Graphene-SGX: A Practical Library OS for Unmodified Applications on SGX

Chia-Che Tsai
Stony Brook University

Donald E. Porter
University of North Carolina at Chapel Hill and Fortanix

Mona Vij
Intel Corporation

Abstract

Intel SGX hardware enables applications to protect themselves from potentially-malicious OSes or hypervisors. In cloud computing and other systems, many users and applications could benefit from SGX. Unfortunately, current applications will not work out-of-the-box on SGX. Although previous work has shown that a library OS can execute unmodified applications on SGX, a belief has developed that a library OS will be ruinous for performance and TCB size, making application code modification an implicit prerequisite to adopting SGX.

This paper demonstrates that these concerns are exaggerated, and that a fully-featured library OS can rapidly deploy unmodified applications on SGX with overheads comparable to applications modified to use "shim" layers. We present a port of Graphene to SGX, as well as a number of improvements to make the security benefits of SGX more usable, such as integrity support for dynamically-loaded libraries, and secure multi-process support. Graphene-SGX supports a wide range of unmodified applications, including Apache, GCC, and the R interpreter. The performance overheads of Graphene-SGX range from matching a Linux process to less than 2× in most single-process cases; these overheads are largely attributable to current SGX hardware or missed opportunities to optimize Graphene internals, and are not necessarily fundamental to leaving the application unmodified. Graphene-SGX is open-source and has been used concurrently by other groups for SGX research.

1 Introduction

Intel SGX introduces a number of essential hardware features that allow an application to protect itself from the host OS, hypervisor, BIOS, and other software. With SGX, part or all of an application can run in an *enclave*. Enclave features include confidentiality and integrity protection for the enclave's virtual address space; restricting control flow into well-defined entry points for an enclave; integrity checking memory contents at start time; and remote attestation. SGX is particularly appealing in cloud computing, as users might not fully trust the cloud provider. That said, for any sufficiently-sensitive application, using SGX may be prudent, even within one administrative domain, as the security track record of commodity operating systems is not without blemish. Thus, a significant number of users would benefit from running applications on SGX as soon as possible.

Unfortunately, applications do not "just work" on SGX. SGX imposes a number of restrictions on enclave code that require application changes or a layer of indirection. Some of these restrictions are motivated by security, such as disallowing system calls inside of an enclave, so that system call results can be sanitized by *shielding code* in the enclave before use. Our experience with supporting a rich array of applications on SGX, including web servers, language runtimes, and command-line programs, is that a number of software components, orthogonal to the primary functionality of the application, rely on faithful emulation of arcane Linux system call semantics, such as `mmap` and `futex`; any SGX wrapper library must either reproduce these semantics, or large swaths of code unrelated to security must be replaced. Although this paper focuses on SGX, we note that a number of vendors are developing similar, but not identical, hardware protection mechanisms, including IBM's SecureBlue++ [16] and AMD SEV [27]—each with different idiosyncrasies. Thus, the need to adapt applications to use hardware security features will only increase in the near term.

As a result, there is an increasingly widespread belief that adopting SGX necessarily involves significant code changes to applications. Although Haven [15] showed that a library OS could run unmodified applications on SGX, this work pre-dated availability of SGX hardware. Since then, several papers have argued that the library OS approach is impractical for SGX, both in performance overhead and trusted computing base (TCB) bloat, and that one must instead refactor one's application for SGX. For instance, a feasibility analysis in the SCONE paper concludes that "On average, the library OS increases the TCB size by 5×, the service latency by 4×, and halves the service throughput" [14]. Shinde et al. [49] argue that using a library OS, including libc, increases TCB size by two orders of magnitude over a thin wrapper.

This paper demonstrates that these concerns are greatly exaggerated: one can use a library OS to quickly deploy applications in SGX, gaining immediate security benefits without crippling performance cost or TCB

For my initial bringup, have the Kernel start in Single-user mode... no user-mode processes running. Then, try to bringup the first few SGX processes (like a static return rand() — a program that's even simpler than Hello World). I'll likely need my own hardware for initial bringup as I'd prefer to not rely on a network stack to talk to a cloud-based rig.

bloat. We present a port of the Graphene library OS [52] to SGX, called Graphene-SGX, and show that the performance overheads are comparable to the range of overheads presented in SCONE; the authors of Panoply also note that Graphene-SGX is actually 5-10% faster than Panoply [49]. Arguments about TCB size are more nuanced, and a significant amount of the discrepancies arise when comparing incidental choices like libc implementation (e.g., musl vs. glibc). Graphene, not including libc, adds 53 kLoC to the application's TCB, which is comparable to Panoply's 20 kLoC or SCONE's 97 kLoC. Our position is that the primary reduction to TCB comes from either compiling out unused library functionality, as in a unikernel [38] and measured by our prior work [53]; or further partitioning an application into multiple enclaves with fewer OS requirements. When one normalizes for functionality required by the code in the enclave, the design choice between a library OS or a smaller shim does not have a significant impact on TCB size.

To be clear, SGX-specific coding has benefits, but we must not let the perfect be the enemy of the good. For example, privilege separating a complex application into multiple enclaves may be a good idea for security [40, 44, 49], and replacing particularly expensive operations can improve performance on SGX. The goal of Graphene is to bring up rich applications on SGX quickly, and then let developers optimize code or reduce the TCB as needed.

Graphene-SGX runs unmodified Linux binaries on SGX; to this end, this paper also contributes a number of usability enhancements, including integrity support for dynamically-loaded libraries, enclave-level forking, and secure inter-process communication (IPC). Users need only configure features and cryptographically sign the configuration.

Graphene-SGX is also useful as a tool to accelerate SGX research. Graphene-SGX has been open-sourced since June 2016[1]. Although our focus is unmodified applications, Graphene-SGX can also run smaller pieces of code in an enclave, as in a partitioned application. Several papers already compared against or extended Graphene-SGX [28, 43, 49] and we are aware of ongoing projects using Graphene-SGX.

The contributions of this paper are:

- A framework, called Graphene-SGX, to isolate unmodified, Linux applications in enclaves.
- Several usability enhancements for SGX, including dynamic loading, `fork`, and IPC.
- A thorough evaluation of the performance of unmodified applications on Graphene-SGX, indicating that the costs of a feature-rich library OS on SGX are in-band with purportedly lighter-weight solutions that require application changes. For example, lighttpd throughput and latency on Graphene-SGX are comparable to a Linux process. Overheads are generally under 2× (cf. SCONE overheads up to 1.6× on comparable workloads). In a few cases, Graphene-SGX overheads are higher, but these are internal to the library OS or fundamental to enclave limitations, not because the application is unmodified.

[1] Available at https://github.com/oscarlab/graphene

2 Background

This section summarizes SGX, and current design points for running or porting applications on SGX.

2.1 Software Guard Extensions (SGX)

The primary SGX abstraction is an *enclave*: an isolated execution environment within the virtual address space of a process. The code and data in enclave memory do not leave the CPU package unencrypted; when memory contents are read back into cache, the CPU decrypts the contents, and checks the integrity of cache lines and the virtual-to-physical mapping. SGX also cryptographically measures the integrity of enclaves at start-up, and provide attestation to remote systems or other enclaves.

SGX enables a threat model where one only trusts the Intel CPUs and the code running in the enclave(s). SGX protects applications from three different types of attacks on the same host, which are summarized in Figure 1: untrusted application code inside the same process but outside the enclave; operating systems, hypervisors, and other system software; other applications on the same host; and off-chip hardware. A SGX enclave can also trust a remote service or enclave, and be trusted after inter-platform attestation [13].

2.2 SGX Software Design Space

This subsection summarizes the principal design choices facing any framework for running applications on SGX. We explain the decisions in recent systems for SGX applications, and the trade-offs in this space.

How much functionality to pull into the enclave? At one extreme, a library OS like Haven [15] pulls most of the application-supporting code of the OS into the enclave. On the other extreme, thin "shim" layers, like SCONE [14] and Panoply [49] wrap an API layer such as the system call table. Pulling more code into the enclave increases the size of the TCB, but can reduce the size and complexity of the interface, and attack surface, between the enclave and the untrusted OS.

The impact of this choice on performance largely depends on two issues. First, entering or exiting the enclave is expensive; if the division of labor reduces enclave border crossings, it will improve performance. The second is the size of the Enclave Page Cache (EPC), limited to 128MB on version 1 of SGX. If a large support-

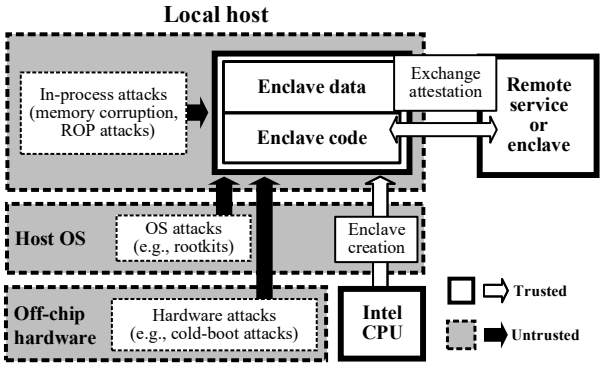


Figure 1: The threat model of SGX. SGX protects applications from three types of attacks: in-process attacks from outside of the enclave, attacks from OS or hypervisor, and attacks from off-chip hardware.

ing framework tips the application's working set size past this mark, the enclave will incur expensive swapping.

Shielding complexity. SGX hardware can isolate an application from an untrusted OS, but SGX alone can't protect an application that requires functionality from the OS. *Iago attacks* [18] are semantic attacks from the untrusted OS against the application, where an unchecked system call return value or effect compromises the application. Iago attacks can be subtle and hard to comprehensively detect, at least with the current POSIX or Linux system call table interfaces.

Thus, any SGX framework must provide some *shielding* support, to validate or reject inputs from the untrusted OS. The complexity of shielding is directly related to the interface complexity: inasmuch as a library OS or shim can reduce the size or complexity of the enclave API, the risks of a successful Iago attack are reduced.

Application code complexity. Common example applications for SGX in the literature amount to a simple network service running a TLS library in the enclave, putting minimal demands on a shim layer. Even modestly complex applications, such as the R runtime and a simple analytics package, require dozens of system calls providing wide-ranging functionality, including `fork` and `execve`. For these applications, the options for the user or developer become: (1) modifying the application to require less of the runtime; (2) opening and shielding more interfaces to the untrusted OS; (3) pulling more functionality into a shim or a library OS. The goal of this paper is to provide an efficient baseline, based on (3), so that users can quickly run applications on SGX, and developers can explore (1) or (2) at their leisure.

Application partitioning. An application can have multiple enclaves, or put less important functionality outside of the enclave. For instance, a web server can keep cryptographic keys in an enclave, but still allow client requests to be serviced outside of the enclave. Similarly, a privilege-separated or multi-principal application might create a separate enclave for each privilege level.

This level of analysis is application-specific, and beyond the focus of this paper. However, partitioning a complex application into multiple enclaves can be good for security. In support of this goal, Graphene-SGX can run smaller pieces of code, such as a library, in an enclave, as well as coordinate shared state across enclaves.

3 Design Overview

This section discusses the threat model, how Graphene-SGX defends against attacks from the untrusted OS, and how users configure policies for defenses.

3.1 Threat Model

Graphene-SGX follows a typical threat model for SGX applications. The following components are untrusted: (1) hardware outside of the Intel CPU package(s), (2) the OS, hypervisor, and other system software, (3) other applications executing on the same host, including unrelated enclaves, and (4) user-space components that reside in the application process but outside the enclave. Our design only trusts the CPUs and any code running inside the enclave, including the library OS, the unmodified application, and its supporting libraries.

We also trust `aesmd`, an enclave provided by the Intel's SGX SDK, which verifies attributes in the enclave signature and approves the enclave creation. Currently, any framework that uses SGX for remote attestation must trust `aesmd`. Graphene-SGX uses, but does not trust, the Intel SGX kernel driver. Other than `aesmd` and the driver, Graphene-SGX does not use or trust any part of the SDK.

Denial of service, side channels, and controlled-channel attacks [54] are vulnerabilities common to all SGX frameworks, and are beyond the scope of this work.

3.2 User Policy Configuration

Before an application is first executed using Graphene-SGX, the user must make certain policy decisions. Our goal is to balance policy expressiveness with usability.

As with Graphene and several other systems, each application requires a *manifest* to specify which resources the application is allowed to use, including a unioned, chroot-style view of the file system (comparable to aufs), and a set of iptables-style network rules. In Graphene, a program cannot access any resources not declared in the manifest. The original intention of the manifest was to protect the host: a reference monitor can easily identify the resources an application might use, and reject an application with a problematic manifest.

In Graphene-SGX, the manifest is extended to protect the application from the host file system. Specifically,

Graphene is the first to demonstrate how a previously shared library needs to be shared using enclaves.

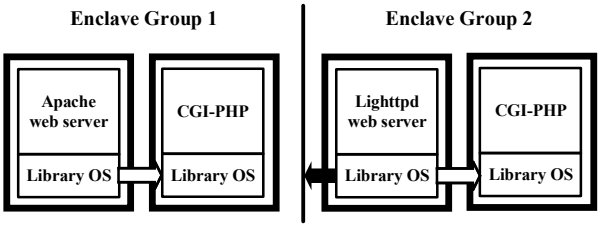


Figure 2: Two enclave groups, one running Apache and the other running Lighttpd, each creates a child enclave running CGI-PHP. Graphene-SGX distinguishes the child enclaves in different enclave groups.

Why manifests? Do I need them?

the manifest can specify secure hashes (using SHA-256) of trusted files (generally read-only, including dynamic libraries). As part of opening a file, Graphene-SGX verifies the integrity of trusted files by checking the secure hashes. A trusted file is only opened if the secure hash matches. The manifest can also specify files or directories that can be accessed but are not trusted, such as a write-only output file. Graphene-SGX includes a signing utility that hashes all trusted files and generates a signed manifest that can be used at runtime.

They replace a few system calls for dynamic allocation w/ a Mainifest file.

SGX requires that certain resources be specified at initialization time, including the number of threads, the maximum size of the enclave, and the starting virtual address of the enclave. Thus, we also extend the manifest syntax for the user to specify these options. Other security-sensitive manifest options inherited from Graphene, such as enabling debug output, are also protected as part of the signed manifest.

3.3 Multi-Process Applications

Graphene supports multi-process applications by running a separate library OS instance in each process [52]. Each library OS instance coordinates state via message passing. Graphene implements Linux multi-process abstractions in the user-space, including `fork`, `execve`, signals, and System V semaphores and message queues.

Graphene-SGX extends the multi-process support of Graphene to enclaves by running each process with a library OS instance in an enclave. For instance, `fork` creates a second enclave and copies the parent enclave's contents over message passing. We call a group of coordinating enclaves an *enclave group*. Figure 2 shows two mutually-untrusting enclave groups running on a host.

Because multi-process abstractions are implemented in enclaves, securing these abstractions from the OS is straightforward. Graphene-SGX adds: (1) the ability for enclaves to authenticate each other via local attestation, and thereby establish a secure channel, and (2) a mechanism to securely fork into a new enclave, adding the child to the enclave group (§4.3).

4 Shielding Linux Abstractions

This section discusses how Graphene-SGX implements and shields the Linux ABI for applications in enclaves.

4.1 Shielding Dynamic Loading

To run unmodified Linux binaries, Graphene-SGX implements dynamic loading and run-time linking. In a major Linux distribution like Ubuntu, more than 99% of binaries are dynamically linked [53]. Static linking is popular for SGX frameworks because it is simple and facilitates the use of hardware enclave measurements. Dynamic linking requires rooting trust in a dynamic loader, which must then measure the libraries. For Haven [15], the enclave measurement only verifies the integrity of Haven itself, and the same measurement applies to any application running on the same Haven binary.

Graphene-SGX extends the Haven model to generate a unique signature for any combination of executable and dynamically-linked libraries. Figure 3 shows the architecture and the dynamic-loading process of an enclave. Graphene-SGX starts with an untrusted Platform Adaption Layer (`pal-sgx`), which calls the SGX drivers to initialize the enclave. The initial state of an enclave, which determines the measurement then attested by the CPU, includes a shielding library (`libshield.so`), the executable to run, and a manifest file that specifies the attributes and loadable binaries in this enclave. The shielding library then loads a Linux library OS (`libLinux.so`) and the standard C libraries (`ld-linux-x86-64.so` and `libc.so`). After enclave initialization, the loader continues loading additional libraries, which are checked by the shielding libraries. If the SHA-256 hash does not match the manifest, the shield will refuse to open the libraries.

They use Intel's drivers...

This goes against their initial claims because they import a ton of code into the TCB.

To reiterate, a manifest includes integrity measurements of all components and is signed; this manifest is unique for each application and is measured as part of enclave initialization. This strategy does require trust in the Graphene (in-enclave) bootloader and shielding module to correctly load binaries according to the manifest and reject any errant binaries offered by the OS. This is no worse than the level trust placed in Haven's dynamic loader, but differentiates applications or even instances of the same application with different libraries.

Memory permissions. By default, the Linux linker format (ELF) often places code and linking data (e.g., jump targets) in the same page. It is common for a library to temporarily mark an executable pages as writable during linking, and then protect the page to be execute-only. This behavior is ubiquitous in current Linux shared libraries, but could be changed at compile time to pad writable sections onto separate pages.

The challenge on version 1 of SGX is that an application cannot revoke page permissions after the enclave

As yet, nobody has proposed modifying a compiler.

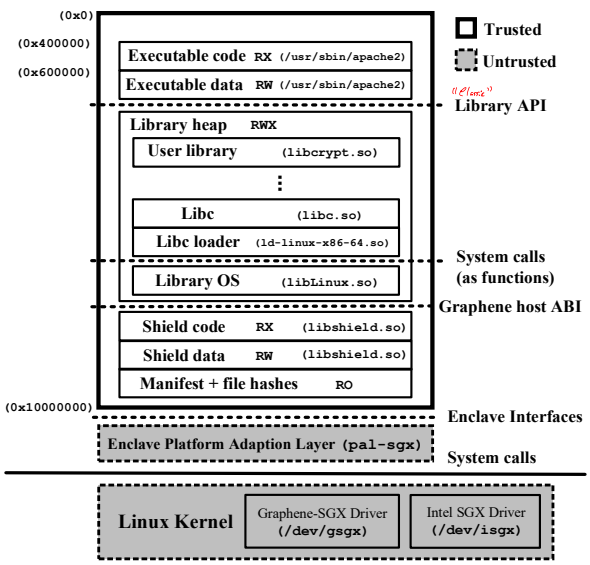


Figure 3: The Graphene-SGX architecture. The executable is position-dependent. The enclave includes an OS shield, a library OS, libc, and other user binaries.

starts. In order to support this ELF behavior, we currently map all enclave pages as readable, writable, and executable. This can lead to some security risks, such as code injection attacks in the enclave. In a few cases, this can also harm functionality; for instance, some Java VM implementations use page faults to synchronize threads. Version 2 of SGX [41] will support changing page protections, which Graphene-SGX will adopt in the future.

? I don't fully understand this bit yet. If we have PIC code, then we can put it anywhere.

Position-dependent executables. SGX requires that all enclave sizes be a power-of-two, and that the enclave starts at a virtual address aligned to the enclave size. Most Ubuntu Linux executables are compiled to be position-dependent, and typically start at address 0x400000. The challenge is that, to create an enclave that includes this address and is larger than 4MB, the enclave will necessarily need to include address zero.

An extreme corner case, but it's a result of a deep consciousness of security and an abundance of caution.

We see including address zero in the enclave as a net positive, but not strictly necessary, as we are reluctant to make strong claims in the presence of code that follows null pointers. Graphene-SGX can still mark this address as unmapped in an enclave. Thus, a null pointer will still result in a page fault. On the other hand, if address zero were outside of the enclave, there is a risk that the untrusted OS could map this address to dangerous data [10], undermining the integrity of the enclave.

4.2 Shielding Single-Process Abstractions

For a single-process application running on Graphene-SGX, most Linux system calls are serviced inside the enclave by the library OS. A Graphene-SGX enclave includes both the same library OS in "classic" Graphene,

Classes	**Safe**	Benign	DoS	Unsafe
Enter enclaves & threads	**2**	0	0	0
Clone enclaves & threads	**2**	0	0	0
File & directory access	**3**	0	0	2
Exit enclave	**1**	0	0	0
Network & RPC streams	**5**	2	0	0
Scheduling	**0**	1	1	0
Stream handles	**2**	2	1	0
Map untrusted memory	**2**	0	0	0
Miscellaneous	**1**	1	0	0
Total	**18**	6	2	2

//28

Table 1: 28 enclave interfaces, including *safe* (host behavior can be checked), *benign* (no harmful effects), *DoS* (may cause denial-of-service), and *unsafe* (potentially attacked by the host) interfaces.

Define API and ABI

that would also run on a Linux or FreeBSD picoprocess, as well as an SGX-specific platform adaptation layer (PAL), which implements 36 functions of the host ABI that the library OS is programmed against. This PAL funnels to a slightly smaller set of 28 interfaces which the enclave calls out to the untrusted OS (Table 1).

Not in my threat model

The evolution of the POSIX API and Linux system call table were not driven by a model of mutual distrust, and retrofitting protection onto this interface is challenging. Checkoway and Shachman [18] demonstrate the subtlety of detecting semantic attacks via the POSIX interface. Projects such as Sego [33] go to significant lengths, including modifying the untrusted OS, to validate OS behavior on subtle and idiosyncratic system calls, such as `mmap` or `getpid`.

The challenge in shielding an enclave interface is carefully defining the expected behavior of the untrusted system, and either validating the responses, or reasoning that any response cannot harm the application. By adding a layer of indirection under the library OS, we can define an enclave ABI that has more predictable semantics, which is, in turn, more easily checked at run-time. For instance, to read a file, Graphene-SGX requests that untrusted OS to map the file at an address outside the enclave, starting at an absolute offset in the file, with the exact size that the library OS needs for checking. After copying chunks of the file into the enclave, but before use, the contents can be hashed and checked against the manifest. This enclave interface limits the possible return values to one predictable answer, and thus reduces the space that the OS can explore to find attack vectors to the enclave. Many system calls are partially (e.g., `brk`) or wholly (e.g., `fcntl`), absorbed into the library OS, and do not need shielding from the untrusted OS.

Table 1 lists our 28 enclave interfaces, organized by risk. 18 interfaces are *safe* because responses from the OS are easily checked in the enclave. An example of a safe interface is `FILE_MAP`, which maps a file outside

the enclave, to copy it into the enclave for system calls like `mmap` or `read`, as discussed below. 6 interfaces are *benign*, which means, if a host violates the specification, the library OS can easily compensate or reject the response. An example of a benign interface is `STREAM_FLUSH`, which requests that data be sent over a network or to disk; cryptographic integrity checks on a file or network communication can detect when this operation is ignored by untrusted software.

Add to threat model

Like any SGX framework, Graphene-SGX does not guarantee liveness of enclave code: the OS can refuse to schedule the enclave threads. Two interfaces are susceptible to liveness issues (labeled *DoS*): `FUTEX_WAIT` and `STREAM_POLL`. In the example of `FUTEX_WAIT`, a blocking synchronization call may never return, violating liveness but not safety. A malicious OS could cause a futex wait to return prematurely; thus, synchronization code in the PAL must handle spurious wake-ups and either attempt to wait on the futex again, or spin in the enclave.

Add to threat model

Finally, only two interfaces, namely `FILE_STAT` and `DIR_READ`, are *unsafe*, because we do not protect integrity of file metadata. We leave this issue for future work, adopting one of several existing solutions [21].

File authentication. As with libraries and application binaries, configuration files and other integrity-sensitive data files can have SHA256 hashes listed in the signed manifest. At the first `open` to ones of the listed files, Graphene-SGX maps the whole file outside the enclave, copies the content in the enclave, divides into 64KB chunks, constructs a Merkle tree of the chunk hashes, and finally validates the whole-file hash against the manifest. In order to reduce enclave memory usage, Graphene-SGX does not cache the whole file after validating the hash, but keeps the Merkle tree to validate the untrusted input for subsequent, chunked reads. The Merkle tree is calculated using AES-128-GMAC.

Good, but we won't be doing this

Memory mappings. The current SGX hardware requires that the maximum enclave size be set at creation time. Thus, a Graphene-SGX manifest can specify how much heap space to reserve for the application, so that the enclave is sufficiently large. This heap space is also used to cache file contents.

Threading. Graphene-SGX currently uses a 1:1 threading model, whereas SCONE and Panoply support an m:n threading model. The issue is that SGX version 1 requires the maximum number of threads in the enclave to be specified at initialization time. We see this as a short-term problem, as SGX version 2 will support dynamic thread creation. We currently have users specify how many threads the application needs in the manifest.

This choice affect performance, as one may be able to use m:n threading and asynchronous calls at the enclave boundary to reduce the number of exits. This is a good idea we will probably implement in the future. Eleos [43] addresses this performance problem on unmodified Graphene-SGX with application-level changes to issue asynchronous system calls. The benefits of this optimization will probably be most clear in I/O-bound network services that receive many concurrent requests.

Exception handling. Graphene-SGX handles hardware exceptions triggered by memory faults, arithmetic errors, or illegal instructions in applications or the library OS. SGX does not allow exceptions to be delivered directly into the enclave. An exception interrupts enclave execution, saves register state on a thread-specific stack in the enclave, and returns to the untrusted OS. When SGX re-enters the enclave, the interrupted register state is then used by Graphene-SGX to reconstruct the exception, pass it to the library OS, and eventually deliver a signal to the application.

We note that the untrusted OS may deliberately trigger memory faults, by modifying the page tables, or not deliver the exceptions (denial of service). Direct exception delivery within an enclave is an opportunity to improve performance and security in future generations of SGX, as designed in Sanctum [19].

By handling exceptions inside the enclave, Graphene-SGXcan emulate instructions that are not supported by SGX, including `cpuid` and `rdtsc`. Use of these instructions will ultimately trap to a handler inside the enclave, to call out to the OS for actual values, which are treated as untrusted input and are checked.

4.3 Shielding Multi-Process Abstractions

Many Linux applications use multi-process abstractions, which are implemented using copy-on-write fork and in-kernel IPC abstractions. In SGX, the host OS is untrusted, and enclaves cannot share protected memory. Fortunately, Graphene implements multi-process support including `fork`, `execve`, signals, and a subset of System V IPC, using message passing instead of shared memory. Thus, Graphene-SGX implements multi-process abstractions in enclaves without major library OS changes. This subsection explains how Graphene-SGX protects multi-processing abstractions from an untrusted OS.

Process creation in Graphene-SGX is illustrated in Figure 4. When a process in Graphene-SGX forks into a new enclave, the parent and child will be running the same manifest and binaries, and will have the same measurements. Similar to the process creation in Graphene, the parent and child enclaves are connected with a pipe-like RPC stream, through the untrusted PAL. As part of initialization, the parent and child will exchange a session key over the unsecured RPC stream, using Diffie-Hellman. The parent and child use the CPU to generate attestation reports, which include a 512-bit field in the report to store a hash of the session key and a unique en-

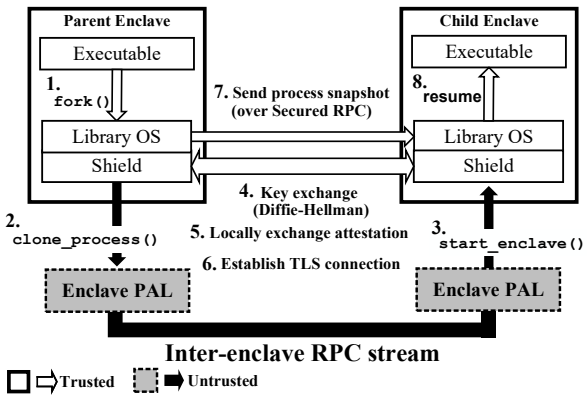


Figure 4: Process creation in Graphene-SGX. Numbers show the order of operations. When a process forks, Graphene-SGX creates a new, clean enclave on the untrusted host. Then the two enclaves exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process snapshot.

A very interesting way of implementing fork(). I'd like to do it differently, but I'm not sure my way will be better.

clave ID. The parent and child exchange these reports to authenticate each other. Unlike remote attestation, local attestation does not require use of Intel's authentication service (IAS). Once the parent and child have authenticated each other, the parent establishes a TLS connection over the RPC stream using the session key. The parent can then send a snapshot of itself over the TLS-secured RPC stream, and the snapshot is resumed in the child process, making it a clone of its parent. This mutual attestation and encryption strategy prevents a man-in-the-middle attack between the parent and child.

Once a parent enclave forks a child, by default, the child is fully trusted. To create a less trusted child, the parent would need to sanitize its snapshot, similar in spirit to the close-on-exec flag for file handles. For example, a pre-forked Apache web server may want to keep worker processes isolated from the master to limit a potential compromise of a worker process. Graphene-SGX inherits a limited API from Graphene, for applications to isolate themselves from untrusted child processes, but developers are responsible for purging confidential information before isolation.

Supporting `execve`. Unlike `fork`, `execve` starts a process with a specific executable, often different from the caller. When a thread calls `execve` in Graphene-SGX, the library OS migrates the thread to a new process, with file handles being inherited. Although the child does not inherit a snapshot from its parent, it can still compromise the parent by exploiting potential vulnerabilities in handling RPC, which are not internally shielded. In other words, Graphene-SGX is not designed to share library OS-internal with untrusted children. Thus, Graphene-SGX restricts `execve` to only launch trusted executables, which are specified in the manifest.

Inter-process communication. After process creation, parent and child processes will cooperate through shared abstractions, such as signals or System V message queues, via RPC messages. While messages are being exchanged between enclaves, they are encrypted, ensuring that these abstraction are protected from the OS.

The libraries do a lot of great work

5 Evaluation

Graphene-SGX is designed to be general-purpose, supporting a broad range of server and command-line applications. We thus evaluate performance overheads of unmodified Linux applications, using binaries from an Ubuntu installation. Depending on the workload, we measure application throughput or latency.

In order to differentiate SGX-specific overheads from Graphene overheads, we use both Linux processes and Graphene on a Linux host without SGX as baselines for comparison. Note that Graphene includes two optional kernel extensions: one that creates a reference monitor to protect the host kernel from the library OS, and one that optimizes fork by with copy-on-write for large (page-sized) RPC messages. Neither of these extensions are currently supported in Graphene-SGX.

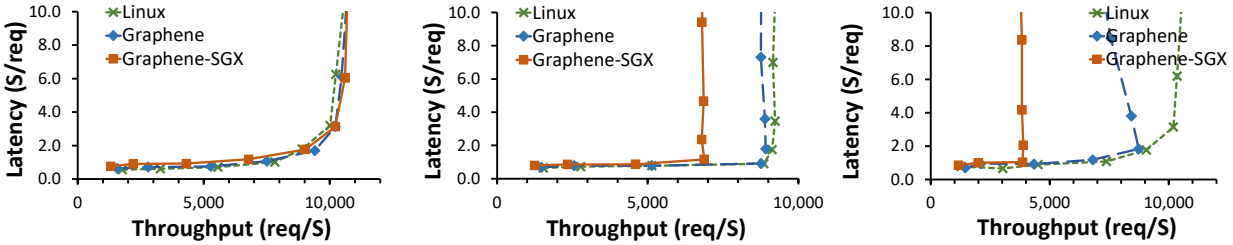
Experimental setup. We use a Dell Optiplex 790 Small-Form Desktop, with a 4-core 3.20 GHz Intel Core i5-6500 CPU (no hyper-threading, with 6MB cache), 8 GB RAM, and a 512GB, 7200 RPM SATA disk. The host OS is Ubuntu 16.04.4 LTS, with Linux kernel 4.4.0-21. Each machine uses a 1Gbps Ethernet card connected to a dedicated local network. We use version 1.8 of the Intel SGX Linux SDK [24] and driver [23].

I like the way they cited the SDK and driver.

5.1 Server applications

One deployment model for SGX is to host network services on an untrusted cloud provider's hardware. We measure three widely-used Linux web servers, including **Lighttpd** [6] (v1.4.35), **Apache** [2] (v2.4.18), and **NGINX** [7] (v1.10). For each workload, we use ApacheBench [1] to download the web pages on a separate machine. The concurrency of ApacheBench is gradually increased during the experiment, to test the both the per-request latency and the overall throughput of the server. Figure 5 shows the throughput versus latency of these server applications in Graphene-SGX, Graphene and Linux. Each workload is discussed below.

Lighttpd [6] is a web server designed to be lightweight, yet robust enough for commercial uses. Lighttpd is multi-threaded; we test with 25 threads to process HTTP requests. By default, Lighttpd uses the `epoll_wait` system call to poll listening sockets. At peak throughput and load, both Graphene and Graphene-SGX have marginal overhead on either latency or throughput of the Lighttpd server. The overheads of



(a) Lighttpd (25 threads) **(b) Apache (5 processes)** **(c) NGINX (event-driven)**

Figure 5: Throughput versus latency of web server workloads, including Lighttpd, Apache, and NGINX, on native Linux, Graphene, and Graphene-SGX. We use an ApacheBench client to gradually increase load, and plot throughput versus latency at each point. Lower and further right is better.

The graph and the narrative don't match.

Graphene are more apparent when the system is more lightly loaded, at 15–35% higher response time, or 13–26% lower throughput. Without SGX, Graphene induces 11–15% higher latency or 13-17% lower throughput over Linux; the remaining overheads are attributable to SGX—either hardware or our OS shield.

The top half doesn't make sense, the bottom half does.

Apache [2] is one of the most popular production web servers. We test Apache using 5 preforked worker processes to service HTTP requests, in order to to evaluate the efficiency of Graphene-SGX across enclaves. This application uses IPC extensively—the preforked processes of a server use a System V semaphore to synchronize on each connection. Regardless of the workload, the response time on Graphene-SGX is 12–35% higher than Linux, due to the overhead of coordination across enclaves over encrypted RPC streams. The peak throughput achieved by Apache running in Graphene-SGX is 26% lower than running in Linux. In this workload, most of the overheads are SGX-specific, such as exiting enclaves when accessing the RPC, as non-SGX Graphene has only 2–8% overhead compared to Linux.

This makes sense.

Also, I liked how they distinguish three web servers by their implementation.

NGINX [7] is a relatively new web server designed for high programmability, for as a building block to implement different services. Unlike the other two web servers, NGINX is event-driven and mostly configured as single-threaded. Graphene-SGX currently only supports synchronous I/O at the enclave boundary, and so, under load, it cannot as effectively overlap I/O and computation as other systems that have batched and asynchronous system calls. Once sufficiently loaded, NGINX on both Graphene and Graphene-SGX performs worse than in a Linux process. The peak throughput of Graphene-SGX is 1.5× lower than Linux; without SGX, Graphene only reaches 79% of Linux's peak throughput. We expect that using tools like Eleos [43] to reduce exits would help this workload; in future work, we will improve asynchronous I/O in Graphene-SGX.

5.2 Command-Line Applications

We also evaluate the performance of a few commonly-used command-line applications. Three off-the-shelf applications are tested in our experiments: **R** (v3.2.3) for statistical computing [9]; **GCC** (v5.4), the general GNU C compiler [4]; **CURL** (v7.74), the default command-line web client on UNIX [3]. These applications are chosen because they are frequently used by Linux users, and each of them potentially be used in an enclave to handle sensitive data—either on a server or a client machine.

I'm really impressed that they got all of these to run in an enclave

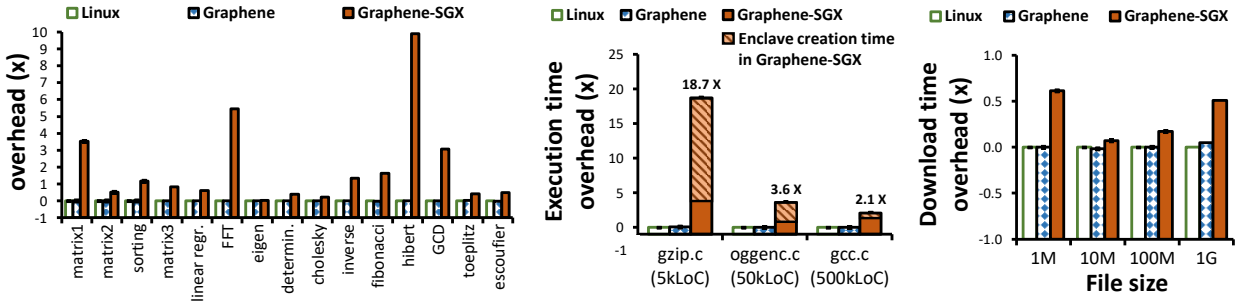
We evaluate the latency or execution time of these applications. In our experiments, both R and CURL have internal timing features to measure the wall time of individual operations or executions. On a Linux host, the time to start a library OS is higher than a simple process, but significantly lower than booting a guest OS in a VM or starting a container. Prior work measured Graphene (non-SGX) start time at 641 μs [52], whereas starting an empty Linux VM takes 10.3s and starting a Linux (LXC) container takes 200 ms [12].

This is the first time I've seen this.

On SGX, the enclave creation time is relatively higher, ranging from 0.5s (a 256MB enclave) to 5s (a 2G enclave), which is a fixed cost that any application framework will have to pay to run on SGX. Enclave creation time is determined by the latency of the hardware and the Intel kernel driver, and is primarily a function of the size of the enclave, which is specified at creation time because it affects the enclave signature. For non-server workloads that create multiple processes during execution, such as GCC in Figure 6, the enclave creation contributes a significant portion to the execution time overheads, illustrated as a stacked bar.

R [9] is a scripting language often used for data processing and statistical computation. With enclaves, users can process sensitive data on an OS they don't trust. We use an R benchmark suite developed by Urbanek et al. [8], which includes 15 CPU-bound workloads such as matrix computation and number processing. Graphene-SGX slows down by less than 100% on the majority of the workloads, excepts the ones which involve allocation and garbage collection: (`matrix1` creates and destroys matrices, and both `FFT` and `hilbert` involve heavy garbage collection.) Aside from garbage collection, these R benchmarks do not frequently interact with

Similar to other papers



(a) R (b) GCC (c) CURL

Figure 6: Performance overhead on desktop applications, including latency of R, execution time of GCC compilation, download time with CURL. The evaluation compares native Linux, Graphene, and Graphene-SGX.

the host. We further note that non-SGX Graphene is as efficient as Linux on all workloads, and these overheads appear to be SGX-specific. In our experience, garbage collection and memory management code in managed language runtime systems tends to be written with assumptions that do not match enclaves, such as a large, sparse address space or that memory can be demand paged nearly for free (SGX version 1 requires all memory to be mapped at creation); a useful area for future work would be to design garbage collection strategies that are optimized for enclaves.

GCC [4] is a widely-used C compiler. By supporting GCC in enclaves, developers can compile closed-source applications on customers' machines, without leaking the source code. GCC composes of multiple binaries, including `cc1` (compiler), `as` (assembler), and `ld` (linker). Therefore, GCC is a multi-process program using `execve`. We test the compilation of thee source files with varied sizes, using single C source files collected by MIT [5]. Each GCC execution typically creates five processes, and we run each process in a 256MB enclave by default. For a small workload like compiling `gzip.c` (5 kLoC), running in Graphene-SGX (4.1s) is 18.7× slower than Linux (0.2s). The bulk of this time is spent in enclave creation, taking 3.0s in total, while the whole execution inside the enclaves, including initialization of the library OS and OS shield, takes only 1.1s, or 4.2× overhead. For larger workloads like `oggenc.c` (50 kLoC) and `gcc.c` (500 kLoC), the overhead of Graphene-SGX is less significant. For `gcc.c` (500 kLoC), we have to enlarge one of the enclaves (`cc1`) to 2GB, but running on Graphene-SGX (53.1s) is only 2.1× slower than Linux (17.2s), and 7.1s is spent on enclave creation. The overhead of non-SGX Graphene on GCC is marginal.

CURL [3] is a command-line web downloader. Graphene-SGX can make CURL into a secure downloader that attests both server and client ends. We evaluate the total time to download a large file, ranging from 1MB to 1GB, from another machine running Apache. Graphene has marginal overhead on CURL, and Graphene-SGX adds 7–61% overhead to the downloading time of CURL, due to the latency of I/O.

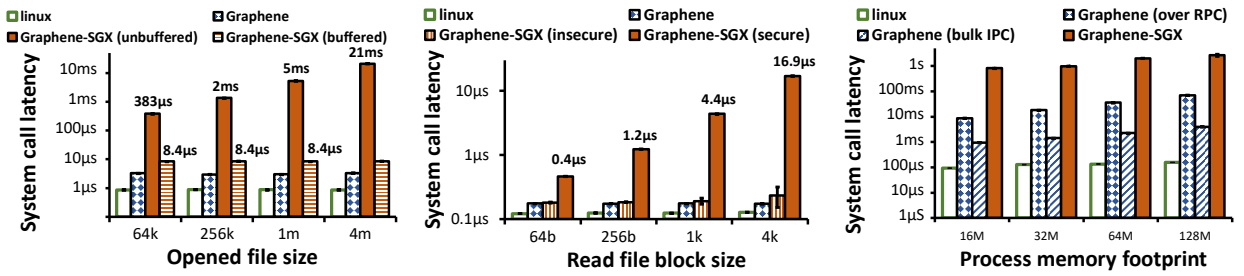
5.3 Performance Overhead Analysis

In this section we evaluate a few system operations that are heavily impacted by the Graphene-SGX design. We measure the `open`, `read`, and `fork` system calls using LMbench 2.5 [42]. A primary source of the overheads on these system calls is the cost of shielding applications, with run-time checks on the inputs. Cryptographic techniques are used to: (1) validate the file against the secure hash, at `open`, (2) check the file chunks against the Merkle tree, at `read`, and (3) establish a TLS connection over inter-enclave RPC, at `fork`. The remaining overheads contribute to exiting the enclave for host system calls, and bringing memory into the EPC (enclave page cache) or decrypting memory on a last-level cache miss.

Figure 7(a) shows the overhead for authenticating files in `open`. Depending on the file size, the latency of `open` on Graphene-SGX is 383μs (64KB file) to 21ms (4MB file), whereas on Linux, the latency is constant at 0.85μs. We note that this is where enclaves are at a disadvantage, as `open` normally does not need to read file content; whereas here Graphene-SGX uses `open` as a point at which to validate file content. For a subsequent `open`, when the Merkle tree is already generated, the overhead of simply exiting enclave for `open`, and searching the file list in the manifest, is about 9×.

One might be able to optimize further for cases where only part of a file is accessed with incremental hashing. However, in the common case where nearly all of the file is accessed, these costs are difficult to avoid when host file system is untrusted. Another opportunity is to create the Merkle tree offline, when the manifest is created.

Figure 7(b) shows the overhead for authenticating files in `read`, which is lower than `open`. Since the whole file has been verified at `open`, the sequential `read` only verifies the chunks of files it is reading from untrusted memory. Depending on the size of blocks being read, the latency on Graphene-SGX is 0.5μs (64-byte `read`)



(a) Open a file **(b) Read a file** **(c) Fork a process**

Figure 7: Latency of some expensive system calls in Graphene-SGX, including opening and reading a secured (authenticated) file, and forking a new process. The results are compared with native Linux and Graphene.

Components	**Graphene-SGX**	SCONE	Panoply
libc (ld, libm, pthread)	**1,292**	88	–
	(glibc-2.19)	(musl)	
Library OS	**34**	–	–
PAL / OS Shield	**22**	99	10
Total	**1,348**	187	10

Table 2: TCB size (in thousands of lines of code) of Graphene-SGX, SCONE, and Panoply.

to 16.9μs (4KB `read`). The latency of `read` on Linux is $\sim$0.1μs for any block size below 4KB. If the file is not authenticated, Graphene-SGX only copies the file contents into the buffer, and the overhead reduces to 48% (64-byte `read`) to 83% (4KB `read`).

Figure 7(c) shows the overhead of forking a process. As described in 4.3, the latency of `fork` in Graphene-SGX is affected by three factors: creation of a new enclave, local attestation of the integrity, and duplicating the process state over an encrypted RPC stream. Combining these factors, `fork` is one of the most expensive calls in Graphene-SGX. The default enclave size is 256MB. Our evaluation shows that the latency of forking a process is around 0.8s (16MB process) to 2.7s (128MB process), but can be more expensive if the parent process uses more memory. The trend matches the performance of Graphene without the bulk IPC optimization.

One way to further optimize `fork` is to reduce or avoid enclave creation time; one can potentially pre-launch a child enclave, and then migrate the process contents later when `fork` is called. There might be another opportunity to improve the latency of process migration, if copy-on-write sharing of enclave pages can be supported in future generations of SGX. **Yikes!!**

5.4 TCB Size and Shielded Functionality

In this section we measure the increase in TCB size of Graphene-SGX, as well as the OS functionality shielded by the framework. We compare to SCONE and Panoply, using numbers reported in their papers. A smaller TCB is generally easier to review or possibly verify, and is assumed to have fewer vulnerabilities.

Table 2 lists the lines of code in each components within the TCB of Graphene-SGX, SCONE, and Panoply. By comparing the total TCB size, Graphene-SGX is 9× larger than SCONE, and 134× larger than Panoply. However, the primary difference is the selection of libc: for maximum compatibility, Graphene uses glibc. SCONE uses the smaller musl libc, which lacks some features of glibc. Panoply excludes libc from its TCB, to fit into the range of automated formal verification, as they shield at the libc interface. In principle, Graphene could easily support musl as well as glibc for applications that do not need the additional features of glibc. We also see the benefit of removing unused code from libraries, especially in an unsafe language, similar to the approach taken in unikernels [38]. On balance, this choice of libc implementation is largely orthogonal to the issue of how general-purpose the shields are.

If we focus on the TCB size of the library OS and the shields, Graphene-SGX is 44% smaller than SCONE. We cannot analyze the size of SCONE because it is closed source. Panoply has a smaller TCB in its shield, but within the same order of magnitude. Panoply only shields 91 out of 256 supported POSIX functions; for context, POSIX 1003.1 defines 1,191 APIs [11].

All three of these compatibility layers or shields are within the same order of magnitude in code size, and the differences are likely correlated with different ranges of supported functionality. A recent study indicates that only order-of-magnitude differences in code size correlate with reported CVE vulnerabilities; within the same order-of-magnitude, the data is inconclusive that there is a meaningful difference in risk [25]. Thus, increased generality does not necessarily come with increased risk.

6 Related Work

Good chapter - re-read

Protection against untrusted OSes. Protecting applications from untrusted OSes predates hardware support. Virtual Ghost [20] uses both compile-time and run-time monitoring to protect an application from a potentially-compromised OS, but requires recompilation of the guest OS and application. Flicker [40], MUSHI [56],

I'd do well to mention all of these, and focus on Flicker as a representative example.

SeCage [37], InkTag [21], and Sego [32] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koberl et al. [30], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [34] built a 2-way sandbox for x86 by separating the Native Client (NaCl) [55] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [39] to protect the piece of application logic from the untrusted OS. Jang et al. [26] build a secure channel to authenticate the application in the Untrusted area isolated by the ARM TrustZone technology. Song et al. [50] extend each memory unit with an additional tag to enforce fine-grained isolation at machine word granularity in the HDFI system.

Need to do XOM.

Trusted execution hardware. XOM [35] is the first hardware design for trusted execution on an untrusted OS, with memory encryption and integrity protection similar to SGX. XOM supports containers of an application to be encrypted with a developer-chosen key. This encryption key is encrypted at design-time using a CPU-specific public key, and also used to tag cache lines that the containers are allowed to access. XOM realizes a similar trust model as SGX, except a few details, such as lack of paging support, and allowing `fork` by sharing the encryption key across containers.

Write about how TrustZone and SEV are different than SGX

Besides SGX, other hardware features have been introduced in recent years to enforce isolation for trusted execution. TrustZone [51] on ARM creates an isolated environment for trusted kernel components. Different from SGX, TrustZone separates the hardware between the trusted and untrusted worlds, and builds a trusted path from the trusted kernel to other on-chip peripherals. IBM SecureBlue++ [16] also isolates applications by encrypting the memory inside the CPU; SecureBlue++ is capable of nesting isolated environments, to isolate applications, guest OSes, hypervisors from each other.

Do IBM SecureBlue

AMD is introducing a feature in future chips called SEV (Secure Encrypted Virtualization) [27], which extends nested paging with encryption. SEV is designed to run the whole virtual machines, whereas SGX is designed for a piece of application code. SEV does not provide comparable integrity protection or the protection against replay attacks on SGX. Graphene-SGX provides the best of both worlds: unmodified applications with confidentiality and integrity protections in hardware.

Do Sanctum

Sanctum [19] is a RISC-V processor prototype that features a minimal and open design for enclaves. Sanctum also defends against some side channels, such as page fault address and cache timing, by virtualizing the page table and page fault handler inside each enclave.

SGX frameworks and applications. Besides shielding systems [14, 15, 49], SGX has been used in specific applications or to address other security issues. VC3 [45] runs MapReduce jobs in SGX enclaves. Similarly, Brenner et al. [17] run cluster services in ZooKeeper in an enclave, and transparently encrypt data in transit between enclaves. Ryoan [22] sandboxes a piece of untrusted code in the enclave to process secret data while preventing the loaded code from leaking secret data. Opaque [57] uses an SGX-protected layer on the Spark framework to generate oblivious relational operators that hide the access patterns of distributed queries. SGX has also been applied to securing network functionality [47], as well as inter-domain routing in Tor [29].

Probably all of these are worth a quick look and a sentence — like they do here... then add all work up to preset.

Innovative Technology
SCONE
Haven

Several improvements to SGX frameworks have been recently developed, which can be integrated with applications on Graphene-SGX. Eleos [43] reduces the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. SGXBOUND [31] is a software technique for bounds-checking with low memory overheads, to fit within limited EPC size. T-SGX [48] combines SGX with Transactional Synchronization Extensions, to invoke a user-space handler for memory transactions aborted by page fault, to mitigate controlled-channel attacks. SGX-Shield [46] enables Address Space Layout Randomization (ASLR) in enclaves, with a scheme to maximize the entropy, and the ability to hide and enforce ASLR decisions. Glamdring [36] uses data-flow analysis at compile-time, to automatically determine the partition boundary in an application.

7 Conclusion

This paper demonstrates that the costs of running an unmodified application in SGX on a library OS are marginal compared to thinner shims. The major costs of using SGX are still hardware limitations of SGX. As SGX and similar technologies mature, these design choices may have more impact. In the interim, Graphene-SGX serves as a simple, open-source tool to quickly bring up existing applications on SGX, and then incrementally adapt the code to improve performance and security on SGX.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Mihai Christodorescu, for their insighful comments on the work. We also thank the users of Graphene for contributing bug reports, code patches, and suggestions for the project, as well as their patience with bug fixes. Part of this work was completed while Porter's primary affiliation was Stony Brook University. This work was supported in part by NSF grants CNS-1149229, CNS-1161541, CNS-1228839, CNS-1405641, VMware, and an SGX pre-release equipment loan from Intel.

`8839 is close to us, but it's expired. All of these are coming through Stony Brook in NY. NSF' CNS is a good fit for our work, though.

References

[1] Apache HTTP benchmarking tool. `http://httpd.apache.org/docs/2.4/programs/ab.html`.

[2] Apache HTTP server project. `https://httpd.apache.org/`.

[3] CURL, command line tool and library for transferring data with url. `https://curl.haxx.se`.

[4] GCC, the GNU compiler collection. `https://gcc.gnu.org`.

[5] Large single compilation-unit C programs. `http://people.csail.mit.edu/smcc/projects/single-file-programs/`.

[6] Lighttpd. `https://www.lighttpd.net/`.

[7] NGINX. `https://www.nginx.com/`.

[8] R benchmark 2.5. `http://www.math.tamu.edu/osg/R/R-benchmark-25.R`.

[9] The R project for statical computing. `https://www.r-project.org/`.

[10] CVE-2009-2692. Available at MITRE, `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2692`, August 2009.

[11] IEEE International Standard for Information Technology – Portable Operating System Interface (POSIX) Base Specifications, Issue 7. Standard, IEEE, September 2009.

[12] K. Agarwal, B. Jain, and D. E. Porter. Containing the hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, 2015.

[13] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata. Innovative technology for CPU based attestation and sealing. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy at Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, 2013.

[14] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'Keeffe, M. L. Stillwell, D. Goltzsche, D. Eyers, R. Kapitza, P. Pietzuch, and C. Fetzer. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[15] A. Baumann, M. Peinado, and G. Hunt. Shielding applications from an untrusted cloud with haven. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 267–283, 2014.

[16] R. Boivie and P. Williams. SecureBlue++: CPU support for secure executables. Technical report, IBM Research, 2013. Available at `http://domino.research.ibm.com/library/cyberdig.nsf/papers/BE73A643EFE8274B85257B51006760C0`.

[17] S. Brenner, C. Wulf, and R. Kapitza. Running ZooKeeper coordination services in untrusted clouds. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, 2014.

[18] S. Checkoway and H. Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *SIGPLAN Not.*, pages 253–264, March 2013.

[19] V. Costan, I. Lebedev, and S. Devadas. Sanctum: Minimal hardware extensions for strong software isolation. In *USENIX Security*, volume 16, pages 857–874, 2016.

[20] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Citeseer, 2014.

[21] O. S. Hofmann, S. Kim, A. M. Dunn, M. Z. Lee, and E. Witchel. Inktag: secure applications on an untrusted operating system. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 265–278. ACM, 2013.

[22] T. Hunt, Z. Zhu, Y. Xu, S. Peter, and E. Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2016.

[23] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX driver. `https://github.com/01org/linux-sgx`.

[24] Intel Corporation. Intel software guard extensions for Linux OS - Intel SGX SDK. `https://github.com/01org/linux-sgx`.

[25] B. Jain, C.-C. Tsai, and D. E. Porter. A clairvoyant approach to evaluating software (in)security. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2017.

[26] J. S. Jang, S. Kong, M. Kim, D. Kim, and B. B. Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[27] D. Kaplan, J. Powell, and T. Woller. AMD memory encryption. White paper, April 2016. Available at `http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf`.

[28] S. Kim, J. Han, J. Ha, T. Kim, and D. Han. Enhancing security and privacy of tors ecosystem by using trusted execution environments. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017.

[29] S. Kim, Y. Shin, J. Ha, T. Kim, and D. Han. A first step towards leveraging commodity trusted execution environments for network applications. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, page 7. ACM, 2015.

[30] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, page 10. ACM, 2014.

[31] D. Kuvaiskii, O. Oleksenko, S. Arnautov, B. Trach, P. Bhatotia, P. Felber, and C. Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, EuroSys '17, 2017.

[32] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *SIGOPS Oper. Syst. Rev.*

[33] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 277–290. ACM, 2016.

[34] Y. Li, J. McCune, J. Newsome, A. Perrig, B. Baker, and W. Drewry. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the USENIX Annual Technical Conference*, pages 409–420, 2014.

[35] D. Lie, C. A. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. *ACM SIGOPS Operating Systems Review*, 37(5):178–192, 2003.

[36] J. Lind, C. Priebe, D. Muthukumaran, D. O'Keeffe, P.-L. Aublin, F. Kelbert, T. Reiher, D. Goltzsche, D. Eyers, R. Kapitza, C. Fetzer, and P. Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, Santa Clara, CA, 2017. USENIX Association.

[37] Y. Liu, T. Zhou, K. Chen, H. Chen, and Y. Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[38] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2013.

[39] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. TrustVisor: Efficient TCB reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 143–158, 2010.

[40] J. M. McCune, B. J. Parno, A. Perrig, M. K. Reiter, and H. Isozaki. Flicker: An execution infrastructure for TCB minimization. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 315–328, 2008.

[41] F. McKeen, I. Alexandrovich, I. Anati, D. Caspi, S. Johnson, R. Leslie-Hurd, and C. Rozas. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, New York, New York, USA, June 2016. ACM Press.

[42] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *Proceedings of the USENIX Annual Technical Conference*, pages 23–23, 1996.

[43] M. Orenbach, P. Lifshits, M. Minkin, and M. Silberstein. Eleos: ExitLess OS services for SGX enclaves. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[44] N. Provos, M. Friedl, and P. Honeyman. Preventing privilege escalation. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 16–16, Berkeley, CA, USA, 2003. USENIX Association.

[45] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz, and M. Russinovich. VC3: Trustworthy data analytics in the cloud using SGX. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 38–54. IEEE, 2015.

[46] J. Seo, B. Lee, S. Kim, M.-W. Shih, I. Shin, D. Han, and T. Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[47] M.-W. Shih, M. Kumar, T. Kim, and A. Gavrilovska. S-NFV: Securing NFV states by using SGX. In *Proceedings of the ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization (SDN-NFV Security)*, pages 45–48. ACM, 2016.

[48] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[49] S. Shinde, D. L. Tien, S. Tople, and P. Saxena. PANOPLY: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[50] C. Song, H. Moon, M. Alam, I. Yun, B. Lee, T. Kim, W. Lee, and Y. Paek. HDFI: Hardware-assisted data-flow isolation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[51] ARM TrustZone technology overview. Available at `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[52] C.-C. Tsai, K. S. Arora, N. Bandi, B. Jain, W. Jannen, J. John, H. A. Kalodner, V. Kulkarni, D. Oliveira, and D. E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

[53] C.-C. Tsai, B. Jain, N. A. Abdul, and D. E. Porter. A study of modern Linux API usage and compatibility: What to support when you're supporting. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[54] Y. Xu, W. Cui, and M. Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (Oakland)*. IEEE Institute of Electrical and Electronics Engineers, May 2015.

[55] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 79–93. IEEE, 2009.

[56] N. Zhang, M. Li, W. Lou, and Y. T. Hou. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *Military Communications Conference, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[57] W. Zheng, A. Dave, J. G. Beekman, R. A. Popa, J. E. Gonzalez, and I. Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

99% of Ubuntu binaries are dynamically linked.

ASLR in SGX?

this could be kinda cool

Graphiene's dynamic loader brings up a very interesting problem... where a traditional loader can fixup addresses at load-time, an SGX loader can't make changes to the enclave prior to init() time. This raises a number of interesting problems and questions.

Q1) How, exactly, Does a process or library make an SGX call into another enclave?

Q2) Do downstream libraries have a trust relationship with upstream libraries? For example, does libc need to trust HelloWorld? What about callbacks?

Q3) can we do better than Graphene, which makes all pages RWX, by ensuring that .text pages are RX and .data pages are RW?

Q4) How do we prevent library MitM attacks? Who is responsible for preventing it? I think this is a very tricky problem.

Q5) How can we maintain the rich, dynamic development of programs and libraries while maintaining security without creating tight binding between them?

I have a few implementations thoughts...

A1) I think each module has a role to play in trusting other modules. All ELF/PE files have a manifest of libraries+functions they import and functions they export. To this, I think we need to add a block of code that decides how to "trust" an imported library. For example, something that verifies that the "glibc" enclave I'm about to trust is is signed by Gnu.

A2) After each imported library is trusted, the program saves its enclave ID by the imported library name (in RW memory). After that, it generates a secret key exclusively for that enclave (library) and saves it for later use.

A3) today, most loaders verify the imported library exports all of the functions it needs. This is load-time binding. We need to decide if we want to do load time or use-time binding.