SGXKernel: A Library Operating System Optimized for Intel SGX

Hongliang Tian*
Tsinghua University
Beijing, China
tatetian@gmail.com

Yong Zhang, Chunxiao Xing†
Tsinghua University
Beijing, China
{zhangyong05,xingcx}@tsinghua.edu.cn

Shoumeng Yan
Intel Corporation
shoumeng.yan@intel.com

ABSTRACT

Intel Software Guard Extensions (SGX) is an emerging trusted hardware technology. SGX enables user-level code to allocate regions of *trusted* memory, called *enclaves*, where the confidentiality and integrity of code and data are guaranteed. While SGX offers strong security for applications, one limitation of SGX is the lack of system call support inside enclaves, which leads to a non-trivial, refactoring effort when protecting existing applications with SGX. To address this issue, previous works have ported existing library OSes to SGX. However, these library OSes are suboptimal in terms of security and performance since they are designed without taking into account the characteristics of SGX.

In this paper, we revisit the library OS approach in a new setting—Intel SGX. We first quantitatively evaluate the performance impact of *enclave transitions* on SGX programs, identifying it as a performance bottleneck for any library OSes that aim to support system-intensive SGX applications. We then present the design and implementation of SGXKernel, an *in-enclave* library OS, with highlight on its *switchless* design, which obviates the needs for enclave transitions. This switchless design is achieved by incorporating two novel ideas: *asynchronous cross-enclave communication* and *preemptible in-enclave multi-threading*. We intensively evaluate the performance of SGXKernel on microbenchmarks and application benchmarks. The results show that SGXKernel significantly outperforms a state-of-the-art library OS that has been ported to SGX.

CCS CONCEPTS

• **Software and its engineering** → **Operating systems**; *Multi-threading*; • **Security and privacy** → *Trusted computing*;

KEYWORDS

Library OS, Intel SGX, Trusted Hardware

*This work was done while the author was an intern at Intel.
†Also with, Research Institute of Information Technology, Tsinghua National Laboratory for Information Science and Technology,Tsinghua University.

CF'17, Siena, Italy

978-1-4503-4487-6/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3075564.3075572

ACM Reference format:
Hongliang Tian, Yong Zhang, Chunxiao Xing, and Shoumeng Yan. 2017. SGXKernel: A Library Operating System Optimized for Intel SGX . In *Proceedings of ACM International Conference on Computing Frontiers, Siena, Italy, May 15-17, 2017 (CF'17),* 10 pages.
DOI: http://dx.doi.org/10.1145/3075564.3075572

1 INTRODUCTION

Intel® Software Guard Extensions (SGX)[3] is a set of new instructions that enables user-level code to allocate regions of *trusted* memory, called *enclaves*, where the confidentiality and integrity of code and data are guaranteed. SGX is compatible with traditional system software like OSes and hypervisors, which are still able to manage resources, including enclave-related ones. SGX allows enclaves to run simultaneously on multiple processes or threads, thus utilizing the parallelism of multi-core processors. Performance evaluation of SGX on CPU-intensive benchmarks show an average performance overhead of only 5%[11]. Due to these advantages, SGX has been leveraged to secure a variety of applications, e.g., big data platform[16], network infrastructure[14], machine learning[13], etc.

However, due to security reasons, certain CPU instructions are forbidden inside enclaves, including `INT`, `SYSCALL` and `SYSENTER`, which are the ones that enable user programs to invoke system calls. As a consequence, the C standard library `tlibc` shipped with Intel SGX SDK[3] is stripped of any APIs that rely on system calls, e.g., `printf`, `open`, `socket`, etc. This lack of OS support inside enclaves makes it difficult to port legacy programs, reuse existing libraries or even develop new applications.

To bridge this gap, one natural idea is library OS, in which the operating system services are provided in the form of libraries. By linking to such a library at the compile time of an enclave, the user code inside the enclave can gain access to system calls. This approach has been demonstrated to be viable by previous works[2, 9], where existing library OSes (Drawbridge[15] and Graphene[19], respectively) have been successfully ported into enclaves. However, as will be shown later in this paper, these existing library OSes are suboptimal in terms of security and performance for SGX applications since they are designed without taking into account the characteristics of SGX.

In this paper, we revisit the library OS approach in a new setting—Intel SGX. Our key observation on SGX is that to library OSes that aim to support system-intensive applications (e.g., web servers and databases) inside enclaves, one potentially major source of performance overhead is *enclave transitions*, which happen when the

trusted code executed inside an enclave invokes untrusted functions outside the enclave, or vice versa. Another observation is that keeping the codebase of the library OS small should be an important design consideration for both security and memory consumption reasons.

Inspired by these observations, we design and implement SGXKernel, a library OS designed specifically for running a *single-process* application *securely* and *efficiently* inside enclaves. While SGXKernel is intended to be useful for a wide range of SGX applications, a particularly interesting class of target applications are *system-intensive* ones. As this class of workloads rely heavily on system calls, it is most challenging to library OSes; yet, SGXKernel is well-equipped for them.

To provide highly efficient system calls, we adopt a *switch-less* architecture for SGXKernel, which aims to completely eliminate expensive enclave transitions. The switch-less architecture of SGXKernel consists of two halves: a *trusted* one inside an enclave and an *untrusted* one outside the enclave. The two halves never initiate any enclave transitions to each other; in fact, they run *concurrently* in two disjoint set of SGX or OS threads. To make this switch-less architecture work, we incorporate two novel ideas:

1. Asynchronous cross-enclave communication. The trusted and untrusted halves communicate with each other *asynchronously* by sharing and accessing *cross-enclave data structures*, which have two strong properties: *non-blocking concurrency* and *tamper resistance*. By leveraging cross-enclave data structures, we implement three asynchronous cross-enclave communication primitives—delegated calls (or `DCalls`), I/O streams, and virtual interrupt signals—which enable efficient communication between the two halves of SGXKernel without triggering any enclave transitions.

2. Preemptible in-enclave multi-threading. The overheads of context switching and thread synchronization among SGX threads are extremely high due to the enclave transitions incurred. To address this issue, SGXKernel implements a multi-threading mechanism entirely inside enclaves (thus no enclave transitions involved). This *in-enclave* multi-threading mechanism maps M in-enclave threads to N SGX threads in order to enable true parallelism among in-enclave threads on different SGX threads and to achieve fast context switching among in-enclave threads on the same SGX thread. In addition, fairness among in-enclave threads is ensured by *preemptive scheduling*.

As designed for empowering *single-process* SGX applications to access services of the host OS, a great portion of OS functionalities in SGXKernel can be simplified (e.g. `execve`, `mmap`, etc.) or redirected to the host OS with little or no mediation (e.g. `getpid`, `mkdir`, etc.). Thus, we end up with a codebase of only 7,000 LoC for the trusted half of SGXKernel. In addition, we choose `musl`[4] instead of the de facto `glibc`[1] as the implementation of the companion C standard library for its greatly reduced codebase. The result is a combined TCB (trusted computing base) of around 70,000 LoC, which is at least one order of magnitude smaller compared to the existing library OSes[15, 19] that have been ported to SGX.

Our experiment results show that SGXKernel significantly outperforms Graphene-SGX[2], a state-of-the-art, open-source library OS for SGX: the improvement of syscall throughput is up to 15×; I/O throughput, up to 56×; inter-thread synchronization, 7×. Powered by SGXKernel, we successfully ported Redis[5]—an in-memory database with a reputation of high request throughput—into enclaves with minimal code modification (around 50 LoC) and modest performance overhead (23% on average) compared to running on Linux. We believe an overhead of this level can be acceptable in many scenarios when taking into account the enhanced security by SGX.

The main contributions of this paper are:

1. We analyze and quantify the performance overhead of enclave transitions, identifying it as a potential performance bottleneck for any library OSes that aim to support system-intensive SGX applications (section 2).

2. We present the *switch-less* design of SGXKernel, which obviates the needs for enclave transitions. We highlight on the two novel ideas: *asynchronous cross-enclave communication* and *preemptible in-enclave multi-threading* (section 3).

3. We intensively evaluate the performance of SGXKernel on microbenchmarks and application benchmarks. The results demonstrate that SGXKernel significantly outperforms the state-of-the-art library OS for SGX, and its overhead is acceptable for practical uses (section 4).

2 THE (HIGH) COSTS OF ENCLAVE TRANSITIONS

In this section, we analyze and quantify the overhead incurred by enclave transitions and their performance impact on SGX programs. We start by giving some background knowledge about enclave transitions, and then present the experimental results on micro-benchmarks.

2.1 Understanding Enclave Transitions

In Intel x86 architecture, there are a number of execution modes—e.g., privileged mode, user mode and enclave mode—and an even larger number of transitions between them. An *enclave transition* happens when the execution mode of a CPU switches from non-enclave mode to enclave mode, or vice versa.

To aid readers' understanding of enclave transitions, let's first have a quick review on the classic concept of user-privileged transitions. In modern x86-64 machines, application software (in user mode) uses `SYSCALL` instruction to jump to a predefined location of OS kernel (in privileged mode); and the kernel uses `SYSRET` to switch back to the calling function in user mode. Another form of user-privileged mode switching is when a hardware interrupt occurs: the interrupted user code is forced to give up the control of CPU to the privileged code for interrupt handling.

The switching mechanism for enclave mode is similar. User programs can jump into a predefined location in an enclave via `EENTER` instruction and exit back to user mode via `EEXIT`. Additionally, if a hardware interrupt occurs while a logical processor is running in enclave mode, the logical processor is taken out of enclave mode by *Asynchronous Enclave Exit* (AEX), before switching into privileged mode for interrupt handling. The interrupted execution of the enclave can be continued by invoking `ERESUME` instruction.

These SGX CPU instructions—`EENTER`, `EEXIT` and `ERESUME`—are, of course, not intended to be used directly by SGX application developers; they use more user-friendly APIs from Intel SGX SDK—

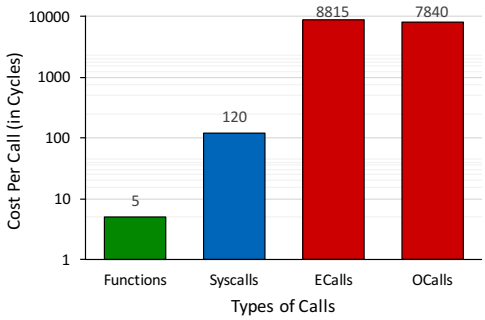


Figure 1: The overheads of different types of calls.

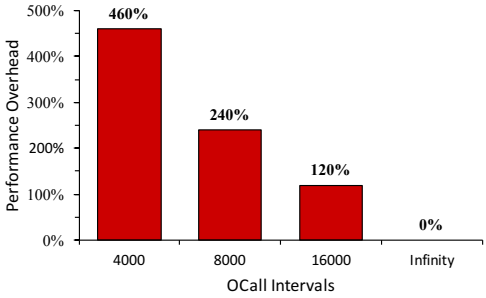


Figure 2: The performance overhead caused by `OCalls` (invoked at various intervals)

`ECalls` and `OCalls`. An `ECall` (or enclave call) is a function call from the outside of an enclave to a trusted function inside the enclave; and an `OCall` (or outside call) is a function call from the inside of an enclave to an untrusted function outside the enclave. Intel SGX SDK provides tools that help developers specify and implement user-defined `ECalls` and `OCalls` that suit their specific needs.

While transparent to SGX application developers, every `ECall` issues both `EENTER` and `EEXIT` instructions at least once to complete the switching in and out of enclave mode; and the same is true for an `OCall`, just in the opposite order. As `ECalls` and `OCalls` are the de facto mechanisms to communicate between the trusted part and the untrusted part of a SGX program, and the only means available for SGX application developers to initiate enclave transitions, we mainly focus on the enclave transitions incurred by `ECalls` and `OCalls`, and measure the performance of empty `ECalls` or `OCalls` to quantify the overhead of enclave transitions.

The primary sources of the overhead of enclave transition are complicated bookkeeping and security checks, saving and restoring CPU states, and the TLB cache misses caused by flushing TLBs.

2.2 Microbenchmarks

In the remainder of this section, we quantify the overhead of enclave transitions and their impact on the performance of SGX applications.

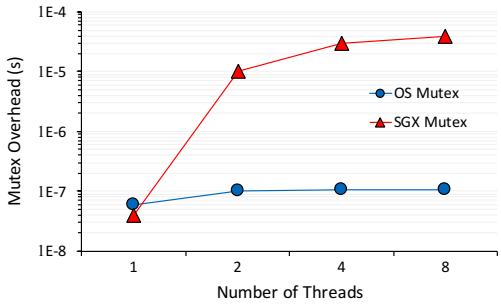


Figure 3: The overheads of SGX and OS mutexes.

All the microbenchmarks use an Intel Core i7-6670HQ CPU at 2.6GHz with 8 hyper-threads (2 per core) and 6MB cache. The machine runs on an Ubuntu 14.04 server with Intel SGX SDK 1.7 installed.

The overhead of enclave transitions. The first microbenchmark compares the overheads of different types of call mechanism: function calls, system calls, `ECalls` and `OCalls`. In this microbenchmark, all the calls use "empty" implementation. Thus, our result should represent the overhead associated with a specific kind of call mechanism. As shown in in figure 1, an empty OS call, which spends its time mainly on switching into the OS kernel and then back to the user space, is fairly fast as well, requiring only 120 cycles, which is even smaller than the latency of a memory access that misses CPU cache. In a sharp contrast, the time required to complete an empty `ECall` or `OCall` is as high as 8815 cycles and 7840 cycles, respectively; in another word, *the overhead of an enclave transition is over* 60× *higher than that of a system call.*

Next, we examine the performance impact of enclave transitions on applications.

Performance impact on a single-threaded SGX application. In the second experiment, we ported a single-threaded, CPU-intensive program (the bzip2 test from SPEC CPU 2006 benchmark) into enclaves and modified it so that empty `OCalls` were invoked periodically. As shown in figure 2, when invoking one `OCall` for every 4,000 CPU instructions, the runtime was lengthened by 460%, and when invoking one `OCall` for every 16,000 CPU instructions, the runtime was lengthened by 120%. This range of `OCalls`' frequencies are comparable to that of system calls in system-intensive workloads (as shown in [9, 18]), thus the result obtained is practical.

Performance impact on a multi-threaded SGX application. In the third experiment, we benchmarked a simple multi-threaded program that concurrently increased a shared counter protected by a mutex. We compared two versions of the program, one was a normal C program using `pthread`, the other was a multi-threaded SGX application based on the multi-threading library provided in Intel SGX SDK. As depicted in figure 3, the overhead of SGX mutexes was comparable to that of OS mutexes as long as there was no lock contention; however, SGX mutexes performed two magnitudes worse when the number of threads was greater than 1. This is because for every unlock operation, a SGX mutex has to do an `OCall` to wake up other SGX threads that wait on the mutex. This dramatically degrades the performance of SGX mutexes.

The last two micro-benchmarks demonstrate that *frequent enclave transitions can cause significant performance degradation on SGX applications.*

3 SGXKERNEL: DESIGN AND IMPLEMENTATION

In this section, we start with a discussion on the design principles of SGXKernel. We then give an overview of the *switch-less* architecture of SGXKernel. The remaining of this section details the design and implementation of the system.

3.1 Design Principles

We argue that to construct a *secure* and *efficient* library OS for SGX, two guiding design principles should be followed:

1. Offloading inessential responsibilities to the host OS. Previous library OSes that have been ported to SGX environment, e.g. Drawbridge and Graphene, prioritize application compatibility, security isolation, and independent system evolution benefits. To this end, a large portion of an ordinary OS has to be part of the resulting library OSes. The codebase of the library OS and its dependencies is well beyond 1,000,000 LoC in Drawbridge and more than 500,000 LoC in Graphene, leading to a bloated TCB. A TCB of this large is almost certain to contain security vulnerabilities (such examples are abundant[6, 17]). Moreover, the bloated codebases imply larger memory footprints. This may not be a problem for regular programs as a dynamically-linked library (say, a library OS) can be shared across different processes; but for SGX programs, sharing trusted memory across different enclaves is forbidden by design, thus trusted memory consumption by a (huge) library OS increases (quickly) with the number of running enclaves. Note that on current Intel processors with SGX, the cache of trusted memory is only 128MB; when the total size of the working memory of running enclaves exceeds this threshold, the overhead of trusted memory access dramatically increases[10]. Therefore, in order to minimize the TCB and the memory footprint, a library OS for SGX demands a minimal design, offloading inessential responsibilities to the host OS as much as possible.

2. Reducing the CPU switches between library OS and host OS. The implementation of certain system calls of a library OS inevitably relies on the host OS to perform privileged operations. To facilitate the library OS to request services from the host OS, Drawbridge and Graphene use a dedicated component named Hardware Adaption Layer (HAL), which exposes a small set of functions with well-defined semantics to the upper-layer library OS. For example, when the library OS receives a `socket` system call from the user code, it will eventually call a HAL function named `DkStreamOpen`, which then in turn call `socket` (on Linux) or `CreateSocket` (on Windows) on behalf of the library OS to do the real job. In the context of library OS for SGX, these HAL functions like `DkStreamOpen` must be 1) callable inside enclaves yet still 2) able to access normal system calls. Thus, it is not surprising that HAL functions perform enclave transitions. As a matter of fact, Graphene-SGX implements HAL functions as `OCalls`. Unfortunately, as have shown in section 2, enclave transitions (or `ECalls`/`OCalls`) are extremely expensive. Should a library OS trigger enclave transitions on every system call, its performance for system-intensive workloads would degrade significantly. Thus, reducing enclave transitions should be an important design consideration for SGX library OSes.

At first glance, these two goals of offloading *more* yet switching *less* to the host OS seem to be contradictory. Yet, the rest of this paper will demonstrate how it can be achieved.

3.2 System Overview

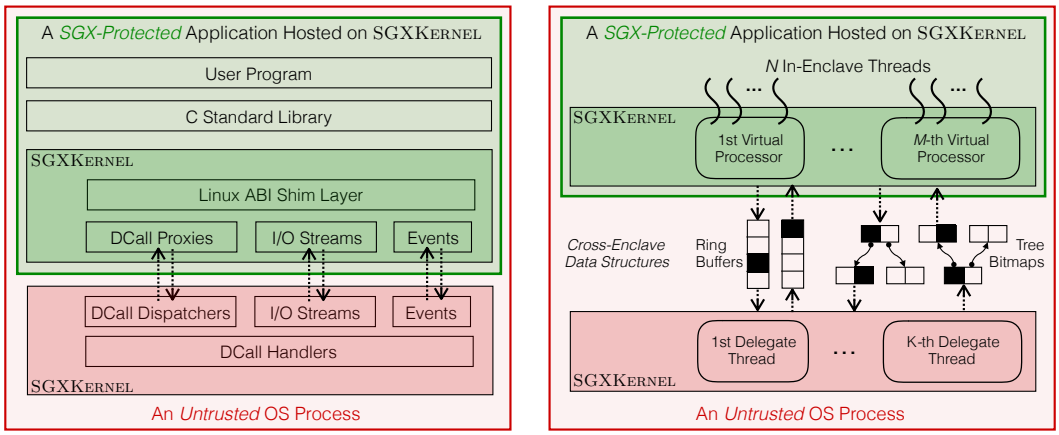
With the aforementioned principles in mind, we design and implement SGXKernel, a library OS designed specifically for running a *single-process* application *securely* and *efficiently* inside enclaves.

While SGXKernel is intended to be useful for a wide range of SGX applications, a particularly interesting class of target applications are *system-intensive* ones, e.g. web servers and database systems, which 1) are commonly deployed on (potentially malicious) public cloud infrastructure, and thus likely to benefit from the protection of SGX, and 2) have workload characteristics like high I/O throughput, high concurrency, and intensive synchronization. As this class of workloads relies heavily on system calls, it is most challenging to library OSes; yet, SGXKernel is well-equipped for them.

To provide highly efficient system calls, we adopt a *switch-less* architecture for SGXKernel, which aims to completely eliminate expensive enclave transitions from the"critical path" of executing a system call. As shown in figure 4a, the architecture of SGXKernel, like any other SGX applications, consists of two halves: a *trusted* one inside an enclave and an *untrusted* one outside the enclave; yet, unlike normal SGX applications, the two halves never trigger any enclave transitions to each other; in fact, they run *concurrently* in two disjoint set of SGX or OS threads. To make this switch-less architecture work, we incorporate two novel ideas::

1. Asynchronous cross-enclave communication. The trusted and untrusted halves share and access special data structures outside the enclave to communicate with each other *asynchronously*. These data structures are called *cross-enclave data structures*, which have two strong properties: *non-blocking concurrency* and *tamper resistance*. Specifically, we implement in SGXKernel two types of cross-enclave data structures: *cross-enclave ring buffers* and *cross-enclave tree bitmaps*. By leveraging these cross-enclave data structures, we implement in SGXKernel three asynchronous cross-enclave communication primitives—delegated calls (or `DCalls`), I/O streams, and virtual interrupt signals—which enable efficient communication between the two halves of SGXKernel without triggering any enclave transitions. For example, SGXKernel completely gets rid of `OCalls` by using `DCalls`. `DCalls` are issued from the trusted half by pushing the requests into FIFO queues (which are cross-enclave ring buffers), and asynchronously executed by the untrusted half after popping `DCall` requests from the queues.

2. Preemptible in-enclave multi-threading. To SGXKernel, providing efficient multi-threading is a top priority for two reasons: on the one hand, workloads that depend heavily on multi-threading are common; on the other hand, fast context switching leaves more time for other runnable threads to do useful work while threads are waiting for the completion of the `DCalls` they have issued. However, as we have shown in 2, the overhead of thread synchronization



(a) Component View (b) Runtime View

Figure 4: The system architecture of SGXKernel, which consists of a trusted half and an untrusted half, executing in enclave mode and user mode, respectively.

among SGX threads are extremely high due to the enclave transitions incurred. For the reasons above, SGXKernel implements a multi-threading mechanism entirely inside the enclave (no enclave transitions involved), thus offering very fast context switching and synchronization primitives (e.g., mutexes). This *in-enclave* multi-threading mechanism maps M in-enclave threads to N SGX threads (each of which runs a *SGX virtual processor*) to achieve true parallelism on multi-core processors (see figure 4b). In addition, fairness among in-enclave threads is ensured by *preemptive scheduling*.

Besides minimizing the number of enclave transitions, SGXKernel also minimizes its TCB. As designed for running single-process applications, some OS functionalities, like process management (e.g., `execve`) and memory management (e.g., `mmap`), can be greatly simplified in SGXKernel; as a matter of fact, since the memory pages inside enclaves must be statically allocated (on the current hardware platform) and are private to its owner enclave, it is impossible to fully simulate these system calls (e.g., `execve`, `mmap`, etc.) anyway. Furthermore, a great number of system calls (e.g., `getpid`, `mkdir`, etc.) can be redirected by the trusted half via `DCalls` to the untrusted half with little or no mediation. With a great portion of system calls simplified or redirected, we end up with a codebase of only 7,000 lines for the trusted half of SGXKernel. We emphasize that a codebase of this size is amenable for formal verification (as has been done in [12]). As user programs use wrapper functions from C standard library instead of invoking system calls directly, we accompany SGXKernel with a slightly-modified version of `musl`[4], which is a modern libc implementation with a much smaller codebase than the de facto `glibc`[1]. This results in a combined TCB of around 70,000 lines of code, which is one order of magnitude smaller compared to the existing library OSes[15, 19] that have been ported to SGX.

The rest of this section elaborates on the two salient aspects of SGXKernel: asynchronous cross-enclave communication and preemptible in-enclave multithreading.

3.3 Asynchronous Cross-Enclave Communication

The trusted and untrusted halves of SGXKernel communicate with each other *asynchronously* via **cross-enclave data structures**, which have two strong properties:

1. *Non-blocking concurrency*, which enables concurrent access by both trusted halves and untrusted halves in a non-blocking way. Based on these non-blocking data structures, the resulting asynchronous cross-enclave communication primitives that are used by either of the two halves are guaranteed to make progress regardless of any delay of the SGX or OS threads that the other half runs on. Common sources of delay include scheduling preemption and page faults from the host OS.

2. *Tamper resistance*, which guarantees the trusted half not exploitable by any adversary who can access the untrusted memory used by cross-enclave data structures. This means, in spite of any possible malicious modification to the untrusted memory, no security loopholes, e.g., buffer overflow or unsafe pointer manipulation, would exist in the trusted half of SGXKernel.

In SGXKernel, we design and implement two types of cross-enclave data structures: *cross-enclave ring buffers* and *cross-enclave tree bitmaps*. Due to the two strong properties required by cross-enclave data structures, their implementation is tricky to get right. The implementation details are out of the scope of this paper; we only list here several key points. SGXKernel's cross-enclave data structures 1) are designed with lock-free algorithms rather than lock-based ones; 2) are array-based instead of link-based (in order to avoid dynamic allocation of untrusted memory and to minimize manipulation of untrusted memory pointers inside enclaves); and 3) replicate and validate security-critical fields inside the enclave instead of maintaining them only outside the enclave.

Based on these cross-enclave data structures, we implement three asynchronous cross-enclave communication primitives that are used by the Linux ABI shim layer of SGXKernel:

Delegated calls. SGXKernel completely gets rid of OCalls by introducing DCalls (or delegated calls). DCalls are cross-enclave function calls that are issued by in-enclave threads running on (trusted) SGX virtual processors, and executed asynchronously by (untrusted) delegate threads. To process DCalls, one delegate thread is assigned to each SGX virtual processor; and multiple SGX virtual processors may be served by one delegate thread, depending on configuration. DCall requests are passed from a SGX virtual processor to its assigned delegate thread via a FIFO queue (which is essentially a cross-enclave ring buffer); so are the DCall responses, just in the opposite direction. After issuing a DCall request, the current in-enclave thread is blocked, and another runnable in-enclave thread is scheduled to run on the SGX virtual processor. Once the DCall response is received, the corresponding blocked in-enclave thread is set as runnable and ready to be rescheduled. All the arguments of DCalls that are output by the untrusted delegate threads are validated before passed to the trusted callers. DCalls do not trigger expensive enclave transitions and its throughput tends to increase with the number of in-enclave threads (see the next subsection for more about in-enclave threads).

I/O streams. While DCalls are much less expensive than OCalls, the asynchronous nature of DCalls limits the lower bound of their latency. Due to the ubiquitous of I/O operations, we consider it a top priority to provide efficient I/O operations. For this reason, SGXKernel implements I/O streams, which are the foundation of all I/O related system calls. The opening of I/O streams is initiated by the trusted half of SGXKernel via DCalls. The corresponding DCall handlers executing in delegate threads open the underlying OS file descriptors in non-blocking mode for the I/O streams and associate each I/O stream with a write buffer and/or a read buffer (which are cross-enclave ring buffers). The write buffers are flushed to the underlying OS files (or sockets) by the delegate threads as long as it is not empty; the read buffers are filled with data (prefetched when possible) by the delegate threads as long as it is not full. In this way, when an in-enclave thread attempts to perform write (or read) operations on a stream, it is likely that they can write to (or read from) the underlying cross-enclave ring buffer of the stream *immediately*. If the write (or read) buffer of the stream is not ready, the current in-enclave thread is blocked until any interesting I/O events of the stream occur.

Virtual interrupt signals. Like hardware interrupt signals to real CPUs, virtual interrupt signals draw the attention of SGX virtual processors or delegate threads to handle interesting events; for instance, the readiness of I/O streams, and the arrival of POSIX inter-process signals. Virtual interrupt signals can be emitted by delegate threads to SGX virtual processors, the latter of which periodically check for any pending signals. Similarly, virtual interrupt signals can also be emitted by virtual processors to delegate threads. In addition, virtual interrupt signals allow a much larger number of interrupt request (IRQ) lines than hardware interrupt signals; for instance, each open stream of SGXKernel is associated with a virtual IRQ line for SGX virtual processors and another for delegate threads. As virtual interrupt signals can be conceptually viewed as an array of bits, we implement them in SGXKernel using cross-enclave tree bitmaps, which allow setting and extracting bits concurrently and efficiently.

```
// Variables local to OS/SGX threads
__thread tcs_t* tcs; // Thread Control Structure (TCS) of a SGX thread
__thread volatile cputime_t will_preempt_at = CT_MAX; // Set by vproc

// Start a trusted virtual processor (vproc) on the current OS/SGX thread
void start_vproc() {
  // Normal enclave entry with aep_preempt as the AEP
  sgx_enter(ECALL_RUN, &&aep_preempt, tcs);
  return;
aep_preempt:
  cputime_t uptime = os_get_thread_cpu_time();
  if (uptime >= will_preempt_at) {
    // Nested enclave entry with aep_resume as the AEP
    sgx_enter(ECALL_PREEMPT, &&aep_resume, tcs);
  }
aep_resume:
  sgx_resume(tcs);
}
```

Listing 1: Simplified C++ code of the preemptive scheduling in SGXKernel: the *untrusted* part

```
// Variables local to SGX threads
__thread vproc* this_vproc = NULL; // Local virtual processor

void ecall_run() { // For ECALL_RUN
  assert(sgx_get_nested_ecall_depth() == 0);
  vproc::init_this_vproc(); // Initialize the local virtual processor
  inenclave_thread::init_default_threads(); // e.g., the main thread
  this_vproc->reschedule(false);
}

void ecall_preempt() { // For ECALL_PREEMPT
  assert(sgx_get_nested_ecall_depth() == 1);
  if (this_vproc->is_preemption_disabled()) return;
  this_vproc->reschedule(true);
}

void vproc::reschedule(bool is_preempt) {
  inenclave_thread *p = this->current(), *n = NULL;
  this->disable_preemption(); // Disable preemption for rescheduling
  this->update_thread_runtime(p); // Update runtime of current thread
  if (p->is_runnable()) this->runqueue->push(p); // Re-enqueue
  do { // Find the next runnable thread
    this->handle_events_and_signals(); // e.g. DCall completion
  } while ((n = this->runqueue->pop()) != NULL);
  will_preempt_at = this->assign_cputime_for(n); // Assign CPU time
  n->switch_to(is_preempt); // Do context switching
  this->enable_preemption(); // Enable preemption after rescheduling
}

void inenclave_thread::entry_point() {
  this_vproc->enable_preemption(); // Enable preemption after reschedule
  void* retval = this->func(this->data); // Run this thread
  this->exit(retval); // Exit with a statu code
  this_vproc->reschedule(0); // Schedule another thread to run
}

void inenclave_thread::switch_to(bool is_preempt) {
  inenclave_thread *p = this_vproc->current(), *n = this;
  if (is_preempt) {
    ssa* ssa = this_vproc->sgx_thread->ssa;
    p->setcontext_full(ssa); // Save the CPU state of p from SSA
    n->getcontext_full(ssa); // Restore the CPU state of n to SSA
    return;
  }
  if (!p->setcontext()) // Save the current CPU state for p
    n->getcontext(); // Restore the CPU state of n
}
```

Listing 2: Simplified C++ code of the preemptive scheduling in SGXKernel: the *trusted* part

3.4 Preemptible In-Enclave Multi-Threading

SGXKernel implements a multi-threading mechanism (almost) entirely inside enclaves. The salient features of this *in-enclave multi-threading* are:

M-on-N mapping. The in-enclave multi-threading maps M in-enclave threads to N SGX virtual processors, each of which runs on a SGX thread. The number of SGX virtual processors N is no more than the number of CPU cores, and each of them is pinned to a specific CPU core to maximize locality. This M-on-N threading model combines the best of both 1-on-1 and M-on-1 model: allowing true parallelism among in-enclave threads on different virtual processors and achieving fast context switching between in-enclave threads on the same SGX virtual processor.

Fast context switching. Most of the context switches in SGXKernel happen when the running in-enclave thread voluntarily gives up the CPU, and these non-preemptive context switches are extremely fast. Non-preemptive context switches occur, for example, when a `DCall` is issued, the running thread is put to sleep, and another runnable thread is scheduled; or when a system call like `sched_yield` or `futex` is invoked. Regardless of the causes, non-preemptive context switches in SGXKernel are always initiated by normal function calls; according to Linux calling convention, only a handful of general CPU registers are required to save (and later restore). In addition, the thread schedulers are local to SGX virtual processors, maintaining its own per-processor run queue. Run queues are rebalanced by periodically sending threads from busy run queues to idle ones. Since each virtual processor makes rescheduling decisions for itself, the rescheduling algorithm is fast. With fewer registers to maintain and the fast rescheduling algorithm, the non-preemptive context switching in SGXKernel is extremely fast.

Preemptive scheduling. Preemptive scheduling, which avoids starvation among threads, improves program responsiveness, and frees programmers from crafting application-specific scheduling logic, is generally considered superior to cooperative scheduling. Thus, all modern OSes support preemptive scheduling for OS threads. However, the situation is less fortunate for user-level threads: either kernel modifications are required[7], or preemptive scheduling is replaced with simpler cooperative scheduling[18]. As for our in-enclave threads, one would expect preemptive-scheduling is even harder to realize as there are more restrictions for code in enclave mode than in user mode. Interestingly, we find the opposite is true: preemptive scheduling can be achieved simply and efficiently by leveraging the full strength of SGX instructions.

Our key observation is that while hardware interrupts (e.g., periodic timer interrupts, which OSes rely on to do preemptive scheduling) are transparent to normal OS threads, this is not the case for SGX threads. As we briefly mentioned in section 2, whenever a running enclave is interrupted, an asynchronous enclave exit (AEX) is performed by the CPU, which mainly consists of two steps: 1) saving the CPU state of the interrupted enclave to a predefined data area (called State Save Area, or SSA) inside the enclave, and 2) transferring the control to a pre-specified instruction address (called Asynchronous Exit handler Pointer, or AEP) outside the enclave, replacing CPU registers with synthetic values. These two properties of AEX are the basis of enabling preemptive scheduling inside enclaves.

The preemptive scheduling for in-enclave threads in SGXKernel proceeds as following (see listing 1 and 2): whenever an AEX happens (most likely, a timer interrupt), the AEX handler (line u10) checks whether the assigned runtime of the current in-enclave thread has exhausted. If not, just resume the execution of interrupted enclave (line u16); otherwise, enter the enclave again by evoking a nested `ECall` (line u14) to reschedule for the next runnable thread (line t17). If the context switching is initiated for preemption (line t39), save the CPU state of the previous thread by copying from the SSA, and then replace the SSA with the state of the next thread. Finally, exit the nested `ECall` and resume the first `ECall` (line u16). Since the SSA has been replaced with the state of the next thread, resuming the enclave executes the next thread.

Table 1: SGXKernel's implementation status of the mostly used Linux system calls (a total of 130 system calls with unweighted popularities ≥ 0.1)

Linux System Calls			
Impl.?	Pct.	Comments on implementation	Examples
Yes	15%	Implement mainly inside the enclave	`clone`, `dup`, `sched_yield`
Yes	18%	Implement inside and outside the enclave	`open`, `socket`, `read`, `epoll`
Yes	33%	Implement mainly outside the enclave	`getpid`, `stat`, `lchown`, `kill`
No	26%	Will implement in the future	`openat`, `readv`, `poll`, `recvmsg`
No	8%	Will NOT implement	`vfork`, `execve`, `shmdt`

4 EVALUATION

In this section, we present a thorough evaluation of SGXKernel. Specifically, we aim to answer the following questions:

1. To what extent does SGXKernel support system APIs inside enclaves?
2. Is the switch-less design of SGXKernel effective in enabling efficient system calls inside enclaves?
3. How is the performance of system-intensive applications running on SGXKernel compared to Linux?

We compare SGXKernel with Linux and/or Graphene-SGX in a series of benchmarks. Graphene-SGX is a state-of-the-art library OS that supports SGX. For the time being, Graphene-SGX is the only open-source library OS for SGX and is in active development. All results on Graphene-SGX reported in this paper were acquired on the latest version (commit `fba92d`, submitted on Jan 14th, 2017) at the time of writing.

4.1 The Coverage of System APIs

One major motivation of constructing library OSes for SGX is to facilitate the development of SGX applications by giving programmers access to the system APIs, including system calls and the C

Table 2: The supported APIs of the C standard library inside enclaves. The popularity means the probability of an API being used for at least once by an application.

	Number of Supported APIs in `libc`		
Popularities	Linux (baseline)	SGX SDK	SGXKernel
≥ 0.8	17 (100%)	8 (47%)	**17 (100%)**
≥ 0.6	46 (100%)	17 (37%)	**45 (98%)**
≥ 0.4	122 (100%)	27 (22%)	**118 (97%)**
≥ 0.2	234 (100%)	44 (19%)	**219 (94%)**

standard library. For this reason, we report some statistics on the system APIs supported by SGXKernel.

Currently, we have selectively implemented 102 system calls (out of >300 Linux system calls) according to the importance of system calls and the demands of porting existing applications. Our implementation status of the mostly used Linux system calls is summarized in table 1. There is a total of 130 Linux system calls with an unweighted popularity[20] ≥0.1 (meaning the system call is used by at least one out of ten applications). Among these 130 Linux system calls, 66% have been implemented in our prototype implementation of SGXKernel; 26% will be supported in future versions; and, the remaining 8% of the system calls (e.g., `vfork`, execve, etc.) that are related to process creation and memory sharing among processes are not to be supported since SGXKernel aims at running *single-process* applications inside enclaves, and SGX is designed to isolate rather than share trusted memory.

While it may seem that our current coverage of Linux system calls is only modest, a large number of APIs in C standard library have been made accessible to SGX applications thanks to SGXKernel's support of system calls. As shown in table 2, there are 234 APIs from C standard library with unweighted popularities ≥0.2 (meaning the API is used by at least two out of ten applications). While the `tlibc` that is shipped with Intel SGX SDK covers only 19% of these most popular APIs, *the SGXKernel's port of* `musl` *achieves a coverage of 94%.* This can already enable a wide range of possible applications inside enclaves.

4.2 Microbenchmarks

This subsection and the next present experimental results of SGXKernel. All experiments were conducted on an Intel Core i7-6670HQ CPU at 2.6GHz with 8 hyper-threads (2 per core) and 6MB cache. The machine has a total of 32GB main memory and runs on an Ubuntu 14.04 server with Intel SGX SDK 1.7 installed. All data points are averaged on 10 runs. If not stated otherwise, SGXKernel is configured to run with a single SGX virtual processor and a single delegate thread.

System call throughput. In figure 5, we present the throughput of three system calls that are representative of three different classes. The first class of system calls is those that can be trivially cached or simulated inside the library OS, e.g. `getpid` and `getppid`. For this class of system calls, both Graphene-SGX and SGXKernel outperform Linux. In the case of `getppid`, both library OSes perform exactly the same as it is simply returning an integer value. The second class of system calls are those that can be

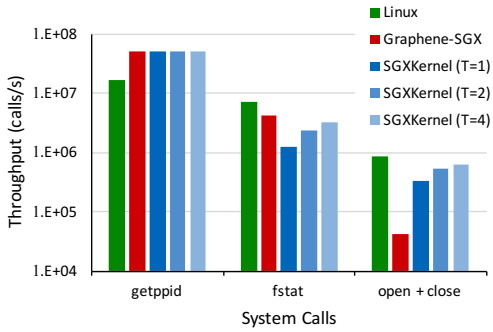


Figure 5: The throughput of system calls on different (library) OSes. *T* is the number of in-enclave threads running in SGXKernel.

handled (largely) inside Graphene-SGX but not SGXKernel. One such example is `fstat`: while `fstat` can be handled by the virtual file system of Graphene-SGX, it must be redirected to the host OS in SGXKernel. Naturally, in situations like this, the performance of Graphene-SGX is better than SGXKernel, and comparable to Linux. Note that the good performance for the second class of system calls in Graphene-SGX are made possible at the price of duplicated layer of virtual file systems inside the library OS. Finally, the third class of system calls is the ones that must interact with the host OS. Take open and `close` as an example: both Graphene-SGX and SGXKernel must turn to the host OS for opening and closing I/O stream on the requested file. In such cases where the library OS must interact with the host OS, SGXKernel can significantly outperform Graphene-SGX since the overhead of `DCalls`, which doesn't trigger any enclave transitions, are much smaller than `OCalls`.

An extra performance edge can be gained by `DCalls` if some useful work can be done at the trusted half of SGXKernel while waiting for the completion of `DCalls`. This is exactly the situation when multiple in-enclave threads are issuing `DCalls` simultaneously. As shown in figure 5, the throughput of `fstat` on SGXKernel increases significantly with the number of in-enclave threads. The throughput of open+`close` follows a similar pattern, showing a performance improvement with more in-enclave threads.

Overall, *the system call throughput of SGXKernel is* 0.3×-15× *that of Graphene-SGX* depending on the classes of system calls.

I/O throughput. Figure 6 and 7 show the I/O throughput of sequential file write and read, respectively. In these two experiments, we limit the size of the target file so that the data of the entire file can fit into the disk cache inside Linux kernel. This ensures that the I/O throughput of the disk is not a bottleneck. The maximum write throughput of Graphene-SGX, which relies on `OCalls`, is only 19% as much as that of Linux, and the read throughput is only 27%. In contrast, SGXKernel achieves over 70% as much as Linux does for large writes and reads, and even outperforms Linux for small writes and reads (since the ring buffer of SGXKernel can combine multiple small writes/reads into large OS writes/reads). Overall, *SGXKernel outperforms Graphene-SGX in terms of I/O throughput by a factor ranging from* 2.8× *to* 56× *depending on the I/O sizes.*

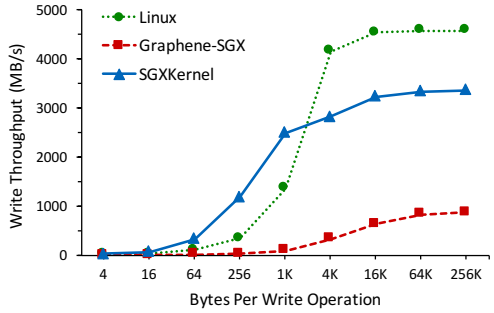


Figure 6: The throughput of sequential file write on different (library) OSes.

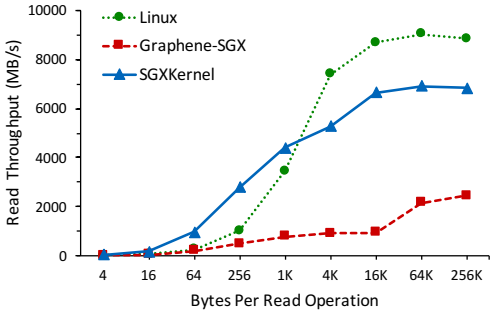


Figure 7: The throughput of sequential file read on different (library) OSes.

Table 3: The cost of context switching among threads on different (library) OSes (in microseconds).

Linux (baseline)	Graphene-SGX	SGXKernel
2.74 (0%)	4.62 (+69%)	**0.63 (-77%)**

The overhead of context switching between threads. Table 3 shows the overhead of context switching among threads on different (library) OSes. The results are collected by leveraging mutexes and conditional variables to force repeated context switching between two threads. These threads are normal OS threads for Linux and Graphene-SGX, and in-enclave threads for SGXKernel. Graphene-SGX, which performs threads scheduling and synchronization by invoking `futex` system calls of the host OS (Linux) via `OCalls`, incurs an additional 69% overhead for context switching compared to Linux. In contrast, *SGXKernel uses 77% less time than Linux to do context switches, which is 7× faster than Graphene-SGX.* This performance improvement is achieved thanks to the efficient in-enclave multi-threading mechanism of SGXKernel.

4.3 Application Benchmark

We present our results on running Redis, an in-memory data-structure store, on top of SGXKernel inside enclaves. We choose

Table 4: Redis's performance on different (library) oses.

	Request throughput (k.req/s)	
Tests	Linux (baseline)	SGXKernel (overhead)
`ping_inline`	188 (0%)	**139 (26%)**
`ping_bulk`	183 (0%)	**139 (24%)**
`set`	191 (0%)	**145 (24%)**
`get`	186 (0%)	**145 (22%)**
`sadd`	196 (0%)	**145 (26%)**
`spop`	194 (0%)	**151 (22%)**
`lpush`	199 (0%)	**145 (27%)**
`lrange`	81 (0%)	**70 (14%)**
Average	177 (0%)	**135 (23%)**

Redis[5] as our target application for two reasons: 1) Redis has a reputation of supporting high request throughput, thus generating heavy load on the underlying (library) OS; 2) the codebase of Redis is of a modest size (within 100,000 LoC), thus easier to debug and diagnose when porting with SGXKernel. Some modifications (less than 50 LoC) are made to the source code of Redis in order to disable features that SGXKernel currently doesn't support. No features of Redis are fundamentally difficult to be fully supported by SGXKernel. We just disable some features for the convenience of testing.

We compared Redis between running on Linux and SGXKernel. We did not include Graphene-SGX in this benchmark because we found a Redis server hosted on Graphene-SGX did not respond to any requests. While we were not able to track the root of this bug and run this test on Graphane-SGX, we have some indirect evidence (e.g., the poor file I/O throughput of Graphene-SGX shown in the previous subsection) that suggests a poor performance of Redis hosted on Graphene-SGX due to excessive `OCalls` triggered by frequent system calls like `accept`, `read`, `write`, `epoll`, etc.

We use `redis-benchmark` utility to generate the workload. The number of parallel connections is set to 50. The data size of each `SET`/`GET` value is 512 bytes. The results are summarized in table 4. *Redis running on SGXKernel shows a consistent throughput on various tests, with a performance degradation of 23% on average compared to Linux.* Taking into account the enhanced security provided by SGX, we believe a performance overhead of this level can be acceptable in many scenarios.

While we only provide SGXKernel's macro-benchmark results on Redis, the performance degradation shown in this test can be seen as an "upper-bound" due to the fact that Redis is among the most system-intensive applications. We plan to port more complex applications with SGXKernel in the future.

5 RELATED WORK

Haven[9] and Graphene-SGX[2] are two state-of-the-art library OSes that have been ported to SGX. Haven is based on Drawbridge[15] library OS, and Graphene-SGX is based on Graphene[19] library OS. Both Drawbridge and Graphene are intended to realise the benefits of virtual machines, thus prioritizing application compatibility and security isolation.

SGXKernel is radically different from Haven and Graphene-SGX. First, one primary goal of Haven and Graphene-SGX is to virtualize the resources of the host OS. To this end, the two library OSes duplicate a major portion of OS functionalities, e.g. virtual file system and process management. In addition, both of them feature a security monitor that isolates instances of the library OS from each other and from the host OS. This amounts to a larger TCB and a memory footprint. In contrast, the motivation of SGXKernel is to enable efficient system calls for SGX applications, thus its complexity can be greatly reduced. Second, both Haven and Graphene-SGX are built on the assumption that system calls from within the library OS to the host OS (via `OCalls`) are relatively cheap. But as shown in this paper, this assumption can only be held in user mode, not enclave mode. In SGXKernel, we use `DCalls` to request services from the host OS *asynchronously*. Third, both Haven and Graphene-SGX reuse the multi-threading mechanism of the host OS in order to reduce the complexity of the library OSes. However, as we have demonstrated, the context switching and synchronisation primitives of SGX threads are far more expensive than that of OS threads, thus the in-enclave multi-threading mechanism of SGXKernel is highly desirable.

FlexSC[18] proposes asynchronous system calls, which does not require user-privillged transitions. By getting rid of these transitions, FlexSC improves the CPU locality for system-intesive applications. FlexSC requires extensive support from both user-level libraries and OS kernel, yet only improves application performance to a limited degree. Thus, FlexSC can barely justify all the extra complexity introduced to both user and kernel code. While similar to FlexSC in spirit, SGXKernel leverages asynchronous communication to eliminate enclave transitions, which, as demonstrated in this paper, can improve the performance of system-intensive SGX applications by a one order of magnitude. Therefore, the techniques described in this paper are more likely to be adopted in practice.

Scone[8], which is a concurrent work to ours, designs a SGX-secured container mechanism for Docker. The library OS layer of Scone, which is similar to FlexSC and our work in spirit, also features asynchronous system calls and user-level multi-threading. While the asynchronous system calls in Scone requires loading a special kernel module and running multiple kernel threads, the asynchronous cross-enclave communication mechanism in SGXKernel is completely implemented at user level. Moreover, the multi-threading mechanism in the library OS of Scone does not support preemptive scheduling, which can lead to unfairness or even starvation among threads.

6 CONCLUSION

In this paper, we revisit the library OS approach in a new setting—Intel SGX. To address the issue of expensive enclave transitions, we propose a switch-less architecture for SGXKernel, which incorporates two novel ideas: asynchronous cross-enclave communication and preemptible in-enclave multi-threading. The experimental results demonstrate the effectiveness of our approach on improving the performance of system calls inside enclaves.

ACKNOWLEDGMENTS

Special thanks to Mona Vij at Intel Coporation for her valuable insights and concrete suggestions. We would also like to thank the anonymous reviewers for their useful comments.

This work was supported by NSFC(91646202), the National High-tech R&D Program of China(SS2015AA020102), Research/Project 2017YB142 supported by Ministry of Education of The People's Republic of China Research Center for Online Education Qtone Education Group Online Education Fund, the 1000-Talent program, Tsinghua University Initiative Scientific Research Program.

REFERENCES

[1] The GNU C Library (glibc). https://www.gnu.org/software/libc/
[2] Graphene-SGX Library OS. https://github.com/oscarlab/graphene
[3] Intel(R) Software Guard Extensions SDK. https://software.intel.com/en-us/sgx-sdk/documentation
[4] musl libc. http://www.musl-libc.org
[5] Redis, an in-memory data structure store. http://redis.io/
[6] Security Vulnerabilities of Linux Kernel. https://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/Linux-Linux-Kernel.html
[7] Thomas E. Anderson, Brian N. Bershad, Edward D. Lazowska, and Henry M. Levy. 1992. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. *ACM Trans. Comput. Syst.* 10, 1 (1992), 53–79.
[8] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David M. Eyers, Rüdiger Kapitza, Peter R. Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* 689–703.
[9] Andrew Baumann, Marcus Peinado, and Galen C. Hunt. 2015. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.* 33, 3 (2015), 8:1–8:26.
[10] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter R. Pietzuch, and Rüdiger Kapitza. 2016. SecureKeeper: Confidential ZooKeeper using Intel SGX. In *Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 - 16, 2016.* 14.
[11] Shay Gueron. 2016. A Memory Encryption Engine Suitable for General Purpose Processors. *IACR Cryptology ePrint Archive* 2016 (2016), 204.
[12] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2009. seL4: formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP 2009) Big Sky, Montana, USA.* 207–220.
[13] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. 2016. Oblivious Multi-Party Machine Learning on Trusted Processors. In *25th USENIX Security Symposium, USENIX Security 16, Austin, TX, USA, August 10-12, 2016.* 619–636.
[14] Nicolae Paladi and Christian Gehrmann. 2017. TruSDN: Bootstrapping Trust in Cloud Network Infrastructure. *CoRR* abs/1702.04143 (2017). http://arxiv.org/abs/1702.04143
[15] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the library OS from the top down. In *16th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2011, Newport Beach, CA, USA, March 5-11, 2011.* 291–304.
[16] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. 2015. VC3: Trustworthy Data Analytics in the Cloud Using SGX. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* 38–54.
[17] Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.* 44, 3 (2012), 11:1–11:46.
[18] Livio Soares and Michael Stumm. 2010. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2010), Vancouver, BC, Canada.* 33–46.
[19] 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. 2014. Cooperation and security isolation of library OSes for multi-process applications. In *Ninth Eurosys Conference 2014, EuroSys 2014, Amsterdam, The Netherlands, April 13-16, 2014.* 9:1–9:14.
[20] Chia-che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. 2016. A study of modern Linux API usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems, EuroSys 2016, London, United Kingdom, April 18-21, 2016.* 16:1–16:16.