Eleos: ExitLess OS Services for SGX Enclaves

Meni Orenbach, Pavel Lifshits, Marina Minkin, Mark Silberstein
Technion - Israel Institute of Technology

Abstract

Intel Software Guard eXtensions (SGX) enable secure and trusted execution of user code in an isolated *enclave* to protect against a powerful adversary. Unfortunately, running I/O-intensive, memory-demanding server applications in enclaves leads to significant performance degradation. Such applications put a substantial load on the in-enclave system call and secure paging mechanisms, which turn out to be the main reason for the application slowdown. In addition to the high direct cost of thousands-of-cycles long SGX management instructions, these mechanisms incur the high indirect cost of *enclave exits* due to associated TLB flushes and processor state pollution.

We tackle these performance issues in *Eleos* by enabling *exit-less* system calls and exit-less paging in enclaves. Eleos introduces a novel Secure User-managed Virtual Memory (SUVM) abstraction that implements application-level paging inside the enclave. SUVM eliminates the overheads of enclave exits due to paging, and enables new optimizations such as sub-page granularity of accesses.

We thoroughly evaluate Eleos on a range of microbenchmarks and two real server applications, achieving notable system performance gains. `memcached` and a face verification server running in-enclave with Eleos, achieves up to $2.2\times$ and $2.3\times$ higher throughput respectively while working on datasets up to $5\times$ larger than the enclave's secure physical memory.

1. Introduction

Intel Software Guard Extensions (SGX) [2, 8, 16, 20] are a new set of CPU instructions which enable trusted and *isolated* execution of selected sections of application code in hardware containers called *enclaves*. An enclave acts as a reverse sandbox: its private memory and execution state are isolated from any software outside the enclave, including an OS and/or a hypervisor, yet the code running in the enclave may access *untrusted* memory of the owner process.

While SGX provides the convenience of a standard `x86` execution environment inside the enclave, there are important differences in the way enclaves manage their private memory and interact with the host OS.

First, because an enclave may only run in user mode, OS services, e.g., system calls, are not directly accessible. Instead, today's SGX runtime forces the enclave to exit, that is, to *securely transition* from trusted to untrusted mode, and to re-enter the enclave after the privileged part of the system call completes. Similar exits are caused by asynchronous events such as page faults and signals.

Second, an enclave may access an isolated *trusted* memory space called the *enclave page cache*, or *EPC*, which is accessible only from that enclave. Physical memory that stores the EPC contents is limited to the size of trusted *processor reserved memory (PRM)* (128MB today). Therefore, EPC introduces an extra level of virtual memory with its own demand paging system. Under PRM pressure, EPC pages are securely evicted to untrusted memory and paged in on-demand by the SGX driver in response to EPC page faults.

Low-overhead system calls and efficient EPC paging mechanisms are essential to running I/O and memory-demanding server applications such as key-value stores (KVS) inside enclaves. These applications are naturally executed in enclaves, because most of their code is dedicated to manipulating private data in response to client requests. Unfortunately, running such workloads in enclaves today results in a significant slowdown. For example, Figure 1 shows that a simple KVS we implement runs $10\times$ to $33\times$ slower in the enclave, losing performance as its memory requirements grow. Our thorough analysis in §2 indicates the root cause of the slowdown: the high direct and indirect costs of *exiting* and *re-entering* an enclave per system call or EPC page fault. We seek to mitigate these costs in this work.

Eleos is a runtime system for *exit-less* OS services in enclaves. Eleos enables exit-less system calls by transparently and securely delegating them to a *remote procedure call (RPC)* service running in another application thread, without exiting the enclave. Furthermore, Eleos offers a novel *Secure User-managed Virtual Memory (SUVM)* abstraction. This abstraction provides the same security and functional-

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

 ISBN 978-1-4503-4938-3/17/04…$15.00

DOI: http://dx.doi.org/10.1145/3064176.3064219

ity as SGX EPC, but brings the paging mechanism *into the enclave*, therefore eliminating enclave exits due to paging. A user allocates buffers in SUVM via a special allocator and obtains a *secure pointer* or s*pointer*, which can then be used as a regular pointer in the application. A *s*pointer accessing evicted SUVM pages triggers a *software page fault*, which is handled entirely inside the enclave. Notably, these mechanisms are designed to improve enclave performance without weakening the original SGX security guarantees.

There are several key aspects of our design that together contribute to its performance advantages:

Reduced cache pollution due to system calls. As we show in §2, Last Level Cache (LLC) misses are expensive in enclaves, thus amplifying the application overheads of system call-induced cache pollution [28]. Limiting the LLC space available to the RPC thread using the Cache Allocation Technology can reduce the cache pollution effects, saving up to 25% of the enclave execution time (§6.1.1).

Application-managed paging. SUVM is a user-level library which implements its own per-enclave page table and page cache in EPC along with a secure backing store in host memory. Thus, it complies with hardware-enforced memory protection and SGX memory management. However, SUVM carefully sizes its page cache to avoid EPC page faults, achieving up to 7× faster accesses to secure memory buffers 10× the size of PRM (§6.1.2).

Low-overhead software address translation. Memory accesses via *s*pointers are resolved to the SUVM page cache or trigger a software page fault to a page in evicted pages. To save lookups on every *s*pointer access, the page cache pointer is *cached* in the *s*pointer, while the runtime ensures that the page is not swapped out for as long as the *s*pointer is in use. Thus, the page table lookup is performed once per page, making the *s*pointer accesses only 15-25% slower than accesses to EPC (§6.1.2).

Graceful handling of multiple enclaves. PRM is shared by all enclaves. Under PRM pressure, e.g., due to new enclave invocation, the SGX driver may evict part of the SUVM page cache, undermining its performance benefits. Therefore, SUVM coordinates the size of its page cache with the SGX driver to avoid thrashing, similarly to ballooning in virtual machines [31]. As a result, performance of multi-enclave execution improves by up to 3.5× (§6.1.2).

Optimized eviction and memory access policies. Exposing SUVM management to the application enables optimizations which cannot be implemented with SGX hardware paging. We introduce two such optimizations: preventing write back of clean pages to the backing store, and providing direct access to the backing store at sub-page granularity. We show that these optimizations boost performance by up to 1.7× and 1.6× respectively (§6.1.2).

The main observation that drives our work is that enclave exits are an impediment to application performance. Therefore, Eleos strives to reduce or eliminate the exits due to in-

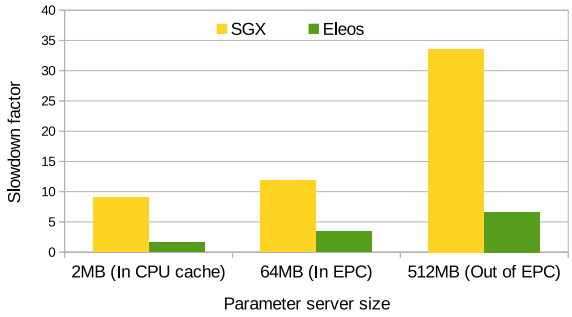


Figure 1: Parameter server slowdown in the enclave over untrusted execution, with and without Eleos (§2.1)

enclave system calls and EPC page faults. The concept of avoiding costly transitions to interact with system services from isolated environments has been studied in several contexts, e.g., virtual machines [14] and GPU computing accelerators [18, 27]. It has also been applied in FlexSC to accelerate system calls [28]. The SCONE project was the first to introduce batched exit-less system calls from SGX enclaves [9], independently and concurrently with our work.

Eleos retrofits some of these concepts under SGX constraints, combining them with novel design ideas such as user-level paging and software address translation, which together achieve significant performance gains. In particular, we show that handling EPC page faults in software inside the enclave is 3× to 4× faster than SGX hardware-implemented page faults, as it both avoids the exit overheads and sidesteps the complexity of managing EPC paging by an untrusted SGX driver. Moreover, through a systematic analysis of SGX overheads, we find that *indirect* costs of enclave exits dominate system performance. Therefore, with the expected drop in direct enclave management costs in the next hardware generations, the Eleos exit-less design that eliminates indirect costs will become even more important for achieving high performance in enclaves.

We evaluate Eleos on a range of microbenchmarks and two real server workloads: a face verification server [18] and `memcached`. Running the face verification server with Eleos on a dataset of 450 MB (about 4× PRM size), we achieve 95% of the throughput of the server execution without SGX, and up to 2.3× higher throughput compared to vanilla SGX. We modify `memcached` to use SUVM (75 LOC changed), and run it in enclave using the Graphene library OS [4, 29], enhanced with Eleos's RPC mechanism for faster system calls. With 500MB of data, `memcached` runs 2.2× faster with Eleos than with Graphene alone. Notably, the throughput of SUVM+`memcached` operating on 500MB of data is only 15% lower than the throughput of Graphene+`memcached` operating on a small 20MB data set that does not incur EPC page faults at all.

2. Motivation

We show that executing a simple server application in enclaves results in significant performance degradation. We then use this application to thoroughly analyze the main sources of in-enclave execution overheads. We refer to running code outside the enclave as an *untrusted* execution. The measurement methodology and the hardware specifications are presented in §6.

Workload. Parameter servers are commonly used in distributed machine learning systems to store shared model parameters (e.g., weights of a neural network) across a cluster of workers which train the model [11]. Each worker issues in-place updates, e.g., increments one or more parameters or retrieves their values. This workload serves as a proxy for large memory footprint server applications, yet it remains simple enough to analyze. It is also a fairly realistic application for SGX which operates on private data in public clouds.

We implement a simple parameter server and compare its performance inside and outside an enclave. The server stores the parameters in a hash table in the enclave's private memory, 8-byte keys, 8-byte values. The server listens to client requests, computes the new values, and updates the relevant entries in the hash table. Each client request updates one or more values. We use `OCALL` SGX SDK API to invoke `recv()` from the enclave. `OCALL` internally calls the `EEXIT` and `EENTER` SGX instructions.

In the experiments, the network requests are encrypted by the clients and decrypted by the server using AES-NI instructions. The load is generated on a separate machine connected via a dedicated 10Gb/s network.

2.1 End-to-end performance

We configure the load generator to issue 100,000 random single-value updates, and then measure the performance for three server data sizes: 2MB, which fits into the *last level cache (LLC)*, 64MB, which fits into the EPC, and 512MB, which exceeds the EPC.

Figure 1 compares the performance of trusted and untrusted execution. The enclaves incur dramatic slowdown: $9\times$ slower than the untrusted execution when the server data fits in LLC and up to $34\times$ for the out-of-EPC configuration. For comparison, we include the outcome of the same experiment with the best-performing Eleos configuration.

Our analysis presented next, shows that when the server data fits in EPC the slowdown is caused by the overheads of SGX exits due to system call invocations. When the data size exceeds the EPC size, the slowdown increases due to the overhead of the SGX paging mechanism.

2.2 The cost of system calls

Direct costs. We refer to the latency of SGX `EEXIT` and `EENTER` instructions invoked for each system call as a direct cost of running code in an enclave. We find that the latency of these two instructions is about 3,300 and 3,800 cycles, which is an order of magnitude higher than the cost of a regular system call (about 250 cycles [28]). The SDK `OCALL` API adds about 800 additional cycles, bringing the total cost of exits to about 8,000 cycles.

Operation	Sequential access	Random access
READ	5.6×	5.6×
WRITE	6.8×	8.9×
READ and WRITE	7.4×	9.5×

Table 1: Relative cost of LLC misses when accessing EPC vs. accesses to untrusted memory.

Our measurements show that when the parameter server is initialized to 2MB of data (fits in the LLC), processing a single request takes about 9,000 and 1,000 cycles inside and outside the enclave respectively. Therefore, the enclave application slowdown is rooted in direct enclave execution costs alone.

2.2.1 Indirect cost

When an enclave exits, its execution state gets partially evicted from the caches and other micro-architectural buffers in the CPU. As a result, when the enclave execution is resumed, the enclave state needs to be restored. We refer to the associated overhead as indirect costs.

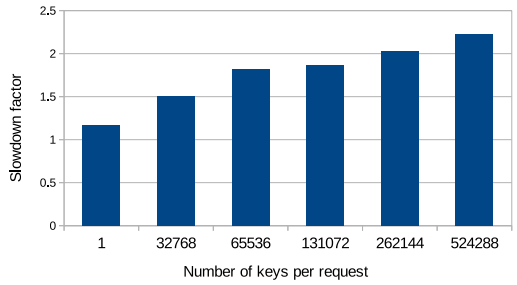
As we show below, the indirect costs might grow quite large when executing a system call. In fact, a similar impact on application performance was observed earlier with system calls alone [28]. However, in combination with enclave exits, the indirect cost of a system call is much larger.

The cost of LLC pollution. System calls, and in particular I/O system calls such as `recv()`, use additional internal buffers that compete with the application state in the LLC. Moreover, as enclaves operate on confidential data, decrypting/encrypting its inputs/outputs also consume part of the LLC space. However, in our experiments, we observe that the effective LLC size available to an application running in the enclave is smaller than in untrusted runs. We speculate that this is due to the SGX memory encryption engine storing its integrity tree in the EPC [15], which competes for LLC space as well.

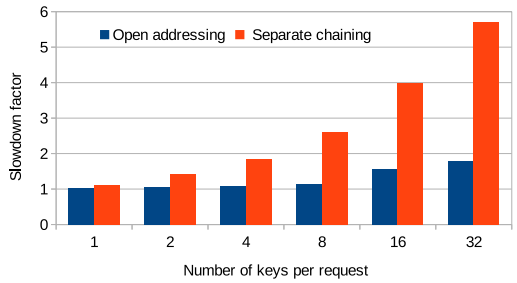
The cost of LLC misses, however, is dramatically higher in enclaves. Indeed, for every miss, the enclave performs integrity checks and encryption/decryption which slow down memory accesses.

Table 1 shows the slowdown of LLC misses when accessing EPC over accesses to untrusted memory. Read and write operations exhibit different performance most likely because for the write workload, there is a need to encrypt dirty data when evicting it from the cache.

Higher LLC contention and LLC miss costs in the enclave result in more costly system calls than in an untrusted environment, as we show in Figure 2a. We run a server with 64MB of data. For this experiment, we issue 100K random requests limited to "hot" keys, such that the parameter server



(a) LLC cache pollution costs for "hot" requests on a 64MB parameter server



(b) The cost of TLB flushes for a 2MB parameter server

Figure 2: Indirect costs of system call induced exits in a parameter server, as a function of the number of keys updated in each request. Measured in-enclave.

only updates 8MB out of the 64MB it holds (8MB is the size of the LLC). We measure the impact of the request size on the in-enclave execution time. Ideally, no impact should be observed. However, as we increase the number of keys updated per request, hence, cache pollution, the enclave execution becomes up to $2.2\times$ slower over the untrusted run.

The cost of TLB flushes. Figure 2b shows the effect of mandatory TLB flushes performed upon enclave exits. We compare two implementations of the parameter server: one that uses a hash table with open addressing and another with chaining; both are loaded with 2MB of data, which fits in the LLC. Both hash tables have the same number of buckets. We configure the load generator to issue 100K random requests, varying the number of keys to update in each request (up to 32 keys). We measure the in-enclave execution time without including the direct costs of exits and system calls. Since open addressing does not involve pointer chasing, this implementation is insensitive to TLB misses. However, the hash table with chaining suffers from the growing slowdown proportional to the number of hash table lookups (items accessed) per request.

2.3 The cost of EPC page faults

EPC page faults occur when the enclave accesses EPC pages that are not resident in the PRM. In today's systems, PRM is limited to 128 MB [2]. In practice, however, only about 90MB is available to applications. The rest is used by enclave page tables and metadata.

Direct cost. We instrument the SGX SDK and the EPC page fault handler in the SGX driver to count the number of page faults. Specifically, we count faults in which a single page is evicted. The cost of such faults is about 12k cycles, excluding the time to exit and re-enter the enclave. As a sanity check, we measure the combined cost of eviction and paging-in together, by performing random accesses to a 200 MB array in an enclave. We find the combined cost to be about 25k cycles.

Indirect cost. We measure the execution times of performing 100k random write operations to a small 60MB array (t_{nopf}) and to a 200MB array (t_{pf}). The first run did not experience page faults, while the second incurred about 2.9×10^9 page faults. For the second run, we have already measured the time required to handle page faults in the driver: C_{drct}=25,000 cycles. The total cost of a single page fault, which includes both eviction and paging-in as observed by the enclave, is computed as $C_{total} = \frac{(t_{pf} - t_{nopf})}{\#pagefaults}$. This cost includes the cost of exiting and re-entering the enclave: $C_{exit} = 7,000$ cycles (§2.2). The indirect cost is $C_{indrct} = C_{total} - C_{drct} - C_{exit}$. We conclude that C_{total}=40,000 cycles, thus $C_{indrct} = 8,000$ cycles.

We note that these measurements are slightly biased because accessing the 200MB buffer results in a 98% LLC miss rate compared to a 93.4% LLC miss rate for the 60MB buffer. We believe, however, that this bias is insignificant for the purposes of evaluating the indirect costs of EPC paging.

We conclude that running applications in SGX lead to significant performance slowdown due to costly system calls and EPC page faults. Eleos aims to eliminate or significantly reduce the number of enclave exits in these cases to recover application performance.

3. Design

Eleos provides two essential services inside the enclave: exit-less OS system call invocation via a Remote Procedure Call (RPC) mechanism, and Secure User-managed Virtual Memory (SUVM) with exit-less paging.

Our design goals are as follows:

- **Performance.** We seek to reduce the cost of running memory-demanding I/O-intensive server applications in enclaves.
- **Small TCB.** Eleos adds relatively little code into the enclave's TCB compared to Intel's SDK or library OSes like Graphene [29] (1000 LOC in total).
- **Ease-of-use and ease-of-tuning.** Eleos is intended for use by application developers, so it only introduces two new memory management functions, while RPC services are integrated transparently with Intel SDK. However, it also exposes a low-level tuning interface for expert runtime developers.

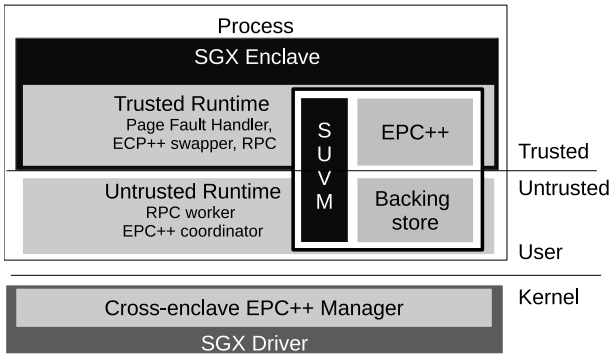


Figure 3: Eleos high-level design

- **Security.** Eleos does not change the original SGX security guarantees, nor does it improve or weaken them.

Design overview. Figure 3 shows the high-level design. There are three components: (1) the trusted runtime, which provides the RPC (§3.1) and SUVM (§3.2) inside the enclave; (2) an untrusted runtime running in a separate application thread to handle RPC requests and to interact with the SGX driver; (3) the SGX driver module, which exposes the interface for coordinating SUVM memory allocation across enclaves (§ 3.3). We now describe each component in detail.

3.1 RPC for system calls

The RPC mechanism enables invocation of blocking calls into untrusted code without exiting the enclave. The actual call is delegated to a *worker thread*, which executes in the *untrusted context* of the enclave's owner process. As an enclave may execute multiple threads, Eleos maintains a thread pool with multiple worker threads.

The thread pool interacts with the enclave via a shared job queue located in untrusted memory. To perform the call, the enclave enqueues the pointer to the untrusted function and its parameters in the job queue and blocks (via polling) until its completion. The threads in the thread pool poll the queue, invoke the requested functions, and transfer the results back via the untrusted shared buffer. The synchronization between the trusted and untrusted contexts is performed via polling because the enclave's threads cannot use standard OS synchronization primitives like mutexes. To reduce the cost of polling, Eleos invokes long running system calls like `poll()` via the naive `OCALL` mechanism.

Cache Partitioning. As we discussed in §2.2.1, the RPC mechanism alone is insufficient to recover in-enclave performance because of the LLC pollution induced by I/O system calls. This is because I/O buffers used by the worker threads still compete with the enclave threads for the LLC.

To reduce the impact of cache pollution, Eleos augments the plain RPC mechanism by partitioning the cache between the RPC thread (25%) and the enclave (75%) using Intel Cache Allocation Technology (CAT) [12]. Eleos's partitioning scheme favors enclave threads, assuming the untrusted calls executed in the worker threads require a smaller fraction of the cache. We evaluate the benefits of this optimization in §6.1.1.

Integration. The original Intel SGX SDK provides an `OCALL` interface for performing system calls from the enclave. Eleos's RPC infrastructure replaces Intel's `OCALL` mechanism while maintaining the same API, thus allowing applications to enjoy the performance benefits transparently without any code modifications.

3.2 Secure user-managed virtual memory

The *Secure User-managed Virtual Memory (SUVM)* abstraction provides a user-space mechanism for managing secure memory on top of the enclave's EPC, eliminating costly EPC hardware page faults and the associated enclave exits.

3.2.1 Design overview

SUVM is designed as an additional level of virtual memory (VM) on top of hardware VM. The enclave program allocates memory by calling `suvm_malloc()`, which returns a special pointer we call a *secure active pointer*, or s*pointer*. *S*pointers implement the same semantics as regular pointers, but they encapsulate the address translation mechanism which steers respective memory accesses to use SUVM.

SUVM implements a full-fledged paging system, with its own page table and a page cache in the EPC, and a backing store in untrusted memory of the enclave's owner process (see Figure 3). The SUVM page cache, *EPC++*, caches the contents of the backing store in the trusted memory. The SUVM page table maintains the mapping between EPC++ pages and the location of the cached content in the backing store. When an application accesses pages not resident in EPC++, an equivalent of a page fault occurs, but it is triggered by and handled in software. Specifically, the enclave handles the page fault by transferring the page content from the backing store into EPC++. SUVM software paging does not require exiting the enclave. SUVM mimics the behavior of the original SGX paging, maintaining privacy, integrity, and freshness of the evicted pages. We discuss the SUVM mechanisms in §3.2.2 and §3.2.3.

SUVM management in user-space provides several benefits beyond the exit-less paging. In particular, it enables the application-tailored eviction policies and memory access optimizations we discuss in §3.2.4.

To ensure efficient operation of SUVM with multiple enclaves, we extend the SGX driver to support *coordinated* allocation of EPC++ space per enclave. Thus, we avoid occasional eviction of EPC++ pages by the SGX driver due to growing memory pressure when running multiple enclaves. We discuss this further in §3.3.

We now describe each component of SUVM in detail.

3.2.2 Software address translation via ActivePointers

The key challenge in designing software-managed virtual memory is to overcome the overheads incurred on every

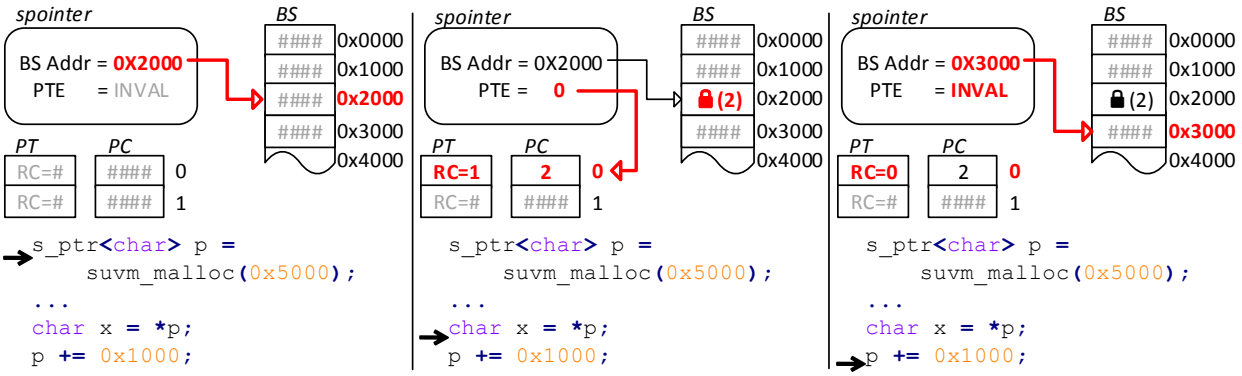


Figure 4: *S*pointers in action. BS: encrypted backing store, PT: page table, PC: page cache, RC: reference count. See 3.2.2 for details.

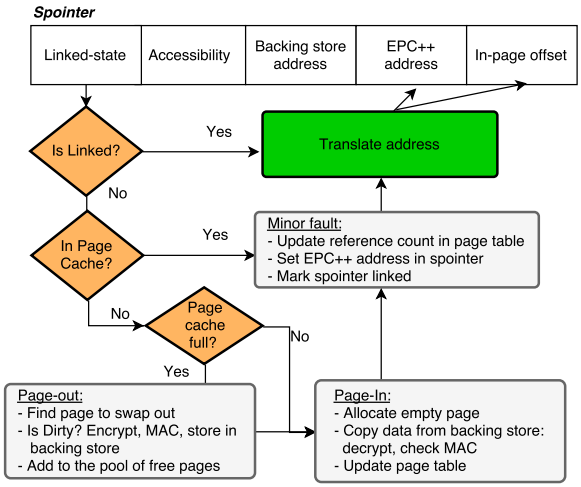


Figure 5: *S*pointer address translation in SUVM.

memory access due to the extra level of address translation. In our solution, we retrofit the *ActivePointers* software address translation mechanism used in GPUs [25].

*S*pointers provide an abstraction of a large linear address space on top of a small page cache backed by a large backing store. The page cache is stored in the EPC, and the backing store securely stores evicted pages in untrusted memory. The *s*pointers perform address translation, while the SUVM paging mechanism automatically swaps relevant data in and out of the backing store on-demand.

The *s*pointer address translation scheme is depicted in Figure 5. This mechanism is similar to the ordinary VM translation, with the notable exception that *user code has full control over the spointer's page table, page size, and eviction policy.* To save costly page table lookups on every memory access, each *s*pointer caches the translated address of the respective page in a local variable upon the first access (EPC++ address in the figure). The *s*pointer with the cached address is marked as *linked*. Thus, accesses via linked *s*pointers involve no page table lookup, reducing the number of lookups to only one per page. As a result, the translation overhead is reduced for the common case of spatial and temporal locality of accesses to a page. Access to an unlinked *s*pointer trigger either a major or minor page fault. Major page faults page data into EPC++, possibly evicting other EPC++ pages if no space is left in the cache. Minor page faults occur if the page is already resident in EPC++, but the respective *s*pointer is unlinked.

Two basic ideas make the address caching possible without sacrificing correctness. First, the system pins the pages in the page cache for as long as the *s*pointers that cache them are accessible by the application. A pinned page cannot be evicted from the page cache. The total number of *s*pointers pointing to a given page is kept in a per-page reference counter in the page table.

Second, the system strives to minimize the number of pinned pages. To that end, when a linked *s*pointer is destroyed (e.g., leaving the program scope), or moved beyond the boundary of the linked page, the *s*pointer gets *unlinked*. Then, SUVM drops the cached address, and decreases its respective reference count in the page table. Pages with their reference counts equal to zero become unpinned and can be evicted from the page cache.

Spointers in action. An example in Figure 4 illustrates the the *s*pointer address translation while executing a simple program. The figure shows the state of the *s*pointer data structures (changes in red) after the execution of the respective code at the bottom.

`suvm_malloc()` initializes a *s*pointer by allocating memory in the SUVM backing store (address 0x2000). Initially, the *s*pointer (Figure 4, left) is still unlinked, and the page is not cached in the EPC++ until the first access. As the program executes, the page gets accessed, cached, and eventually evicted. Figure 4, middle, depicts the state of the backing store after the page has been evicted and securely stored in it (value '2' in encrypted form).

When the *s*pointer is dereferenced (Figure 4, middle), the system realizes that the page is not found in EPC++,

and triggers a major page fault. The system allocates a new EPC++ page (page 0 in the figure), and populates it with the contents of the respective location in the backing store, decrypting and integrity-checking the data first. The address of the page in EPC++ is stored in the *s*pointer (PTE = 0), and the page reference count is increased (RC=1).

When the *s*pointer is incremented (Figure 4, right), it crosses the page boundary, so the cached value of the EPC++ page becomes invalid. *s*pointer is marked unlinked, and the reference count of the page in the page table is decremented.

Heuristics for reducing the number of pinned pages. The system enforces two rules that help reduce the number of linked *s*pointers and consequently the number of pinned pages: (1) when assigning a linked *s*pointer to another *s*pointer, the new *s*pointer is initialized *unlinked* (2) a *s*pointer gets unlinked when it is destroyed (e.g., automatic variables) and when it is iterated outside of the page boundary. These rules establish a reasonable tradeoff between the translation overhead of unlinked *s*pointers and the number of pinned pages. In particular, they enable efficient use of *s*pointers in data containers, e.g., hash tables, which is the main use case for *s*pointers in this paper. In such containers, even though the container contents are stored in buffers pointed to by *s*pointers, all these *s*pointers remain unlinked. As a result, SUVM enables creating data containers of arbitrarily large sizes, whose content is stored securely in the backing store.

3.2.3 *s*pointers and SUVM

SUVM API. The SUVM API has been designed for C++ applications. A *s*pointer is a C++ template which can be instantiated with any type and used instead of regular pointers. A *s*pointer is initialized by allocating memory via `suvm_malloc()`, and deleted via `suvm_free()` call. For applications written in C, we provide a lower level API for operating on the *s*pointer data type. This interface requires more effort to adapt an application to using SUVM, as we discuss in §5. Last, to achieve better performance with data containers and large data blobs, we design an optimized API for operating on memory buffers, e.g., `suvm_memcpy`, `suvm_memcmp` and `suvm_memset`.

Secure backing store. A secure backing store is allocated in untrusted memory of the process that owns the enclave. The data is initialized inside the enclave, but is written back to the backing store when a page is evicted from EPC++. Upon eviction, the page is first encrypted with a random per-page nonce and signed using a random per-application key stored in the EPC. When the page is paged in, its integrity is checked to avoid replay and data manipulation attacks. The nonce and the page signature are stored in the page table inside the enclave. The encryption, signing, and validation operations use AES-GCM just like the `EWB` SGX instruction, as described in the SGX manual [2].

Periodic page eviction. In vanilla SGX, EPC page eviction is the responsibility of the SGX driver. In Eleos, the eviction logic runs inside the enclave. Eviction may occur in three cases: (1) when EPC++ is full and a new page has to be paged in due to a page fault, (2) when an *EPC++ swapper thread*, which is periodically invoked by the untrusted runtime, removes some pages to maintain enough pages in the EPC++ free memory pool, and (3) when the swapper thread removes pages to reduce the size of EPC++ upon request of the SGX driver, e.g., when another enclave is started, as we explain in §3.3.

The eviction logic runs inside the enclave and is *trusted*, obviating the need to use costly *untrusted* EPC page table manipulation instructions such as `EWB`, `EBLOCK` and `ETRACK`.

Multithreading. The SUVM implementation supports multi-threaded enclaves. The page cache and page table are protected via fine-grained spin-locks built atop `x86` atomic instructions.

The use of SUVM in multi-threaded enclaves has an additional benefit of eliminating TLB shootdown costs associated with EPC page eviction. Specifically, when a page is evicted from EPC by the SGX driver, the driver first determines whether the page address is cached in TLBs of other cores via `ETRACK` instruction. The driver then performs the TLB shootdown which involves sending *Inter Processor Interrupts (IPI)* to other cores. A core which receives the IPI forces the running SGX thread to exit the enclave (via AEX [2]), resulting in an additional application performance penalty. In SUVM, instead, evicting a page from EPC++ does not require IPIs. We evaluate the associated performance savings in §6.

3.2.4 Memory access-specific optimizations

SUVM enables optimizations which are not yet available in vanilla SGX and might be hard to implement with SGX hardware virtual memory. We show two examples of such optimizations: avoiding write-back for clean pages and sub-page direct accesses to the backing store.

Avoiding write-back for clean pages. Page eviction is a computationally heavy operation on the critical path in large memory footprint workloads. However, if the page to be evicted is already present in the backing store and has not been modified since the previous eviction, the write-back is unnecessary and the page can be discarded. This optimization is not implemented in SGX today, and it is unclear whether such implementation is at all possible. This is because the only SGX memory eviction instruction (`EWB`) forces the page being evicted to be written to the backing store regardless of whether it has been modified.

In SUVM we add a dirty bit to the *s*pointer data structure that is updated on every write access. When the *s*pointer becomes unlinked, the dirty bit is copied into the page table. Unfortunately, C++ does not distinguish between deref-

erencing operators for read and for write. Therefore, to leverage this optimization, a user should access *s*pointers via `get/set` macros. The default behavior assumes write access.

Sub-page direct access. When the data access pattern exhibits no spatial/temporal locality, caching the contents in the page cache only decreases performance. Paging in incurs the overhead of copying, decrypting and validating the integrity of the whole page, even though only a small fraction of that page might actually be accessed.

To optimize for such an access pattern, we implement a special mode in which accesses to *s*pointers effectively bypass the page cache, and access data directly in the backing store, at sub-page granularity. The system ensures that reads are consistent by checking that the page is not resident in the page cache first. This mode is akin to `O_DIRECT` mode for direct access to storage.

When accessing the data in this mode, decryption, freshness and integrity check operations are performed at sub-page granularity too. Otherwise, the whole page would have to be read anyway, and most of the benefits of sub-page direct access would be lost. Therefore each sub-page is encrypted and signed separately, with its own nonce.

This optimization appears to be impossible to implement in SGX hardware today because SGX uses standard hardware virtual memory that may access data only from the page cache.

We note that allowing sub-page access to the page cache would have been equivalent to using SUVM with a small page size. In contrast, sub-page accesses to the backing store are suitable for workloads with small random accesses which exhibit no data reuse, as we show in §6.1.2.

3.2.5 SUVM security

As mentioned, SUVM follows the SGX secure paging design in terms of encryption and integrity checking mechanisms, ensuring that data integrity, privacy, and freshness of the evicted pages in untrusted memory are as protected as they would have been in the original SGX. All the security-related metadata is stored in the secure enclave memory. Moreover, the SUVM page table is stored in the secure enclave memory and can be accessed only from trusted code. The paging operations are performed inside the enclave as well. Therefore, neither the data nor SUVM management meta-data are exposed to untrusted code.

SGX is known to be vulnerable to controlled channel attacks which learn enclave's access pattern to EPC pages [33]. Similar techniques can be used to attack SUVM EPC++. However, SUVM would not leak any information beyond the page access pattern, hence it does not improve, neither reduces the security guarantees of the original SGX paging.

3.3 Multi-enclave memory allocation

Any EPC page, and in particular EPC++ memory, may be evicted from the EPC by the SGX driver. Unfortunately, this renders the SUVM exit-less paging mechanism useless, because accessing evicted SUVM pages would still result in a hardware page fault, and even incur additional SUVM translation penalty. This problem becomes severe with multiple enclaves, because the driver redistributes the secure memory dynamically as enclaves get invoked, without notifying the enclaves of the EPC allocation changes.

We, therefore, extend the SGX driver to coordinate EPC page eviction and allocation with the untrusted user-space runtime. Specifically, the runtime periodically queries the SGX driver to determine the secure memory space available to the enclave, adjusting the EPC++ allocation accordingly. To adjust the allocation of running enclaves the runtime invokes the SUVM swapper thread, which enters the enclave and frees or adds pages to its EPC++.

Similarity to ballooning. We note that the basic idea of collaborative management of EPC++ across enclaves is similar to that of memory ballooning [31], used by a hypervisor to manage memory allocation among virtual machines. However, whereas a hypervisor has no direct control of the OS memory consumption, the Eleos trusted runtime can directly modify the enclave's working set by evicting or adding new pages to its EPC++.

3.4 Discussion

SUVM offers an alternative design for secure memory management, enabling application-optimized, simpler, more flexible and thus more efficient paging mechanisms than in the original SGX. The exit-less design not only saves the overheads associated with page-fault induced exits, but also, requires no complex virtual memory management instructions, because the page faults are handled in a trusted environment.

SUVM targets only a single-application case, which matches the typical usage scenario for enclaves, similarly constrained to the context of one process. This simple scenario might not require the full power of generic hardware paging used in SGX today. Thus, it might enable more efficient hardware support as well, e.g., by adding application-level unprivileged page faults for certain memory regions. The SUVM application-level paging system would still be useful in such a case.

Despite the benefits, we consider SUVM complementary to the SGX paging system, because it relies on hardware protection mechanisms and global inter-enclave memory management.

Usability. Our experience with SUVM shows that using it for developing new SGX applications is easy. However, porting existing applications, especially C-based, requires significant effort as we discuss further in § 5. One option to improve the usability of SUVM for existing applications

is to employ mechanisms first used in distributed shared memory systems, e.g., Shasta [23]. This approach also holds the potential to both reduce current SUVM overheads by using aggressive compiler-optimizations.

Page size. The EPC++ page size is configured at compilation time. Increasing the page size may be useful to reduce the memory consumption of SUVM page tables and free more space for the application data.

Hardware support. SUVM could benefit from hardware support, for example, boundary checking [25], to speed up its critical path for page-fault free cases.

4. Implementation

We implement the Eleos prototype for Linux on Skylake SGX-enabled CPUs. We omit the details of the RPC mechanism, which is fairly straightforward, and focus on SUVM and SGX driver modifications.

4.1 SUVM implementation

Page tables. SUVM maintains two page tables: (i) The EPC++ inverse page table, which maps a page in the backing store to the EPC++ page. It stores the reference count of all linked *s*pointers referring to that page. This small page table has an entry for every EPC++ page, and is used upon every page fault. (ii) The crypto-metadata page table, which holds HMAC and nonce for each page in the backing store. This table is accessed only during the paging, and may grow fairly large.

Both tables are stored in EPC, and are implemented as hash tables with fine-grained locking, using separate spinlocks for each bucket. To ease contention and minimize rehashing, we pre-allocate the tables to be large. Under PRM pressure, unused entries get evicted through native SGX paging. However, frequently accessed mappings will remain resident in secure physical memory.

EPC++. We pre-allocate the memory pool for EPC++ pages in EPC, and keep track of unused pages in a free list. When EPC++ has to be downsized due to growing PRM pressure, we remove the pages from the free list and stop using them in EPC++, but do not de-allocate them. Since they are unused, the SGX driver eventually evicts them while keeping the other EPC++ pages intact.

Backing store. The backing store uses a slab memory allocator from the SQLite project [1]. The allocator implements the standard buddy system to reduce fragmentation, with a minimum allocation of 16 bytes in our configuration. We allocate the slab in the application heap in untrusted memory.

We implement all the cryptographic operations for the backing store using AES-NI [7] via the native SGX SDK's cryptographic library, which in turn relies on the IPPCP cryptographic library [30].

***s*pointers.** A *s*pointer is a C++ templated class. It stores the address translation data (See Figure 5), which includes the current page index, in-page offset, dirty bit, and a pointer to the EPC++ page when the *s*pointer is linked The data used in address translation of a linked *s*pointer is limited to 16 bytes to reduce LLC space, which is important to the reduction of fault-free access overheads.

Eleos uses operator overloading [1], providing the convenience of regular pointers.

SGX driver modifications. We add an `ioctl()` to query the amount of PRM available for a given enclave. Today's driver splits the PRM evenly among the enclaves, and therefore our implementation returns the number of active enclaves as a simple heuristic.

4.2 Limitations

SUVM space overheads. Our implementation can be improved by reducing the memory consumption of the *s*pointer object, which might, in turn, speed up applications that use many *s*pointers. In addition, we are overly conservative in our choice to store all the cryptographic metadata for the backing store in EPC. Some of its fields might be stored in untrusted memory, reducing the EPC consumption.

SUVM metadata eviction. SUVM does not evict its own metadata, relying on the native SGX paging mechanism.

Direct access to the backing store. Our implementation currently lacks working support for operating on both direct access to the backing store and page table access.

Misaligned data. Misaligned data in the backing store accessed via *s*pointers requires fetching the data from two subsequent entries in the secured backing store. Our current implementation currently lacks support for this mechanism.

EPC++ resizing. Our implementation currently lacks working support for dynamic EPC++ resizing and thus we resort to initialization-time configuration.

SUVM for code. SUVM does not support eviction of code.

5. Applications

We evaluate Eleos by using it to execute in enclave two real server applications, `memcached` and face verification [18]. We augment them to decrypt/encrypt each request/response from within the enclave using AES-NI hardware acceleration in CTR mode with a randomized 128-bit key. In this section, we describe the modifications to allow in-enclave execution of these applications with Eleos.

5.1 Memcached

`Memcached` is a popular in-memory key-value store widely used in production. It is an OS-intensive application with extensive use of system calls. To run it in the enclave we use the Graphene in-enclave library OS prototype [4, 29], which conveniently allows system call invocation from the enclave.

[1] *s*pointers currently overload dereference, assignment and arithmetic operators.

We integrate the Eleos's RPC mechanism with Graphene to enable exit-less system calls.

Modifications to memcached. `memcached` is written in C, therefore we cannot use the *s*pointers C++ interface, and instead employ the C API described in Section 3.2.3.

One non-trivial challenge is posed by the `memcached` memory allocation system. When integrating it with SUVM, one obvious option would be to place the memory pool used by `memcached`'s allocator in SUVM memory. Such a design, however, would require significant code modifications. Specifically, the `memcached` memory allocator stores the data (key and value) and metadata, i.e., slab class affiliation, last access time, *together* in the same memory pool. A large part of `memcached`'s logic accesses the metadata, all via memory pointers. Manually modifying all these pointers to use SUVM would require massive code changes.

Fortunately, less intrusive modifications (75 lines of code) are needed if we limit our use of SUVM only to store key-value pairs and their respective sizes. The size of a key-value pair is the only sensitive metadata that requires privacy and integrity protection of SGX. The rest of the metadata is security-insensitive, and thus is stored in the clear. This implementation enjoys the benefits of SUVM software paging because the key-value pairs constitute the main bulk of the `memcached` working set.

Implementation. We allocate `memcached`'s original memory pool in untrusted memory, and use it to store security-insensitive metadata. These include the slab class affiliation, last access time, expiration time, and hash chain pointers. We then allocate another memory pool via SUVM, and use it to *securely* store the keys, values and their sizes while replacing all references to these fields in the original code with *s*pointers. As a result, the memory pool in SUVM is managed by the `memcached` original allocator, while SUVM transparently takes care of demand paging.

5.2 Face verification

We adapt a biometric identity checking server like the one used in border and password control kiosks [18] to run in-enclave. This server stores a large database of sensitive biometric data for each individual using a hash table. A client queries the server to validate the claimed identity. The server retrieves the stored biometric data for the given identity and compares it with the measurements provided by the client.

We store the server's hash table in SUVM. We use a standard face recognition benchmark [22] for performance evaluation. The server receives an encrypted request from the client over the network. The request contains a person's identity and her image. The server fetches the images matching the person's identity from the hash table and invokes the LBP face verification algorithm [6] to compare them with the image in the request. The server then returns an encrypted response to the client whether the claimed identity has been verified. For this workload, the server's hash table contains 40-byte keys (the person ID) and 232KB values (the image histogram).

6. Evaluation

We evaluate Eleos using microbenchmarks and real workloads. First, we evaluate RPC and SUVM separately, seeking to characterize the sources of performance gains and overheads under various conditions. We then evaluate the entire system using the server workloads described in detail in § 5.

Setup. We use a Dell OptiPlex 7040 machine, with Intel Skylake i7-6700 4-core CPU with 8MB LLC, 128 MB PRM (about 93MB available for applications), 16 GB RAM, and 256 GB SSD drive. The machine runs Ubuntu Linux 14.04 64-bit, Linux 4.2.0-36, and the latest Intel SGX driver [5], SDK and platform software (PSW) [3] with our modifications for performance measurements. We use GCC 4.8.4, and compile using the SGX SDK *Prerelease* configuration, which has the same performance as production enclaves [2].

Measurement methodology. Measuring in-enclave performance is a non-trivial challenge: hardware performance counters cannot be used inside the enclave because reading them (including `RDTSC`) from enclave is not supported [2]. In addition, profilers that use them rely on interrupts for sampling, which in turn induce enclave exits and distort the actual values of the counters.

Instead, we use a measurement thread outside the enclave to sample the timer. The thread is signaled from the enclave via a shared flag to measure in-enclave execution time. The measurement error is about 200 cycles, which is an order of magnitude smaller than the values we measure in the experiments.

Unless specified, we measure end-to-end performance, run each experiment 60 times, with the first ten invocations as warm-up, and report the average of the rest. The standard deviation is within 5% across all the experiments and is not reported.

6.1 Microbenchmarks

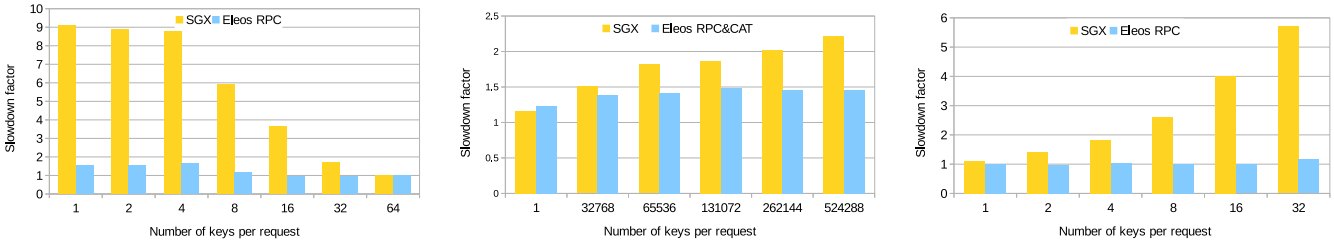
We evaluate the RPC and SUVM mechanisms on several microbenchmarks. We first report the results for RPC and then discuss SUVM performance.

6.1.1 RPC for system calls

We use the parameter server described in §2. All the experiments here match the respective experiments in §2.

Eliminating direct costs of `EENTER/EEXIT`. Figure 6a shows the slowdown of the server in the enclave compared to the untrusted run, while serving 100K random single-value requests (same experiment as in §2.2). The data size is 2MB, and entirely fits in the LLC. We observe that exit-less system calls result in 6× improvement for small requests, but are on par with `OCALLS` for batches with 64 updates.

[2] Attempts to issue `RDTSC` or `RDPMC` on our platform resulted in #UD.



(a) Eliminating EENTER/EEXIT costs (end-to-end) (b) Reducing LLC pollution overheads (in-enclave) (c) Eliminating TLB flush overheads (in-enclave)

Figure 6: Slowdown of the parameter server in enclave with and without the RPC mechanism. Lower is better.

Eliminating LLC pollution. We initialize the parameter server with 64MB of data, while serving only a pool of 8MB of "hot" random requests (which fits in the LLC). We compare in-enclave execution time (same experiment as in §2.2.1). Eleos allocates 25% and 75% of the LLC to the RPC thread and the enclave thread respectively. Figure 6b shows that cache partitioning enables over 25% improvement, in particular for larger I/O buffers.

Eliminating TLB flushes. We initialize the parameter server with 2MB of data, configure the load generator to generate 100k random requests while varying the number of updates per request, and measure in-enclave execution time. We use chaining in the parameter server's hash table to highlight the effects of TLB misses (same experiment as in §2.2.1). Figure 6c shows that Eleos eliminates the overheads of TLB flushes performed as part of enclave exits, resulting in up to 5.5× faster execution.

6.1.2 SUVM

Page-fault intensive workloads: large memory buffers. In each experiment, the program creates an array of *s*pointers, each pointing to an SUVM memory buffer allocated with a separate call to `suvm_malloc()`. The accesses are performed at random array locations via *s*pointers. This experiment emulates the access pattern to a hash table implemented with *s*pointers, but without hash contentions. We set the EPC++ size to 60MB and run the experiment for arrays with different numbers of *s*pointers.

Figure 7a shows the speedup of SUVM over native SGX for one thread, together with the number of hardware page faults in each setup. When the application buffer exceeds the available EPC, SUVM is about 5.5× and 3× faster for reads and writes respectively. Writes are slower than reads due to the write-back of pages upon eviction. For buffers of up to 1GB, SUVM eliminates all the hardware page faults, but as the application working set grows, SUVM management data structures get evicted from EPC.

Multi-threaded runs and TLB shootdown. Figure 7b shows the SUVM performance with multiple enclave threads. SUVM speedup is higher than in a single-threaded experiment. The SGX performance is slower due to the extra penalty of TLB shootdowns and associated Inter Processor Interrupts (IPIs) issued by the SGX driver.

Threads in enclave	#IPI SGX(SUVM)	#Faults SGX(SUVM)	Speedup
1	50.2e3 (84)	116.1e3 (151.2e3)	4.5×
4	77.9e3 (119)	115.1e3 (151.2e3)	5.5×

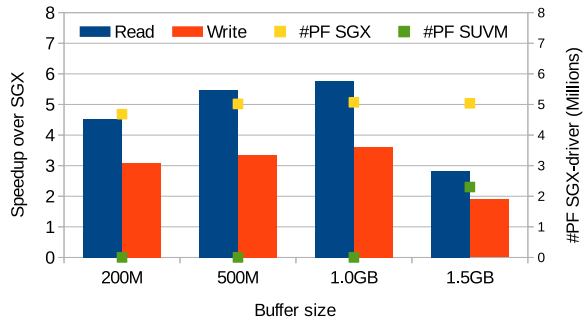
Table 2: Inter Processor Interrupts (IPIs) in SGX (SUVM), and SGX page faults (SUVM page faults) for 100k random 4K reads from 200MB buffer.

To confirm this hypothesis, we measure the number of IPIs and SUVM software/SGX hardware page faults for 100k random 4KB reads from a 200MB buffer. Table 2 shows that SUVM handles more faults, because it is configured with 60MB of EPC++, compared to about 90MB of PRM space available to SGX. Changing the number of threads has a negligible effect on the number of page faults. However, with multiple enclave threads the SGX driver performs 1.5× more IPIs than with one thread [3]. In contrast, no IPIs during SUVM execution are issued regardless of the number of threads, suggesting that the lack of IPIs is the primary reason for better SUVM multithreaded performance.

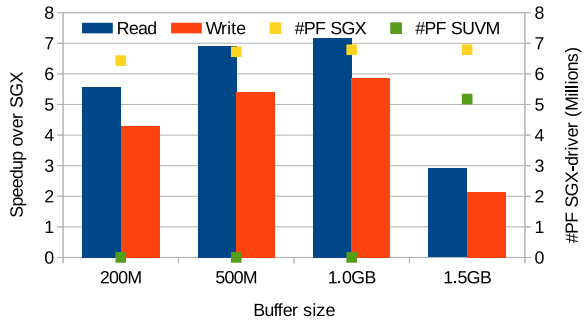
Page-fault free workloads: small memory buffers. We evaluate the overheads of memory accesses through *s*pointers by allocating a large array in SUVM with a single `suvm_malloc()` and accessing it through an *s*pointer at page-aligned locations. To avoid major page faults, the array is pre-fetched into EPC++. We vary the size of accessed elements to estimate the granularity of access at which minor page fault cost becomes amortized. We use two configurations: when the array fits into the LLC (the worst case for *s*pointers due to the low memory access cost), and when it fits into PRM (LLC miss, no page faults).

Figure 8a shows that *s*pointers introduce up to 22% overhead for reads and up to 25% for writes, while when the working set exceeds the LLC in Figure 8b the overhead is less than 20%.

[3] There are IPIs even for single-threaded enclaves due to an asynchronous swapper thread in the driver.

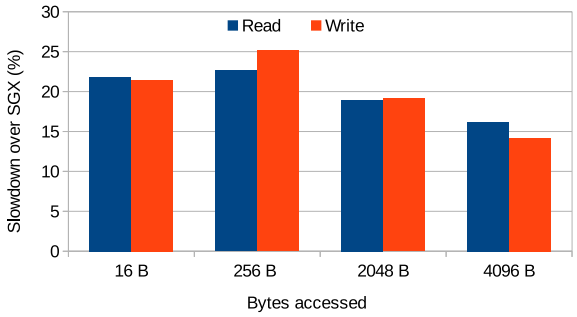


(a) 4K random accesses, one thread. Higher is better.

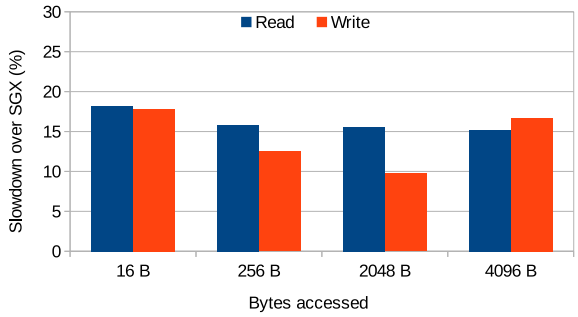


(b) 4K random accesses, four threads. Higher is better.

Figure 7: SUVM speedup over native SGX paging mechanism.



(a) Data in LLC (2MB). Lower is better.



(b) Data in PRM (60MB). Lower is better.

Figure 8: SUVM slowdown for fault-free accesses over regular access to memory.

Direct access to the backing store. Table 3 compares the performance of direct accesses with sub-page granularity (1KB sub-pages) vs. EPC++ accesses (4KB pages). Short reads enjoy up to 58% improvement, but larger ones are on par or slower than EPC++. There are two reasons: (a) the overhead of handling extra sub-page encryption and integrity checks when reading more than one sub-page, (b) about 25% of EPC++ accesses are hits.

Bytes/access	16	256	2048	4096
Speedup	58%	41%	-3%	-17%

Table 3: Direct accesses with 1KB sub-page granularity vs. EPC++ accesses with 4KB pages.

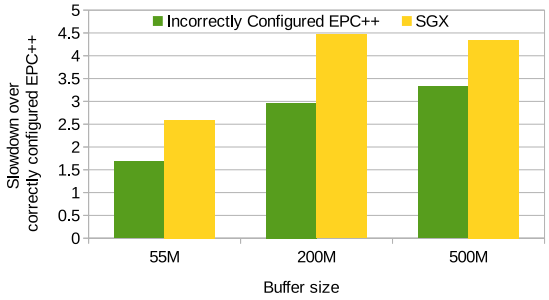


Figure 9: EPC++ resizing: slowdown of running two enclaves with SGX and incorrectly configured EPC++ over two enclaves with correct EPC++ size. Lower is better.

Coordinated allocation of EPC++ across enclaves. Each experiment measures the throughput of 4K random reads for three different sizes of arrays. We test three configurations: native SGX, SUVM with correctly configured EPC++ =30MB (fitting in PRM with two enclaves), and SUVM with incorrectly configured EPC++ =50MB (which causes thrashing with two enclaves). Figure 9 confirms that the EPC++ size has to be adjusted in response to PRM pressure. Running two enclaves with incorrectly configured EPC++ causes both SUVM and SGX faults, which results in up to 3.4× lower throughput compared to correctly configured EPC++.

SUVM software page faults vs. SGX hardware page faults. We measure the latency (in CPU cycles) of SUVM page faults as observed by the application, similarly to the way we evaluate native SGX faults in §2. Handling native SGX faults (including page eviction and paging in) requires about 40k CPU cycles (see §2). In SUVM, the combined cost of evicting and paging-in a page (as occurs in write

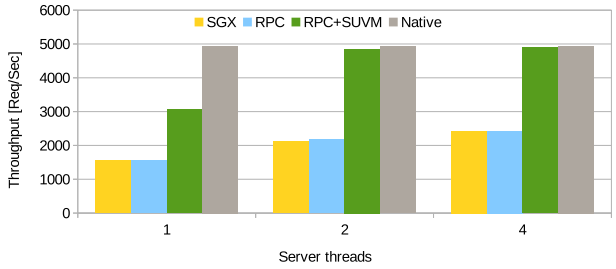


Figure 10: Face verification throughput. Higher is better.

Value size	Threads in enclave	SGX +Graphene	Eleos +Graphene	Native
1KB	1	21.4 (11.1×)	43.4 (5.2×)	229
1KB	4	57.8 (7.1×)	128 (3.2×)	406
4KB	1	16.6 (10×)	36 (4.5×)	163
4KB	4	41.8 (6.6×)	86 (3.2×)	274

Table 4: Throughput (Kops/s) of `memcached` running in enclave with and without Eleos vs. native execution (Slowdown factor).

workloads) is about 14K cycles, whereas the cost of paging in alone (read-only accesses) is about 8.5K cycles. Thus, software page faults are about 5× faster for read workloads and about 3× faster for write workloads, which correlates well with the results shown in Figure 7a.

6.2 End-to-end evaluation

We run all the experiments by using a separate machine for load generation. We use a 6-core Intel Xeon CPU E5-2608 v3 at 2GHz with 32 GB of RAM running Ubuntu 14.04. The server and the client machines are connected back to back via a 10Gb NIC.

6.2.1 Face verification server

We use a standard FERET face recognition benchmark [22]. Each image is resized to 512x512 and reformatted as raw grayscale. The data set is preprocessed and stored in a hash table, consuming 450 MB of memory. We use a request generator on a separate machine to saturate the server. We measure the throughput for different number of server threads.

We measure the throughput of a native server (no SGX), vanilla SGX run, and then Eleos with RPC mechanism alone, and finally with SUVM. Each request yields a single has table read of 232KB.

Figure 10 shows that the throughput of the native server is bounded by the network. Eleos's RPC alone is not effective, because the `OCALL` cost is hidden by network latency. The use of SUVM recovers most of the native performance, achieving 95% of the maximum throughput with two threads.

6.2.2 Memcached

We evaluate `memcached` using the popular *memaslsap* [34] load generation for `memcached` servers, which measures end-to-end throughput. We use memaslap to simulate enough clients to saturate the server. The client first inserts items to fill up `memcached` with values, and then requests these values via a GET command. We configure memaslap to issue requests to all the items stored in the memory pool (500MB, 4.5× the size of PRM). Finally, we set the key size to 20B, and experiment with two value sizes of 1KB or 4KB.

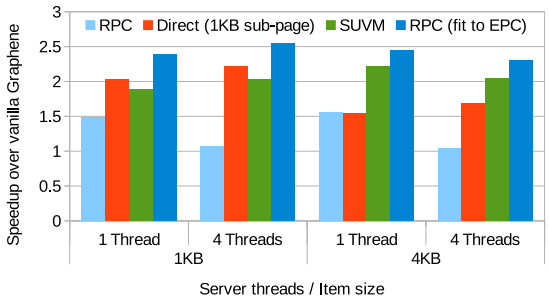


Figure 11: Throughput of `memcached` for different value sizes normalized to vanilla Graphene-SGX run. Higher is better.

We use `memcached` with Graphene as a baseline. To evaluate `memcached` with Eleos we modify Graphene to use Eleos's RPC mechanism, and run SUVM in two configurations: with EPC++ and with direct access (1KB sub-pages). To determine the upper bound on the SGX execution of `memcached` with Graphene, we evaluate a page fault-free execution with a small dataset (20MB) that fits into EPC entirely.

Figure 11 and Table 4 show the results. We observe that RPC is only effective while running `memcached` in 1 thread. SUVM with direct access offers the highest speedup of up to 2.2× over Graphene. Eleos with large data set achieves within 17% of the throughput of a page-fault free execution with Graphene. Finally, SUVM with direct access to the backing store is faster than EPC++ for shorter reads, but slower for larger accesses, matching the results of microbenchmarks (§6.1.2).

Metadata in untrusted memory. We evaluate the performance improvements due our changes to store the metadata in untrusted memory (§5). The modified version is about 3%-7% faster than the original one which stores all its data in trusted memory. Therefore, this change is not the primary source of the performance gains we report here.

7. Related Work

We are not aware of prior work which investigates exitless user-managed virtual memory. However, Eleos draws on similar ideas that have been considered in other contexts, such as optimized I/O and memory management in virtual machines and GPUs, and low-overhead system calls.

System support for trusted execution. Intel SGX SDK [8, 13, 16, 20] introduces the `OCALL` interface to allow untrusted function calls from enclaves, which force the enclave

to exit to perform such a call. Eleos replaces `OCALL` with a more efficient exit-less implementation.

The authors of Intel SGX 2 [19, 32] acknowledges the performance overheads of `OCALL`. However, they do not consider indirect costs.

Haven [10], Graphene [29] and PANOPLY [26] provide secure execution for legacy applications inside SGX enclaves without application code modifications, by providing a compatibility layer that deals with enclave execution. Our work is complementary, and can be used to improve applications performance, as we do with Graphene (§ 6). Finally, the integration of Eleos adds only a few hundred lines of code into the TCB.

VC3 [24] uses SGX to achieve confidentiality and integrity as part of the MapReduce framework. Ryoan [17] is a system used to execute enclaves in a sandbox and distributed environment. Ryoan proposed use cases include health analysis and image processing modules, which like VC3 are both I/O and memory-demanding. Thus, using Eleos with it might be beneficial.

Closest to our work, SCONE [9] leverages SGX to provide isolated execution for Linux containers [21]. SCONE employs an independently developed technique that is similar to Eleos's RPC mechanism. However, the authors do not analyze the costs of exits, as we do in this paper. Furthermore, Eleos enhances the RPC mechanism to reduce LLC pollution by using CAT. Finally, we extend the scope of exit-less services to virtual memory, and show significant performance benefits for workloads exceeding the size of PRM.

Asynchronous system calls. The authors of FlexSC [28] observe the need to reduce user/kernel transitions to optimize the system call performance, proposing asynchronous system call execution with batching. Eleos's RPC service is similar. Furthermore, our analysis of LLC pollution and TLB flushes was inspired by that of FlexSC.

System services for GPUs. This work adapts some of the ideas introduced earlier to provide system services on GPUs. Specifically, GPUfs [27] and GPUnet[18] are systems for efficient I/O abstractions for GPUs. Like them, Eleos uses an RPC infrastructure to reduce transition costs. ActivePointers [25] is a software address translation system for GPUs that provides support for memory mapped files. Eleos adopts this concept for *s*pointers but extends it by redesigning its paging system to support secure paging and optimizing it for execution on CPUs.

Virtual machine ballooning. Eleos applies the idea of coordinated memory management among virtual machines [31] to enclaves. Thus enclaves, like virtual machines, may evict pages according to their eviction policy. However, unlike VM ballooning, Eleos adds its own trusted swapping thread, and can directly modify the enclave's working set.

Distributed shared memory. Shasta [23] is a software based distributed shared memory system, which supports a shared address space across a cluster. To maintaining coherency in fine-grain granularity, Shasta instruments load and store instructions to test for memory state validity. Eleos, adapts this concept into *s*pointers, yet extends it to support full virtual memory management in a secure fashion.

Exit-less interrupts for optimized I/O in VMs. The concept of exit-less interrupt handling in Virtual Machines introduced in ELI [14] inspired us to consider techniques for eliminating costly exits in enclaves. ELI, however, focuses on interrupt handling in the context of optimized I/O performance, and does not consider avoiding exits due to page faults.

8. Conclusions

We introduce Eleos, an in-enclave trusted runtime for accelerating the execution of I/O and memory intensive applications in SGX enclaves. Eleos achieves its performance gains by mitigating the costs of exits in SGX enclaves. It offers an exit-less RPC mechanism to reduce the overhead of untrusted OS services invocation, and Secure User-managed Virtual Memory with exit-less page faults to reduce the cost of SGX secure memory paging.

Our evaluation shows that Eleos can significantly improve the performance of I/O and memory intensive applications executing in enclave, for example, up to $2.2\times$ better throughput for `memcached` and a face verification server operating on datasets $5\times$ larger than the secure physical memory.

Eleos advocates for moving system management and control into enclave, thereby exposing more opportunities for application-specific performance optimizations in the constrained enclave environment. Further, following this philosophy, Eleos might be extended to provide new services, i.e., inter-enclave shared memory, which are not currently supported in SGX. Last, we believe that future hardware support for enhanced execution control and memory management inside the enclave might not only help improve performance of Eleos in future systems, but will also help strengthen SGX security guarantees, reducing the reliance on untrusted OS services.

The source code for Eleos and all the enclave applications used in this paper is publicly available at `https://github.com/acsl-technion/eleos`.

9. Acknowledgements

We would like to thank Ittai Anati, Jack Doweck and Christof Fetzer for their insightful comments. Mark Silberstein was supported by the Israel Science Foundation (grant No. 1138/14). Meni Orenbach was partially supported by HPI-Technion Research School.

References

[1] SQlite zero-malloc allocation system. https://www.sqlite.org/malloc.html. Accessed: 2016-10.

[2] Intel Software Guard Extensions Programming Reference. https://software.intel.com/sites/default/files/managed/48/88/329298-002.pdf, 2016. Accessed: 2016-10.

[3] Intel Software Guard Extensions SDK for Linux OS Developer Reference. http://download.01.org/intel-sgx/linux-1.6/docs/Intel_SGX_SDK_Developer_Reference_Linux_1.6_Open_Source.pdf, 2016. Accessed: 2016-10.

[4] Graphene-SGX Library OS - A Library OS for Linux Multi-process Applications, with Intel SGX support. https://github.com/oscarlab/graphene, 2017. Accessed: 2017-02.

[5] Intel SGX Linux Driver. https://github.com/01org/linux-sgx-driver, 2017. Accessed: 2017-01.

[6] Timo Ahonen, Abdenour Hadid, and Matti Pietikainen. Face description with local binary patterns: Application to face recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):2037–2041, 2006.

[7] Kahraman Akdemir, Martin Dixon, Wajdi Feghali, Patrick Fay, Vinodh Gopal, Jim Guilford, Erdinc Ozturk, Gil Wolrich, and Ronen Zohar. Breakthrough AES performance with Intel AES new instructions. *Intel White paper, June*, 2010.

[8] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, volume 13, 2013.

[9] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L Stillwell, et al. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016.

[10] Andrew Baumann, Marcus Peinado, and Galen Hunt. Shielding applications from an untrusted cloud with haven. *ACM Transactions on Computer Systems (TOCS)*, 33(3), 2015.

[11] Trishul Chilimbi, Yutaka Suzue, Johnson Apacible, and Karthik Kalyanaraman. Project adam: Building an efficient and scalable deep learning training system. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 571–582, 2014.

[12] Intel corp. Improving Real-Time Performance by Utilizing Cache Allocation Technology. *Intel White paper, April*, 2015.

[13] Victor Costan and Srinivas Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086.

[14] Abel Gordon, Nadav Amit, Nadav Har'El, Muli Ben-Yehuda, Alex Landau, Assaf Schuster, and Dan Tsafrir. ELI: bare-metal performance for I/O virtualization. *ACM SIGPLAN Notices*, 47(4):411–422, 2012.

[15] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors.

[16] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP@ ISCA*, 2013.

[17] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: a distributed sandbox for untrusted computation on secret data. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 533–549. USENIX Association, 2016.

[18] Sangman Kim, Seonggu Huh, Xinya Zhang, Yige Hu, Amir Wated, Emmett Witchel, and Mark Silberstein. GPUnet: Networking abstractions for GPU programs. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 201–216, 2014.

[19] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel Software Guard Extensions (Intel SGX) Support for Dynamic Memory Management Inside an Enclave. In *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*. ACM, 2016.

[20] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP@ ISCA*, 2013.

[21] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux Journal*, 2014(239), 2014.

[22] P Jonathon Phillips, Harry Wechsler, Jeffery Huang, and Patrick J Rauss. The FERET database and evaluation procedure for face-recognition algorithms. *Image and vision computing*, 16(5):295–306, 1998.

[23] Daniel J Scales and Kourosh Gharachorloo. Design and performance of the Shasta distributed shared memory protocol. In *Proceedings of the 11th international conference on Supercomputing*, pages 245–252. ACM, 1997.

[24] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. VC3: trustworthy data analytics in the cloud using SGX. In *2015 IEEE Symposium on Security and Privacy*, pages 38–54. IEEE, 2015.

[25] Sagi Shahar, Shai Bergman, and Mark Silberstein. Active-pointers: a case for software address translation on GPUs. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 596–608. IEEE Press, 2016.

[26] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux Applications with SGX Enclaves. *National University of Singapore, Tech. Rep*, 2016.

[27] Mark Silberstein, Bryan Ford, Idit Keidar, and Emmett Witchel. GPUfs: integrating a file system with GPUs. In *ACM SIGPLAN Notices*, volume 48, pages 485–498. ACM, 2013.

[28] Livio Soares and Michael Stumm. FlexSC: flexible system call scheduling with exception-less system calls. In *Proceedings of the 9th USENIX Conference on Operating Systems De-*

sign and Implementation, pages 33–46. USENIX Association, 2010.

[29] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E Porter. Cooperation and security isolation of library OSes for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*. ACM, 2014.

[30] Anastasiya Vladimirova. Cryptography for Intel Integrated Performance Primitives 2017 Developer Reference. `https://software.intel.com/sites/default/files/managed/3c/05/ippcp_0.pdf`, 2015. Accessed: 2016-10.

[31] Carl A Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, 2002.

[32] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. 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, 2016.

[33] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Security and Privacy (SP), 2015 IEEE Symposium on*, pages 640–656. IEEE, 2015.

[34] Mingqiang Zhuang and Brian Aker. memaslap: Load testing and benchmarking tool for memcached. `http://docs.libmemcached.org/bin/memaslap.html`.