging overheads related to EPC size, the total memory size of a given workload remains relatively unchanged with additional enclave threads. However, additional enclave threads imply more concurrent memory accesses that stress the LLC whose performance is impacted by the centralized MEE engine. On one hand, more concurrent long latency memory accesses provide more opportunities to hide long latency stalls with hardware-level parallelism. On the other hand, single thread performance is reduced due to more long latency stalls per thread.

With exit-based calling, enclave threads interact with the untrusted OS using the explicit exit-based ecall/ocall mechanism. Increasing the number of threads in this case results in performance scaling similar to that of non-SGX applications, although thread-level performance is limited by serialization effects of expensive secure context switches between the enclave threads and the OS. However, performance scaling with exitless calling is less straight forward to reason about. Exitless calling provides better single-threaded performance by eschewing the explicit ecall/ocall overheads, but additional responder threads limit the performance scaling available to enclave application threads. Furthermore, in cases where responder threads have gone to sleep due to insufficient requests, periodically waking up these threads incurs additional thread invocation overheads as well as longer request response latencies. Therefore, it is helpful to separately consider exitless calling for compute and memory-bound workloads. In compute-bound workloads, enclave threads have a higher probability of requesting a system call in any given time interval. Consequently, the responder threads are constantly saturated with work and thus never go to sleep. Moreover, responder threads incur fewer polling overheads because state information corresponding to the exitless implementation (such as the request queue and synchronization variables) has higher cache locality. In memory-bound workloads that stress the LLC, enclave threads have a lower probability of requesting a system call in any given time interval due to higher latency memory accesses. Consequently, responder threads have a higher probability of going to sleep because of insufficient requests, which results in additional thread invocation overheads and longer request response latencies. Moreover, because the LLC is saturated by the workload, exitless state information must be reloaded on every responder thread poll which incurs additional overhead.

The following sections outline the methodology to quantify the aforementioned tradeoffs in representative parallel applications, and present the characterization on a real SGX-enabled hyper-threaded Intel machine.

IV. Evaluation Methodology

The evaluation is performed on an Intel Core i7-6700K (4 physical cores, 8 logical cores at 4 GHz) machine with 24 GB, 2133 MHz DDR4 memory. The CPU provides an 8 MB shared last-level cache (LLC), and SGX1 support with 128 MB PRM. The system uses Ubuntu 18.04 with kernel version 5.4.111, and SGX driver version 2.11. The exact version of Graphene-SGX used can be found at [20].

All evaluations are performed using Graphene without SGX [21], Graphene-SGX with exit-based calling, and Graphene-SGX with exitless calling. SGX applications must pre-declare the maximum number of application threads, so every application is allowed to spawn up to 32 threads. For exitless calling configurations, the responder thread pool size is set to the maximum number of enclave application threads to ensure an optimal one-to-one mapping between enclave (requester) and responder threads.

A. Parallelized Enclave Microbenchmark

To systemically study parallel performance as a function of enclave threading, we develop a microbenchmark that measures the throughput and latency of a memory copy operation for a configurable workload size, number of worker threads, number of copy iterations, and level of system call activity. By adjusting the size of the workload, we vary the amount of stress on the LLC. The memory copy operation is split evenly between a configurable number of worker threads so that N worker threads copy a non-overlapping M/N section of the workload buffer per thread, where M is the configured workload size. System call activity is controlled such that each worker thread makes A system calls every B copy iterations, where A and B are configurable parameters. Both total and per-thread throughput and latency are measured as the performance metrics.

We first evaluate the microbenchmark for various workload sizes with a single thread and no system calling activity to determine compute-bound and memory-bound workload sizes. We then evaluate a 2 MB compute-bound configuration that fits entirely within the LLC, and an 8 MB memory-bound configuration that stresses the limits of the machine's 8 MB LLC. For each configuration, we evaluate both light system calling activity where no system calls are made during the benchmark, and heavy system calling activity where 2 system calls are made after every copy iteration. For each configuration, we evaluate 1 to 8 worker threads that make 10,000 copy iterations each. We also measure LLC Misses Per Kilo Instruction (MPKI) for each configuration to quantify the amount of stress on the LLC.

B. In-Memory Database Workload

Memcached is a multi-threaded in-memory key-value store that is widely deployed as a caching layer between web servers and databases. A comprehensive study of Memcached workloads can be found in [22]. Memcached performance is measured by the number of server operations per second. Memcached is evaluated using the *memtier_benchmark* tool developed by Redis Labs [23]. It uses the

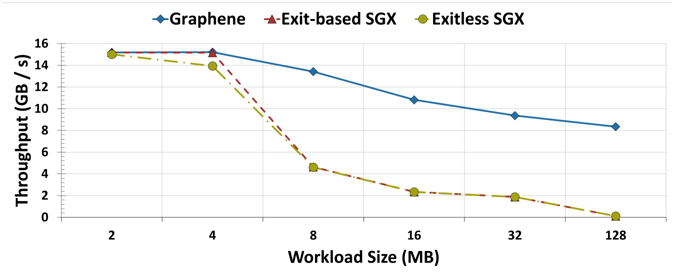


Fig. 1: Single-threaded microbenchmark performance for various workload sizes. Workloads smaller than the 8 MB LLC are compute-bound whereas larger workloads are memory-bound. Memory-bound SGX performance suffers due to memory encryption and decryption overheads.

same benchmark configuration as described in [12] with the exception that we use 4 concurrent memtier_benchmark client threads and vary the number of Memcached server threads. In summary, a total of 4,000,000 2 KB requests from 4 concurrent clients are issued. The MPKI of the Memcached server threads and the total number of ecalls/ocalls for SGX configurations are also measuured.

C. *Web Server Workloads*

Apache [24] and Nginx [25] are both high-performance web servers, and are therefore evaluated in the same manner. Web server performance is measured by the number of requests served per second and the latency of each request as reported by *Apache Benchmark* [26]. The number of web server threads are varied while the number of Apache Benchmark client threads are fixed. Apache and Nginx serve 2 and 4 clients respectively, which result in the best overall performance for each application. In contrast to Memcached, a single client thread cannot dispatch requests concurrently, and a single web server thread can only serve one request at a time. The benchmark is configured to make a total of 10,000 requests for a 10 KB webpage filled with random static data. Like the Memcached evaluation, we also measure MPKI of the server threads and the total number of ecalls/ocalls for SGX configurations.

V. Evaluation

A. *Microbenchmark*

1) Workload Size Sweep: We first evaluate a single thread with no system calling activity for various workload sizes to determine which workloads are compute-bound and which are memory-bound. Figure 1 depicts a significant performance drop under all configurations for workloads greater than or equal to the machine's 8 MB LLC. Consequently, 2 MB and 8 MB are used respectively as the compute-bound and memory bound workload sizes. Memory-bound workloads under both SGX configurations experience further performance degradation due to SGX's memory encryption/decryption overheads. As workloads larger than the 128 MB EPC perform even worse due to excessive page swapping, we do not consider any workloads of this size for the remaining evaluations.

2) Compute-bound Workload: The 2 MB compute-bound workload fits entirely within the 8 MB LLC, and a negligible MPKI of less than 0.002 is observed for all thread counts. As the workload fits entirely within the 128 MB EPC, no page swapping performance limitations are observed.

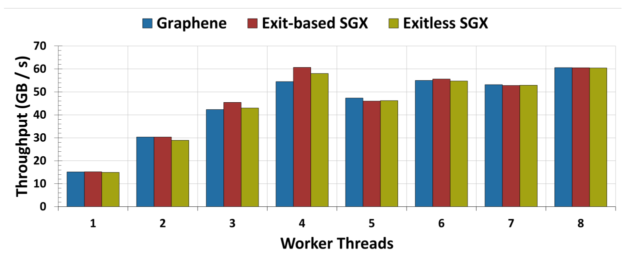
In the light system calling configuration (Figure 2a), consistent performance across all environments is observed for all threading configurations. In this case, main memory accesses are virtually non-existent under all environments, so neither the exit-based nor exitless SGX configurations suffer from memory encryption/decryption overheads. No system calls are made, so exit-based SGX threads do not incur expensive ecall/ocall costs, which allows exit-based SGX performance to match that of the non-SGX baseline. The responder threads in exitless SGX quickly go to sleep due to a lack of system call activity, which avoids the additional threading and polling overheads incurred by exitless calling. Consequently, exitless SGX performance is also able to match the non-SGX baseline. The expected performance scaling under all environments is observed up to 4 worker threads. Beyond 4 threads, performance is mostly unchanged due to limited latency hiding opportunities for the hyper-threaded cores under the compute-bound configuration.

In the heavy system calling configuration (Figure 2b), memory accesses remain unchanged from the light system calling configuration. However, exit-based SGX performance is significantly worse than the baseline in all threading configurations because the high number of system calls incur expensive ecall/ocall overheads. Moreover, the performance gap between exit-based SGX and the baseline widens with increasing thread count because the number of system calls increases with the number of threads. Despite this, the baseline and exit-based SGX observe the same thread scaling trend up to 4 threads as observed in the light system calling configuration. Exitless SGX is able to completely overcome the ecall/ocall overheads and match the performance of the baseline up to 3 enclave threads. This is because the available hyper-threaded cores are able to sufficiently handle the additional exitless responder threading and polling overheads. Beyond 3 enclave threads, the combined total of requester and responder threads encroaches upon the threading capability of the machine. Consequently, the additional exitless threading overheads cannot be mitigated by the hyper-threaded cores, and the performance gap between exitless SGX and the baseline widens. Overall performance scaling past 4 enclave threads in exit-based, and past 3 for exitless, is mostly unchanged from that observed in the light system calling configuration.

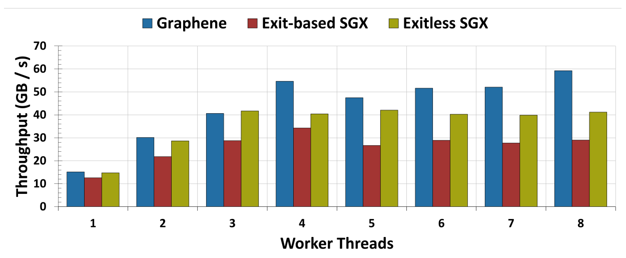
3) Memory-bound Workload: The 8 MB memory-bound workload stresses the 8 MB LLC, and MPKI measurements up to 0.443 are observed.

In the light system calling configuration (Figure 2c), both exit-based and exitless SGX perform significantly worse compared to the baseline due to longer latency cache misses under SGX because of memory encryption/decryption and security checking overheads. Otherwise, exit-based and exitless SGX achieve similar performance trends for the same reasons described in the equivalent 2 MB configuration. Interestingly, all environments observe performance scaling up to 6 enclave threads. This is primarily because higher latency memory accesses due to more cache misses across all environments provide opportunities for latency hiding by the hyper-threaded cores. Beyond these, no new insights from the equivalent 2 MB configuration are observed.

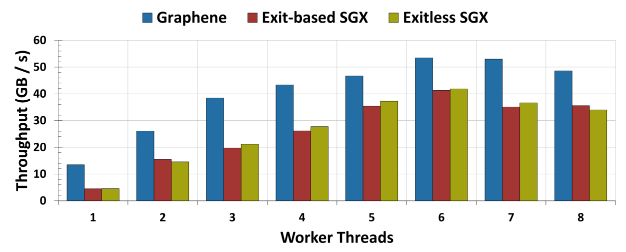
In the heavy system calling configuration (Figure 2d), both exit-based and exitless SGX performance suffers due to the previously described effects of increased LLC misses. However, exit-based SGX performance degrades further due to the increased ecall/ocall overheads. Unlike the 2 MB workload, the exitless SGX performance is no longer able to match the baseline. This is because the longer memory access latencies imply a higher probability that responder threads will fall asleep more often, and thus incur additional thread invocation costs and increased response latencies. This is especially evident with 1–3 threads where these additional costs bring exitless



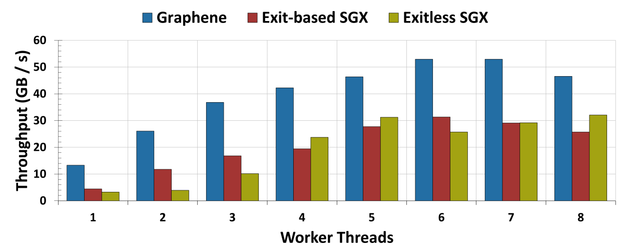
(a) Compute-bound workload with light system calling activity.



(b) Compute-bound workload with heavy system calling activity.



(c) Memory-bound workload with light system calling activity.



(d) Memory-bound workload with heavy system calling activity.

Fig. 2: Microbenchmark memory throughput performance for compute and memory-bound workloads. The number of worker threads and level of system calling is varied for each workload.

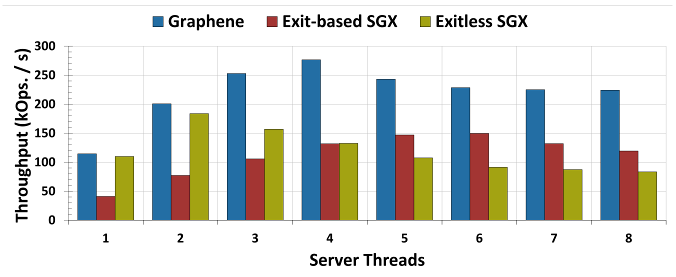


Fig. 3: Memcached performance measured in operations per second for a variable number of server threads. 4 clients are served concurrently.

performance below that of exit-based SGX. Moreover, because the LLC is stressed, exitless thread state information is not always cached and needs to be fetched on each responder thread poll. This explains why exit-based and exitless SGX performance are mostly equivalent even when responder threads are sufficiently saturated with requests.

B. Memcached

We evaluate Memcached for a variable number of server threads that serve 4 concurrent clients (Figure 3). For all configurations, the number of ecalls/ocalls measured for exit-based SGX is around 10,000,000, indicating heavy system calling activity. MPKI never exceeds 0.115, indicating a workload that is lightly memory-bound. As the workload fits entirely within the EPC, page swapping exerts no influence on performance.

At 1 server thread, exit-based SGX performance is significantly worse compared to the baseline due to high ecall/ocall overheads. On the other hand, exitless SGX is able to completely overcome these overheads and match baseline performance. Moreover, responder threads are sufficiently saturated and do not incur a performance overhead. This aligns with the compute-bound, heavy system call activity microbenchmark evaluation.

At 2 server threads, exit-based SGX performance scales in line with our compute-bound, heavy system call activity microbenchmark results, but otherwise still performs significantly worse than the baseline due to high ecall/ocall overheads. Exitless SGX continues to mitigate ecall/ocall overheads and performs much better than exit-based SGX, but is now unable to match baseline performance. This is because the total number of threads in this configuration (8) encroaches upon the total core count of the machine and the hyper-threaded cores cannot completely overcome the additional exitless threading and polling overheads.

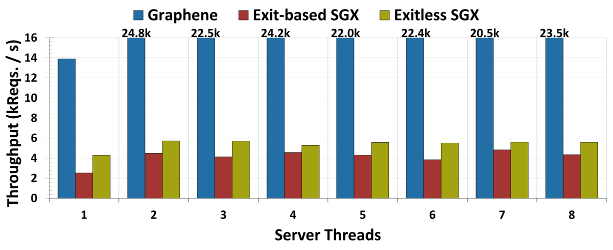
At 3 server threads, exit-based SGX performance continues to scale, but exitless SGX performance is now significantly worse than the baseline and comparable to exit-based SGX. This is because the total number of threads (10) exceed the total number of hyper-threaded processor cores and thus the hardware is overwhelmed by the additional threading and polling overheads.

At 4 server threads, exit-based SGX performance continues to scale as before, but exitless SGX performance degrades to the point that it matches exit-based SGX.

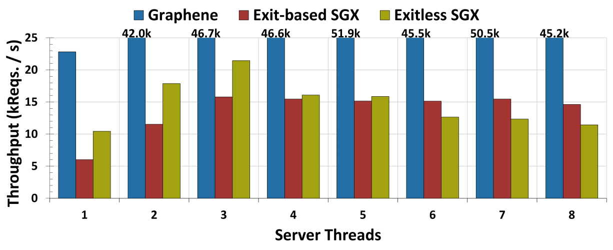
At 5 server theads, exit-based SGX continues to scale and reaches peak performance whereas exitless SGX performance continues to degrade. Crucially, the performance of exit-based SGX is now comparable to the peak performance of exitless SGX observed at 2 server threads. This is a key insight: under the right threading configuration, both exit-based and exitless SGX configurations achieve similar performance for the Memcached workload.

Beyond 5 threads, performance of both exit-based and exitless SGX continues to degrade, but exit-based SGX maintains a sizable advantage over exitless SGX.

These observations indicate that as the total number of threads approach the total core count of the machine, exitless calling becomes less beneficial due to additional threading and polling overheads. However, exit-based SGX continues to show performance scaling with increased server threads.



(a) Apache



(b) Nginx

Fig. 4: Web server performance measured in requests per second. The number of server threads for each application are varied, but the number of clients is fixed at 2 and 4 for Apache and Nginx respectively.

C. Apache and Nginx

As Apache and Nginx are both web servers, we evaluate them together. These workloads consider a variable number of web server threads that serve a fixed number of client threads, as described in the methodology. The number of ecall/ocalls in Apache and Nginx peak at 600,000 and 400,000 respectively. Consequently, both applications exhibit moderately high system call activity, albeit not to the same extent as Memcached. Moreover, the peak MPKI observed is 0.9 and 1.2 for Apache and Nginx respectively, indicating memory-bound behavior. As both workloads fit entirely within the EPC, page swapping exerts no influence on performance.

For all Apache configurations (Figure 4a), performance scaling is observed up to 2 server threads. In contrast to the Memcached workload, client threads do not dispatch requests concurrently, so peak performance for all configurations is achieved when the number of server thread matches the number of client threads (2 for Apache). Consequently, no performance scaling beyond 2 server threads is observed. Neither exit-based nor exitless SGX is able to match the performance of the baseline. Crucially, memory overheads dominate in both SGX configurations and neither approach offers a clear advantage over the other. Interestingly, the presence of additional responder threads under exitless SGX does not seem to further impact performance, even at higher thread counts. This is likely because the additional threading and polling overheads under exitless SGX are relatively small compared to the memory encryption/decryption overheads.

For the Nginx workload (Figure 4b), neither exit-based nor exitless SGX is able to match the performance of the baseline for the same reasons as described for the Apache workload. Both exit-based and exitless SGX see performance scaling up to 3 server threads. Moreover, peak performance for both SGX configurations is achieved at this server thread count, with exitless maintaining a sizable advantage over exit-based. Beyond 3 server threads, exit-based SGX performance remains about constant. At 4 and 5 server threads, exitless SGX performance degrades to match that of exit-based, and falls below exit-based for the remaining server thread configurations. In contrast to Apache, the additional threading and polling overheads incurred by exitless SGX become more severe as total thread count approaches the machine's core count. In this regard, Nginx's behavior under SGX is more similar to that of Memcached.

VI. Seeding Enclave-level Parallelism and Conclusion

In summary, enclave overheads interact in complex ways in the presence of increased enclave parallelism, which makes reasoning about real parallel workload performance difficult. Crucially, we identify that both exit-based and exitless calling scale with increased enclave parallelism, but in divergent ways that make deciding whether to use one over the other highly dependent on application threading behavior and machine capabilities. Overall, we observe that exit-based and exitless calling can perform about the same if each is configured properly, however exitless calling generally performs better than exit-based calling at lower thread counts whereas the opposite is true for higher thread counts.

On the back of this characterization, we identify the development of methods that make optimal configuration decisions for enclave applications based on known performance tradeoffs as a promising area of future research for both existing and emerging state-of-the-art secure processors. Specifically, our characterization reveals an opportunity for application-aware configuration or scheduling mechanisms that consider the degree to which enclave application threads (i) interact with the OS and (ii) are compute or memory-bound.

Acknowledgements

This research was supported by the National Science Foundation under Grants No. CNS-1929261 and CNS-1950600. Brandon D'Agostino was also supported by the U.S. Department of Education GAANN Fellowship. The authors wish to acknowledge Jared Nye who contributed to this research during his Research Experiences for Undergraduates (REU) at the University of Connecticut.

References

[1] D. C. Chou, "Cloud Computing: A Value Creation Model," *Computer Standards & Interfaces*, vol. 38, pp. 72–77, Feb. 2015.

[2] V. Costan and S. Devadas, "Intel SGX Explained." [Online]. Available: https://eprint.iacr.org/2016/086

[3] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption."

[4] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security-Enabling Trusted Computing in Embedded Systems."

[5] V. Costan, I. Lebedev, and S. Devadas, "Secure Processors Part I: Background, Taxonomy for Secure Enclaves and Intel SGX Architecture," vol. 11, no. 1, pp. 1–248. [Online]. Available: http://www.nowpublishers.com/article/Details/EDA-051

[6] V. Costan, I. Lebedev, and S. Devadas, "Sanctum: Minimal Hardware Extensions for Strong Software Isolation," pp. 857–874. [Online]. Available: https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/costan

[7] T. Bourgeat, I. Lebedev, A. Wright, S. Zhang, Arvind, and S. Devadas, "MI6: Secure Enclaves in a Speculative Out-of-Order Processor," vol. 1812, p. arXiv:1812.09822. [Online]. Available: http://adsabs.harvard.edu/abs/2018arXiv181209822B

[8] H. Omar and O. Khan, "IRONHIDE: A Secure Multicore that Efficiently Mitigates Microarchitecture State Attacks for Interactive Applications," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 111–122.

[9] R. Boivie and P. Williams, "SecureBlue++: CPU Support for Secure Executables." [Online]. Available: https://dominoweb.draco.res.ibm.com/BE73A643EFE8274B85257B51006760C0.html

[10] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer, "SCONE Secure Linux Containers with Intel SGX," pp. 689–703. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov

[11] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein, "Eleos: ExitLess OS Services for SGX Enclaves," in *Proceedings of the Twelfth European Conference on Computer Systems*, ser. EuroSys '17. Association for Computing Machinery, pp. 238–253. [Online]. Available: http://doi.org/10.1145/3064176.3064219

[12] 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. Association for Computing Machinery, pp. 81–93. [Online]. Available: http://doi.org/10.1145/3079856.3080208

[13] A. Hasan, R. Riley, and D. Ponomarev, "Port or Shim? Stress Testing Application Performance on Intel SGX," in *2020 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 123–133.

[14] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: a practical library OS for unmodified applications on SGX," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, ser. USENIX ATC '17. USENIX Association, pp. 645–658.

[15] 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*, ser. SysTEX '18. Association for Computing Machinery, pp. 22–27. [Online]. Available: https://doi.org/10.1145/3268935.3268942

[16] Which platforms support intel software guard extensions... [Online]. Available: https://www.intel.com/content/www/us/en/support/articles/000058764/software/intel-security-products.html

[17] S. Gueron, "A Memory Encryption Engine Suitable for General Purpose Processors." [Online]. Available: http://eprint.iacr.org/2016/204

[18] 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," pp. 285–298. [Online]. Available: https://www.usenix.org/conference/atc17/technical-sessions/presentation/lind

[19] Y. Shen, H. Tian, Y. Chen, K. Chen, R. Wang, Y. Xu, Y. Xia, and S. Yan, "Occlum: Secure and efficient multitasking inside a single enclave of intel SGX," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20. Association for Computing Machinery, pp. 955–970. [Online]. Available: http://doi.org/10.1145/3373376.3378469

[20] graphene (commit 75962a8). oscarlab. [Online]. Available: https://github.com/oscarlab/graphene/

[21] 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*, ser. EuroSys '14. Association for Computing Machinery, pp. 1–14. [Online]. Available: http://doi.org/10.1145/2592798.2592812

[22] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload analysis of a large-scale key-value store," vol. 40, no. 1, pp. 53–64. [Online]. Available: http://doi.org/10.1145/2318857.2254766

[23] memtier_benchmark. Redis Labs. [Online]. Available: https://github.com/RedisLabs/memtier_benchmark

[24] Apache HTTP Server Project. Apache Software Foundation. [Online]. Available: https://httpd.apache.org/

[25] nginx. Nginx, Inc. [Online]. Available: https://nginx.org

[26] Apache Benchmark. Apache Software Foundation. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html