# Writing Plan

Access the following Google Doc:
`https://docs.google.com/a/cs.stonybrook.edu/document/d/1Wk4dk2mgFPXlB_3hRwfPiyQdBI16B9eT4xA5qpJ24iA/edit?usp=sharing`

**A Library Operating System for Compatibility and Security Isolation**

A Dissertation presented

by

**Chia-Che Tsai**

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

**Doctor of Philosophy**

in

**Computer Science**

Stony Brook University

**August 2017**

**Stony Brook University**

The Graduate School

**Chia-Che Tsai**

We, the dissertation committee for the above candidate for the

Doctor of Philosophy degree, hereby recommend

acceptance of this dissertation

---

**R. Sekar - Chairperson of Defense**
Professor, Computer Science Department

---

**Donald E. Porter - Dissertation Advisor**
Research Assistant Professor, Computer Science Department

---

**Michael Ferdman**
Assistant Professor, Computer Science Department

---

**Timothy Roscoe**
Professor, Computer Science, ETH Zürich

This dissertation is accepted by the Graduate School

Charles Taber
Dean of the Graduate School

Abstract of the Dissertation

**A Library Operating System for Compatibility and Security Isolation**

by

**Chia-Che Tsai**

For the Degree of

**Doctor of Philosophy**

in

**Computer Science**

Stony Brook University

**2017**

**To be written.**

## Dedication Page

This page is optional.

# Table of Contents

# List of Tables

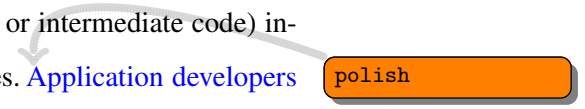
# List of Figures

# Chapter 1

# Introduction

Operating systems simplify the programming of an application, to utilize a wide range of hardware. A UNIX-style OS [139] encapsulates hardware resources and abstractions, using a system interface. Without a system interface, an application will have to be programmed against hardware interfaces defined by manufacturers. The problem of programming against bare hardware is that the execution of applications is bound to specific hardware options. Operating systems allow application developers to program against a consistent, hardware-independent system interface, so that application developers can avoid making assumption or restricting users to run an application on specific hardware.

An application developed in an OS is dependent on the system interface to remain compatible with previous OS generations. This thesis defines **compatibility** as the ability of reproducing a system interface that satisfies the hard-coded requirement of applications. The development of a fully-compiled application (not as a script or intermediate code) includes hard-coding the usage of a system interface in the binaries. Application developers make the decisions of embedding the requests and parameters of interacting with a system interface inside of an application. If a milestone of OS development involves modifying a part of the system interface, an application becomes obsolete if it contains outdated system interface specifications. Although compilers and libraries can reconstruct or recast some OS functions, a principle-level or assumption-level change in the system interface would still be difficult to recover. The common practice in OS development is to preserve the



whole system interface, as application developers previously observe when building an application.

OS developers often struggle to maintain compatibility while improving the relationship between the OS and applications. An system interface may contain outdated specifications of OS features and application programming interfaces (APIs); however, OS developers generally avoid modifying the system interface, despite of missed opportunities to improving both OS and application sides. For example, Linux and similar OSes introduce system calls such as `openat()` as an unracy version of `open()`. Unfortunately, complete deprecation of the unsafe `open()` system call would be a painful and lengthy process, wherein every applications have to be reprogrammed accordingly. Moreover, if a system interface is changed on a large scale, the result may be catastrophic: a well-known example is the cancellation of an early version of Windows Vista, codenamed "Longhorn", which introduced a brand-new user interface API and a database-like file system, but sacrificed compatibility [155]. In general, OS developers treat compatibility as an important factor to the usability of a system, but also an unwelcome distraction in the early stage of innovation.

polish

Sometimes, the adoption of an innovative hardware poses unexpected challenges to maintaining compatibility in OS development. A new hardware that follows a similar design as its predecessors can be adopted by the OS using a standardized driver. However, a cutting-edge or research hardware may not fall into the typical abstractions encapsulated by the driver interfaces, and thus demands extra attention for adoption. The challenges emerge as more ground-breaking hardware technologies have been released in the industry. For example, CPU technologies like Intel SGX (software guard extensions) [120] change the relationship between the OS and applications. SGX supports developing an application with self-protection against malicious OSes, hypervisors, and administrators; however, SGX also raises several compatibility issues, including reverting the trusted nature of OS services. Other examples can be found among research-type architectures, such as an asymmetric multi-processing architectures without inter-connected memory [55, 78].In the long

polish

Issue 1.2.b (done): Specify SGX as a motivating example

polish

term, OS developers should consider gradually promoting a new system interface that can encauplate the latest hardware, but existing applications still need a timely solution to resolve the urgent compatibility issues.

A practical reason for maintaining compatibility is the fact that application developers may be unwilling to modify an application, at the stage of either source code or compiled binaries. The development of a commercial application requires a thorough process of testing and code inspection to ensure the correctness and safety of execution. Application developers may consider modifying or recompiling an application a risk to application stability. Even if users are willing to modify applications on their own, application developers may be reluctant to release the proprietary source code. These dilemmas call for a solution to translate an alternative system interface for unmodified applications.

To formalize the problem, OS developers can't or shouldn't avoid modifying a system interface, yet existing application tend to rely on former system interface definitions to remain functional. A reasonable approach is to insert a compatibility layer between the application and the OS, to bridge the gap of system interface definitions, including both the APIs and the nature of OS features. Take SGX for example: a compatibility layer for supporting unmodified applications on SGX must redirect the requests for system API to the host OS, as well as validate the results of OS services in case the host OS is malicious. Another goal is to reduce the effort of developing such a compatibility layer, so that maintaining compatibility would not become a burden in OS development and innovation. OS developers should not have to reimplement every features of a system interface in a compatibility layer; A modern OS such as Linux can contain up to hundreds or thousands of functions, with lots of options and corner cases to implement [13]. Ultimately, this thesis is presenting a solution for building a compatibility layer that reproduces a rich-feature system interface, with less development effort.

The development of a compatibility layer can benefit from virtualizing the upper layers of an OS, or the whole OS. An OS contains several components that are independent from encapsulating hardware or upholding OS assumptions (e.g., multiprocessing).

polish

Issue 1.1.b (done): Specify the problem as reducing the gap of interface idiosyncrasy

polish

polish

For example, a major task of the API components (e.g., system call table) is to interpret the user inputs, as well as to update or retrieve the related OS states. Virtualization can preserve these OS components, using an intermediate interface exported by the host OS. Virtualization allows a VM (virtual machine) to carry an unmodified OS kernel, as a full-stack compatibility layer for applications. A VM relies on a virtual hardware interface as the intermediate to the host; the virtual hardware interface requires a CPU architecture with virtualization support [133] such as Intel VT (virtualization technology) [164]. Virtualization can provide full compatibility by reusing an OS implementation, as long as the architecture is virtualizable.

Issue 1.2.c (done): discuss library OS vs VM

As an alternative to a VM, OS developers can partition an OS, to virtualize only necessary components into a thin compatibility layer—a **library OS** [29, 72, 116, 134] loaded inside of an application. The purpose of a library OS is to reproduce the features and API of a system interface, inside a user-space, reusable *library* that executes upon a virtualized host interface. Unlike the virtual hardware interface, a virtualized host interface to the library OS is redefined for the simplicity of OS development upon each host OS and hardware. The host interface is derived from a "pinch point" found inside an OS, to partition the high-level, hardware-independent OS components, into a library OS. A library OS can be reused to build a compatibility layer for unmodified applications upon any host, as long as the host interface is implemented.

polish

polish

This thesis presents **Graphene**, a library OS for running unmodified applications upon innovative hardware and alternative system interfaces. Graphene chooses the rich-feature Linux system interface (i.e., system calls) as the target for reproducing. The goal of Graphene is to build a compatibility layer for reusing a wide range of Linux commercial applications, to benefit the highly-customizable server and cloud environments. The Graphene library OS is developed upon a narrowed host interface, defined as a host ABI (application binary interface), to be easily ported to various host options. The host ABI isolates the implementation of Linux features and API, from addressing the low-level restriction and idiosyncrasy. The strength of Graphene is to fine-tune the host ABI, to simplify the host

polish

development and drop the assumptions on host OSes and hardware. So far Graphene has been targeting several host options, including innovative CPU platforms (e.g., SGX), alternative kernels (e.g., Windows, OSX, and FreeBSD), and research-type OSes (e.g., L4 microkernels [104], Barrelfish [43]).

This thesis tries to strike a better balance between simplicity of the host interface and sufficiency of the library OS functionality. According to a study of Linux system API [161], system calls are not equally important in term of the impact on application compatibility. Also, not every applications are demanded to be reused upon different host OSes and hardware [163]. The study shows that a portion of Linux system calls are defined for administrative purposes, such as configuring ethernet cards and rebooting. These system calls are exclusively used by system software, such as `ifconfig` and `reboot`. Graphene is primarily targeting three types of applications: server and cloud applications (e.g., Apache, Memcached), command-line programs (e.g., Bash, GCC), and language runtimes (e.g., Python, OpenJDK). The Graphene library OS is sufficient for running these applications, by implementing a portion of the Linux system calls (145 out of 318), upon a narrowed host ABI containing 42 functions.

Issue 1.2.d (done): Specify target applications

Graphene inherits a foundation of the host ABI from Drawbridge [134], a previous library OS developed for reusing Windows applications. The purpose of Drawbridge is to use the library OS as a lightweight VM, to run applications in a guest environment. Because the library OS is built by partitioning OS components, it requires far less memory resource than running a VM with a full, unmodified OS kernel. The Drawbridge design is later adopted by Bascule [44] to show the feasibility of implementing single-process, Linux functionality, and inserting host-level extension layers. Haven [45] later ports Drawbridge to SGX, to shield Windows application against malicious inputs from the host OS. Established upon the previous work, Graphene presents an overall solution for running unmodified applications upon a variety of host OSes and hardware, and defines a host interface that is simple to port and sufficient to support a full-feature library OS.

polish

polish

This thesis has several contributions over previous work.

rewrite this

## 1.1 Motivating Examples

This section shows two examples in which developing a compatible OS for existing applications can be challenging, to motivate the library OS design.

Issue 1.1.b (done): motivating examples

### 1.1.1 Unmodified applications on SGX

SGX [120] are new extensions to the six-generation Intel CPUs, which can protect signed application code from attacks in compromised OSes, hypervisors, and other system software. In the isolated environment of SGX, or **enclaves**, applications can securely utilize the resources of an untrusted host, such as a public cloud machine for rent, or a client machine, with both confidentiality and integrity. SGX also offers attestation for the integrity of remote enclaves, as well as proving the integrity of the Intel CPUs.

To utilize SGX, common expectation is that developers have to partition a piece of the application code to run inside an enclave. The restriction is for both security and simplicity reasons. The enclave code is mostly statically compiled, for making code verification straightforward. Development of enclave code also involves removing OS function calls and instructions which are not supported in an enclave. The restriction on the instructions is also for the clarity of application protection. For example, SGX forbids `cpuid` in case of combining with the hardware virtualization (VT), to avoid trapping to hypervisors. SGX also forbids `rdtsc`, in order to rule out potential physical attacks on the Time Stamp Counter (TSC).

SGX excludes OS services from the trust model, except functions which can be fully ported to user space (e.g. `malloc()`). The absence of trusted OS services is an issue for porting any application. Existing solutions combines a modified C library with applications, to redirect system interfaces to the untrusted OS [38, 150]. The problem, however, is in checking the results of system interfaces, because the OS is not trusted. Previous work [57] shows that checking untrusted system interface results can be subtle, because the existing system interfaces are not designed for an untrusted or compromised

Issue 1.2.c (done): discuss challenges of SGX shielding

OS.

In summary, the existing porting models of Intel SGX always requires modifying application binaries, and reasoning about the completeness of checking system interfaces. This thesis argues that, by introducing a library OS into enclaves, the interaction with the untrusted OS can be restricted to OS services which have clear semantics for checking. By implementing the host ABI inside an enclave, users can easily run an unmodified Linux application, such as an Apache server or an OpenJDK runtime, with a trusted Graphene library OS instance.

### 1.1.2 Multi-process applications

Issue 1.3.b (done): Using multi-process as a motivating example

One characteristic of a UNIX program is the opportunity of utilizing multiple processes, created by either `fork()` or `exec()`, to program self-contained sessions or commands in applications. Especially, `fork()` is an unique feature in UNIX-style OSes, such as Linux and BSD, to clone a process into another child process, with isolation from the parent. The multi-process abstractions are convenient for creating a temporary session for processing incoming requests or commands, and destroying the session without corrupting the parent process.

Between multiple processes, there are several mechanisms of inter-process communication (IPC) available for programming. The Linux IPC combines the UNIX System V features, including message queues and semaphores, and POSIX abstractions, such as signaling and namespaces. Figure 1.1 shows a code example of two Linux programs ("`sh`" and "`kill`") running in parallel as part of a multi-process application and communicating with signals. The destination of signaling is determined by a unique process identifier (PID) known by all processes. These kinds of identifiers or names are globally shared, as part of the POSIX namespaces, among applications or processes visible to each other.

Implementing IPC mechanisms in an OS used to rely on a coherent kernel space, but can be challenging when an OS makes the opposite assumption. The easiest design is to store the states and namespace in the kernel memory accessible to all processes. Sharing

(Parent process: "sh")

```
char pid[10], *argv[]={"kill",pid,0};
itoa(getpid(), pid, 10);
if (!fork()) //clone a process
  execv("/bin/kill", argv);
wait(NULL);  //wait for signal
```

(Child process: "kill")

```
pid=atoi(argv[1]);
//send a signal
kill(pid, SIGKILL);
```

Figure 1.1: Sample code for Linux applications using process cloning and inter-process communication (IPC).

kernel states, however, is prone to attacks in an OS with process sandboxing, or in an application isolated by hardware like Intel SGX. An isolated OS design tends to avoid sharing a fully-trusted, coherent kernel space shared with other applications. For example, on SGX, an enclave cannot share trusted memory with other enclaves.

Moreover, not all architectures assume inter-connected, coherent memory. Several recent multi-CPUs architectures choose not to implement memory coherence for simplicity [55, 78]. Barrelfish [43] demonstrates an efficient OS design, called multikernels, which runs distributed OS nodes on CPUs with inter-node coordination by message passing. The distributed OS design resonates with the Graphene library OS, which uses RPC (remote procedure call) streams to implement multi-process abstractions. Since Graphene does not assume a coherent kernel space, it can be a flexible option for porting multi-process applications to a variety of OSes and architectures.

## 1.2 Security Isolation

Issue 1.1.d: separate the motivation for security isolation

## 1.3 Evaluating API Compatibility

rewrite this

This thesis studies the compatibility requirement in Linux and POSIX from the perspectives of applications and users. The study begins with building a metric for compatibility, weighted by the API usage in applications, and application importance (or popularity). Using the metric, OS developers can quantitatively determine the priority in implementing a

system interface, and evaluate the developed results. The study shows that most applications do not require half of the Linux system calls, which are either for administrative use (e.g., halting the machine), or unpopular among applications.

## 1.4 Summary

This thesis contributes a library OS design, called Graphene, which demonstrates the benefits on reusing unmodified Linux applications, upon new hardware or OS prototypes. Compared with ad-hoc translation layers, a library OS with a rich of Linux functionality (145 system calls) can be reused on various host platforms, as an adaptable layer with compatibility. Graphene can adapt to the restrictions and limited hardware abstractions on a host, with acceptable performance and memory footprint. This thesis further reasons about the sufficiency of a library OS for running frequently-reused applications. The reasoning is based on a metric which can evaluate the partial compatibility of a system interface. Graphene prioritizes indispensable system calls over administrative or unpopular features, to reuse a wide range of applications, from server applications to language runtimes.

**Previous publications.** The initial design of the Graphene library OS is presented in [160], which emphasizes on security isolation, between mutually-untrusted applications. A later publication [25] focuses on porting the host ABI to Intel SGX, and demonstrates the security benefit over a thin redirection layer, and the usability feature to run unmodified applications. [161] presents the compatibility metrics for compatibility, with a study of the Linux API usage among Ubuntu users and applications.

## 1.5 Organization

The rest of this thesis is organized as follows: Chapter 2 describes the overview of Graphene (including the host ABI and the library OS) and the design principles behind the implementation. Chapter 3 formally defines the host ABI, and provides a specification of the host-specific PAL (platform adaption layer). Chapter 4 discusses the library OS in details.

Chapter 5 describes the PAL on Linux, as an example of implementing the host ABI and security isolation between library OS instances. Chapter 6 discusses SGX-specific challenges to application porting, and the PAL implementation inside a SGX enclave. Chapter 7 evaluates the performance and memory footprint of Graphene and Graphene-SGX, and presents a security study. Chapter 8 presents a quantitative metric for compatibility, to evaluate the completeness of Linux functionality in Graphene. Chapter 9 presets a study of the Linux API importance, to give an insight about prioritizing API implementation. Chapter 10 discusses the related work. Chapter 11 concludes the thesis.

# Chapter 2

# System Overview

This chapter gives an overview of Graphene. The design of Graphene is divided into two parts: a host ABI (application binary interface) for porting to new OSes and hardware, and the Graphene library OS. This chapter first introduces the host ABI, and its design principles. The rest of chapter discusses the library OS, including its architecture, approaches to implementing Linux functionality, and ~~the~~ potential trade-offs.

Issue 1.1.a (done): reorganization: Graphene overview

## 2.1 The Host ABI

Graphene facilitates compatibility by partitioning an OS at a narrowed interface that contains OS abstractions ~~that are~~ essential to application execution. The host ABI separates the low-level, hardware management features, from the idiosyncrasy of system interface. Graphene moves the upper layer of OS components, including the system calls and namespaces, into an library OS, leaving the host ABI as a narrowed interface to the host OSes and hardware.

Issue 1.1.b (done): describe the host ABI specification

We define a **host** as an OS or a hypervisor which run an application or a VM on a physical machine. For example, a commodity OS, such as Linux, BSD, or Windows, can be a host to the library OS, with portability on different hardware. An innovative hardware abstraction like SGX (software guard extensions) imposes unique assumptions and restrictions on a commodity OS, and thus creates a special host above the OS.

The PAL ABI determines a boundary which partitions several upper-level OS com-

ponents, into a library OS, in order to isolate the host idiosyncrasy. The strategy is also used in OSes: An example is the Linux virtual file system (VFS), an internal interface which encapsulates operations of file system drivers. Similar to VFS, the host ABI is intended to be a more ubiquitous interface, which encapsulates ~~any~~ host-specific behavior and semantic inside the host OS.

The PAL ABI shares several characteristics of a virtual hardware interface for running a VM. A hypervisor usually export a generic, but backward-compatible virtual hardware interface, to port an unmodified OS into a VM. For example, a virtual ethernet card in VMware workstation or QEMU emualates an Intel E1000 device. However, unlike a virtual hardware interface, the host ABI does not reuse a whole, unmodified OS. Instead, the PAL ABI expects the library OS to be rewritten and customized for the PAL ABI, similar to a para-virtualizated VM. Moreover, Graphene deduplicates OS components, such as scheduler, page fault handler, file system, and network stack, between the host and the library OS. Overall, we can put a library OS and a VM on a spectrum: while a VM tries to reuse most of an OS and runs on a virtual hardware interface, a library OS partitions an OS at a narrowed interface.

**Simplicity of the host ABI.** To reduce the burden of OS developers, the host ABI is designed to simple to port on a new host. The amount of porting effort is subjective to OS developers, but can be reduced with two strategies: the first is to narrow the size of the host ABI; Although narrowness does not indicate simplicity, removing redundant OS functions or corner cases makes porting easier. The second strategy is to include functions which can be directly mapped to OS functions exported by the host. An observation of OS development is that similar OS functions tend to exist in different OSes. We can find system call lookalikes in many OSes: for example, `read()` in Linux, BSD, and POSIX looks similar to `ReadFile()` in Windows, except the data types. Most functions in the host ABI can be easily translated to host system interfaces in various styles.

**Sufficiency of the host ABI.** The host ABI is responsible for exporting sufficient functionality to the library OS, for requesting hardware or OS abstractions which cannot be

easily virtualized in user space. ~~For sufficiency~~, the host ABI ~~covers a set of typical OS functions. The typical OS functions in most OSes~~ includes process creation, memory management, I/O (typically, files and network connection), security and protection [67]. Linux defines multiple system calls with similar purposes. For example, both `mmap()` and `brk()` can allocate memory in a process: `mmap()` allocates memory regions at page granularity, whereas `brk()` grows a single, continuous heap space. Graphene shows that a basic OS function is sufficient to implement all the ~~idiosyncratic~~ variants in the Linux system interfaces. Both `mmap()` and `brk()` can be implemented using `VirtMemAlloc()` in the host ABI, with bookkeeping of allocated memory regions in the library OS.

The definition of the host ABI in Graphene is based on ~~a prior work ca~~ Drawbridge [134]. Drawbridge is a library OS for running single-process Windows applications in a lightweight, guest environment. Drawbridge defines a host ABI, including 36 functions, to develop its library OS. The Drawbridge host ABI is ported to multiple hosts, including Windows, Linux, Barrelfish, and Intel SGX [44, 45, 134, 156], and is capable of running a library OS for single-process, Linux applications, with a few host ABI changes [44]. Although running Windows and Linux applications introduces a different set of challenges, the nature of the Linux and Windows APIs is actually quite similar, with some exceptions. During the development of Graphene, we found a few occasions in which we have to extend the host ABI to deal with Linux-specific challenges. The Linux-specific extensions will be further discussed in Section 2.1.1 and Chapter 3.

**Checkpointing and migration.** The Graphene library OS shares several features of VMs, including the convenience of checkpointing a running application and migrating to other hosts. The migration feature is also the key to implementing copy-on-write forking for applications, on a host which disallows memory sharing (e.g., SGX). For a VM, checkpointing and migration is based on using the virtual hardware interface as a clean boundary for snapshotting the application and OS state. The host ABI in Graphene shares the same quality with the virtual hardware interface, as being as *stateless* as possible. The statelessness of the host ABI guarantees any states in the hosts are temporary to the applications and the

library OS, and can be reproduced without any snapshotting in the hosts.

### 2.1.1 Host ABI functions

Table 2.1 lists 42 functions defined in the host ABI of Graphene. Among these functions, 27 are inherited from the Drawbridge host ABI, including functions to managing I/O (e.g., `StreamOpen()`), memory allocation (e.g., `VirtMemAlloc()`), scheduling processes and threads (e.g., `ThreadCreate()`), and several miscellaneous functions (e.g., `ThreadCreate()`). Only 8 functions are added by Graphene, in order to implement Linux-specific features. For example, unlike Windows or OSX, Linux generally delivers hardware exceptions to a process as signals . Linux also requires the x86-specific segment registers (i.e., FS/GS registers) to determine the location of thread-local storage (TLS), which can be hard-coded in application binaries by a compilation mode of GCC. However, in Windows or OSX, the x86-specific segment registers are mostly ignored, and even frequently reset to avoid being manipulated by attackers.

The host ABI also include 5 functions for remote procedure call (RPC), in order to implement Linux multi-process abstractions. The rationale of the multi-process support in Graphene is to reduce the complexity of inter-process communication in the host. A host to Graphene needs not ~~to~~ understand all the Linux-specific multi-process behaviors, but only sees a pipe-like RPC stream for message passing between processes. To ensure performance to be acceptable, Graphene defines an optional, bulk IPC feature in the host ABI as optional host functions, to send large chunks of memory across processes. The bulk IPC feature works similarly as sending the memory through RPC streams, but is much faster because it avoids copying the memory in the host kernels.

### 2.1.2 PAL (platform adaption layer)

Graphene relies on each host to implement the host ABI, using the native system interfaces on the host. For each host, a platform translation layer (PAL) is loaded below the library OS, to translate each functions in the host ABI to native system interfaces. The development of

| Abstraction | Function Names | Description |
|---|---|---|
| Streams | `StreamOpen`<br>`StreamMap`<br>`StreamFlush`<br>`StreamSetLen`<br>`StreamRead`<br>`StreamWrite`<br>`StreamWaitforClient`<br>`StreamAttrQuery`<br>`StreamAttrQuerybyHandle`<br>`StreamAttrSetbyHandle`<br>`StreamDelete` | Opening streams using URIs, with prefixes representing stream types (e.g., `file:`,`tcp:`,`pipe:`), as well as common stream operations, including transmission of data, and query to the stream attributes. |
| Memory | `VirtMemAlloc`<br>`VirtMemFree`<br>`VirtMemProtect` | Allocation, deallocation, and protection of a chunk of virtual memory. |
| Threads & scheduling | `ThreadCreate`<br>`ThreadExit`<br>`ThreadDelayExecution`<br>`ThreadYieldExecution`<br>`SemaphoreCreate`<br>`SemaphoreRelease`<br>`EventCreate`<br>`EventSet`<br>`HandlesWaitAny` | Creation and termination of threads; Using scheduling primitives, including suspension, semaphores, events, and pollable IO events. |
| Process | `ProcessCreate`<br>`ProcessExit` | Creating or terminate a process with a library OS instance. |
| Miscellaneous | `SystemTimeQuery`<br>`RandomBitsRead` | Querying system time, and random number generation. |
| Exceptions † | `ExceptionSetHandler`<br>`ExceptionReturn` | Setting an exception handler, and returning from the handler. |
| TLS † | `SegmentRegisterSet` | Setting the `FS`/`GS` registers. |
| Remote Procedure Call † (optional) | `RpcSendHandle`<br>`RpcRecvHandle`<br>`PhysicalMemoryStoreOpen`<br>`PhysicalMemoryCommit`<br>`PhysicalMemoryMap` | Sending opened stream handles or physical memory across processes. |

Table 2.1: An overview of the 42 functions in the host ABI of Graphene. The ones marked with the symbol † are introduced in the initial publication of Graphene [160] or later extended for this thesis. The rest are inherited from Drawbridge [134].

a PAL is a effort paid per host, whereas the library OS is reusable on every hosts. Based on the simplicity of the host ABI, the PAL development is supposed to be straightforward and easy for average developers, except corner cases on a few eccentric hosts, such as SGX.

Graphene is experimented with several "classic" hosts, including Windows, Linux, OSX, FreeBSD, and Linux with SGX. For most of these hosts, implementing the host ABI is straightforward, because most OSes have exported a version of the POSIX API, or similar API. Few exceptions where the porting has been challenging are mostly caused by ~~usual~~ assumptions made in the host OSes. For example, the Windows API disallows directly resizing or protecting part of a memory region, which is essential to implementing the `munmap()` and `mprotect()` system calls. A workaround for porting the host ABI to Windows is to change the memory region at the physical page level, but requires running the PAL in administer permission.

Based on the experience in Graphene, even a host ABI specialized for simplicity cannot guarantee to be portable on every hosts. It is possible that a host simply lacks the ~~typical~~ functionality provided in other OSes. As a principle, Graphene does not require each host to implement the whole host ABI, but designs the library OS to flexibly switch emulation strategies when a function is unavailable. For example, when the bulk IPC feature is not available on the host, the library OS can always switch back to the RPC-based IPC, with performance penalty. In the worst case, if there is no emulation strategies to compensate for the missing of a function, at least users can predict whether an application will be affected and thus cannot run on certain hosts.

### 2.1.3 Security isolation

To target multi-tenant environments, such as cloud, Graphene has to ensure ~~at least complete~~ security isolation between mutually-untrusted applications running on the same host. The security isolation of Graphene is comparable to running each application in a VM, featuring a fully-isolated guest OS. Similar to the virtual hardware interface isolating the VMs, the host ABI also enforces security isolation between library OS instances, according to the

trust model of the applications.

Graphene delegates the enforcement of security isolation to the hosts. The library OS and the application are mutually-trusted, as long as they are loaded in the same process. The host ABI decouples the implementation of OS functionality from the enforcement of security isolation. On each host, a reference monitor will enforce security isolation policies assigned to the application, to control the access to the hardware abstractions managed by the host ABI, including files, network sockets, and RPC streams. The hosts need not to be aware of the system interfaces and coordination of the OS states inside of the library OS. The security isolation in the hosts is much more straightforward, simply based on monitoring the references to the host abstractions.

A host of Graphene will isolate applications in a *sandbox*, a container including one or multiple mutually trusting processes. For a multi-process application, Graphene creates multiple library OS instances, which will coordinate to construct a unified OS view. As the library OS instances can coordinate shared OS states using simple RPC streams, it is easy for the hosts to enforce security isolation. The reference monitor simply has to block any RPC streams crossing the sandbox boundary, to prevent applications in different sandboxes from interfering each other through manipulating IPC. The current design focuses on security isolation in an all-or-nothing fashion, although we do expect to extend the design for more sophisticated policies in the future.

**Threat model.** For most of the Graphene hosts (except a SGX host), an application running inside Graphene fully trusts the the library OS, as well as the host OS or hypervisor for exporting the host ABI and enforcing security isolation. the application also trusts other library OS instances in the same sandbox. However, other applications or library OS instances running in a separate sandbox will not be trusted. Other sandboxes on the same host will be adversary to the hosts and benign application clients, by exploiting any vulnerabilities on the host ABI, or in the reference monitor. ~~the Graphene design reduces the attack surface between the hosts and the library OS instances, to defend against a malicious application.~~

The threat model of Graphene on a SGX host (i.e., Graphene-SGX) is similar to other hosts, except the applications running on SGX also distrust the host OS, hypervisor, or other system software. An untrusted OS or hypervisor holds a wide attack surface to invade applications or VMs, using Iago attacks [57]. The challenges to porting Graphene to SGX is not limited to patching the compatibility issues of enclaves, but also requires defending the applications and the library OS against potential Iago attacks.

## 2.2 Graphene Overview

Graphene is a library OS designed for compatibility. Existing library OSes have been built for compatibility against single-process applications [134]. The library OSes map the high-level system APIs onto a few narrowed interfaces to the host kernel. A library OS is comparable to a partial, guest OS running in a virtual machine. However, compared with an actual VM, a library OS eliminates duplicated features between the guest to the host kernel, such as the CPU scheduler or file system drivers, and thus reduce the memory footprint of a guest library OS [116, 134]. Library OSes have also been proven useful for reusing applications on new hardware platforms, such as SGX enclaves [45].

A key drawback for prior library OSes is the limitation on supporting unmodified, multi-process applications. Many existing applications, such as network servers (e.g., Apache) and shell scripts (e.g., GNU makefiles), create multiple processes for performance scalability, fault isolation, and programmer convenience. In order for the efficiency benefits of library OSes to be widely applicable, especially for unmodified Unix applications, library OSes must provide commonly-used multi-process abstractions, such as `fork()`, signals, System V IPC message queues and semaphores, sharing file descriptors, and exit notification. Without sharing memory across processes, the library OS instances must coordinate shared OS states to support multi-process abstractions. For example, Drawbridge [134] cannot simulate process forking because copy-on-write memory sharing is not a universal OS feature.

In Graphene, multiple library OS instances collaboratively implement POSIX ab-

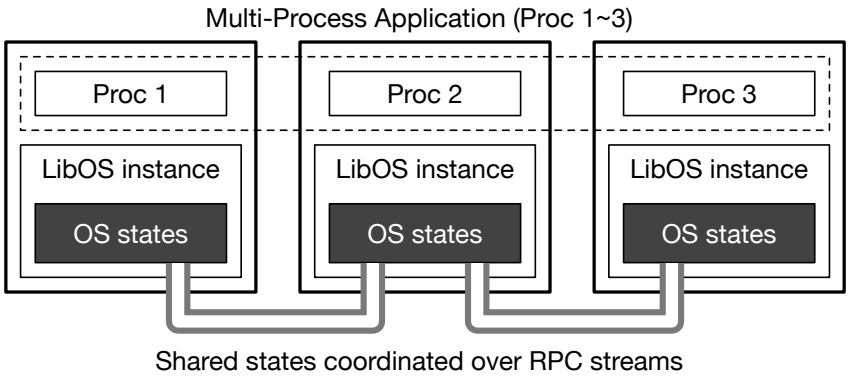



Figure 2.1: Multi-process support model of Graphene library OS. For each process of an application, a library OS instance will serve system calls and keep local OS states. States of multi-process abstractions are shared by coordinating over host-provided RPC streams, creating an illusion of running in single OS for the application.

stractions, yet appear to the application as a single, shared OS. Graphene instances coordinate state using RPC streams bridging the processes. In a distributed POSIX implementation, placement of shared state and messaging complexity are first-order performance concerns. By coordinating shared states across library OS instances, Graphene is able to create an illusion of running in a single OS for multiple processes in an application (Figure 2.1).

### 2.2.1 The architecture

A library OS typically executes in either a paravirtual VM [16, 116] or an OS process, called a *Picoprocess* [44, 134], with interfaces restricted to a narrowed host ABI. The library OS deduplicates features for hardware management in both the guest and host kernels. Graphene executes within a picoprocess (Figure 2.2), which includes an *unmodified* application and its supporting libraries, both of which run on a library OS instance. The library OS is implemented over the kernel ABI designed to expose very generic abstractions that can be easily implemented on any host OS.

The library OS shows that the host ABI is sufficient to implement the guest OS functionality. As an example of this layering, consider the heap memory management abstraction. Linux provides applications with a data segment—a legacy abstraction dating

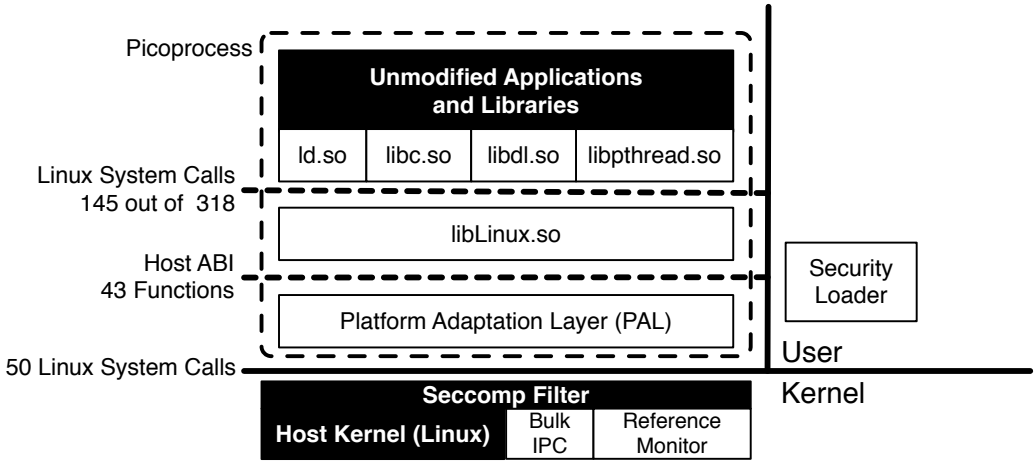



Figure 2.2: Building blocks of Graphene. Black components are unmodified. We modify the four lowest application libraries on Linux: `ld.so` (the ELF linker and loader), `libdl.so` (the dynamic library linker), `libc.so` (standard library C), and `libpthread.so` (standard threading library), that issue Linux system calls as function calls directly to `libLinux.so`. Graphene implements the Linux system calls using a variant of the Drawbridge ABI, which is provided by the platform adaption layer (PAL). A trusted reference monitor that ensures library OS isolation is implemented as a kernel module. Another small module is added for fast bulk IPC, but it is optional for hosts other than Linux.

back to original Unix and the era of segmented memory. The primary thread's stack is at one end of the data segment, and the heap is at another. The heap grows up (extended by `brk()`) and the stack grows down until they meet in the middle. In contrast, the host ABI provides only simple abstraction that allocate, deallocate, or protect regions of virtual memory. This clean division of labor encapsulates idiosyncratic abstractions in the library OS.

At a high level, these library OS designs scoop the layer just below the system call table out of the OS kernel and refactor this code as an application library. The driving insight is that there is a natural, functionally-narrow division point one layer below the system call table in most OS kernels. Unlike many OS interfaces, the host ABI generally minimize the amount of application state in the kernel, facilitating migration: a library OS instance can programmatically read and modify its own OS state, copy the state to another instance, and the remote instancey OS can load a copy of this state into the OS—analogous to hardware registers. A picoprocess may not modify another picoprocess's OS state.

### 2.2.2 Multi-process abstractions

Issue 1.3.b (done): An overview of multi-process support

A key design feature of Unix is that users compose simple utilities to create larger applications. Thus, it is unsurprising that many popular applications for Unix or Linux require multiple processes—an essential feature missing from current library OS designs. The underlying design challenge is minimally expanding a tightly-drawn isolation boundary without also exposing idiosyncratic kernel abstractions or re-duplicating mechanisms in both the host kernel and the library OS.

For example, consider the process identifier (PID) namespace. In current, single-process library OSes, `getpid()` could simply return a fixed value to each application. This single-process design is isolated, but the library OS cannot run a shell script, which requires `fork()`'ing and `exec()`'ing multiple binaries, signaling, waiting, and other PID-based APIs.

**Design Options.** There are two primary design options: (1) implement processes and scheduling in the library OS, and (2) treat each library OS instance as a process, and distribute the shared POSIX implementation across a collection of library OSes. Graphene follows the second option, which imposes fewer host assumptions.

Implementing the multi-process abstractions inside the library OS is also possible using hardware MMU virtualization, similar to Dune [46], but this reintroduces a duplicate scheduler and memory management. Moreover, Intel and AMD have similar, but mutually incompatible MMU virtualization support, which would complicate live migration across platforms. None of these problems are insurmountable, and it would be interesting in future work to compare both options.

In Graphene, multiple library OS instances in multiple picoprocesses collaborate to implement shared abstractions. The supported Linux abstractions include copy-on-write `fork()`, signals, exit notification, and System V IPC semaphores and message queues. For instance, when process A signals process B on Graphene, A's library OS instance issues a query to B's instance over a host RPC stream (similar to a Unix pipe), and B's instance then calls the appropriate signal handler.

Graphene implements all shared abstractions by cooperatively managing the abstraction states over RPC streams. Single-process applications still service system calls from local state, and Graphene, includes optimizations to place state where it is most likely to be used, minimizing RPC overheads. The host reference monitor can easily isolate picoprocesses by blocking all RPC messages, without the need to understand the library OS details or semantics of these abstractions. In the PID example, only mutually-trusting picoprocesses can signal each other.

The Graphene library OS is designed to gracefully handle disconnection from other library OSes, facilitating dynamic application sandboxing. RPC streams may be disconnected at any time by either the reference monitor or at the request of a library OS. When a picoprocess is disconnected, the library OS will handle the subsequent divergence, *transparently* to the application. For instance, if a child process is disconnected from the parent by the reference monitor, the library OS will interpret the event as if the other process terminated—closing any open pipes, delivering exit notifications, etc.

**Comparison with microkernels.** The building blocks of Graphene are very similar to the system abstractions of a microkernel [26, 42, 59, 71, 99, 111, 112], except a microkernel often has a even narrower, more restricted interface than the host ABI. Unlike a multi-server microkernel system, such as GNU Hurd [76] or Mach-US [157], which implements Unix abstractions across a set of daemons that are shared by all processes in the system, Graphene implements system abstractions as a library in the application's address space, and can coordinate library state among picoprocesses to implement shared abstractions. Graphene guarantees isolation equivalent to running an application on a dedicated VM; this isolation could be implemented on a multi-server microkernel by running a dedicated set of service daemons for each application.

The Graphene host ABI could be described as a hybrid microkernel, which also exposes the file system and network of the host kernel. Similarly, picoprocesses are assumed to be provided by a production OS, like Linux or Windows, or by a Type 2 hypervisor. A bare metal hypervisor could potentially export a PAL, but would require services from

a trusted VM, such as Xen's dom0 [41]. Arguably, recent library OS designs might be improved by rethinking the division of labor, but this is beyond the scope of this thesis.

**Alternatives.** Another approach to support multi-process applications in a library OS would be to use hardware MMU virtualization such as nested paging used by a system like Dune [46] in order to implement a second process abstraction, memory manager, and scheduler in the library OS. This approach threatens the efficiency benefits of deduplicating these features. A final option is exposing additional system interfaces, such as signals, by adding more system calls to a picoprocess. This approach undermines compatibility, as many of these coordination abstractions tend to be very OS-specific.

## 2.3 Summary

The Graphene design centers around building a para-virtualized layer to reuse typical OS components (e.g., system calls, namspaces), for reproducing the OS features which are prerequisite to applications. Graphene defines a host ABI, as a new boundary between the OS and user space. The host ABI is designed to be simple enough to port on a new host (containing 42 functions), but expose sufficient functionality from the host to run the virtualized OS components, as a library OS. The host ABI disconnects the complexity of reproducing existing system interfaces for reusing applications, from resolving host-specific challenges occurred in OS development, such as defending applications on SGX.

The Graphene library OS implements a rich of Linux system calls, for both single-process and multi-process applications. To reproduce the multi-process abstractions of Linux, the library OS chooses a design of distributed POSIX namespaces, coordinated using message-passing over simple RPC streams. RPC-based coordination is more adaptable than sharing memory among library OS instances, or virtualizing paging.

# Chapter 3

# The Host ABI

This chapter specifies ~~the formal definition of~~ the host ABI ~~as a component~~ of the Graphene architecture. The host ABI acts as a boundary between the host and the guest (i.e., the library OS), and defines several UNIX-like features. The specification of the host ABI is primarily based on two criterions: *simplicity*, for reducing the development effort per host, and *sufficiency*, for encapsulating enough host-specific system abstractions. The chapter discusses the rationale behind the definition of the current host ABI, according to the experience of porting the host ABI to several host examples, such as Linux, Windows, and SGX.

[dP]: A sentence on motivation might be good here. Same with implication. Like, what is the high level goal? What properties would a good abi have (or how to judge goodness)?

polish

## 3.1 PAL Calling Convention

The host ABI inherits the x86-64 Linux convention. The PAL contains a simple run-time loader that can load the library OS as an ELF (executable and linkable format) binary [74], similar to `ld.so` for loading a user library. Inheriting the Graphene-SGX Linux convention simplifies the dynamic linking between the application, library OS and PAL, and enables compiler-level optimizations for linking, such as function name hashing. Another benefit is to simplify the debugging with GDB, which only recognizes one calling convention at a time.

[dP]: Make this sentence more direct: "The library OS is loaded by the host OS as user libraries."

**Host differences.** A PAL is responsible for translating the calling convention between the host ABI and a system interface on the host. For example, Windows or OSX applications

follow a different calling convention and binary format from Linux applications. On all hosts, the PAL includes a simple ELF loader (even on a Linux host). On a host like Windows or OSX, the PAL is not itself ELF, but rather a host-specific binary that includes an ELF loader.

[dP]: Make this point clearer

**Error codes.** For explicity, a function in the host ABI only returns two types of results: a non-zero number or pointer if the function succeeds, or zero if it fails. Unlike the Linux call convention, the host ABI does not return negative values as error codes (e.g., `-EINVAL`). Instead, the host ABI delivers the failure of a function call as an exception, with the library OS capturing the failure by an ~~assigned~~ exception handler. The design avoids confusing semantics when interpreting return values.

**Dynamic linking vs static linking.** Graphene dynamic links the application, library OS, and PAL in a process. Dynamic linking ensures the complete reuse of an unmodified application, as well as an unmodified library OS implementation. Graphene allows the binaries of application, library OS, and PAL to be deployed individually to the users, and be swapped with different implementations. The dynamic linking of applications is a prerequisite to running most of the Linux applications without modification or recompilation.

However, there are cases where static linking is preferred on a host (e.g., SGX), and recompilation is acceptable to the users. Compiling an application, library OS, and PAL into a single binary is similar to the technique of unikernels [116], which has the benefit of compiling out unnecessary code and execution paths from the binary. Theoretically, it is possible for Graphene to statically link an application with the library OS and a PAL, but this technique is out of the scope of this thesis.

## 3.2 The PAL Calls

This section defines the PAL calls in the host ABI, as a developer's guide to implementing the host ABI for a new host. This section describes the usage of each PAL call in details, followed by a justification of the necessity and simplicity for a host to export such a PAL

[dP]: make this chapter self-contained; should be like a manual of the host ABI; avoid taking about libOS and focus on PAL

call.

### 3.2.1 Stream I/O

I/O is part of the foundation of an OS, to allow an application to interact with other machines, users, applications, or system software. An OS typically supports three types of I/O: **storage**, for externalizing data to a permanent store; **network**, for exchanging data with another machine over internet; and **RPC** (remote procedure call), for connecting concurrent applications or processes. An OS must contain features for all three types of I/O abstractions, and manages the resources on I/O devices, such as hard drives and NICs (network interface controllers). Therefore, unless an I/O device is virtualized and dedicated to an application or a guest, a host OS must take a major role in I/O management; for the least, a host OS has to share the resources among multiple applications or guests, and contain the drivers to interface with the I/O devices.

Issue 1.2.a (done): discuss resource management at host level (I/O)

The basic I/O abstraction in the host ABI is a simple byte stream. A byte stream allows sending or receiving information over an I/O device as a continuous byte sequence. According the type of I/O, a byte stream is restructured as the I/O device demands; for example, on a storage device, a byte stream is logically stored as a sequential file, but physically divided into blocks; on a NIC, a byte stream is transfered as packets, and identified by IP address and port number bound to a network socket; a RPC stream can be simply a FIFO (first-in-first-out), which applications or processes use to pass messages. The host ABI for I/O is similar to the API of a UNIX-style OS, which treats "everything as a file descriptor" and allows utilizing different types of I/O devices through the same file system APIs, including `read()` and `write()`. Managing I/O as byte streams simplifies the development of both the library OS and PALs.

[dP]: Perhaps you want to start with defining a single byte stream abstraction? And then talk about how different URIs leads to different subclasses?

The host ABI identifies I/O streams by URIs (unified resource identifier). A URI is a unique name which describes both the subclass of an I/O stream, and the information for locating or identifying an I/O stream on an I/O device or inside the host OS. The subclass of an I/O stream is identified by the URI prefix, a keyword that represents different types of

I/O: "`file:`" for regular files; "`tcp:`" and "`udp:`" for network connections; and "`pipe:`" for RPC streams. The rest of the URI represents an identifier of the I/O stream: for example, a file can be identified by a path located in a hierachical file system; a network connection can be identified by the socket address. The URIs standardize the way of identifying I/O resources inside various host OSes.

The PAL calls defined in the host ABI for I/O are as follows: `StreamOpen()` creates or opens an I/O stream; `StreamRead()` and `StreamWrite()` send and receive data over an opened I/O stream; `StreamMap()` maps a regular file to the application's memory; `StreamAttrQuery()` and `StreamAttrQuerybyHandle()` retrieves the file metadata and I/O attributes; `StreamWaitForClient()` blocks and creates an I/O stream for incoming network or RPC connection; `StreamSetLength()` truncates a regular file; `StreamFlush()` clears the I/O buffer inside the host OS. The following sections will discuss these PAL calls in details.

[dP]: Define these semantics without a reference to POSIX, so that the document is self-contained.

### Opening or creating an I/O stream

```
HANDLE StreamOpen (const char *stream_uri,
                   u16 access_flags, u16 share_flags,
                   u16 create_flags, u16 options);
```

`StreamOpen()` opens an I/O stream, according to a URI given by `stream_uri` as a string argument. The specification of `StreamOpen()` includes interpreting the URI prefixes and syntaxes of `stream_uri`, and allocating the associated resources in the host OS and on the I/O devices. If `StreamOpen()` succeeds, it returns a **stream handle**. A stream handle is stored by the guest as an identifier to the opened I/O stream. A stream handle is an opauqe pointer, which means the guest should only reference it as an identifier, and never try to interpret the content. On the other hand, if `StreamOpen()` fails (e.g., invalid arguments or permission denied), it returns a null pointer with the failure reason delivered with an exception.

[dP]: The term ''Opaque pointer'' is useful here

Other arguments of `StreamOpen()` specify the options for opening an I/O stream:

[dP]: Need a listing of all the values of flags

- `access_flags` specifies the access mode of the I/O stream, which can be either `RDONLY` (read-only), `WRONLY` (write-only), `APPEND` (append-only), and `RDWR` (readable-writable). The first three access modes are only available for regular files; if the opened stream is a network or RPC stream, the access mode is always `RDWR`. The access modes specify the basic access permissions that an application can request when opening a file. The access permissions are validated by the host OS, based on user configurations. For example, a file configured as append-only for the running application can only be opened in the `APPEND` mode.
- `share_flags` specifies the permissions for sharing a regular file (ignored for other types of I/O streams) with other applications, either in Graphene or in the host OS. `share_flags` can be a combination of six different values: `OWNER_R`, `OWNER_W`, and `OWNER_X` represent the permissions to be read, writed, and executed by the creater of the file; `OTHER_R`, `OTHER_W`, and `OTHER_X` represent the permissions to be read, writed, and executed by everyone else. The permissions are externalized to the host file system; access modes given in future execution are validated against the permissions.
- `create_flags` specifies whether to create a regular file, when it does not exist in the host file system. If `create_flags` is given as `TRY_CREATE`, it creates the file no matter if the file exists. If `create_flags` is given as `ALWAYS_CREATE`, it fails if the file already exists.
- `options` specifies a set of miscellaneous options to configure the opened I/O stream. Currently `StreamOpen()` only accepts one option: `NONBLOCK` specifies that the I/O stream will never block whenever the guest attempts to read or write data. The non-blocking I/O option is necessary for performing asynchronous I/O in the guest, to overlap the blocking time of multiple streams by polling (using `ObjectsWaitAny()`).

According to consecutive operations, handles returned by `StreamOpen()` can be separated into two types: One is a simple byte stream; the other type is a **server handle**, which waits for remote clients to initiate handshakes for establishing a byte-stream con-

nection. A server handle can be bound as a network server or a RPC server. Because a server handle is not a byte stream, it cannot be directly read or written, but can be given to `ServerWaitForClient()` to block and receive a client connection. The host ABI includes the abstraction of creating server handles because receiving client connections requires control at the TCP/IP layer and allocating host resources, which cannot be implemented in the guest unless the network stack is virtualized.

`StreamOpen()` accepts the following URI prefixes and syntaxes for creating a byte stream or a server handle:

- `file:[path]` creates or opens a regular file on the host file system. The opened file is located by a path—either an absolute path from the root of the host file system, or a relative path. A relative path is located from the initial directory where the application is launched, and will never change afterward. `StreamOpen()` accepts relative paths for the convenience of locating application files packaged and shipped together. Note that there could be security concerns that a relative path may collide with another absolute path, or be ambiguous if the path starts with a "dot-dot" (i.e., walking back a directory). Fortunately, both cases can be checked by the guest, as long as the initial directory is specified by the host.
- `tcp:[address]:[port]` or `udp:[address]:[port]` creates a TCP or UDP connection to a remote server, based on the IPv4 or IPv6 address and port number of the remote end. One a connection is created, it will exists until it is torn down by both sides.
- `tcp.srv:[address]:[port]` or `udp.srv:[address]:[port]` create a TCP or UDP server handle which can receive remote client connections. A TCP or UDP server is bound on a IPv4 or IPv6 address and an idle port number. If the specified port number is smaller than 1024, it may require additional privilege from the host OS.
- `pipe.srv:[name]` or `pipe:[name]` create a named RPC server or a connection to a RPC server. The name of a RPC server is an arbitrary, unique string. An RPC stream is an efficient way for passing messages between applications or processes

`[dP]: Mention CWD is relative to where the app is launched from. It may also be worth noting that this is included for convenience, but there are some security risks to using relative paths.`

running on the same host, compare with using a network stream locally. An RPC stream is supposed to be low-latency, and can scale up to significantly more concurrent connections than the limitation on network streams.

`StreamOpen()` defines the scope of enforcing and configuring security isolation in the hosts. The host ABI restricts the sharing of host resources to type types of simple I/O streams (i.e., file, network, and RPC). Other host resources, such as threads and memory, are local to each process, and thus can be isolated by dedicating the host resources. Therefore, in the host ABI, `StreamOpen()` is the only PAL call which requires permission checks in the hosts. Moreover, a user can configure the policies of sharing I/O streams by whitelisting the URIs that are permitted for an application.

### Reading or writing an I/O stream

```
u64 StreamRead  (HANDLE stream_handle, u64 offset,
                 u64 size, void *buffer);
u64 StreamWrite (HANDLE stream_handle, u64 offset,
                 u64 size, const void *buffer);
```

`StreamRead()` and `StreamWrite()` synchronously read and write data over an opened I/O stream. Both PAL calls receive four arguments: a `stream_handle` for referencing the target I/O stream; `offset` from the beginning of a regular file (ignored if the stream is a network or RPC stream); `size` for specifying how many bytes are expected to be read or written; and finally, a `buffer` for storing the read or written data. At success, the PAL calls return the number of bytes actually being read or written.

`StreamRead()` and `StreamWrite()` avoid the semantics of sequential file access to skip the migration of stream handles. A regular file opened by `StreamOpen()` (not in the append-only mode) can only be read or written at an absolute offset from the beginning of the file. The random file access prevents the host OSes to track the offset as an internal state, and allows a migrated guest to reopen the I/O stream on another host without migrating the host OS states. Because all the host OS states associated with an I/O stream is only meaningful to the host, and can be receated anytime, the I/O stream appears to be *stateless*

to the guest.

The host ABI does not includes asynchronous I/O semantics, or peeking into network or RPC buffers inside the host OS. Asynchronous I/O and peeking the buffers are both common OS features that an application may depend on. Although the features are not included in the host ABI, the guest (i.e., the library OS) is supposed to emulate these features using the synchronous `StreamRead()` and `StreamWrite()`, combined with other PAL calls (e.g., `ObjectsWaitAny()`) to prevent blocking on an I/O stream. The guest can also allocate its own buffer to store data prematurely received from an I/O stream, to serve the buffer peeking feature. More details of these features are discussed in Chapter 4.

**Alternative.** An alternative strategy is to define a host ABI with asynchronous I/O semantics. An asynchronous read or write does not return a result immediately; instead, it creates an event handle which can be polled arbitrarily. An ABI that asynchronously reads and writes an I/O stream potentially has more predictable semantics, because the guest can explicitly tell which PAL calls will be blocking. This strategy is taken by Bascule [44]. Graphene chooses synchronous I/O over asynchronous I/O in the current host ABI, because synchronous I/O is a more common feature in host OSes.

### Mapping a file to memory

```
u64 StreamMap (HANDLE stream_handle, u64 expect_addr,
               u16 protect_flags, u64 offset, u64 size);
```

`StreamMap()` maps a file stream to an address in memory, for reading and writing data, or executing code stored in a binary file. `StreamMap()` creates a memory region as either a copy of the file, or a pass-through mapping which shares file updates with other processes. When calling `StreamMap()`, the guest specifies an expected address in memory for mapping the file, or a null address (i.e., zero) for mapping at a ~~random~~ address decided by the host. `expect_addr`, `offset`, and `size` have to be aligned with the allocation granularity of the hosts (more discussion in Section 3.2.2). `protect_flags` specifies the protection

mode of the memory mapping, as a combination of `READ` (readable), `WRITE` (writable), `EXEC` (executable), and `WRITE_COPY` (writable local copy). At success, `StreamMap()` returns the starting address of the mapped area; otherwise, a null address is returned.

The host ABI includes `StreamMap()` for two reasons. First, memory-mapped I/O is suitable for certain file access patterns of applications, and cannot be fully emulated by the guest using `StreamRead()` and `StreamWrite()`. An application often chooses memory-mapped I/O for avoiding the overhead of memory copy and context switch, for frequent, small, random file reads and writes. Second, memory-mapped I/O is asynchronous by nature. The data written to a file-backed memory mapping can be lazily flushed out to the storage; the same feature is difficult to emulate in the guest without an efficient way of marking recently-updated pages (page table dirty bits can only be accessed in host OSes).

Although `StreamMap()` allows multiple processes to map the same file into memory, it does not guarantee the data to be coherently shared across processes. Because memory-mapped I/O is asynchronous, the data written in the memory is only guaranteed to be flushed to the storage when the memory mapping is unmapped. Also, the host ABI drops the assumption of memory sharing, especially for an isolated environment like SGX. It is optional for the host to flush earlier, or to coherently share the memory across multiple processes.

[dP]: there should be an explicit semantics about when the mapping is visible back to the host, like on a sync, with the option to flush earlier

### Listening on a server

```
HANDLE ServerWaitforClient (HANDLE server_handle);
```

`ServerWaitforClient()` waits on a network or RPC server handle, to receive an incoming client connection. A network or RPC server handle cannot be accessed by `StreamWrite()` or `StreamRead()`; instead, the host OS listens on the server handle, and negotiates the handshakes for incoming connections. Once a connection is fully established, the host OS returns a client stream handle, which can be read or written as a simple byte stream. Before any connection arrives, `ServerWaitforClient()` blocks eternally. If a

connection arrives before the guest calls `ServerWaitforClient()`, the host can optionally buffer the connection in a limited backlog; the maximal size of server backlogs is up to the user configurations. The host will drop incoming connections when the backlog is full.

**File and stream attributes**

```
bool StreamAttrQuerybyHandle (HANDLE stream_handle,
                              STREAM_ATTRS *attrs);
bool StreamAttrQuery (const char *stream_uri,
                      STREAM_ATTRS *attrs);
```

Both `StreamAttrQuerybyHandle()` and `StreamAttrQuery()` query the attributes of an I/O stream, and return the attributes in a `STREAM_ATTRS` data structure. The only difference is that `StreamAttrQuerybyHandle()` queries an opened stream handle, whereas `StreamAttrQuery()` queries a URI without opening the I/O stream. `StreamAttrQuery()` is convenient for querying stream attributes when the guest is not planning to access the data of an I/O stream. Both PAL calls return true or false for whether the stream attributes are retrieved successfully.

```
typedef struct {
    u16 stream_type, access_flags, share_flags, options;
    u64 stream_size;
    u64 recvbuf, recvtimeout;
    u64 sendbuf, sendtimeout;
    u64 lingertimeout;
    u16 network_options;
} STREAM_ATTRS;
```

The `STREAM_ATTRS()` data structure consists of multiple fields specifying the attributes assigned to an I/O stream since creation. `stream_type` specifies the type of I/O stream that the handle references to. `access_flags`, `share_flags`, and `options` are the same attributes assigned to an I/O stream when the stream is created by `StreamOpen()`. `stream_size` has different meanings for files and network/RPC streams: if the handle is a file, `stream_size` specifies the total size of the file; if the handle is a network or RPC

stream, `stream_size` specifies the size of pending data currently received and buffered in the host.

The remaining attributes are specific to network or RPC streams. `recvbuf` and `sendbuf` specify the limitation of buffering the pending bytes, either inbound or outbound. `recvtimeout` and `sendtimeout` specify the receiving or sending timeout (in microsends) before a stream is considered being disconnected abruptly. `lingertimeout` specify the timeout for closing or shutting down a connection to wait for the pending outbound data. `network_options` is a combination of flags that specify the options of configuring a network stream. Currently `network_options` accepts the following generic options: `KEEPALIVE` (enabling keep-alive messages), `TCP_NODELAY` (no delay on sending small data), and `TCP_QUICKACK` (no delay on sending ACK responses).

```
bool StreamAttrSetbyHandle (HANDLE stream_handle,
                            const STREAM_ATTRS *attrs);
```

`StreamAttrSetByHandle()` is a PAL call newly introduced by Graphene. `StreamAttrSetByHandle()` changes the attributes initially assigned to an I/O stream, and externalizes the change to the host OS. `StreamAttrSetByHandle()` is given an updated `STREAM_ATTRS` data structure, which contains the new attributes to be assigned to the I/O stream. `stream_type` cannot be changed, as well as any attributes that violate the limitation imposed by the host.

A dilemma for defining the `STREAM_ATTRS` data structure is to decide which stream attributes, especially the attributes for a network stream (i.e., flags included in `network_option`), should be exposed by the host, A network stream attribute can be derived from an optional feature inside the host network stack, or a configuration at the NIC level. Exposing these stream attributes allows the guest to export APIs for applications to fine-tune the I/O performance. However, exposing too many attributes makes the host ABI less portable on different host OSes, since these attributes may not have their equivalences in certain host OSes. Eventually, a guest should not expect every attributes defined in `STREAM_ATTRS` to be always configurable, and `StreamAttrSetByHandle()` will raise a failure if the guest tries to set an unavailable attribute.

```
bool StreamSetLength (HANDLE stream_handle, u64 length);
```

Finally, `StreamSetLength()` expands or truncates a file stream to a specific length. In general, the data blocks on storage media are allocated dynamically to a file when the file length grows. If `StreamWrite()` writes data beyond the end of a file, it automatically expands the file, by allocating new data blocks on the storage media. However, a file-backed memory mapping created by `StreamMap()` lacks an explicit timing to expand the file when writing to the memory mapped beyond the end of file. `StreamSetLength()` can explicitly request the host to expand a file to an appropriate length, so that consecutive memory write will never raise memory faults. `StreamSetLength()` can also shrink a file to the actual data size if the file has overallocated resources earlier.

**Listing a directory.** Graphene extends the stream I/O feature in the host ABI to retrieve directory information. A file system usually organizes files in directories, and allows applications to retrieve a list of files in a given directory. Instead of adding new PAL calls for directory operations, the host ABI uses existing PAL calls, namely `StreamOpen()` and `StreamRead()`, for listing a directory. When `StreamOpen()` is given a file URI that points to a directory, such as "`file:/usr/bin`", `StreamOpen()` returns a stream handle which allows consecutive `StreamRead()` calls to read the file list as a simple stream. The stream handle that references to a directory can only be read as FIFO (first-in-first-out), and the returned data should contain a series of file names as null-terminated strings. The stream handle cannot be written or mapped into memory.

[dP]: Always start with what you did, and then discuss alternatives.
I think the heart of issue isn't clear. It seems that you are adopting a POSIX model of treating a directory like a file.

**Character devices.** The host ABI also supports reading or writing data over a character device, such as a terminal. A terminal can be connected as a stream handle, using a special URI called `dev:tty`. Other character devices include the debug stream of a process (the URI is `dev:debug`), equivalent to writing to `stderr` in POSIX.

### 3.2.2 Page Management

In the hos ABI, the abstraction for page management is a **virtaul memory area (VMA)**, a page-aligned, nonoverlaping region in the guest's virtual address space. A virtual memory area specifies the memory region that requires the host OS to allocate the page resources, either statically or dynamically. There are two types of VMAs: one is a file-backed VMA, which is created by `StreamMap()` and backs the memory pages with file data blocks. The other type is an anoumous VMA, which is purely in DRAM and not backed by any files. Either types of VMAs is part of the virtual address space, and a VMA should never overlap with others created in the same virtual address space. The host OS can choose to populate all the pages for a VMA immediately at creation of the VMA, or delay the allocation until the first memory access (i.e., demand paging).

Issue 1.2.a (done): discuss resource management at host level (pages)

```
u64  VirtMemAlloc   (u64 expect_addr, u64 size,
                     u16 protect_flags);
```

`VirtMemAlloc()` creates an anonymous VMA in the guest memory. When `VirtMemAlloc()` is given an expected address, the host OS must try to create the VMA at the exact address. Otherwise, if no address is given, `VirtMemAlloc()` can create the VMA ~~at~~ wherever the host OS sees fit, and does not overlap with existing VMAs. Both `expect_addr` and `size` must be page-aligned, and never exceed the permitted range in the guest's virtual address space. `protect_flags` specifies the page protection in the created VMA, and can be given a combination of the following values: `READ`, `WRITE`, and `EXEC` (similar as `StreamMap()` but without `WRITE_COPY`). If `VirtMemAlloc()` succeeds, it returns the starting address of the created VMA, which the guest is permitted to access up to the given size.

```
bool VirtMemFree    (u64 addr, u64 size);
bool VirtMemProtect (u64 addr, u64 size,
                     u16 protect_flags);
```

`VirtMemFree()` and `VirtMemProtect()` modify one or more VMAs, by either freeing the pages or adjusting the page protection in an address range. Both PAL calls

specify the starting address and size of the address range to modify; the given address range must be page-aligned, but can be any part of the guest virtual address apace, and overlap with any VMAs, either file-backed or anonymous. If the given address range overlaps with a VMA, the overlapped part is divided into a new VMA, and be destroyed or protected accordingly.

The challenge to defining the host ABI for page management is to accommodate different allocation models and granularities of the hosts. A POSIX-style OS often assumes dynamic allocation with page granularity (normally with four-kilobyte pages); the assumption is deeply ingrained in the design of page fault handler sand page table management inside an OS like Linux or BSD; the page management component in an OS is usually ~~low-level and~~ closely interacting with the hardware interface, to serve the needs of both the OS and applications. Such an OS design makes it difficult to move page management into the guest, unless using hardware virtualization such as VT [164], which virtualizes page fault handlers and page table management to the guest.

Moved from the beginning of this section

[dP]: Needs a cite, and probaly a more specific mention of what feature of VT you have in mind

### 3.2.3 CPU Scheduling

The host ABI for CPU scheduling includes two abstractions: one is the creation of a **thread**, an execution unit allocated by the guest, to be scheduled to run on a CPU core. The other abstraction is a set of **scheduling primitives**, designed for synchronization and coordination among multiple threads.

Issue 1.2.a (done): discuss resource management at host level (threading)

The thread abstraction requires the host OS to contain a host scheduler. A host scheduler will dynamically assign one of the living threads to each CPU core, to allow the thread to continue execution until rescheduling. The scheduling algorithm of a host scheduler is up to the design and configuration of the host OS; however, a host scheduler does have to ensure every thread to follow its expected behaviors, regardless of the scheduling algorithm. For the least, a host scheduler should avoid completely starving one of the living threads, so that the guest can make progress as expected. Other CPU scheduling criteria such as fairness, throughput, and CPU utilization are still critical to the application perfor-

mance, but the host scheduler is responsible of improving these criteria.

### Creating or terminating a thread

```
HANDLE ThreadCreate (void (*start) (void *),
                     void *param);
```

`ThreadCreate()` creates a living thread that can be immediately scheduled by the host scheduler. The arguments of `ThreadCreate()` specify the initial state of the new thread, including a function to start the thread execution, and a parameter being passed to the function. As soon as `ThreadCreate()` successfully returns, the caller thread and the created thread should both be live in the host OS. When `ThreadCreate()` succeeds, it returns a thread handle that represents the created thread to the caller.

`ThreadCreate()` is simplified in several ways. First, `ThreadCreate()` does not allow the guest to specify the initial stack where the new thread starts execution. Instead, the host OS takes the liberty of allocating an initial, fixed-size stack for the new thread, but the new thread is free to switch to a user-assigned stack afterward. Second, `ThreadCreate()` takes no creation options except a starting function and a parameter. Every threads created by `ThreadCreate()` should look identical to the host, and share every resources assigned to the process.

```
void ThreadExit (void);
```

`ThreadExit()` simply terminates the current thread in the host OS. The PAL call takes no argument, and should never return if it succeeds. The purpose of `ThreadExit()` is to free the resources allocated in the host OS for creating the current thread, including the initial stack.

### Scheduling a thread

The host ABI allows a running thread to voluntarily give up the CPU core, or to be interrupted by other threads. In either ways, the thread is suspended until being

rescheduled to a CPU core. The purpose of rescheduling a thread is to prevent the thread to busily waiting for a specific condition, such as a variable being set to specific value, or a specific time in the future. Busy-waiting wastes CPU cycles, and can potentially prevent other threads from being scheduled, if the host scheduler does not implement a time-slicing scheduling algorithm such as round-robin. Although the guest delegates scheduling to the host scheduler, the guest can proactively request for scheduling to improve CPU throughput.

```
u64  ThreadDelay (u64 delay_microsec);
void ThreadYield (void);
```

Both `ThreadDelay()` and `ThreadYield()` suspend the current thread for rescheduling. `ThreadDelay()` suspends the current thread for a period of time, and the length of suspension is specified by `delay_microsec`, in microseconds. If the thread is suspended successfully and rescheduled after expiration of the specified period, `ThreadDelay()` returns zero and resumes the thread execution. If the thread is rescheduled prematurely, due to interruption of other threads (using `ThreadInterrupt()`), `ThreadDelay()` returns the remaining suspension time in microseconds.

[dP]: Define the semantics a bit more. When you do a delay, presumably wait at least that long; are theere any cases you can return early (e.g., EINTR)?

`ThreadYield()` simply yields the execution of current thread, and the thread can be rescheduled immediately by the host scheduler. `ThreadYield()` allows a thread to request for rescheduling when the thread expects to wait for certain conditions. When `ThreadYield()` is called, the host scheduler will suspend the current time slice, and rerun the scheduling algorithm to select a runnable thread.

[dP]: When you do a yield, under what conditions are you rescheduled?

```
void ThreadInterrupt (HANDLE thread_handle);
```

`ThreadInterrupt()` reschedules another thread, either running or suspended, based on a given thread handle. `ThreadInterrupt()` has two primary purposes. First, `ThreadInterrupt()` can interrupt the suspension of a thread, and force the thread to resume execution immediately. Second, `ThreadInterrupt()` can interrupt the execution of a running thread, so that the thread can instantaneously respond to a sudden event. Without `ThreadInterrupt()`, a running thread can only detect the occurrence of an event at a certain “checkpoint” in the

code.

**Scheduling options.** The host ABI currently contains no scheduling options for the guest to configure the host scheduler. An OS usually allows an application to set certain scheduling options, such as assigning the scheduling priority of a thread, or configuring the scheduling policies. For simplicity, the host ABI completely delegates scheduling to the host scheduler, and only allows host-level, user configuration for setting the scheduling options statically. The simplicity prevents the host from exposing a wide interface for configuring the host scheduler, but such a host ABI would fail to support most of the scheduling options available in Linux (e.g., `sched_setparam()`).

Luckily, most of the scheduling options in Linux does not impact the functionality of an application. For example, without setting the scheduling priority, the application can still make progress, but may suffer performance penalty due to unnecessary CPU idles. The only exception is CPU affinity, as binding a thread to one or multiple CPU cores. CPU affinity is important for an application with a producer-consumer programming model, wherein the consumer busily waits for the producer. Such a producer-consumer model requires the producer and consumer threads to be scheduled on different CPU cores, to prevent being deadlocked by the scheduler. We propose adding a PAL call called `ThreadSetCPUAffinity()` to support binding a thread to CPU cores:

```
bool ThreadSetCPUAffinity (u8 cpu_indexes[], u8 num);
```

`ThreadSetCPUAffinity()` binds the current thread to a list of CPU cores, as specified in `cpu_indexes`. `cpu_indexes` is an array of non-negative integers, which are smaller than the total number of CPU cores.

[dP]: Explain how the arguments work

**Scheduling primitives**

The host ABI includes two scheduling primitives: one is mutex (mutually-exclusive) locking, and the other is a waitable event object. The purpose of including scheduling primitives in the host ABI is to improve the user-space synchronization, which is generally

implemented by atomic or compare-and-swap (CAS) instruction; with user-space synchronization, a thread will spin on a CPU core until the state of a lock or an event is atomically changed. The host-level scheduling primitives can avoid the spinning by suspending a blocking thread in the host scheduler, until being signaled by another thread.

```
HANDLE MutexCreate (void);
void   MutexUnlock (HANDLE mutex_handle);
```

`MutexCreate()` creates a handle that can be used as a mutex lock. A mutex lock enforces atomic execution in a critical section: if multiple threads are competing over a mutex lock before entering the critical section, only one thread can proceed while other threads will block until the lock is released again. `MutexUnlock()` releases a mutex lock held by the current thread. To acquire a mutex lock, a generic PAL call, `ObjectsWaitAny()` (defined later), can be used to compete with other threads, or wait for the lock release if the lock is held.

```
HANDLE SynchronizationEventCreate (void);
HANDLE NotificationEventCreate    (void);
void   EventSet   (HANDLE event_handle);
void   EventClear (HANDLE event_handle);
```

`SynchronizationEventCreate()` and `NotificationEventCreate()` create two different types of waitable event objects, as synchronization and notification event objects. Any thread can use `EventSet()` to signal an event. Signaling a synchronization event object allows exactly one waiting thread to continue its execution, immediately or in the future. A synchronization event object can be used to coordinate threads that cooperate as producers and consumers; a producer thread can signal exactly one consumer at a time. On the other hand, a notification event object can stay signaled until another thread manually resets the event object, using `EventClear()`. A notification event object can be used for notifying the occurrence of a one-time event, such as the start or termination of an execution. Similar to acquiring a mutex lock, `ObjectsWaitAny()` can also be used to wait for an event object to be signaled.

### Waiting for scheduling events

```
HANDLE ObjectsWaitAny (HANDLE *handle_array,
                       u8 handle_num, u64 timeout);
```

The host ABI defines a generic PAL call, `ObjectsWaitAny()`, to wait for various kinds of events that are assoicated with a given handle array (specified by `handle_array` and `handle_num`). A common usage of `ObjectsWaitAny()` is to perform a blocking operation on a scheduling primitive, such as acquiring a mutex lock, or waiting for the signaling of an event object. If the mutex lock is successfully acquired, or the event is signaled, `ObjectsWaitAny()` stops blocking and returns the target handle. `ObjectsWaitAny()` only accepts one handle if the handle is a mutex lock or an event object; waiting for multiple mutex locks or event objects is not supported by the host ABI, because the feature has no concrete use case in the guest, and can be tricky to implement due to lack of similar system API.

`ObjectsWaitAny()` also receives a `timeout` argument to prevent the current thread to wait for the events indefinitely. If the timeout expires before the occurrence of any events related with the given handles, `ObjectsWaitAny()` stops blocking and returns a null handle.

`ObjectsWaitAny()` can also be used for polling multiple stream handles, to wait for I/O events such as receiving inbound data or sudden failure. Unlike a mutex lock or an event object, a stream handle can be associated with multiple I/O events. Therefore, the host ABI introduces a PAL call, `StreamGetEvent()`, to create a stream event handle that represents a specific I/O event of the given stream handle. The definition of `StreamGetEvent()` is inspired by Bascule [44].

```
HANDLE StreamGetEvent (HANDLE stream_handle, u16 event);
```

`StreamGetEvent()` receives a stream handle and a specific I/O event. The `event` argument can be given one of the following values: `READ_READY`, for notifying that there are inbound data ready to be read; `WRITE_READY`, for notifying that a network connection

[dP]: Could you accomplish the smae thing by creating more than one handle for a network socket?

is fully established and ready to be written; and `ERROR`, for notifying that certain failures occur on the stream.

### Thread-local storage

Some OSes, such as Linux and Windows, requires a thread-local storage (TLS), either to store thread-private variables, or to maintain a thread control block (TCB). A TLS area can potentially be frequently accessed by an application or library. Therefore, on x86-64, the TLS is often referenced by one of the FS and GS segment registers, which is assigned a unique pointer. A thread can access a state in its own TLS by referencing a specific offset from the FS/GS register. Also, retrieving or assigning the value of the FS/GS register is a privilege operation that must be performed in the host OS kernel.

```
u64 SegmentRegisterAccess (u8 register, u64 value);
```

The host ABI introduces a PAL call, `SegmentRegisterAccess()`, for reading or writing the FS/GS register value. The `register` argument can be either `WRITE_FS` or `WRITE_GS`, with the `value` argument being a pointer that references to the TLS area. Otherwise, the `register` argument can be `READ_FS` or `READ_GS`, to retrieve the FS/GS register value. Regardless, `SegmentRegisterAccess()` returns the current value of FS/GS register.

Unfortunately, the feasibility of implementing `SegmentRegisterAccess()` deoends on the host OS. Linux and similar OSes allow the usage of FS/GS register, primarily because the FS register is heavily used in the standard C library. However, in other OSes, especially Windows and OSX, changing the FS/GS register is forbidden by the OS kernels. The Windows 7, 8, and 10 kernels confiscate the FS register for storing a thread control block (TCB), and thus forbid users to change the FS register value. OSX's xnu kernel considers FS/GS registers to be of no concrete use. These OS kernels have been aggressively resetting the FS/GS register values to mitigate any user attempt of changing them. Therefore, the host ABI declares `SegmentRegisterAccess()` as one of the optional PAL calls, as the guest should consider developing a workaround if the PAL call is not supported on a certain host.

Issue 1.3.e (done): discuss the FS/GS limitation

### 3.2.4 Processes

The host ABI creates a process as a new guest to run on the current host. A process in the host perspective is a *picoprocess*, which consists of brand-new instances of the PAL, the library OS, and a specific application. The host ABI defines a process as a simple abstraction, which owns a new address space, and starts with a clean state of no guest VMAs, no held I/O resources, and no allocated handles. Moreover, a new process will not share any memory with former processes. The definition of the process abstraction is meant to simplify the host design for expanding a single-process execution to multiple processes.

[dP]: don't mix perspective; focus in this chapter on the PAL model only. I would do minimal discussion of what the libOS does here.

```
HANDLE ProcessCreate (const char *application_uri,
                      const char *manifest_uri,
                      const char **args, uint flags);
```

`ProcessCreate()` creates a process (or picoprocess) to run an application specified by `application_uri`, a URI that identifies the application executable. `ProcessCreate()` allows specifying the user configuration according to a given manifest file (i.e., `manifest_uri`), as well as passing command-line arguments (i.e., `args`) to the new process. The effect of calling `ProcessCreate()` is mostly equivalent to relaunching the specified application in Graphene, except two distictions: `ProcessCreate()` returns a process handle to its caller; also, a process created by `ProcessCreate()` belongs to the same *sandbox*—an isolated container of related processes—with its parent process. The detail of the sandbox abstraction is discussed in Section 3.2.5.

To bootstrap the inter-process communication, a process handle also works as an unnamed RPC stream connecting the parent and child processes. The guests in the parent and child processes can use this RPC stream to share internal states, as well as to inherit I/O stream handles from each other.

#### Sharing a handle

Due to the statelessness of handles, a guest can cleanly migrate its state to a new process, and recreate all handles afterward. Unfortunately, not all I/O streams can be recre-

ated in a new process, due to the host limitations; for example, in most host OSes, a network connection is bounded to a process, and can only be shared through inheriting the network handle or file descriptor from the parent process. Since every process created by `ProcessCreate()` is a clean picoprocess without inheriting any stream handles, a guest needs a host feature to share a network stream handle with other processes.

```
void   RpcSendHandle (HANDLE rpc_handle, HANDLE cargo);
HANDLE RpcRecvHandle (HANDLE rpc_handle);
```

The host ABI introduces `RpcSendHandle()` and `RpcRecvHandle()` for sharing I/O stream handles over a RPC stream (a process handle is also used as a RPC stream). `RpcSendHandle()` migrates the host state of a stream handle, specified by `cargo`, over a RPC stream handle. `RpcSendHandle()`, which is called inside another process, then receives the migrated host states from the RPC stream. `RpcSendHandle()` will grant the receiving process permissions to access the I/O stream handle. If `RpcSendHandle()` succeeds, it returns a handle that references to the shared I/O stream. The abstraction is similar to a feature in Linux and similar OSes that shares file descriptors over a UNIX domain socket.

**Bulk IPC (physical memory store)**

The host ABI introduces an optional bulk IPC feature, as a faster alternative to RPC stream. The optimization brought by the feature is to reduce the latency of sending large chunks of data across processes. The main abstraction of bulk IPC is a physical memory store. Multiple processes can open the same memory store; a processes sends the data in a piece of page-aligned memory to the store, while another process maps the data to its memory. Since the host can enable the copy-on-write sharing on the data mapped to both processes, the latency can be much shorter than copying the data over a RPC stream.

```
HANDLE PhysicalMemoryStore  (u32 index);
```

`PhysicalMemoryStore()` creates or attaches to a physical memory store, based on

a given index number. The indexing of physical memory stores is independent for each sandbox (the container abstraction discussed in Section 3.2.5), so unrelated guests cannot share a physical memory store by specifying the same index number. If `PhysicalMemoryStore()` succeeds, it returns a handle that references to the physical memory store. The store is alive until every related processes close the corresponding store handles, and no data is left in the store.

```
u64    PhysicalMemoryCommit (HANDLE store_handle,
                             u64 addr, u64 size);
u64    PhysicalMemoryMap    (HANDLE store_handle,
                             u64 addr, u64 size,
                             u16 protect_flags);
```

`PhysicalMemoryCommit()` commits the data in a memory range to a physical memory store. Both `addr` and `size` must be aligned to pages, so that the host can enable copy-on-write sharing if possible. `PhysicalMemoryMap()` maps the data from a physical memory store to a memory range in the current process. `protect_flags` specifies the page protection assigned to the mapped memory ranges.

### 3.2.5 Sandboxing

The security isolation of Graphene is based on a **sandbox**, a container isolating a number of coordinating library OS instances. When Graphene launches an application, the application begins running inside a standalone sandbox. By default, a new process cloned by the application share the sandbox with its parent process. To configure the isolation policies, developers provide a **manifest** file for each application. The policies are enforced by a reference monitor in the host. A manifest file contains run-time rules for sandboxing resources which can be shared in the host, including files, network sockets, and RPC streams.

Sandboxing delegates security isolation to the host. An application doesn't have to trust the library OS to enforce security policies, on every applications running on the same host. If a library OS instance is compromised by the application, the threat will be contained inside the sandbox, and cannot cross the sandbox boundary, unless the host

is also compromised. For each sandbox, the isolation policies are statically assigned, in the manifest file given at the launch. The isolation policies cannot be subverted during execution.

The host ABI also introduces a PAL call, `SandboxSetPolicy()`, to dynamically move a process to a new sandbox. Sometimes, an application needs to reassign the rules of security isolation, for enforcing stricter rules inside the application. A multi-sandbox environment can protect an application with multiple privilege levels, or an application that creates session for separating the processing for each client. With `SandboxSetPolicy()`, a process that requires less security privilege or serves a separate session can voluntarily moves itself to a new sandbox, with stricter rules. `SandboxSetPolicy()` can dynamically assign a new manifest file that specifies the new rules, to be applied to the new sandbox created for the current process.

```
bool SandboxSetPolicy (const char *manifest_uri,
                       u16 sandbox_flags);
```

`SandboxSetPolicy()` receives a URI of the manifest file that specifies the sandboxing rules, and an optional `sandbox_flags` argument. The `sandbox_flags` argument currently can only contain one value: `SANDBOX_RPC`, for isolating the RPC streams between the original sandbox and the new sandbox.

### 3.2.6 Miscellaneous

Besides managing host resources, some miscellaneous features is needed from the host OSes. Some features, such as exception handling, are specific to the implementation of Linux functionality in the guest. This section lists these miscellaneous PAL calls defined in the host ABI.

#### Exception handling

The exception handling in the host ABI is strictly designed for returning hardware exceptions, or failures inside the PAL. The host ABI allows the guest to specify a **handler**,

which the execution will be redirected to, when a specific exception is triggered. The feature of assigning handlers to specific exceptions grants a guest the ability of recovering from hardware or host OS failures.

```
typedef void (*EXCEPTION_HANDLER)
            (void *exception_obj, EXCEPTION_INFO *info);
```

The host ABI defines `EXCEPTION_HANDLE` as the data type of a valid handler function. A valid handler accepts two arguments. The first is an exception object, as an opague pointer which the host OS maintains to store a host-specific state regarding the exception. The second is a piece of exception information that is revealed to the exception handler. The content of the exception information is defined as follows:

```
typedef struct {
    u8  exception_code;
    u64 fault_addr, registers[REGISTER_NUM];
} EXCEPTION_INFO;
```

The `EXCEPTION_INFO` data structures consists of three fields. `exception_code` specifies the type of exception. `fault_addr` specifies the address that triggers a memory fault, or an illegal instruction. `registers` returns the value of all x86-64 general-purpose registers when the exception is raised. The exception code can be one of the following values:

- `MEMFAULT`: a protection or segmentation fault.
- `DIVZERO`: a divide-by-zero fault.
- `ILLEGAL`: an illegal instruction fault.
- `TERMINATED`: terminated by the host.
- `INTERRUPTED`: interrupted by `ThreadInterrupt()` (defined in Section 3.2.3).
- `FAILURE`: a failure in the host ABI.

When an exception is raised, the current execution is interrupted and redirected to the assigned handler function. The handler function can try to recover the execution, based on the information given in the `EXCEPTION_INFO` data structure. For example, a

handler function can print the interrupted register values to the terminal. Once a handler function finishes processing the exception, it can return to the original execution, by calling `ExceptionReturn()` with the exception object given by the host.

```
void ExceptionReturn     (void *exception_obj);
```

The host ABI defines `ExceptionReturn()` to keep the semantics of exception handling clear and flexible across host OSes. A handler function does not assume that it can return to the original execution using the `ret` or `iret` instruction. Instead, a handler function must explicitly call `ExceptionReturn()`, so that the host can destroy the frame that belongs to the handler function, and return to the interrupted frame. Also, `ExceptionReturn()` can update the register values pushed to the interrupted frame, based on the `registers` field in `EXCEPTION_INFO`.

```
bool ExceptionSetHandler (u8 exception_code,
                          EXCEPTION_HANDLER handler);
```

`ExceptionSetHandler()` assigns a handler function to a specific exception, based on the given `exception_code`. The assignment of exception handlers applies to every threads in the same process. If `ExceptionSetHandler()` is given a null pointer as the handler, it cancels any handler previously assigned to the exception. If no handler is ever assigned to a specific exception, the default behavior of handling the exception is to kill the whole process.

### Querying the system time

```
u64 SystemTimeQuery (void);
```

`SystemTimeQuery()` returns the current system time as the number of microseconds passed since the Epoch, 1970-01-01 00:00:00 Universal Time (UTC). Querying the system time requires the host to have a reliable time source. A common reliable, time source on x86-64 is a system timer incremented by the hardware alarm interrupts [115],

combined with the Time Stamp Counter (TSC), a CPU counter tracing the number of cycles since the system reset. `SystemTimeQuery()` exports a reliable, simple time source to the guest, based on the calculation of any arbitrary time sources used in the host.

#### Reading random bits

```
u64 RandomBitsRead (void *buffer, u64 size);
```

`RandomBitsRead()` fills the given buffer with data read from the host random number generator (RNG). If `RandomBitsRead()` successfully reads up to the number of bytes specified by `size`, it returns the number of bytes that are actually read. Based on the host random number generator, `RandomBitsRead()` may block until there is enough entropy for generating the random data.

The purpose of `RandomBitsRead()` is to leverage the hardware random number generators, either on-chip or off-chip. For example, recent Intel and AMD CPUs support the `RDRAND` instruction, which generates random bytes based on an on-chip entropy source. Other hardware RNGs also exist, mostly based on thermodynamical or photoelectric patterns of the hardware. Graphene only requires each host to export one trustworthy source of random data, such as `/dev/random`, a pseudo-device in POSIX.

## 3.3 Summary

The host ABI consists of a sufficient set of simple, UNIX-like OS features. The goal of defining the host ABI is to ensure that the host ABI can be implemented relatively easily on most hosts, and simultaneously, to expose enough host abstractions for developing a library OS that runs a lot of Linux applications. Functions in the host ABI, or so-called PAL calls, either manage a ubiquitous hardware resource, such as pages or CPU cycles, or encapsulate an idiosyncratic feature of the host OSes, such as scheduling primitives or exception handling. For most of the PAL calls, system API with similar functionality and semantics can be found on most host OSes; Few exceptions (e.g., setting segment registers),

[dP]: I think related goals are to show that 1) the ABI can be implemented relatively easily on most hosts. 2) that it is sufficient to run a lot of Linux applications.

which are limited to be implemented on certain host OSes, are defined as optional features, so that the library OS can be prepared with designing workarounds.

# Chapter 4

# The Library OS

This chapter demonstrates how a practical, rich-feature library OS can be developed upon the PAL ABI, to run unmodified Linux applications. The major challenge in building the Graphene library OS, or `libLinux`, is to recreate the features of the Linux system interface, including system calls and namespaces, There are two primary criterion for `libLinux`: compatibility and performance. Compatibility of `libLinux` depends on richness of the implemented Linux features and API, while performance of `libLinux` is determined by the emulation strategies based on characteristics of the PAL ABI.

Issue 1.1.a (done): discuss the technical aspects of libOS

Use ‘‘thelibos’’ macro for the whole paper

## 4.1 Resource Management

`libLinux` relies on the host OSes to manage hardware and privileged OS resources. The PAL ABI defines a set of host abstractions—such as I/O streams, virtual memory areas (VMAs), and threads—for the development of a library OS. These host abstractions delegate the management of a few ubiquitously-installed hardware resources, such as I/O devices, memory, and CPUs, to the host OSes. Other abstractions defined by the PAL ABI, such as a local RPC stream and a system clock, depend on low-level, privileged resources in the host OSes, such as in-kernel queues and time sources. To support `libLinux` as a guest, the PAL ABI avoids the requirement of exposing or virtualizing these hardware and privileged OS resources, by encapsulating the resources as generic, user-space abstractions.

For resource management, the role of `libLinux` is to allocate the host abstractions,

Issue 1.2.a (done): discuss the role of libOS in resource management

as unambiguous requests for the host-managed resources. At a high level, the purpose of `libLinux` is to recreate the Linux abstractions. `libLinux` implements the Linux abstractions based on managing the host abstractions instead of the underlying resources. For example, if a Linux abstraction requires allocating pages for either usage in an application or internal bookkeeping, the implementation in `libLinux` will allocate a VMA, which is the memory abstraction of the PAL ABI, instead of physical pages. Such a library OS design operates on the faith that the host OS will manage and assign pages to VMAs, with reasonable fairness as well as efficiency. Unless the allocation exceeds user quotas or host limitations, the library OS should be allowed to obtain more host-manged resources, by increasing the allocation of a host abstraction.

The role of `libLinux` in resource management is close to a language runtime, such as a Java virtual machine [9, 10, 28], or a Python [18] or Perl [17] runtime. A language runtime generally relies on system APIs exported by the OSes for resource management. A common behavior of a language runtime is to use system calls like `mmap()` to allocate a large heap, which is later chunked into objects and assigned to variables in an application. Similar strategies have been applied to filesystem or threading abstractions in a language runtime.

A primary task and challenge to managing resources in `libLinux` is to reproduce the idiosyncratic features or requirements of Linux, using only the host abstractions defined by the PAL ABI. Take page management for example. Linux contains several memory abstractions, including VMAs allocated by `mmap()`, a self-growing stack in each process, and a fine-grained heap allocated by `brk()`. Since `libLinux` does not directly manage pages, it requires particular emulation strategies to implement the allocation models that applications expect when using these Linux abstractions. A strategy repeatedly used in `libLinux` and language runtimes is to "overallocate" certain host abstractions when an application requests for resources. The purpose of overallocation is to keep the flexibility of adjusting the resources afterward. The caveat of using these kinds of strategies is that they are based on an assumption that the host allocates the resources on demand, instead

of populating the resources all at once. However, such an assumption does not apply to all hosts; for example, the current version of SGX requires a static virtual memory layout, and each VMA is at least fully populated once at enclave creation for checking the integrity of memory data. Therefore, overallocating VMAs slows down the creation of an enclave.

End with a summary of the paragraph?

**Alternatives.** Virtualization is another approach to allow guest-level resource management. There are two primary strategies to fully virtualize hardware resources for a guest OS. The first strategy is to rely on a hypervisor, such as QEMU [19] or VMWare ESX [166], to emulate the physical hardware, giving the guest OS the illusion of having full control of all the hardware resources. Another strategy to leverage the hardware virtualization, such as IOMMU [53], which allows a hypervisor to dedicate physical hardware resources to guest OS instances. Both of the virtualization strategies grant a guest OS with more control on managing hardware resources than `libLinux` in the Graphene architecture.

Exokernel [72] also uses a library OS

### 4.1.1 Virtual address space

`libLinux` allocates memory resources for two reasons. The first reason is to service the requests from the applications and user libraries, through system APIs such as `mmap()`. The second reason is to maintain the internal states of the library OS, for either implementing the Linux features and APIs, or managing the allocated host abstractions.

To allocate memory, `libLinux` creates VMAs (virtual memory areas) in the host OS.

`libLinux` is in control of the whole virtual address space of a picoprocess, except the initial PAL mappings.

Internally, `libLinux` maintains a list of VMA records, for managing the virtual address space.

**VMA bookkeeping.**

**Address Space Layout Randomization (ASLR).**

### 4.1.2 File systems

Issue 1.2.e: describe POSIX file system vs NFS/object stores/other approaches

### 4.1.3 Network connections

### 4.1.4 Threads

POSIX threads, or **pthreads**, are commonly used in Linux and similar OSes, for developing multi-thread applications.

## 4.2 Single-Process Abstractions

`libLinux` implements a subset of the Linux system calls (currently at 145 calls) using only the PAL ABI to interact with the host. We note that Linux exports a very long tail of infrequently used calls. A rough analysis of this tail indicates roughly 100 additional calls that can be implemented with the existing PAL ABI and coordination framework, less than 10 administrative calls that will not make sense to expose to an application, such as loading a kernel module or rebooting the system, and roughly 54 that will require PAL extensions to meaningfully implement, such as controlling scheduling, NUMA placement, I/O privilege, and shared memory. In the last category of system calls, the degree to which actual host details should be exported versus emulated is debatable.

Each time we have tested Graphene with a new application, the number of extra system calls required has dropped—most recently we only added 4 calls (namely, epoll_create, epoll_wait, semget and semop) to support the Apache web server. Thus, we believe Graphene implements a representative sample of Linux calls.

### 4.2.1 Bootstrapping

In order to use `libLinux.so`, we modified 606 lines of `glibc` to replace system instructions with function calls into `libLinux.so`, and to cooperatively manage thread-local storage with `libLinux.so` (Table 5.1).

### 4.2.2 Single-process `execve()` and `vfork()`

### 4.2.3 Exceptions

### 4.2.4 Asynchronous events

### 4.2.5 Pseudo files and devices

### 4.2.6 Miscellaneous

## 4.3 Multi-Process Abstractions

### 4.3.1 Cloning a process

Copy-on-write fork presented a particular challenge. As with a virtual machine, each new picoprocess is created in a "clean" state; fork is implemented in the library OS.

Issue 1.3.b: describe the workflow of forking

Graphene implements file Unix-style `fork` by leveraging portions of the checkpoint and migration code, which can programmatically save and restore OS state (e.g., file handles, and memory mappings). Rather than writing the checkpoint to a file, we developed an efficient bulk IPC mechanism to permit copy-on-write sharing of memory pages among processes. Bulk IPC is a performance optimization over sending each byte of the parent address space over a stream, although `libLinux` can also implement `fork` over a stream. Bulk IPC adds 3 calls to the host ABI, and the host reference monitor only permits bulk IPC among picoprocesses within a sandbox.

Using our bulk IPC mechanism, the sender (parent) can request that the host kernel copy a series of pages, which need not be virtually contiguous, into the receiver's address space. The receiver (child) specifies where these pages should be mapped. In both sender and receiver, the pages are marked copy-on-write. This bulk IPC mechanism sends pages out-of-band on a byte stream and guests also use the stream to send control messages indicating how many pages are being sent and how they should be interpreted.

Our IPC module is 1,131 lines of code (Table 5.1), runs on multiple versions of Linux (2.6 and 3 series kernels), and does not require Linux kernel changes or recompila-

tion.

**Inheriting File Handles.** `libLinux` uses two PAL calls, `RPCSendHandle()` and `RPCRecvHandle()`, to transfer stream handles out-of-band over reviously established byte streams within a sandbox. Handle passing facilitates inheritance and general-purpose RPC. This mechanism is similar to Unix Domain Sockets, which are commonly used by sandboxing systems. This strategy allows a guest to seamlessly and explicitly share an open handle with another guest in the same sandbox, but prevents a guest from sharing a handle with a guest outside of the sandbox.

Issue 1.3.b: discuss alternative strategies of forking

### 4.3.2 Multi-process `execve()`

### 4.3.3 Signaling

### 4.3.4 System V IPC

## 4.4 Coordinating Guest OS States

An application executes on Graphene with the abstraction that all of its processes are running on a single OS. Graphene library OSes service system calls from local library OS state whenever possible, and state is coordinated across picoprocesses via RPC when necessary. Within a sandbox, Graphene picoprocesses coordinate shared state used to implement multi-process abstractions, such as process identifiers, thread groups, and System V IPC including message queues and semaphores, shared file system and file descriptor states (Table **??**). Similar to previous designs [44, 134], Graphene uses the host file system; the library OS implements file handles and translates between POSIX and the host ABI. Identifying the best division of labor for a library OS file system is left for future work.

The rest of this section describes our coordination framework, beginning with the coordination building blocks, and then explains the implementation of several multi-process abstractions. We conclude with lessons learned from optimizing multi-process perfor-

mance.

Although a straightforward implementation worked, tuning the performance was the most challenging aspect of this design; we conclude with a summary of the lessons learned from optimizing the system. We conclude with lessons learned from tuning the performance of the system. then presenting the design and driving insights, followed by representative examples, and concluding with a discussion of failure recovery.

### 4.4.1 Building blocks

The general problem underlying each of these coordination APIs is **namespace management**. In other words, coordinating picoprocesses need a consistent mapping of names, such as a thread ID or System V message queue ID, to the picoprocess implementing that particular item. Because many multi-process abstractions in Linux can also be used by single-process applications, a key design goal is to seamlessly transition between single-process uses, serviced entirely from local library OS state, and multi-process cases, which leverage remote procedure calls (RPCs) to coordinate accesses to shared abstractions.

Graphene creates an **IPC helper** thread within each picoprocess, which exchanges coordination messages with the IPC helper threads of picoprocesses within the sandbox. The IPC helper services RPCs from other picoprocesses and is hidden from the application. GNU Hurd has a similar helper thread to implement signaling among a process's parent and immediate children [76]; Graphene generalizes this idea to share a broader range of abstractions among any picoprocesses within a sandbox. To avoid deadlock among application threads and the IPC helper thread, an application thread may not both hold locks required by the helper thread to service an RPC request and block on an RPC response from another picoprocess. All RPC requests are handled from local state and do not issue recursive RPCs.

Within a sandbox, all IPC helper threads exchange messages using a combination of a **broadcast stream** for global coordination, and **point-to-point** streams for pairwise interactions, minimizing overhead for unrelated operations. The broadcast stream is cre-

ated for the picoprocess as part of initialization. Unlike other byte-granularity streams, the broadcast stream sends data at the granularity of messages, to simplify the handling of concurrent writes to the stream. Point-to-point streams are simply byte streams between two picoprocesses; two processes may establish a point-to-point stream by passing handles through an intermediate stream or over the broadcast stream. The handle-passing ABI is discussed further in Section **??**. If a picoprocess leaves a sandbox to create a new one, its broadcast stream is replaced with a new one, connected only to the picoprocess and any children created in the new sandbox.

Because message exchange over the broadcast stream does not scale well, we reduce the use of the broadcast stream to the minimum. Broadcast stream is merely used for **picoprocess identifier allocation** and **leader recovery**.

One picoprocess in each sandbox serves as the **leader**. The leader is responsible for subdividing each namespace among other picoprocesses in the sandbox. For example, the leader might allocate 50 process IDs to a picoprocess that wishes to create children. The **owner** of the allocation can then allocate process IDs to children from its local allocation without further involving the leader. For a given identifier, the owner is the serialization point for all updates, ensuring serializability and consistency for that resource.

### 4.4.2 Examples and discussion

**Signals.** Inside a `libLinux` instance, signals are implemented using a combination of `sigaction` data structures to track signal masks and pending signals; PAL-provided hardware exception upcalls (e.g., for `SIGSEGV`); and RPCs for cross-picoprocess signals (e.g., for `SIGUSR1`). If a process signals itself, `libLinux` simply uses internal data structures to call the appropriate signal handler directly. Graphene implements all three of Linux's signaling namespaces: process, process group, and thread IDs.

Figure 4.1 illustrates two sandboxes with picoprocesses collaborating to implement a process ID (PID) namespace. Because PIDs and signals are a library OS abstraction, picoprocesses in each sandbox can have overlapping PIDs, and cannot signal each other.

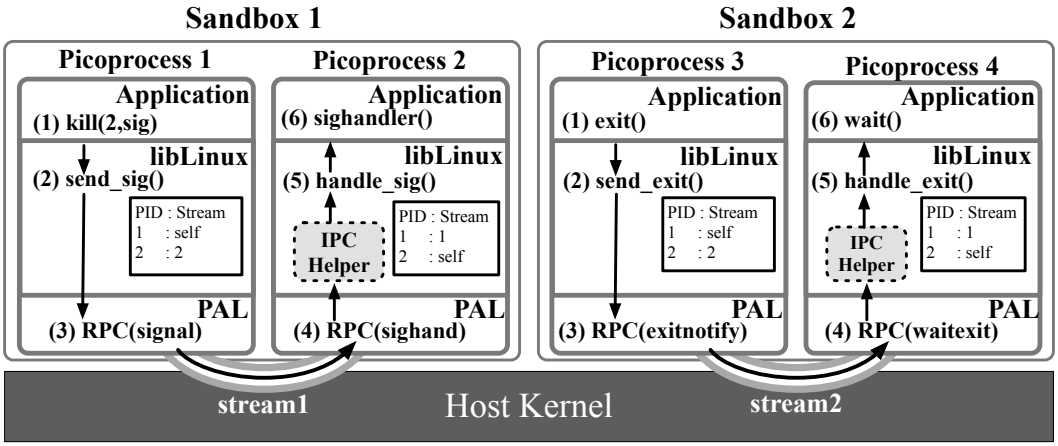



Figure 4.1: Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management. The location of each PID is tracked in `libLinux`; Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation of the `sigaction` interface. Picoprocess 4 waits on an `exitnotify` RPC from picoprocess 3 over stream 2.

Picoprocesses in different sandboxes cannot exchange RPC messages or otherwise communicate.

If picoprocess 1 (PID 1) sends a `SIGUSR1` to picoprocess 2 (PID 2), illustrated in Figure 4.1, the `kill` call to `libLinux` will check its cached mapping of PIDs to point-to-point streams. If `libLinux` cannot find a mapping, it may begin by sending a query to the leader to find the owner of PID 2, and then establish a coordination stream to picoprocess 2. Once this stream is established, picoprocess 1 can send a signal RPC to picoprocess 2 (PID 2). When picoprocess 2 receives this RPC, `libLinux` will then query its local `sigaction` structure and mark `SIGUSR1` as pending. The next time picoprocess 2 makes a `libLinux` call, the `SIGUSR1` handler will be called upon return. Also in Figure 4.1, picoprocess 4 (PID 2) waits on picoprocess 3 termination (in the same sandbox with PID 1). When picoprocess 3 terminates, it invokes the library implementation of exit, which issues an `exitnotify` RPC to picoprocess 4.

The Graphene `libLinux` signal semantics closely match Linux behavior, which delivers signals upon return from a system call or an interrupt or trap handler (PAL upcall). The `libc` signal handling code is unmodified on Graphene. If an application has a signal

pending for too long, e.g., the application is in a CPU-intensive loop, `libLinux` can use a PAL function to interrupt the thread.

**System V IPC.** System V IPC maps an application-specified key onto a unique identifier. All System V IPC abstractions, including message queues and semaphores, are then referenced by this identifier (ID). Similar to PIDs, the leader divides the ID space among the picoprocesses, so that any picoprocess can allocate an ID from local state. The leader also dynamically allocates keys to picoprocesses.

**Message Queues.** In Graphene, the owner of a queue ID is responsible for storing the messages written to the queue; all message sends and receives must go through the owning picoprocess. In our initial implementation, any sends to or receives from a remote queue were several orders of magnitude slower than an access to a local queue. This led to two essential optimizations. First, sending to a remote message queue was made asynchronous. In the common case, the sender can simply assume the send succeeded, as the existence and location of the queue have already been determined. The only risk of failure arises when another process deletes the queue. When a queue is deleted, the owner sends a deletion notification to all other picoprocesses that previously accessed the queue. If a pending message was sent concurrently with the deletion notification (i.e., there is an application-level race condition), the message is treated as if it were sent after the deletion and thus dropped. The second optimization migrates queue ownership from the producer to the consumer, which must read queue contents synchronously.

Because non-concurrent processes can share a message queue, our implementation also uses a common file naming scheme to serialize message queues to disk. If a picoprocess which owns a message queue exits, any pending messages are serialized to a file, and the receiving process may request ownership of the queue from the leader.

**Semaphores.** IPC semaphores follow a similar pattern to message queues, where ownership of a given semaphore is migrated to the picoprocess that most frequently acquires the semaphore. Most of the overhead in the Apache benchmark (§**??**) is attributable to

semaphore overheads.

**Shared File Descriptors.** Open handle descriptors in the Graphene host ABI do not include a seek pointer; Unix-style seek behavior is implemented in the library OS. The default Linux behavior is that children copy the open handles and file seek cursors, but subsequent cursor movements are not shared between parent and child. Shared file descriptor table can be requested by passing the `CLONE_FILES` flag to the `clone` system call. Any new file descriptor opened in a shared table will be visible by every process cloned in this way, as well as subsequent cursor update. If multiple picoprocesses are sharing file descriptor table, the oldest one will coordinate the mapping of each file descriptor to the child picoprocess who owns the seek pointer. Every update to the seek pointer will require coordination if the picoprocess isn't owning it, and similar optimization using migration can be applied here.

**File System States.** In some cases file system states need to be shared across picoprocesses, but the host ABI cannot export the result of Linux-specific behaviors. For example, Linux allows user to perform specialized operations on file system such as opening a FIFO, binding a domain socket, creating a symbolic link, or atomically locking a file. Coordinating these states can be cause significant slowdown on regular file system operations, so we simply export the state in regular files on the host, and atomically update them by renaming.

**Shared Memory.** The Graphene host ABI does not currently permit shared memory among picoprocesses. We expect that a host ABI and existing support for coordinating System V IDs would be sufficient to implement this, with the caveat that the host must be able to handle sandbox disconnection gracefully, perhaps converting the pages to copy-on-write. Thus far we have avoided the use of shared memory in the `libLinux` implementation, both to maximize flexibility in placement of picoprocesses, potentially on different physical machines, and as a rough mechanism to keep all coordination requests explicit.

**Failure and Disconnection Tolerance.** Graphene is designed to tolerate disconnection of collaborating library OS instances, either because of crashes or blocked RPCs. In general, Graphene makes these disconnections isomorphic to a reasonable application behavior, al-

though there may be some edge cases that cannot be made completely transparent to the application.

In the absence of crashes, placing shared state in a given picoprocess introduces the risk that an errant application will corrupt shared library OS state. The microkernel approach of moving all shared state into a separate server process is more resilient to this problem. Anecdotally, Graphene's performance optimization of migrating ownership to the process that most heavily uses a given shared abstraction also improves the likelihood that only the corrupted process will be affected. Making Graphene resilient to arbitrary memory corruption of any picoprocess is left for future work.

**Leader Recovery.** Graphene provides a leadership recovery mechanism when a leader failure is detected. A non-leader picoprocess can detect the failure of a leader by either observing the shutdown of RPC streams or timing out on waiting for responses. Once the picoprocess detects leader failure, it sends out a message on the broadcast stream to volunteer for leadership. After a few rounds of competition, the winning picoprocess becomes the new leader and recover the namespace state by recollecting from every other picoprocess in the sandbox.

### 4.4.3 Lessons learned

The current coordination design is the product of several iterations, which began with a fairly simple RPC-based implementation. This subsection summarizes the design principles that have emerged from this process.

**Service requests from local state whenever possible.** Sending RPC messages over Linux pipes is expensive; this is unsurprising, given the long history of work on reducing IPC overhead in microkernels [59, 111]. We expect that Graphene performance could be improved on a microkernel with a more optimized IPC substrate, such as L4 [71, 99, 112]; we take a complementary approach of avoiding IPC if possible.

An example of this principle is migrating message queues to the "consumer" when a clear producer/consumer pattern is detected, or migrating semaphores to the most frequent

requester. In these situations, synchronous RPC requests can be replaced with local function calls, improving performance substantially. For instance, migrating ownership of message queues reduced overhead for message receive by a factor of $10\times$.

**Lazy discovery and caching improve performance.** No library OS keeps a complete replica of all distributed state, avoiding substantial overheads to pass messages replicating irrelevant state. Instead, Graphene incurs the overhead of discovering the owner of a name on the first use, and amortizes this cost over subsequent uses. Part of this overhead is potentially establishing a point-to-point stream, which can then be cached for subsequent use. For instance, the first time a process sends a signal, the helper thread must figure out whether the process id exists, to which picoprocess it maps, and establish a point-to-point stream to the picoprocess. If they exchange a second signal, the mapping is cached and reused, amortizing this setup cost. For instance, the first signal a process sends to a new processes takes ˜2ms, but subsequent signals take only ˜55 $\mu$S.

**Batched allocation of names minimizes leader workload.** In order to keep the leader off of the critical path of operations like `fork`, the leader typically allocates larger blocks of names, such as process IDs or System V queue IDs. In the case of `fork`, if a picoprocess creates a child, it will request a batch of PIDs from the leader (50 by default). Subsequent child PID allocations will be made from the same batch without consulting the leader. Collaborating processes also cache the owner of a range of PIDs, avoiding leader queries for adjacent queries.

**The coordination within a sandbox is often pairwise.** Graphene optimizes the common case of pairwise coordination, by authorizing one side of the coordination to dictate the abstraction state, but also allows more than two processes to share an abstraction. Based on this insight, we observe that *not all shared state need be replicated by all picoprocesses*. Instead, we adopt a design where one picoprocess is authoritative for a given name (e.g., a process ID or a System V queue ID). For instance, all possible thread IDs are divided among the collaborating picoprocesses, and the authoritative picoprocess either responds to

RPC requests for this thread ID (e.g., a signal) or indicates that the thread does not exist. This trade does make commands like `ps` slower, but optimizes more common patterns, such as waiting for a child to exit.

**Make RPCs asynchronous whenever possible.** For operations that must write to state in another picoprocess, the Graphene design strives to cache enough information in the sender to evaluate whether the operation will succeed, thereby obviating the need to block on the response. This principle is applied to lower the overheads of sending messages to a remote queue.

**Summary.** The current Graphene design minimizes the use of RPC, avoiding heavy communication overheads in the common case. This design also allows for substantial flexibility to dynamically moving processes out of a sandbox. Finally, applications do not need to select different library OSes *a priori* based on whether they are multi-process or single-process—Graphene automatically uses optimal single-process code until otherwise required.

## 4.5 Limitations

## 4.6 Summary

| Component | Lines | (% Changed) |
|---|---|---|
| GNU Library C (`libc`, `ld`, `libdl`, `libpthread`) | 606 | 0.07% |
| Linux Library OS (`libLinux`) | 31,112 | |
| Linux host PAL | 11,644 | |
| Extra code for Linux SGX host PAL | 9.354 | |
| Reference monitor bootstrapper | 3,568 | |
| Linux kernel reference monitor module (`/dev/graphene`) | 888 | |
| Linux kernel IPC module (`/dev/gipc`) | 1,131 | |

Table 5.1: Lines of code written or changed to produce Graphene. Applications and other libraries are unchanged.

# Chapter 5

# The Linux Host

This chapter discribes the implementation of the host ABI on a Linux host.

## 5.1 Implementing the Host ABI

The majority of PAL calls are simple wrappers for similar Linux system calls, adding less than 100 LoC on average for translation between PAL and Linux abstractions. The largest PAL calls are for exception handling, synchronization, and picoprocess creation, which require multiple system calls and range from 500–800 LoC each. Creating a new picoprocesses internally requires a `vfork` and `exec` of a clean application instance, and would be more efficiently implemented in the kernel. Finally, the other major PAL components are an ELF loader (2 kLoC), headers (800 LoC), and internal support code (2.3 kLoC).

Issue 1.1.a: extend the technical sections

**Synchronization.** Perhaps surprisingly, implementing Linux synchronization appears substantially easier than Windows synchronization, as `libLinux` did not require all of the various synchronization ABIs provided by Drawbridge. We believe the reason for this is that Linux has consolidated all user-level synchronization primitives to use futexes [75], which are essentially kernel-level wait queues.

## 5.2 Security Isolation

Graphene ensures that mutually untrusting applications cannot interfere with each other, providing security isolation comparable to running in separate VMs. Graphene reduces the attack surface exposed to applications by restricting access to the host kernel ABI and prevents access to unauthorized system calls, files, byte streams, and network addresses with a *reference monitor*. Graphene contributes a multi-process security model based on the abstraction of a *sandbox*, or a set of mutually trusting picoprocesses. The reference monitor permits picoprocesses within the same sandbox to communicate and exchange RPC messages, but disallows cross-sandbox communication. The current work focuses on all-or-nothing security isolation, although we expect this design could support controlled communication among mutually untrusting library OSes in future work.

Issue 1.1.d: describe the security isolation story for Linux hosts (need polishing)

The only kernel-level sharing abstractions the reference monitor must mediate are files, network sockets, and RPC streams — all other allowed kernel ABIs modify only local picoprocess state. In order for the reference monitor to restrict file access, socket and RPC stream creation, each application includes a *manifest file* [84], which describes a `chroot`-like, restricted view of the local file system (similar to Plan 9's unioned file system views [131]), as well as *iptables*-style [88] network restrictions.

When a new picoprocess is launched by the reference monitor, it begins execution in a new sandbox. Child picoprocesses may either inherit their parent's sandbox, or can be started in a separate sandbox — specified by a flag to the picoprocess creation ABI. A parent may specify a subset of its own file system view when creating a child, but may not request access to new regions of the host file system. The child may also issue an `ioctl`

Explain with a figure.

call to dynamically detach from the parent's sandbox. The reference monitor prevents byte stream creation across sandboxes. When a process detaches from a sandbox — effectively splitting the sandbox — the reference monitor closes any byte streams that could bridge the two sandboxes.

**Threat Model.** We assume a trustworthy host OS and reference monitor, which mediates all system calls with effects outside of a picoprocess's address space, such as file `open` or network socket `bind` or `connect`. We assume that picoprocesses inside the same sandbox trust each other and that all untrusted code runs in sandboxed picoprocesses. We assume the adversary can run arbitrary code inside of one or more picoprocesses within one or more sandboxes. The adversary can control all code in its picoprocesses, including `libLinux` and the PAL.

Graphene ensures that the adversary cannot interfere with any victim picoprocesses in a separate sandbox. The Graphene sandbox design ensures strict isolation: if the only shared kernel abstractions are byte streams and files, and the reference monitor ensures there is no writable intersection between sandboxes, the adversary cannot interfere with any victim picoprocess.

Graphene reduces the system attack surface of the host, but does not change the size of its trusted computing base; however, reducing the effective system call table size of a picoprocess does facilitate adoption of a smaller host kernel, which we leave for future work.

### 5.2.1 System call restriction

Unmodified Linux applications run on Graphene by issuing system calls as library calls to `libLinux`. Application calls are serviced by `libLinux`-internal data structures or PAL calls. The PAL is implemented using 50 host system calls. The host OS must block any native system call that does not appear in the PAL source code. Any allowed system call with external effects is checked by the reference monitor.

Graphene restricts the host system call table using seccomp [143], introduced in

Issue 1.3.d: extend the discussion of SECCOMP filter

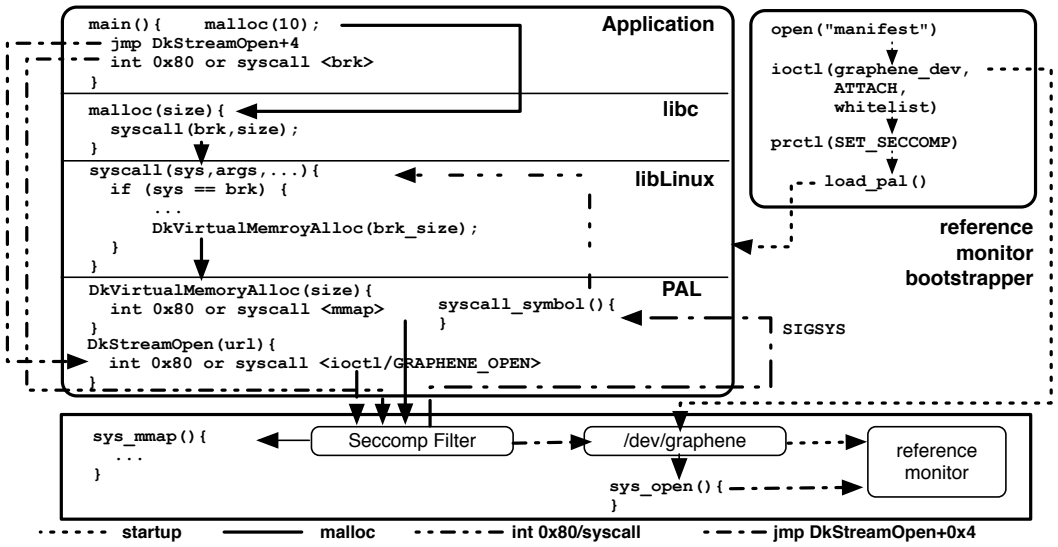



Figure 5.1: System call restriction approach. The reference monitor loads policies into the LSM at startup. A Graphene application requests OS services in three different ways. In the normal case (first line of `main`), `malloc` is invoked causing the invocation of `brk` (`libLinux`) and `mmap` in the PAL. In the second line, the application jumps to an address in PAL, which is permissible. Files are accessed through `ioctl` to `/dev/graphene` and checked by reference monitor. The third line invokes `brk` with an `int` instruction, which is redirected to the `libLinux` function.

Linux 2.6.12. Seccomp allows a process to create an immutable Berkeley Packet Filter (BPF) program that specifies allowed system calls, as well as creates `ptrace` events on certain system calls. The filter can also filter scalar argument values, such as only permitting specific `ioctl` opcodes. If a system call is rejected, the PAL will receive a `SIGSYS` signal, and can either terminate the application or redirect the call to `libLinux`. Seccomp filters cannot be overridden by any picoprocess, and are always inherited. The current Graphene filter is 79 lines of straightforward BPF macros. In our experience, adding more precise argument checks has not significantly changed performance.

Unfortunately, the logic to check for allowed paths network addresses cannot be implemented as a seccomp rule, because it involves reading user memory of unknown sizes. In order to avoid the overhead of trapping to the reference monitor on every use of `open`, `stat`, `bind` or `connect` system calls, we instead force picoprocess to only use `ioctl` system call to Graphene special device (`/dev/graphene`) as alternative interface these

system calls. Direct access to these system calls are banned by seccomp filter.

In order to reduce the impact of bugs in the reference monitor, the reference monitor itself runs with a seccomp filter, blocking unexpected system calls.

**Static Binaries.** For compatibility with statically linked binaries, which compile in system call instructions, we leverage seccomp to redirect these calls back to `libLinux`. For system calls that could also be issued by the PAL, we augment our BPF rules with program counter-based filters. In other words, an `open` system call with a return PC address inside the PAL will be sent to the reference monitor for further inspection; an `open` system call with any other return PC address generates a `SIGSYS` and is ultimately relayed back to `libLinux`. Thus, `libLinux` can catch and differentiate application-issued system calls from those that could also be issued by the PAL. We hasten to note this feature is only for backward compatibility, not security.

**Example.** Figure 5.1 illustrates three possible situations. An unmodified application first invokes the `libc` function `malloc`, which issues a `brk` system call to `libLinux`, which requests memory from the host via a `DkVirtualMemoryAlloc` PAL call, which ultimately issues an `mmap` host system call. The `mmap` host system call is allowed by seccomp because it only affects the picoprocess's address space. The second line of the application jumps to the PAL instruction that issues an `open` system call. From a security perspective, this is permissible, as it is isomorphic to PAL functionality. In practice, this could cause corruption of `libLinux` or application data structures, but the only harm is to the application itself. Because this system call involves the file system, the reference monitor LSM first checks if the file to be opened is included in the sandbox definition (manifest) before allowing the `open` system call in the kernel. Finally, the application uses inline assembly to issue a `brk` system call; because this system call was not issued by the PAL, seccomp will redirect this call back to the PAL, which then calls the `libLinux` implementation.

**Process-specific Isolation.** Sandbox creation in Graphene can provide more options than virtualization, to reflect the security policy of applications at any timing, in the granularity

of picoprocess. A picoprocess can voluntarily detach itself from the current sandbox, dropping its privileges, after finishing security-sensitive operations. If a picoprocess decides one of its children is not trustworthy, it may also start the child under a restricted manifest, or promptly shut down RPC streams to stop sharing OS states. The picoprocess that moves to a separate sandbox will have a restrictive view of the filesystem, and no coordination with the previous sandboxes. In section 7.1, we describe an experiment that improves security isolation of Apache web server without sacrificing functionality.

### 5.2.2 Reference monitor

The reference monitor is a trusted process that runs on the host system. Graphene applications are launched by the reference monitor, which instantiates the seccomp filter and traces all children to check host system calls that could have external effects. The reference monitor interposes using ptrace events, which can be raised for specific system calls by seccomp. We ensure that all processes created within a sandbox are traced by setting the `PTRACE_O_TRACESECCOMP` option on all newly created picoprocesses in the sandbox.

Each application includes a *manifest file*, which specifies restrictions, including network firewall rules and subsets of the host file system sandboxed applications are permitted to access. The reference monitor enforces these rules by interposing on all system calls involving file paths or remote network addresses.

**Privilege.** Although the reference monitor is trusted, it does not run with administrative privilege. Linux 3.5, which we use as our host kernel, introduced the `NO_NEW_PRIVS` bit, which permits a non-privileged process to impose sandboxing restrictions on a child. This flag prevents a process from acquiring root privilege, is inherited by all descendant processes, and cannot be disabled.

**Creating New Sandboxes.** We add a PAL call which permits a picoprocess to request that it be moved into a new sandbox. This call, as well as file system path checks, are implemented as extensions to the AppArmor LSM [35]. The new sandbox call closes any open stream handles that cross sandbox boundaries; mediate path lookups; and create a new

broadcast stream for multi-process coordination (§**??**).

To securely apply seccomp filtering we leveraged the fact that all Graphene processes have the same parent and also the new `NO_NEW_PRIVS` bit introduced for Linux processes starting kernel version 3.5. This bit can be set by any process, is inherited across `fork`, `clone`, and `execve`, and cannot be unset by children processes. Thus, we set the `NO_NEW_PRIVS` bit in the initial Graphene process and apply seccomp filters allowing only system calls with corresponding functions in the PAL. As a result all Graphene processes will inherit the filters and cannot relax or bypass it.

## 5.3 Summary

# Chapter 6

# The SGX Host

This chapter discribes the PAL for running the Graphene library OS in enclaves, to resolve compatibility issues unique to SGX.

## 6.1 Background

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

### 6.1.1 SGX (software guard extensions)

SGX [120] is a feature added in the Intel sixth-generation CPUs, as a hardware support for trusted execution environment (TEE). SGX introduces a number of essential hardware features that allow an application to protect itself from the host OS, hypervisor, BIOS, and other software. 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.

The primary SGX abstraction is an **enclave**, an isolated execution environment within the virtual address space of a process. Enclave features include confidentiality and integrity protection: 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 6.1: 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 [33].

### 6.1.2 SGX frameworks

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++ [50] and AMD SEV [95]—each with different idiosyncrasies. Thus, the need to adapt applications to use hardware security features will only increase in the near term.

Issue 1.2.c (done): compare library OS approach with shim layers

As a result, there is an increasingly widespread belief that adopting SGX necessarily involves significant code changes to applications. Although Haven [45] 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 $5\times$, the service latency by $4\times$, and halves the service throughput" [38]. Shinde et al. [150] 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 bloat. We present a port of the Graphene library OS [160] 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 [150]. 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 [116] and measured by our prior work [161]; 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 [118, 137, 150], 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.

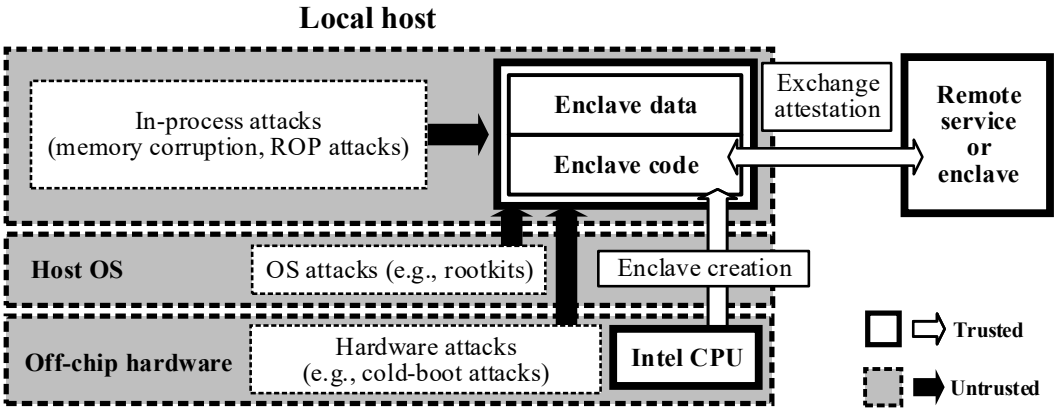



Figure 6.1: The threat model of SGX. SGX protects applications from three types of attacks: in-process attacks from outside of the enclave, attacks from OS or hypervisor, and attacks from off-chip hardware.

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. 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 [97, 126, 150] and we are aware of ongoing projects using Graphene-SGX.

**How much functionality to pull into the enclave?** At one extreme, a library OS like Haven [45] pulls most of the application-supporting code of the OS into the enclave. On the other extreme, thin "shim" layers, like SCONE [38] and PANOPLY [150] 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 supporting 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* [57] 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.

Issue 1.2.f: Clarify the details about Iago attacks

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.

## 6.2 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.

### 6.2.1 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 [171] are vulnerabilities common to all SGX frameworks, and are beyond the scope of this work.

### 6.2.2 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, 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.

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.

### 6.2.3 Multi-process applications

Graphene supports multi-process applications by running a separate library OS instance in each process [160]. 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

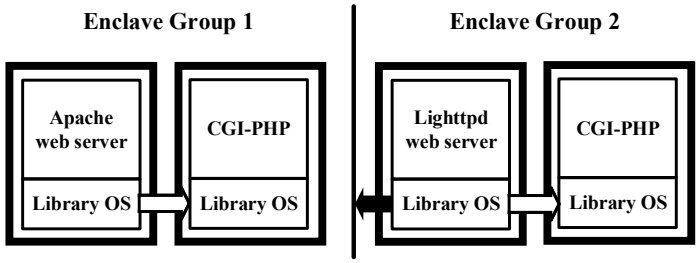



Figure 6.2: Two enclave groups, one running Apache and the other running Lighttpd, each creates a child enclave running CGI-PHP. Graphene-SGX distinguishes the child enclaves in different enclave groups.

group of coordinating enclaves an *enclave group*. Figure 6.2 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 (see Section 6.3.3).

## 6.3 Shielding Linux Abstractions

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

Issue 1.1.d (done): describe the security isolation story for SGX

### 6.3.1 Dynamic loading

To run unmodified Linux binaries, Graphene-SGX implements dynamic loading and runtime linking. In a major Linux distribution like Ubuntu, more than 99% of binaries are dynamically linked [161]. 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 [45], the enclave measurement only verifies the integrity of Haven itself, and the same measurement applies to any application running on the same Haven binary.

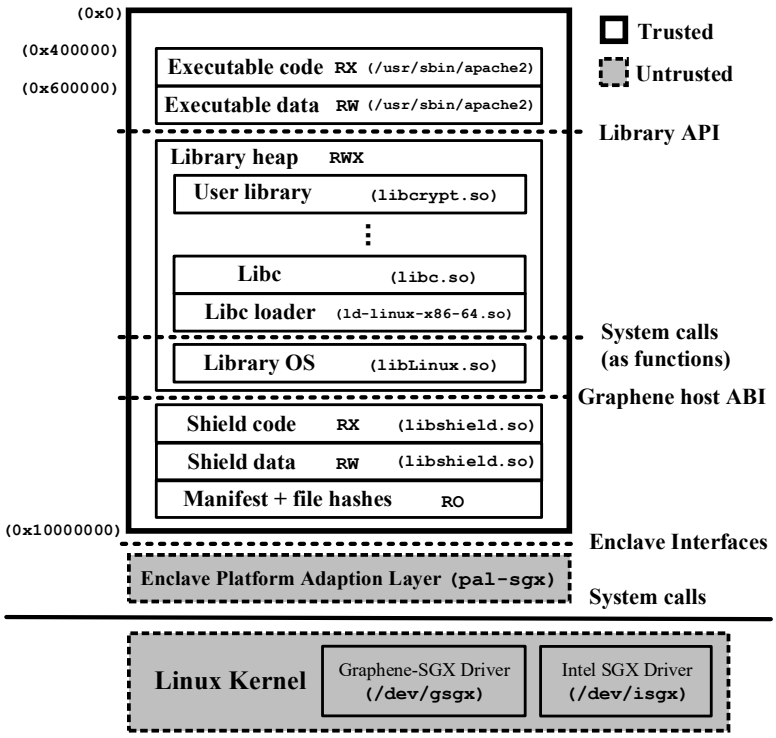



Figure 6.3: The Graphene-SGX architecture. The executable is position-dependent. The enclave includes an OS shield, a library OS, libc, and other user binaries.

Graphene-SGX extends the Haven model to generate a unique signature for any combination of executable and dynamically-linked libraries. Figure 6.3 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.

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 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 [119] will support changing page protections, which Graphene-SGX will adopt in the future.

**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.

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 [23], undermining the integrity of the enclave.

### 6.3.2 Single-process API

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, that would also run on a Linux or FreeBSD pico-process, as well as an SGX-specific platform adaptation layer (PAL), which implements 42 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 6.1).

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 [57] demonstrate the subtlety of detecting semantic attacks via the POSIX interface. Projects such as Sego [103] 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.

| 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 |

Table 6.1: 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.

Table 6.1 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.

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.

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 [82].

**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.

**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 [126] 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 [62].

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.

### 6.3.3 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 6.4. 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 enclave ID. The parent and child exchange these reports

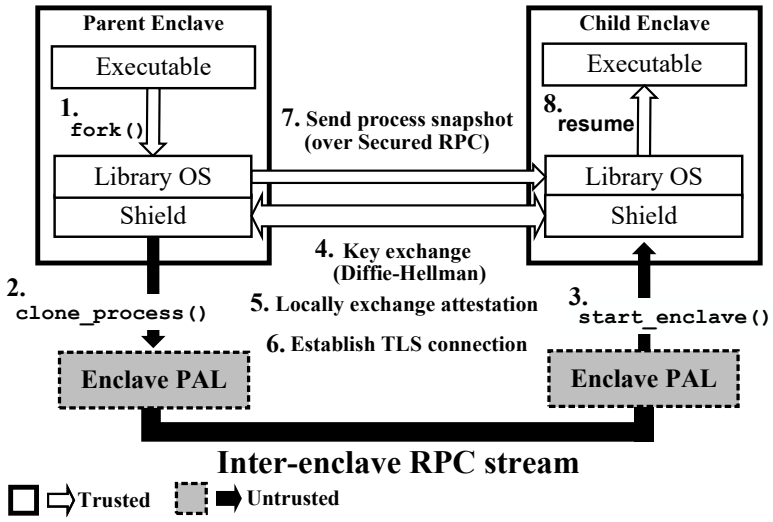



Figure 6.4: Process creation in Graphene-SGX. Numbers show the order of operations. When a process forks, Graphene-SGX creates a new, clean enclave on the untrusted host. Then the two enclaves exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process snapshot.

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.

## 6.4 Summary

# Chapter 7

# Evaluation

## 7.1 Graphene

This section evaluates Graphene's multi-process coordination, security, cross-host migration, memory footprint, and performance. We drive this evaluation with a selection of real-world applications that leverage multiple processes in Graphene, as well as with microbenchmarks and stress tests. We organize the evaluation around the following questions:

1. How do Graphene's startup and migration costs compare to running an application in a dedicated VM?
2. Given that RAM is often the limiting factor in VMs per system, how does Graphene's memory footprint compare to other virtualization techniques?
3. What are the performance overheads of Graphene relative to a native Linux process or VM?
4. What additional overheads are added by the reference monitor?
5. How do Graphene's overheads scale with the number of processes in a sandbox?
6. Does the Graphene reference monitor enforce security isolation comparable to running the application in a VM?
7. What fraction of recent Linux vulnerabilities would Graphene prevent?

Except for scalability measurements, all measurements were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250GB, 7200

| Test | Linux | KVM | | Graphene | |
|---|---|---|---|---|---|
| Start-up | 208 $\mu$S | 3.3 s | 15K× | 641 $\mu$S | 3.1× |
| Checkpoint | N/A | 0.987 s | | 416 $\mu$S | |
| Resume | N/A | 1.146 s | | 1387 $\mu$S | |
| Checkpoint size | N/A | 105 MB | | 376 KB | |

Table 7.1: Startup, checkpoint, and resume times for (1) a native Linux process, (2) a KVM virtual machine, (3) a Graphene picoprocess, (4) a Graphene picoprocess in a SGX enclave, where appropriate. Lower is better. Overheads are relative to Linux.

RPM ATA disk. Our host system runs Ubuntu 12.04 server with host Linux kernel version 3.5, which includes KVM on QEMU version 1.0. Each KVM Guest is deployed with 4 virtual CPU with EPT, 2GB RAM, a 30GB virtual disk image, Virtio enabled for network and disk, bridged connection with TAP, and runs the same Ubuntu and Linux kernel image. We note that recent library OSes are either not openly available or cannot execute unmodified Linux binaries. Unless otherwise noted, Graphene measurements include the reference monitor.

### 7.1.1 Process migration and startup

Graphene supports migration of an application from a picoprocess on one machine to a picoprocess on another machine by checkpointing the application, copying the checkpoint over the network, and then resuming the checkpoint. Table 7.1 shows the time to start up a process, VM, or picoprocess; as well as the checkpoint and resume time for KVM and Graphene. Migration across machines is a function of network bandwidth, so we report checkpoint size instead.

Graphene shows dramatically faster initialization times than a VM. This is not surprising, since Graphene is substantially smaller than an OS kernel. Similarly, checkpointing and restoring a 4 MB application on Graphene is 1–4 times faster than checkpointing or restoring a KVM guest.

### 7.1.2 Memory footprint

We begin by measuring the minimal memory footprint of a simple "hello world" program on Linux (352 KB) and Graphene (1.4 MB). Thus, one would expect roughly 1 MB of extra memory usage for any single-picoprocess application. Because of copy-on-write sharing, however, the incremental cost of adding additional "hello world" children is only about 790 KB per process.

We found that the memory footprints of compilation were a function of the size of the source base, even on Linux; we select compile of `libLinux` as a representative example. Graphene adds less than 15% overhead in all cases.

Unixbench on Graphene uses substantially more memory at a given time than native Linux—more than double. In these samples, however, Graphene also had 3–4× as many processes running; this is because Unixbench simply spawns all of the tasks in the background, rather than executing them sequentially. Because Graphene processes execute more slowly (attributable to a slower `fork`—§**??**), a given sample will include more picoprocesses, pushing total memory usage higher. Thus, we expect that further tuning `fork` performance will lower sampled memory usage.

Across all workloads, Graphene's memory footprint is 3–20× smaller than KVM. For all tests, we used a minimal KVM disk image, generated using debootstrap 1.0.39ubuntu0.3 and supplemented only by packages required to obtain, compile, and run the experiments. In order to make memory footprint measurements as fair as possible to KVM, we used both the virtio balloon driver and kernel same page merging (KSM) [37]. We also reduced the RAM allocated to the VM to the smallest size without harming performance—128MB. We note that memory measured includes memory used by QEMU for VM device emulation, adding a few dozen MB.

If the smallest usable Linux VM consumes about 150 MB of RAM, our measurements indicate that one could run anywhere from 12–188 libOSes within the same footprint.

| | Execution time (s), +/- Conf. Interval, % Overhead | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| **GCC/make** | **Linux** | | **in KVM** | | | **Graphene + SC + RM** | | |
| gcc (helloworld) | 0.020 | .000 | 0.022 | .000 | 7 % | 0.023 | .000 | 12 % |
| gcc (.7MLoC) | 21.64 | .00 | 23.37 | .02 | 8 % | 21.88 | .00 | 1 % |
| make bzip2 | 2.332 | .000 | 2.448 | .000 | 5 % | 2.407 | .000 | 3 % |
| make -j4 bzip2 | 0.888 | .000 | 0.967 | .000 | 9 % | 0.923 | .004 | 4 % |
| make libLinux | 4.832 | .000 | 5.037 | .000 | 4 % | 5.112 | .001 | 6 % |
| make -j4 libLinux | 1.361 | .000 | 1.413 | .000 | 4 % | 1.459 | .000 | 7 % |
| Ap. Bnch | Avg Throughput (MB/s), +/- Conf. Interval, % Overhead | | | | | | | |
| **Apache** | **Linux** | | **in in KVM** | | | **Graphene + SC + RM** | | |
| 25 conc | 6.282 | .005 | 5.327 | .031 | 18 % | 4.586 | .002 | 36 % |
| 50 conc | 6.305 | .002 | 5.420 | .060 | 16 % | 4.555 | .012 | 38 % |
| 100 conc | 6.347 | .010 | 5.152 | .036 | 22 % | 4.572 | .009 | 39 % |
| **Lighttpd** | **Linux** | | **in KVM** | | | **Graphene + SC + RM** | | |
| 25 conc | 6.66 | .01 | 6.46 | .03 | 3 % | 5.65 | .00 | 18 % |
| 50 conc | 6.65 | .13 | 6.41 | .02 | 4 % | 4.79 | .00 | 39 % |
| 100 conc | 6.69 | .01 | 6.39 | .03 | 5 % | 4.56 | .01 | 47 % |
| | Execution Time (s), +/- Conf. Interval, % Overhead | | | | | | | |
| **bash** | **Linux** | | **in KVM** | | | **Graphene + RM** | | |
| Unix utils | 0.87 | .00 | 1.10 | .01 | 26 % | 2.01 | .00 | 134 % |
| Unixbench | 0.55 | .00 | 0.55 | .00 | 0 % | 1.49 | .00 | 192 % |

Table 7.2: Application benchmark execution times in a (1) native Linux process, (2) a process inside a KVM virtual machine, (3) a Graphene picoprocess with the SECCOMP filter (**+SC**) and reference monitor (**+RM**).

### 7.1.3 Application performance

Table 7.2 lists execution time of our *unmodified* application benchmarks (detailed in §**??**). All applications create multiple processes, except for Lighttpd, which only creates multiple threads. Each data point is the average of at least six runs, and 95% confidence intervals are listed in the table.

We exercise `GCC/make` with inputs of varying sizes: bzip2 (v1.0.6, 5KLoC, 13 files), Graphene's `libLinux` (31 KLoC, 78 files) and GCC (v3.5.0, 551 KLoC, collected as a single source file). We benchmark Apache (4 preforked workers) and Lighttpd (4 threads) with ApacheBench, which issues 25, 50, and 100 concurrent requests to download a 100 byte file 50,000 times.

We exercised Bash with 300 iterations of the Unixbench benchmark [3], as well as 300 iterations of a simple shell script benchmark that runs 6 common shell script commands (`cp`, `rm`, `ls`, `cat`, `date`, and `echo`).

Compilation workloads incur overheads ranging from 5–30%. Parallel compilation on both Graphene and Linux yields comparable speedups over sequential, but the percent overhead increases for parallel Graphene. For instance, `make -j4 libLinux` speeds up 3.7× on Linux and 3.4× on Graphene. The compilation overheads are primarily from the reference monitor—nearly all for bzip2 and gcc, and half for `libLinux`. In the case of both Bash workloads, the key bottleneck is the `fork` system call. Profiling indicates that half of the time in `libLinux` is spent on `fork` in both benchmarks. The trend is exacerbated in Unixbench, which creates all of the processes at the beginning and waits for them all to complete; because Graphene cannot create children as quickly as native, this leads to load imbalance throughout the rest of the benchmark.

With the reference monitor disabled, Lighttpd has equivalent throughput to a native Linux process; as discussed in the next subsection, these overheads come from checking paths in the monitor. The Apache web server loses about half of its throughput relative to Lighttpd on Graphene. The primary bottleneck in Apache relative to Lighttpd is System V semaphores, and the remaining overheads are attributable to more time spent waiting for input. The overheads for both Lighttpd and Apache on KVM are primarily attributable to bridged networking.

### 7.1.4 Micro-benchmarks

In order to understand the overheads of individual system calls, Table 7.3 lists a representative sample of tests from the LMbench suite, version 2.5 [122]. Each row reports a mean and 95% confidence interval; we use the default number of iterations for each test case. To measure the marginal cost of the reference monitor, we report numbers with and without the reference monitor.

In general, calls that can be serviced inside the library are faster than native, whereas calls that require translation to a native call incur overheads typically under 100%. For instance, the self-signaling test (sig overhead) just calls the signal handler as a function, which is almost twice as fast as the Linux kernel implementation.

| Test | Linux | | Graphene | | | Graphene + SC + RM | | |
|---|---|---|---|---|---|---|---|---|
| | $\mu$S | +/- | $\mu$S | +/- | %O | $\mu$S | +/- | %O |
| syscall | 0.045 | .000 | 0.014 | .000 | -68.9 | 0.014 | .000 | -68.9 |
| read | 0.115 | .000 | 0.118 | .000 | 2.6 | 0.118 | .000 | 2.6 |
| write | 0.115 | .000 | 0.115 | .000 | 0.0 | 0.115 | .000 | 0.0 |
| stat | 0.332 | .000 | 1.154 | .000 | 247.6 | 1.161 | .000 | 249.7 |
| fstat | 0.107 | .000 | 0.189 | .000 | 76.6 | 0.189 | .003 | 76.6 |
| open/close | 0.880 | .003 | 2.552 | .005 | 190.0 | 2.816 | .002 | 330.0 |
| select file | 3.360 | .001 | 11.536 | .003 | 243.3 | 11.647 | .004 | 246.6 |
| select tcp | 10.590 | .002 | 11.679 | .002 | 10.3 | 12.050 | .000 | 13.8 |
| sig install | 0.126 | .000 | 0.126 | .000 | 0.0 | 0.126 | .000 | 0.0 |
| sigusr1 | 0.951 | .000 | 0.184 | .000 | -80.7 | 0.188 | .000 | -80.2 |
| sigsegv | 0.263 | .000 | 1.467 | .000 | 457.8 | 1.681 | .000 | 539.2 |
| TCP | 8.354 | .001 | 9.121 | .002 | 1.8 | 9.699 | .002 | 16.1 |
| UDP | 7.222 | .002 | 9.450 | .002 | 30.8 | 9.995 | .001 | 38.4 |
| AF_UNIX | 4.937 | .011 | 5.024 | .001 | 1.8 | 6.155 | .010 | 24.7 |
| fork+exit | 66.86 | 0.09 | 380.47 | 3.55 | 469 | 442.83 | 1.39 | 562 |
| fork+fork+exit | 148.32 | 0.55 | 778.00 | 6.87 | 424 | 915.67 | 3.35 | 517 |
| vfork+exec | 141.53 | 0.18 | 487.67 | 5.05 | 245 | 547.30 | 5.22 | 286 |
| fork+exec | 194.93 | 0.20 | 810.00 | 3.27 | 315 | 878.50 | 1.89 | 350 |
| fork+fork+exec | 266.89 | 0.36 | 1,200.60 | 5.20 | 349 | 1,387.75 | 6.21 | 420 |
| fork+sh | 499.64 | 0.67 | 1,726.75 | 6.14 | 245 | 1,912.00 | 3.83 | 283 |

Table 7.3: LMbench comparison among (1) native Linux processes, (2) Graphene picoprocesses on Linux host, both without and with the SECCOMP filter (**+SC**) and reference monitor (**+RM**), and (3) Graphene in SGX enclaves. Execution time is in microseconds, and lower is better. Overheads are relative to Linux; negative overheads indicate improved performance.

The most expensive system calls occur when `libLinux` inadvertently duplicates work with the host kernel. For instance, many of the file path and handle management calls duplicate some of the effort of the host file system, leading to a 1–3× slower implementation than native. As the worst example, `fork+exit` is 5.9× slower than Linux. Profiling indicates that about one sixth of this overhead is in process creation, which takes additional work to create a clean picoprocess on Linux; we expect this overhead could be reduced with a kernel-level implementation of the process creation ABI, rather than emulating this behavior on `clone`. Another half of the overhead comes from the `libLinux` checkpointing code (commensurate with the data in Table 7.3), which includes a substantial amount of serialization effort which might be reduced by checkpointing the data structures in place. A more competitive `fork` will require host support and additional `libLinux` tuning.

We also measure the overhead of isolating a Graphene picoprocess inside the ref-

| Test | | Linux | | Graphene | | |
|---|---|---|---|---|---|---|
| | | $\mu$S | +/- | $\mu$S | +/- | |
| msgget (create) | in-process | 3320 | 0.7 | 2823 | 0.3 | -15 % |
| | inter-process | 3336 | 0.5 | 2879 | 3.6 | -14 % |
| | persistent | N/A | | 10015 | 0.7 | 202 % |
| msgget | in-process | 3245 | 0.5 | 137 | 0.0 | -96 % |
| | inter-process | 3272 | 3.4 | 8362 | 2.4 | 156 % |
| | persistent | N/A | | 9386 | 0.4 | 189 % |
| msgsnd | in-process | 149 | 0.2 | 443 | 0.2 | 191 % |
| | inter-process | 153 | 0.3 | 761 | 1.1 | 397 % |
| | persistent | N/A | | 471 | 0.8 | 216 % |
| msgrcv | in-process | 149 | 0.1 | 237 | 0.2 | 60 % |
| | inter-process | 153 | 0.1 | 779 | 2.2 | 409 % |
| | persistent | N/A | | 979 | 0.6 | 561 % |

Table 7.4: Micro-benchmark comparison for System V message queues between a native Linux process and Graphene picoprocesses. Execution time is in microseconds, and lower is better. overheads are relative to Linux, and negative overheads indicate improved performance.

erence monitor. Because most filtering rules can be statically loaded into the kernel, the cost of filtering is negligible with few exceptions. Only calls that involve path traversals, such as `open` and `exec`, result in substantial overheads relative to Graphene. An efficient implementation of an environment similar to FreeBSD jails [158] would make all reference monitoring overheads negligible.

**System V IPC.** Table 7.4 lists the micro-benchmarks which exercise each System V message queue function, within one picoprocess (in process), across two concurrent picoprocesses (inter process), and across two non-concurrent picoprocesses (persistent). Linux comparisons for persistent are missing, since message queues survive processes in kernel memory.

In-process queue creation and lookup are faster than Linux. In-process send and receive overheads are higher because of locking on the internal data structures; the current implementation acquires and releases four fine-grained locks, two of which could be elided by using RCU to eliminate locking for the readers [121]. Most of the costs of persisting message queue contents are also attributable to locking.

Although inter-process send and receive still induce substantial overhead, the optimizations discussed in §4.4.3 reduced overheads compared to a naive implementation by

a factor of ten. The optimizations of asynchronous sending and migrating ownership of queues when a producer/consumer pattern were detected were particularly helpful.

### 7.1.5 Security study

Issue 1.1.d: Add a kernel coverage study

Demonstrating security is always challenging, as it requires exploring all possible attacks. We instead offer statistics that indicate an overall reduction in attack surface, and qualitative validation where appropriate. Graphene runs substantial Linux applications using less than 15% of the Linux system call table — reducing this attack surface. We wrote several tests that attempt to issue illegal system calls with inline assembly; we validate that all system calls are redirected to `libLinux`, and that signals and other IPC cannot cross sandbox boundaries.

**Isolation experiments.** This subsection tests whether Graphene meets its goal of providing equivalent isolation to a VM. We conducted an evaluation of the security of the isolation mechanisms in Graphene and analyzed whether a malicious Graphene picoprocess could (i) fork a non-Graphene process, (ii) kill processes from another sandbox or a non Graphene process, (iii) access files not prescribed in its Manifest, and (iv) discover secrets from picoprocesses in other sandboxes or from non-Graphene process through side-channels via the `/proc` file system. The first three attacks use inline assembly to directly issue a system call, and are blocked by the reference monitor; the fourth creates an attack similar to Memento [91] and is frustrated by the fact that `/proc` is implemented within `libLinux` and the system `/proc` is inaccessible from Graphene.

**Analysis of Linux vulnerabilities.** Graphene restricts picoprocesses to 15% of the system call table. To evaluate the impact on system security, we *manually* analyzed all reported Linux vulnerabilities from 2011–2013 (a total of 291 vulnerabilities) [24]. We categorized these exploits by kernel component, listed in Table 7.5. Roughly half of these vulnerabilities required a system call which Graphene blocks in order to exploit the system. Graphene would only allow 5 of the relevant vulnerabilities through its system call filtering and reference monitor. The remaining half of the vulnerabilities were entirely within the kernel or

| Category | Total | Prevented by Graphene | |
|---|---|---|---|
| System call | 118 | 113 | 96% |
| Network | 73 | 30 | 41% |
| File system | 33 | 2 | 6% |
| Drivers | 37 | 0 | |
| VM subsystem | 15 | 0 | |
| Application vulnerabilities | 2 | 2 | 100% |
| Kernel other | 13 | 0 | |
| Total | 291 | 147 | 51% |

Table 7.5: Manual analysis of Linux vulnerabilities from 2011-2013 and Graphene's prevention.

modules, such as bugs in the virtual memory subsystem.

Despite the fact that our primary security goal is isolation, these results indicate that moving the system call table into the application has the potential to substantially reduce exploitable system vulnerabilities.

**New opportunities.** To explore new use cases of the Graphene sandboxing model, we modified the Apache `mod_auth_basic.so` module to call the new library OS function `sandbox_create` after user authentication. The worker process that services the user request executes in a separate sandbox with file system access restricted to only data required for that user. Similarly, this worker's libosannot coordinate shared OS abstractions with other worker processes, limiting the risk to other users if this process is exploited. We see interesting opportunities to expand this model in future work.

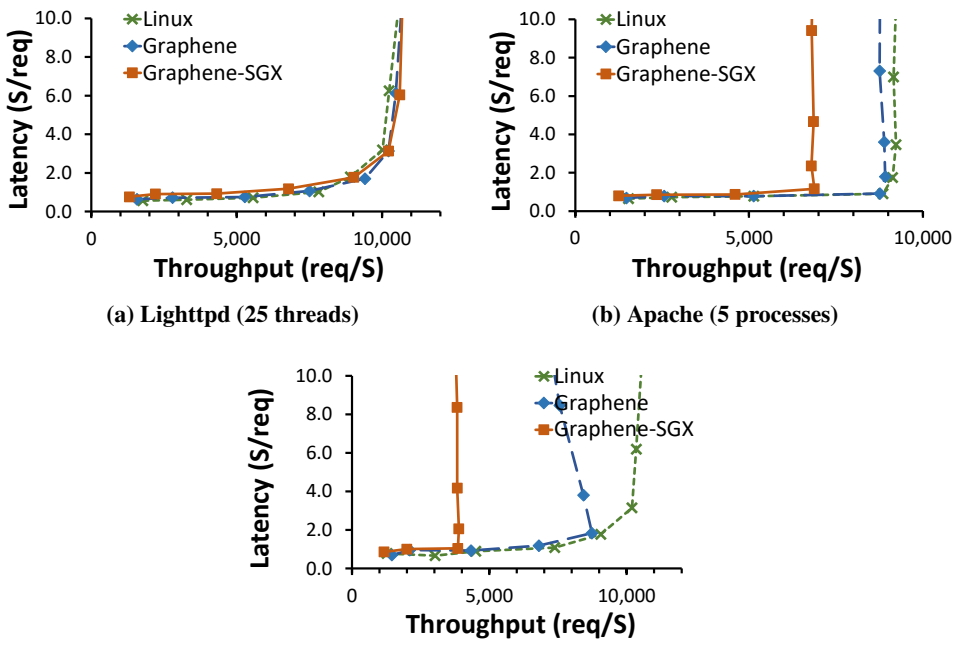
Describe in details.

## 7.2 Graphene-SGX 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.

get rid of the part that we need to compile source code

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




**(a) Lighttpd (25 threads)**

**(b) Apache (5 processes)**

**(c) NGINX (event-driven)**

Figure 7.1: 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.

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 [87] and driver [86].

[dP]: RA asks for a limitations section; I don't think we should do this, but the expensive stuff (events, fork, open) we should add as much as we can about what, if anything, can be done about it, and what can't without more hardware

### 7.2.1 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** [12] (v1.4.35), **Apache** [2] (v2.4.18), and **NGINX** [15] (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 7.1 shows the throughput versus latency of these server applications in Graphene-SGX, Graphene and Linux. Each workload is discussed below.

**Lighttpd** [12] is a web server designed to be light-weight, 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 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.

**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.

**NGINX** [15] 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 [126] to reduce exits would help this workload; in future work, we will improve asynchronous I/O in Graphene-SGX.

### 7.2.2 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 [21]; **GCC** (v5.4), the general GNU C compiler [7]; **CURL** (v7.74), the default command-line web client on UNIX [4]. 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.

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 $\mu$s [160], whereas starting an empty Linux VM takes 10.3s and starting a Linux (LXC) container takes 200 ms [27].

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

added more detailed number

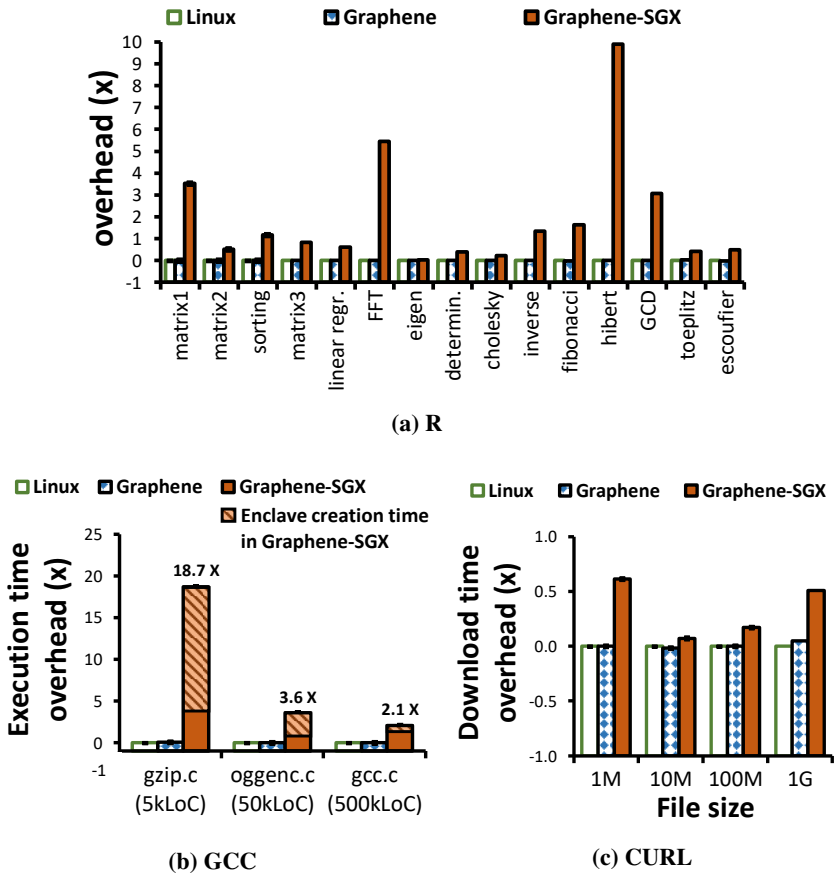



Figure 7.2: Performance overhead on desktop applications, including latency of R, execution time of GCC compilation, download time with CURL. The evaluation compares native Linux, Graphene, and Graphene-SGX.

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 7.2, the enclave creation contributes a significant portion to the execution time overheads, illustrated as a stacked bar.

**R** [21] 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. [20], which includes 15 CPU-bound

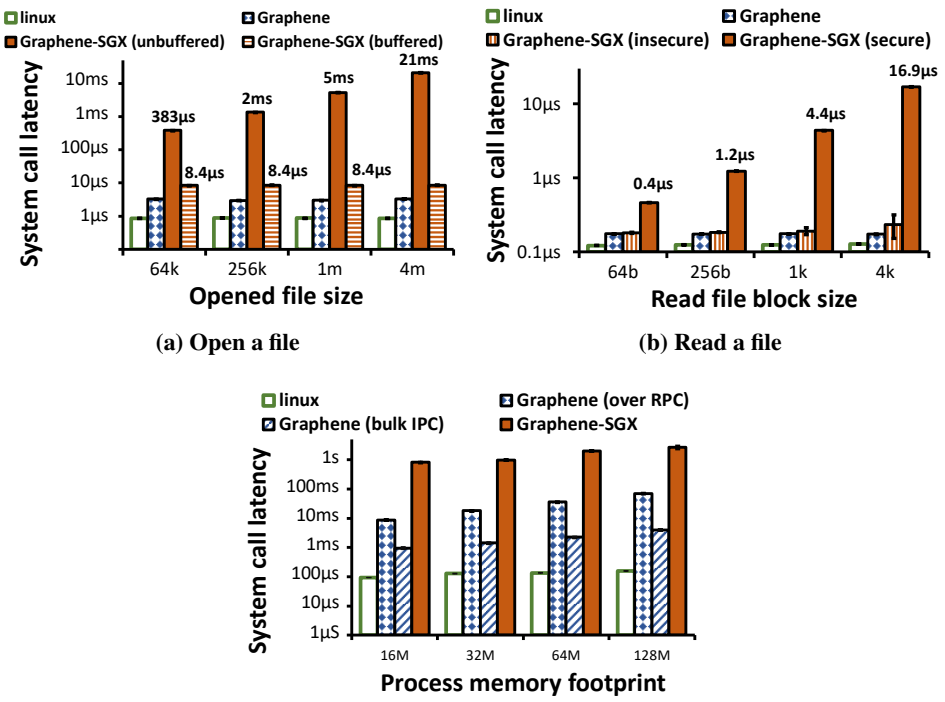
workloads such as matrix computation and number processing. Graphene-SGX slows down by less than 100% on the majority of the workloads, excepts the ones which involve allocation and garbage collection: (`matrix1` creates and destroys matrices, and both `FFT` and `hilbert` involve heavy garbage collection.) Aside from garbage collection, these R benchmarks do not frequently interact with 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** [7] 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 [11]. 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.

it's five, not four

clarified this part, to prevent confusion between latency and overhead. also, GCC numbers got better.

**CURL** [4] 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




**(a) Open a file**

**(b) Read a file**

**(c) Fork a process**

Figure 7.3: 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.

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.

### 7.2.3 Micro-benchmarks

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 [122]. 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 7.3(a) shows the overhead for authenticating files in `open()`. Depending on the file size, the latency of `open()` on Graphene-SGX is 383$\mu$s (64KB file) to 21ms (4MB file), whereas on Linux, the latency is constant at 0.85$\mu$s. 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×.

change overhead to latency

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 7.3(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 0.5$\mu$s (64-byte `read()`) to 16.9$\mu$s (4KB `read()`). The latency of `read()` on Linux is ∼0.1$\mu$s 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()`).

[dP]: Consider doing larger buffers, say up to 64k or even 4 MB

Figure 7.3(c) shows the overhead of forking a process. As described in **??**, 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

| 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 7.6: TCB size (in thousands of lines of code) of Graphene-SGX, SCONE, and PANOPLY.

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.

[dP]: If you want, some thoughts on how this might be improved in the future would be nice... One good suggestion is recycling enclaves, or pre-forking so measurements can be done in parallel

talk about a limitation of improving fork. check this.

### 7.2.4 TCB and 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.

[dP]: Reviewer B asks for memory footprint, which isn't a bad idea

Table 7.6 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 [116]. 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 [**?** ].

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 [90]. Thus, increased generality does not necessarily come with increased risk.

## 7.3 Summary

# Chapter 8

# Evaluating Compatibility

## 8.1 Motivation

When implementing the system APIs and abstractions, system developers routinely make design choice based on what they believe to be the important and unimportant features of the system. However, a developer's view of what APIs are important may be skewed heavily towards that developer's preferred workloads. Similarly, developers struggle to evaluate the impact of a change that affects backward-compatibility, primarily because of a lack of metrics. Deprecating an API is often a lengthy process, wherein users are repeatedly warned and eventually some applications may still be broken. Eliminating or replacing needless, problematic APIs can be good for security, efficiency, and maintainability of OSes, but in practice this is difficult for OS developers to do without tools to analyze API usage.

Many experimental operating systems add a rough Unix or Linux compatibility layer to increase the number of supported applications [36, 40, 70, 174]. Such systems generally support a fraction of Linux system calls, often just enough to run a few target workloads. One metric for compatibility or completeness of a new feature is the count of supported system APIs [44, 48, 135, 160]. System call counts do not accurately estimate the fraction of applications or users that could plausibly use the system. OS researchers would benefit from the ability to translate a set of supported system calls to the fraction of applications that can be directly supported without recompilation. Similarly, it is useful to

know which additional APIs would enable the largest range of additional applications to run on the system. In order to indicate general usefulness, a good compatibility metric should factor in the fraction of users whose choice of applications can be completely supported on a system.

At the root of these problems is a lack of data sets and analysis of how system APIs are used in practice. System APIs are simply not equally important: some APIs are used by popular libraries and, thus, by essentially every application. Other APIs may be used only by applications that are rarely installed. Evaluating compatibility is fundamentally a measurement problem.

## 8.2 API Compatibility Metrics

We started this study from a research perspective, in search of a better way to evaluate the completeness of system prototypes with a Unix compatibility layer. In general, compatibility is treated as a binary property (e.g., bug-for-bug compatibility), which loses important information when evaluating a prototype that is almost certainly incomplete. Papers often appeal to noisy indicators that the prototype probably covers all important use cases, such as the number of total supported system or library calls, as well as the variety of supported applications.

These metrics are easy to quantify, but problematic. Simply put, not all APIs are equally important: some are indispensable (e.g., `read()` and `write()`), whereas others are very rarely used (e.g., `preadv()` and `delete_module()`). A simple count of system calls is easily skewed by system calls that are variations on a theme (e.g., `setuid()`, `seteuid()`, and `setresuid()`). Moreover, some system calls, such as `ioctl()`, export widely varying operations—some used by by *all* applications and many that are essentially never used (§**??**). Thus, a system with "partial support" for `ioctl()` is just as likely to support all or none of the Linux applications distributed with Ubuntu.

One of the ways to understand the importance of a given interface is to measure its impact on end-users. In other words, if a given interface were not supported, how many

users would notice its absence? Or, if a prototype added a given interface, how many more users would be able to use the system? To answer these questions, we must consider both the difference in API usage among applications, and the popularity of applications among end-users. We measure the former by analyzing application binaries, and determine the latter from installation statistics collected by Debian and Ubuntu [65, 163]. An **installation** is a single system installation, and can be a physical machine, a virtual machine, a partition in a multi-boot system, or a chroot environment created by `debootstrap`. Our data is drawn from over 2.9 million installations (2,745,304 Ubuntu and 187,795 Debian).

We introduce two new metrics: one for each API, and one for a whole system. For each API, we measure how disruptive its absence would be to applications and end users—a metric we call **API importance**. For a system, we compute a weighted percentage we call **weighted completeness**. For simplicity, we define a **system** as a set of implemented or translated APIs, and assume an application will work on a target system if the application's API footprint is implemented on the system. These metrics can be applied to all system APIs, or a subset of APIs, such as system calls or standard library functions.

This paper focuses on Ubuntu/Debian Linux, as it is a well-managed Linux distribution with a wide array of supported software, which also collects package installation statistics. The default package installer on Ubuntu/Debian Linux is `APT`. A **package** is the smallest granularity of installation, typically matching a common library or application. A package may include multiple executables, libraries, and configuration files. Packages also track dependencies, such as a package containing Python scripts depending on the Python interpreter. Ubuntu/Debian Linux installation statistics are collected at package granularity and collect several types of statistics. This study is based on data of how many Ubuntu or Debian installations installed a given target package.

For each binary in a package—either as a standalone executable or shared library—we use static analysis to identify all possible APIs the binary could call, or the **API footprint**. The APIs can be called from the binaries directly, or indirectly through calling functions exported by other shared libraries. A package's API footprint is the union of the

API footprints of each of its standalone executables. We weight the API footprint of each package by its installation frequency to approximate the overall importance of each API. Although our initial focus was on evaluating research, our resulting metric and data analysis provide insights for the larger community, such as trends in API usage.

### 8.2.1 API Importance

System developers can benefit from an importance metric for APIs, which can in turn guide optimization efforts, deprecation decisions, and porting efforts. Reflecting the fact that users install and use different software packages, we define API importance as the probability that an API will be indispensable to at least one application on a randomly selected installation. We want a metric that decreases as one identifies and removes instances of a deprecated API, and a metric that will remain high for an indispensable API, even if only one ubiquitous application uses the API.

Intuitively, if an API is used by no packages or installations, the API importance will be *zero*, causing no negative effects if removed. We assume all packages installed in an OS installation are indispensable. As long as an API is used by at least one package, the API is considered *important* for the installation. Appendix A.1 includes a formal definition of API importance.

### 8.2.2 Weighted Completeness

We also measure compatibility at the granularity of an OS, which we call weighted completeness. Weighted completeness is the fraction of applications that are likely to work, weighted by the likelihood that these applications will be installed on a system.

The goal of weighted completeness is to measure the degree to which a new OS prototype or translation layer is compatible with a baseline OS. In this study, the baseline OS is Ubuntu/Debian Linux.

The methodology for measuring the weighted completeness of a target system's API subset is summarized as follows:

1. Start with a list of supported APIs of the target system, either identified from the system's source, or as provided by the developers of the system.
2. Based on the API footprints of packages, the framework generates a list of supported and unsupported packages.
3. The framework then considers the dependencies of packages. If a supported package depends on an unsupported package, both packages are marked as unsupported.
4. Finally, the framework weighs the list of supported packages based on package installation statistics. As with API importance, we measure the effected package that is most installed; weighted completeness instead calculates the expected fraction of packages in a typical installation that will work on the target system.

We note that this model of a typical installation is useful in reducing the metric to a single number, but also does not capture the distribution of installations. This limitation is the result of the available package installation statistics, which do not include correlations among installed packages. This limitation requires us to assume that package installations are independent, except when `APT` identifies a dependency. For example, if packages *foo* and *bar* are both reported as being installed once, we cannot tell if they were on the same installation, or if two different installations. If foo and bar both use an obscure system API, we assume that two installations would be affected if the obscure API were removed. If foo depends on bar, we assume the installations overlap. Appendix A.2 formally defines weighted completeness.

## 8.3 Data Collection

We use static binary analysis to identify the system call footprint of a binary. This approach has the advantages of not requiring source code or test cases. Dynamic system call logging using a tool like `strace` is simpler, but can miss input-dependent behavior. A limitation of our static analysis is that we must assume the disassembled binary matches the expected instruction stream at runtime. In other words, we assume that the binary isn't deliberately

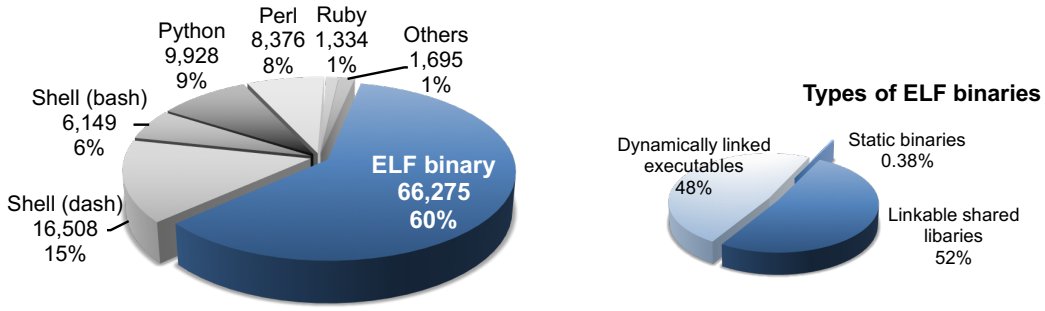



Figure 8.1: Percentage of ELF binaries and applications written in interpreted languages among all executables in the Ubuntu/Debian Linux repository, categorized by interpreters. ELF binaries include static binaries, shared libraries and dynamically-linked executables. Interpreters are detected by *shebangs* of the files. Higher is more important.

obfuscating itself, such as by jumping into the middle of an instruction (from the perspective of the disassembler). To mitigate this, we spot check that static analysis returns as superset of `strace` results.

We note that, in our experience, things like the system call number or even operation codes are fairly straightforward to identify from a binary. These tend to be fixed scalars in the binary, whereas other arguments, such as the contents of a write buffer, are input at runtime. We assume that binaries can issue system calls directly with inline system call instructions, or can call system calls through a library, such as libc. Our static analysis identifies system call instructions and constructs a whole-program call graph.

Our study focuses primarily on ELF binaries, which account for the largest fraction of Linux applications (Figure 8.1). For interpreted languages, such as Python or shell scripts, we assume that the system call footprint of the interpreter and major supporting libraries over-approximates the expected system call footprint of the applications. Libraries that are dynamically loaded, such as application modules or language native interface (e.g.,JNI, Perl XS) are not considered in our study.

Our analysis is based on disassembling binaries inside each application package, using the standard `objdump` tool. This approach eliminates the need for source or recompilation, and can handle closed-source binaries. We implement a simple call-graph analysis

to detect system calls reachable from the binary entry point (`e_entry` in ELF headers). We search all binaries, including libraries, for system call instructions (`int 0x80`, `syscall` or `sysenter`) or calling the `syscall` API of libc. We find that the majority of binaries — either shared libraries or executables — do not directly choose system calls, but rather use the GNU C library APIs. Among 66,275 studied binaries, only 7,259 executables and 2,752 shared libraries issue system calls.

Our call-graph analysis allows us to only select system calls that are actually used by the application, not all the system calls that appear in libc. Our analysis takes the following steps:

- For a target executable or a library, generate a call graph of internal function usage.
- For each library function that the executable relies on, identify the code in the library that is reachable from each entry point called by the executable.
- For each library function that calls another library call, recursively trace the call graph and aggregate the results.

Precisely determining all possible call-graphs from static analysis is challenging. Unlike other tools built on call-graphs, such as control flow integrity (CFI), our framework can tolerate the error caused by over-approximating the analysis results. For instance, programs sometimes make function call based on a function pointer passed as an argument by the caller of the function. Because the calling target is dynamic, it is difficult to determine at the call site. Rather, we track sites where the function pointers are assigned to a register, such as using the `lea` instruction with an address relative to the current program counter. This is an over-approximation because, rather than trace the data flow, we assuming that a function pointer assigned to a local variable will be called. This analysis could be more precise if it included a data flow component.

We also hard-code for a few common and problematic patterns. For instance, we generally assume that the registers that pass a system call number to a system call, or an opcode to a vectored system call, are not the result of arithmetic in the same function. We spot checked this assumption, but did not do the data flow analysis to detect this case.

| Evaluation Criteria | Size |
|---|---|
| Source Lines of Code (Python) | 3,105 |
| Source Lines of Code (SQL) | 2,423 |
| Total Rows in Database | 428,634,030 |

Table 8.1: Implementation of the API usage analysis framework.

Finally, the last mile of the analysis is to recursively aggregate footprint data. We insert all raw data into a `Postgresql` database, and use recursive SQL queries to generate the results. To scan through all 30,976 packages in the repository, collect the data, and generate the results takes roughly three days.

Our implementation is summarized in Table 8.1. We wrote 3,105 lines of code in Python and 2,423 lines of code in SQL (Postgresql). The database contains 48 tables with over 428 Million entries.

## 8.4 Limitations

**Popularity Contest Dataset.** The analysis in this paper is limited by the Ubuntu/Debian Linux's package installer, `APT`, and their package installation statistics. Because most packages in Ubuntu/Debian Linux are open-source, our observations on Linux API usage may have a bias toward open-source development patterns. Commercial applications that are purchased and distributed through other means are not included in this survey data, although data from other sources could, in principle, be incorporated into the analysis if additional data were available.

We assume that the package installation statistics provided by Ubuntu/Debian Linux are representative. The popularity contest dataset is reasonably large (2,935,744 installations), but reporting is opt-in. Moreover, the data does not show how often these packages are actually used, only how often they are installed. Finally, this data set does not include sufficient historical data to compare changes to the API usage over time.

**Static Analysis.** Because our study only analyzes pre-compiled binaries, some compile-time customizations may be missed. Applications that are already ported using macro like `#ifdef LINUX` will be considered dependent to Linux-specific APIs, even though the application can be re-compiled for other systems. Our static analysis tool only identifies whether an API is potentially used, not how frequently the API is used during the execution. Thus, it is not sufficient to draw inferences about performance.

We assume that, once a given API (e.g., `write`) is supported and works for a reasonable sample of applications, handling missed edge cases should be straightforward engineering that is unlikely to invalidate the experimental results of the project. That said, in cases where an input can yield significantly different behavior, e.g., the path given to `open`, we measure the API importance of these arguments. Verifying bug-for-bug compatibility generally requires techniques largely orthogonal to the ones used in this study, and thus this is beyond the scope of this work.

We do not do inter-procedural data-flow analysis. As a result, we were unable to identify system call numbers for 2,454 call sites (4% of the relevant call sites) across all binaries in the repository. As a result, some system call usage values may be underestimated, and may go up with a more sophisticated static analysis.

**Metrics.** The proposed metrics are intended to be simple numbers for easy comparison. But this coarseness loses some nuance. For instance, our metrics cannot distinguish between APIs that are critical to a small population, such as those that offer functionality that cannot be provided any other way, versus APIs that are rarely used because the software is unimportant. Similarly, these metrics alone cannot differentiate a new API that is not yet widely adopted from an old API with declining usage.

## 8.5 System Evaluation

This section uses weighted completeness to evaluate systems or emulation layers with partial Linux compatibility. We also evaluate several libc variants for their degree of complete-

| Systems | # | Suggested APIs to add | Weighted completeness |
|---|---|---|---|
| User-Mode-Linux 3.19 | 284 | `name_to_handle_at`, `iopl`, `perf_event_open` | 93.1% |
| L4Linux 4.3 | 286 | `quotactl`, `migrate_pages`, `kexec_load` | 99.3% |
| FreeBSD-emu 10.2 | 225 | `inotify*`, `splice`, `umount2`, `timerfd*` | 62.3% |
| Graphene | 143 | `sched_setscheduler`, `sched_setparam` | 0.42% |
| Graphene¶ | 145 | `statfs`, `utimes`, `getxattr`, `fallocate`, `eventfd2` | 21.1% |

Table 8.2: Weighted completeness of several Linux systems or emulation layers. For each system, we manually identify the number of supported system calls ("#"), and calculate the weighted completeness ("W.Comp.") . Based on API importance, we suggest the most important APIs to add. (*: system call family. ¶: Graphene after adding two more system calls.)

ness against the APIs exported by GNU libc 2.21.

### 8.5.1 Linux compatibility layers

To evaluate the weighted completeness of Linux systems or emulation layers, the prerequisite is to identify the supported APIs of the target systems. Due to the complexity of Linux APIs and system implementation, it is hard to automate the process of identification. However, OS developers are mostly able to maintain such a list based on the internal knowledge.

We evaluate the weighted completeness of four Linux-compatible systems or emulation layers: User-Mode-Linux [68], L4Linux [81], FreeBSD emulation layer [69], and Graphene library OS [160]. For each system, we explore techniques to help identifying the supported system calls, based on how the system is built. For example, User-Mode-Linux and L4Linux are built by modifying the Linux source code, or adding a new architecture to Linux. These systems will define architecture-specific system call tables, and reimplement `sys_*` functions in the Linux source that are originally aliases to `sys_ni_syscall` (a function that returns `-ENOSYS`). Other systems, like FreeBSD and Graphene, are built from scratch, and often maintain their own system call table structures, where unsupported

| Libc variants | # | Unsupported functions (samples) | Weighted completeness | Weighted completeness (normalized) |
|---|---|---|---|---|
| eglibc 2.19 | 2198 | None | 100% | 100% |
| uClibc 0.9.33 | 1867 | `__uflow`, `__overflow` | 1.1% | 41.9% |
| musl 1.1.14 | 1890 | `secure_getenv`, `random_r` | 1.1% | 43.2% |
| dietlibc 0.33 | 962 | `memalign`, `stpcpy`, `__cxa_finalize` | 0% | 0% |

Table 8.3: Weighted completeness of libc variants. For each variant, we calculate weighted completeness based on symbols directly retrieved from the binaries, and the symbols after reversing variant-specific replacement (e.g.,`printf()` becomes `__printf_chk()`).

systems calls are redirected to dummy callbacks.

Table 8.2 shows weighted completeness, considering only system calls. The results also identify the most important system calls that the developers should consider adding. User-Mode-Linux and L4Linux both have a weighted completeness over 90%, with more than 280 system calls implemented. FreeBSD's weighted completeness is 62.3% because it is missing some less important system calls such as `inotify_init` and `timerfd_create`. Graphene's weighted completeness is only 0.42%. We observe that the primary culprit is scheduling control; by adding two scheduling system calls, Graphene's weighted completeness would be 21.1%.

### 8.5.2 Standard C libraries

This study also uses weighted completeness to evaluate the compatibility of several libc variants — eglibc [6], uClibc [22], musl [14] and dietlibc [5] — against GNU libc, listed in Table 8.3. We observe that, if simply matching exported API symbols, only eglibc is directly compatible to GNU libc. Both uClibc and musl have a low weighted completeness, because GNU libc's headers replace a number of APIs with safer variants at compile time, using macros. For example, GNU libc replaces `printf` with `__printf_chk`, which per-

forms an additional check for stack overflow. After normalizing for this compile-time API replacement, both uClibc and musl are at over 40% weighted completeness. In contrast, dietlibc is still not compatible with most binaries linked against GNU libc — if no other approach is taken to improve its compatibility. The reason of low weighted completeness is that dietlibc does not implement many ubiquitously used GNU libc APIs such as `memalign` (used by 8887 packages) and `__cxa_finalize` (used by 7443 packages).

## 8.6 Summary

Traditionally, the routine procedure for system engineers or researchers to make implementation decisions is mostly based on their anecdotal knowledge, which may be partially credible, but heavily skewed toward their preferred or familiar workloads. The consequence of the lack of information can be unfavorable for developers who are building innovative systems with legacy application support. With the binary, bug-for-bug compatibility, the developers fail to methodologically evaluate and reasonable about the completeness of API implementation in their system prototypes, until the implementation is completed. As produced by this study, a principled approach for determining the priority of API implementation, to enable more applications or more users that can plausible use the system, will guide the developers to make more rewarding decisions.

# Chapter 9

# A Study of System APIs

This chapter present a thorough study of the Linux and POSIX system API, to motivate the design decisions in Graphene for prioritizing the implementation of OS features. This chapter also contributes several insight for the system API usage and compatibility, based on the importance to applications and users.

## 9.1 Linux System Calls

There are 320 system calls defined in x86-64 Linux 3.19 (as listed in `unistd.h`). Figure 9.1 shows the distribution of system calls by importance. The Figure is ordered by most important (at 100%) to least important (around 0%)—similar to inverted CDF. The figure highlights several points of interest on this line.

Over two-thirds (*224 of 320*) of system calls on Linux are indispensable: required by at least one application on every installation. Among the rest, 33 system calls are important on more than ten percent of the installations. 44 system calls have very low API importance: less than ten percent of the installations include at least one application that uses these system calls.

Our study also shows the contributors to an API's importance. For instance, Table 9.1 lists system calls that are only called by one or two particular libraries (e.g., libc). These system calls are wrapped by library APIs, so applications depend on them only because the libraries do. To eliminate the usage of these system calls, developers only have to

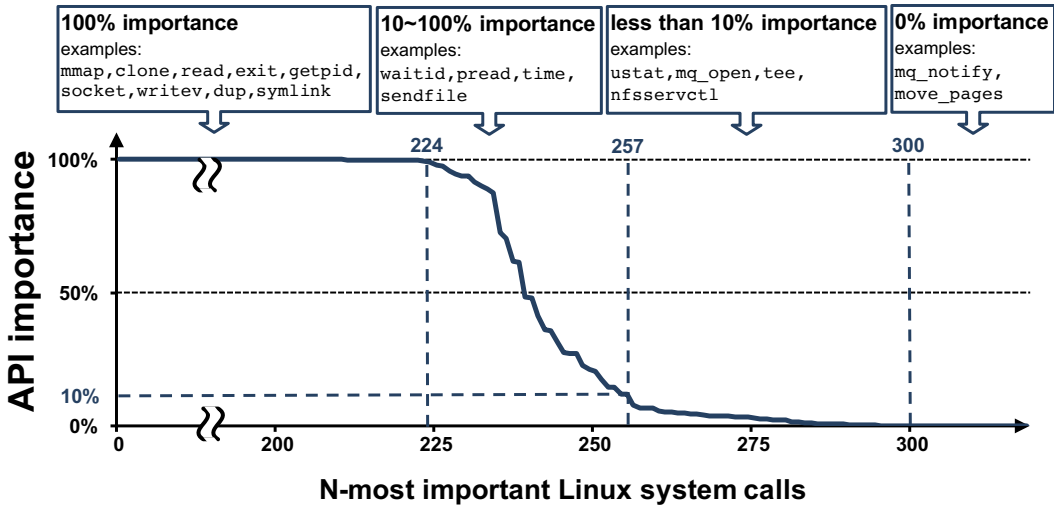



Figure 9.1: The trend of API importance as N-most important system calls among total 320 system calls of Ubuntu Linux 15.04 with Linux kernel 3.19. . Higher is more important; 100% indicates all installations include software that make the system call.

pay minimum efforts to re-implement the wrappers in libraries.

Among the 44 system calls with a API importance above zero but less than ten percent, some are cases where a more popular alternative is available. For instance, Linux supports both POSIX and System V message queues. The five APIs for POSIX message queues have a lower API importance than System V message queues. We believe this is attributable to System V message queues being more portable to other UNIX systems. Similarly, we observed that `epoll_wait` (100%) has a higher API importance than `epoll_pwait` (3%), even though `epoll_pwait` is commonly considered more robust for the same purpose—waiting on file descriptor events. Table 9.2 lists system calls used by only one or two packages—generally special-purpose utilities, such as `kexec_load`, which is used by `kexec-tools`).

In some cases, system calls are effectively offloaded to a file in `/proc` or `/sys`. For instance, some of the information that was formerly available via `query_module` can be obtained from `/proc/modules`, `/proc/kallsyms` and the files under the directory `/sys/module`. Similarly, the information that can be obtained from the `sysfs` system call is now available in `/proc/filesystems`.

| System Calls | API Importance | Used Packages |
|---|---|---|
| clock_settime, iopl, ioperm, signalfd4 | 100% | libc |
| mbind | 36.0% | libnuma, libopenblasp |
| addkey | 27.2% | libkeyutils |
| keyctl | 27.2% | pam_keyutil, libkeyutils |
| requestkey | 14.4% | libkeyutils |
| preadv, pwritev | 11.7% | libc |

Table 9.1: System calls which are only directly used by particular libraries, and their API importance. Only system calls with API importance larger than ten percent are shown. These system calls are wrapped by library APIs, thus they are easy to deprecate by modifying the libraries.

| System Calls | API Importance | Used Packages |
|---|---|---|
| seccomp, sched_setattr, sched_getattr | 1% | coop-computing-tools |
| kexec_load | 1% | kexec-tools |
| clock_adjtime | 4% | systemd |
| renameat2 | 4% | systemd, coop-computing-tools |
| mq_timedsend, mq_getsetattr | 1% | qemu-user |
| io_getevent | 1% | ioping, zfs-fuse |
| getcpu | 4% | valgrind, rt-tests |

Table 9.2: System calls with usage dominated by particular package(s), and their API importance. This table excludes system calls that are officially retired.

We also found five system calls uselib, nfsservctl, afs_syscall, vserver and security system calls that are officially retired, but still have a low, but non-zero, API importance. For instance nfsservctl is removed from Linux kernel 3.1 but still has API importance of seven percent, because it is tried by NFS utilities such as exportfs. These utilities still attempt the old calls for backward-compatibility with older kernels.

In total, 18 of 320 system calls in Linux 3.19 are not used by any application in the Ubuntu/Debian Linux repository. We list these system calls in Table 9.3. In addition to the issues discussed above, Ten of these system calls do not have an entry point, but are still defined in the Linux headers. Five of the unused system calls such as rt_tgsigqueueinfo, get_robust_list, remap_file_pages, mq_notify, lookup_dcookie provide an interface that is not used by the applications. These system calls can be potential candidates for deprecation. However, even though restart_syscall is not used by any application, it is

| Unused System Calls | Reason for Disuse |
|---|---|
| set_thread_area, tuxcall, create_module, and 7 more. | Officially retired. |
| sysfs | replaced by /proc/filesystems. |
| rt_tgsigqueueinfo, get_robust_list | Unused by applications. |
| remap_file_pages | No non-sequential ordered mapping; repeated calls to mmap preferred. |
| mq_notify | Unused: Asynchronous message delivery. |
| lookup_dcookie | Unused: for profiling. |
| restart_syscall | Transparent to applications. |
| move_pages | Unused: for NUMA usage. |

Table 9.3: Unused system calls and explanation for disuse.

internally used by the kernel.

Figure 9.2 shows the optimal path of adding system calls to a prototype system, using a simple, greedy strategy of implementing the N-most important APIs, which in turns maximizes weighted completeness. In other words, the leftmost points on the graph are the most important APIs, but the y coordinate only increases once enough system calls are supported that a simple program, such as "hello world" can execute. Similar to a CDF, this line continues up to 100% of Ubuntu applications. The graph highlights several points of interest in this curve.

Essentially, one cannot run even the most simple programs without at least 40 system calls. After this, the number of additional applications one can support by adding another system call increases steadily until an inflection point at 125 system calls, or supporting extended attributes on files, where weighted completeness jumps to 25%. To support roughly half of Ubuntu/Debian Linux applications, one must have 145 system calls, and the curve plateaus around 202 system calls. On the most extreme end, qemu's MIPS emulator (on an x86-64 host) requires 270 system calls [47]. A weighted completeness of 100% implies that all Linux applications ever used are supported by the prototype.

Table 9.4 breaks down the recommended development phases by rough categories of required system calls. We do not provide a complete ordered list here in the interest of brevity, but this list is available as part of our dataset, at http://oscar.cs.stonybrook.

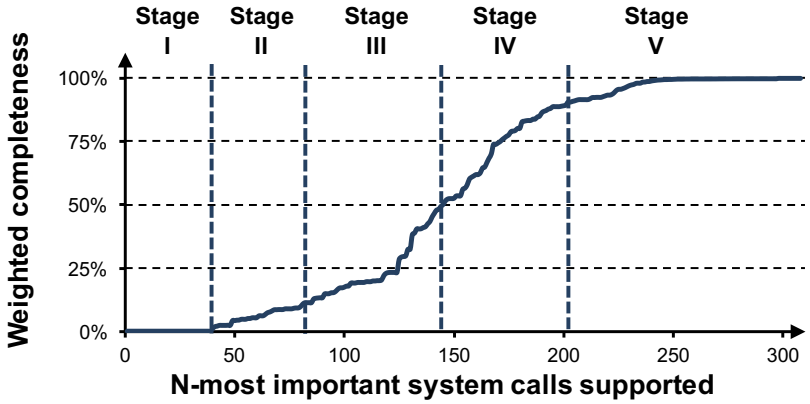



Figure 9.2: Accumulated weighted completeness when N top-ranked system calls are implemented in the OS. Higher is more compliant.

`edu/api-compat-study`.

A goal of weighted completeness is to help guide the process of developing new system prototypes. Section **??** showed that 224 out of 320 system calls on Ubuntu/Debian Linux have 100% API importance. In other words, if one of these 224 calls is missing, at least one application on a typical system will not work. Weighted completeness, however, is more forgiving, as it tries to capture the fraction of a typical installation that could work. Only 40 system calls are needed to have weighted completeness more than 1%.

For simplicity, Table 9.4 only includes system calls, but one can construct a similar path including other APIs, such as vectored system calls, pseudo-files and library APIs. For example, developers need not implement every operation of `ioctl`, `fcntl` and `prctl` during the early stage of developing a system prototype.

## 9.2 Opcodes for Vectored System Calls

Some system calls, such as `ioctl`, `fcntl`, and `prctl`, essentially export a secondary system call table, using the first argument as an operation code. These *vectored* system calls significantly expand the system API, dramatically increasing the effort to realize full API compatibility. It is also difficult to enforce robust security policies on these interfaces, as the arguments to each operation are highly variable.

| Stage | Sample System Calls | # syscalls | Weighted Completeness |
|---|---|---|---|
| I | `mmap`, `vfork`, `exit`, `read`, `gettid`, `fcntl`, `getcwd` `sched_yield`, `kill`, `dup2` | 40 | 1.12 % |
| II | `mremap`, `ioctl`, `access`, `socket`, `poll`, `recvmsg`, `dup`, `unlink`, `wait4`, `select`, `chdir`, `pipe` | +41 (81) | 10.68 % |
| III | `sigaltstack`, `shutdown`, `symlink`, `alarm`, `listen`, `pread64`, `getxattr`, `shmget`, `epoll_wait`, `chroot` | +64 (145) | 50.09 % |
| IV | `flock`, `semget`, `ppoll`, `mount`, `brk`, `pause`, `clock_gettime`, `getpgid`, `settimeofday`, `capset` | +57 (202) | 90.61 % |
| V | All remaining | +70 (272) | 100 % |

Table 9.4: Five stages of implementing system calls based on the API importance ranking. For each stage, a set of system calls is listed, with the work needed to accomplish (# of system calls) and the weighted completeness that can be reached.

The main expansion is from `ioctl`. Linux defines 635 operation codes, and Linux kernel modules and drivers can define additional operations. In the case of `ioctl`, we obverse that there are 52 operations with the 100% API importance (Figure 9.3), each of which are as important as the 226 most important system calls. Of these 52 operations, 47 are frequently used operations for TTY console (e.g., `TCGETS`) or generic operations on IO devices (e.g., `FIONREAD`).

On the narrow end, `fcntl` and `prctl` have 18 and 44 operations, respectively, in Linux kernel 3.19. Unlike `ioctl`, `fcntl` and `prctl` are not extensible by modules or drivers, and their operations tend to have higher API importance (Figure 9.3). For `fcntl`, eleven out of eighteen `fcntl` operations in Linux 3.19 have API importance at around 100%. For `prctl`, only nine out of 44 operations have API importance around 100%, and only eighteen has API importance larger than 20%.

Thus, developers of a new system prototype should support these 47 most important `ioctl` operations, about half of the `fcntl` opcodes, and only 9–20 `prcntl` operations.

Compared to system calls, `ioctl` has a much longer tail of infrequently used operations. Out of 635 `ioctl` operation codes defined by modules or drivers hosted in Linux

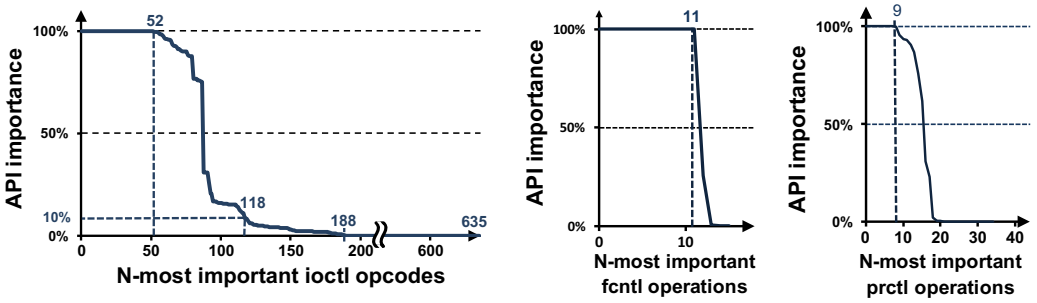



Figure 9.3: Ranking of API importance among ioctl, `fcntl` and `prctl` opcodes. Higher is more important; 100% indicates all installations include software that request the operations.

kernel 3.19, only 188 have API importance more than one percent, and for only 280 we can find usage of the operations in at least one application binary. Those unused operations are good targets for deprecation, in the interest of reducing the system attack surface.

## 9.3 Pseudo Files and Devices

In addition to the main system call table, Linux exports many additional APIs through pseudo-file systems, such as `/proc`, `/dev`, and `/sys`. These are called pseudo-file systems because they are not backed by disk, but rather export the contents of kernel data structures to an application or administrator as if they were stored in a file. These pseudo-file systems are a convenient location to export tuning parameters, statistics, and other subsystem-specific or device-specific APIs. Although many of these pseudo-files are used on the command line or in scripts by an administrator, a few are routinely used by applications. In order to fully understand usage patterns of the Linux kernel, pseudo-files must also be considered.

We apply static analysis to find cases where the binary is hard-coded to use a pseudo-file. Our analysis cannot capture cases where a path to one of these file systems is passed as an input to the application, such as `dd if=/dev/zero`. However, when a pseudo-file is widely-used as a replacement for a system call, these paths tend to be hard-coded in the binary as a string or string pattern. A common pattern we observed was `sprintf(‘‘/proc/%d/cmdline’’, pid)`; our analysis captured these patterns as well.

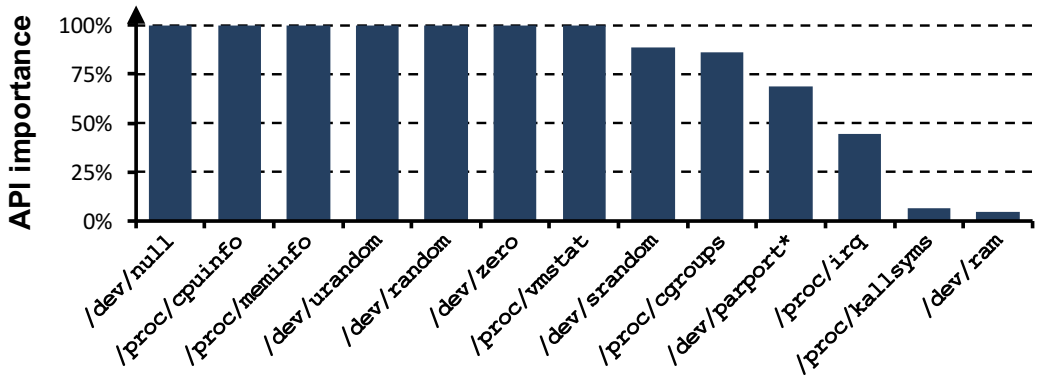



Figure 9.4: API importance distribution over files under `/dev` and `/proc`. Higher is more important; 100% indicates all installations include software that accesses the file.

We also do not differentiate types of access in this study, such as separating read versus write of a pseudo-file; rather we only consider whether the file is accessed or not. Thus, our analysis is limited to strings stored in the binary, but we believe this captures an important usage pattern.

Figure 9.4 shows the API importance of common pseudo-files under `/dev` and `/proc`. These files are ordered from highest API importance; the long tail of files used rarely or directly by administrators is omitted.

Some files are essential, such as `/dev/null` and `/proc/cpuinfo`. These files are widely used in binaries and scripts. Among 12,039 binaries that use a hard-coded path, 3,324 access `/dev/null` and 439 access `/proc/cpuinfo`. However, it is plausible to provide the same functionality in simpler ways. For instance, `/proc/cpuinfo` provides a formatted wrapper for the `cpuinfo` instruction, which one could export directly to userspace using virtualization hardware, similar to Dune [46]. Similarly, `/dev/zero` or `/dev/null` are convenient for use on the command line, but it is surprising that a significant number of applications issue `read` or `write` system calls, rather than simply zeroing a buffer or skipping the `write` (e.g., `grub-install`). Thus, in implementing a Linux compatibility layer, a small number of pseudo-files are essential, and perhaps others could be eliminated with modest application changes.

APIs as pseudo-files or pseudo-devices also have a large subset of infrequently used or unused APIs. Many of them are designed to support one specific application or user. For example, `/dev/kvm` is only intended for `qemu` to communicate with the kernel portion of the KVM hypervisor. Similarly `/proc/kallsyms` is used primarily to export debugging information to kernel developers.

Because so many files in `/proc` are accessed from the command line or by only a single application, it is hard to conclude that any should be deprecated. Nonetheless, these files represent large and complex APIs that create an important attack surface to defend. As noted in other studies, the permission on `/proc` tend to be set liberally enough to leak a significant amount of information [91]. For files used by a single application, an abstraction like a fine-grained capability [147] might better capture this need. For files used primarily by the administrator, carefully setting directory permissions should be sufficient.

In the case of the `/dev` file system, the most commonly used files are pseudo-devices, such as accessing the virtual terminal (`/dev/tty`, `/dev/console`, and `/dev/pts`), or other functionality such as the random number generator (`/dev/urandom`). Even among pseudo-devices, features such as accessing one's standard in and out, or a process's TTY via the `/dev/` interface are not heavily used.

Intuitively, one would not expect many device paths to be hard-coded, and most direct interactions with a device would be done using administrative tools. For instance, we see that some applications do hard-code paths like `/dev/hda` (commonly used for an IDE hard drive), yet an increasing number of systems have a root hard drive using SATA, which would consequently be named `/dev/sda`. Thus, although applications may use paths like `/dev/hda` as a default device path, modern systems are sufficiently varied that these generally need to be searched at runtime.

## 9.4 C Library Functions

In addition to studying kernel interfaces, we also analyze the API importance of APIs defined in core system libraries, such as libc. Most programmers don't directly use the APIs

exported by the kernel, but instead program to more user-friendly APIs in libc and other libraries. For instance, GNU libc [8] exports APIs for using locks and condition variables, which internally use the subtle `futex` system call [75].

Our result shows that among the global function symbols exported by libc — 1,274 in total — 42.8% have a API importance of 100%, 50.6% have a API importance of less than 50%, and 39.7% have a API importance of less than one percent, including some that are not used at all. In other words, about 40% of the APIs inside libc are either not used or only used by few applications. This result implies that most processes are loading a significant amount of unnecessary code into their address space. By splitting libc into several sub-libraries, based on API importance and common linking patterns, systems could realize a non-trivial space savings.

There are several reasons to avoid loading extra code into an application. First, there are code reuse attacks, such as return-oriented programming (ROP) [145], that rely on the ability to find particular code snippets, called gadgets. Littering a process with extra gadgets offers needless assistance to an attacker. Similarly, when important and unimportant APIs are on the same page, memory is wasted. Finally, the space overhead of large, unused jump tables is significant. In GNU libc 2.21, `libc-2.21.so` essentially has 1274 relocation entries, occupying 30,576 bytes of virtual memory. By sorting the relocation table according to API usage, most libc instances could load only first few pages of relocation tables, and leave the remaining relocation entries for lazy loading.

We analyzed the space savings of a GNU libc 2.21 which removed any APIs with API importance lower than 90%. In total, libc would retain 889 APIs and the size would be reduced to 63% of its original size. The probability an application would need a missing function and load it from another library is less than 9.3%(equivalent to 90.7% weighted completeness for the stripped libc). Further decomposition is also possible, such as placing APIs that are commonly accessed by the same application into the same sub-library.

**Effects of standard libraries on API importance.** Libc and the dynamic linker (`ld.so`) also contribute to the system call footprint of every dynamically-linked executable. This has

| System Calls | Libraries |
|---|---|
| access, arch_prctl, mprotect | ld.so |
| clone, execve, getuid, gettid, kill, getrlimit, setresuid | libc |
| close, exit, exit_group, getcwd, getdents, getpid, lseek, lstat, mmap, munmap, madvise, mprotect, mremap, newfsstat, read | libc, ld.so |
| rt_sigreturn, set_robust_list, set_tid_address | libpthread |
| rt_sigprocmask | librt |
| futex | libc, ld.so, libpthread |

Table 9.5: Ubiquitous system call usage caused by initialization or finalization of libc family.

a marked effect on the API importance of some system calls. The APIs used to initialize a program are listed in Table 9.5. In several cases, such as `set_tid_address`, however, libc or libpthread may be the only binaries using these interfaces directly, indicating that changes to some important system interfaces would only require changes in one or two low-level libraries.

## 9.5 Unweighted API Importance

API importance is weighted by the number of installations of applications that use the API. As a result, one ubiquitous application can cause the API importance of an API it uses to be close to 100%. This section observes trends for APIs with multiple variants, using an additional unweighted API importance metric. We remove the weighting by installation frequency to focus on trends in developer behavior.

Once an API has been identified as having a security risk, and a more secure variant is developed, one might wish to know how many vulnerable packages are still in the wild, and how many have moved to less exploit-prone APIs. Similarly, one might want to know how many applications have not migrated away from a deprecated API, even if these applications are not widely used.

One family of APIs prone to security problems are the `set*id()` API family. Many of the `set*id()` APIs have subtle semantic differences across different Unix variants. Chen

et al. [58] conclude that `setresuid()` has the clearest semantics across all Unix flavors. Table 9.6 shows the unweighted API importance of `set*id()` and `get*id()` system calls. Most packages have adopted the more clear and secure interface. System calls `setuid()`, `setreuid()`, and `setresuid()` have unweighted API importance of 15.67%, 1.88% and 99.68% respectively. However, for `get*id` system calls, the unweighted API importance suggests that the `getres*id` system calls are only used by roughly 36% of packages.

Directory operations have a long history of exploitable race conditions [51, 54, 170], or time-of-check-to-time-of-use (TOCTTOU) vulnerabilities. In a privileged application, one system call (e.g., `access`) checks the user's permission, and a second call operates on the file. There are countermeasures that effectively walk the directory hierarchy in user space [159]. This approach replaces calls like `access` with `faccessat`, and similar variants. Table 9.6 shows the current unweighted API importance of `*at` system call variants and their older counterparts. We observed that the unweighted API importance of the race-prone `access` is still high (74.24%), whereas `faccessat` is only 0.63%. This suggests about 75% of the packages use the more vulnerable `access` system call instead of the more secure one.

In addition to security-related hints, unweighted API importance indicates whether obsolete APIs have been replaced by newer variants. For instance, `wait4` system call is considered obsolete [165], and the alternative `waitid` is preferred, as it more precisely specifies which child state changes to wait for. However, unweighted API importance of `wait4` and `waitid` is 60.56% and 0.24%, respectively. This indicates that 60% of the packages are still using the older `wait4` system call. Table 9.7 shows similar trend for some other system calls. Our dataset provides more opportunity for system developers to actively communicate with application developers, in order to speed up the process of retiring problematic APIs.

Some APIs are specific to a particular OS, such as Linux, and often have more portable variants. Table 9.7 shows the comparison between Linux-specific APIs and their generic variants. The results show most developers prefer portable or generic APIs more

| Insecure API | Unweighted API Importance | Secure API | Unweighted API importance |
|---|---|---|---|
| **Unclear vs. Well-defined ID Management Semantics** | | | |
| `setuid` | 15.67% | `setresuid` | 99.68% |
| `setreuid` | 1.88% | | |
| `setgid` | 12.07% | `setresgid` | 99.68% |
| `setregid` | 1.24% | | |
| `getuid` | 99.81% | `getresuid` | 36.19% |
| `geteuid` | 55.15% | | |
| `getgid` | 99.81% | `getresgid` | 36.14% |
| `getegid` | 48.87% | | |
| **Nonatomic vs. Atomic Directory operations** | | | |
| `access` | 74.24% | `faccessat` | 0.63% |
| `mkdir` | 52.07% | `mkdirat` | 0.34% |
| `rename` | 43.18% | `renameat` | 0.30% |
| `readlink` | 46.38% | `readlinkat` | 0.50% |
| `chown` | 24.59% | `fchownat` | 0.23% |
| `chmod` | 39.80% | `fchmodat` | 0.13% |

Table 9.6: Unweighted API importance of secure and insecure API variations. Higher is more important.

than Linux-specific APIs. Except `pipe2`, most API variants that are Linux-specific have unweighted API importance lower than 10 percent.

[dP]: Can you comment on why pipe2 is popular?

Finally, we consider system calls with multiple variants where one version has increased functionality. Table 9.7 shows the difference between these system calls. Interestingly, more developers chose the less powerful variants, such as using `select()` over `pselect6()`, or `dup2()` over `dup3()`. This indicates that more often than not, developers choose simplicity unless a task demands the functionality of a more powerful API variant.

| Old API | Unweighted API Importance | New API | Unweighted API Importance |
|---|---|---|---|
| `getdents` | 99.80% | `getdents64` | 0.08% |
| `utime` | 8.57% | `utimes` | 17.90% |
| `fork` | 0.07% | `clone` | 99.86% |
| `vfork` | 99.68% | | |
| `tkill` | 0.51% | `tgkill` | 99.80% |
| `wait4` | 60.56% | `waitid` | 0.24% |

(a) old (generally deprecated) vs new (preferred)

| Linux Specific API | Unweighted API importance | Portable / Generic API | Unweighted API importance |
|---|---|---|---|
| `preadv` | 0.15% | `readv` | 62.23% |
| `pwritev` | 0.16% | `writev` | 99.80% |
| `accept4` | 0.93% | `accept` | 29.35% |
| `ppoll` | 3.90% | `poll` | 71.07% |
| `recvmmsg` | 0.11% | `recvmsg` | 68.82% |
| `sendmmsg` | 5.17% | `sendmsg` | 42.49% |
| `pipe2` | 40.33% | `pipe` | 50.33% |

(b) Linux-specific vs. portable/generic

| Linux API | Unweighted API importance | Alternative API | Unweighted API importance |
|---|---|---|---|
| `read` | 99.88% | `pread64` | 27.23% |
| `select` | 61.53% | `pselect6` | 4.13% |
| `dup3` | 8.72% | `dup2` | 99.75% |
| | | `dup` | 66.64% |
| `recvmsg` | 68.82% | `recvfrom` | 53.80% |
| `sendmsg` | 42.49% | `sendto` | 71.71% |

(c) comparison of other API variants

Table 9.7: Unweighted API importance among API variants. Higher is more important.

## 9.6 Summary

Traditionally, the routine procedure for system engineers or researchers to make implementation decisions is mostly based on their anecdotal knowledge, which may be partially credible, but heavily skewed toward their preferred or familiar workloads. The consequence of the lack of information can be unfavorable for developers who are building innovative systems with legacy application support. With the binary, bug-for-bug compatibility, the developers fail to methodologically evaluate and reasonable about the completeness of API implementation in their system prototypes, until the implementation is completed. As produced by this study, a principled approach for determining the priority of API implementation, to enable more applications or more users that can plausible use the system, will guide the developers to make more rewarding decisions.

# Chapter 10

# Related Works

## 10.1 Library OSes

**Previous library OSes.** This work extends previous library OSes [16, 44, 70, 116, 134], which focused on single-process applications, to support coordination abstractions required for multi-process applications, such as shell scripts.

Bascule [44] implements a Linux library OS on a variant of the Drawbridge ABI, but does not include support for multi-process abstractions such as signals or copy-on-write fork. The Bascule Linux library OS also implements fewer Linux system calls than Graphene, missing features such as signals. Bascule demonstrates a complementary facility to Graphene's multi-process support: composable library OS extensions, such as speculation and record/replay. OSv is a recent open-source, single-process library OS to support a managed language runtime, such as Java, on a bare-metal hypervisor [16].

A number of recent projects have provided a minimal, isolated environment for web applications to download and execute native code [70, 83, 123, 168, 172]. The term "picoprocess" is adopted from some of these designs, and they share the goal of pushing system complexity out of the kernel and into the application. Unlike a library OS, these systems generally sacrifice the ability to execute unmodified application code, eliminate common UNIX multi-process functionality (e.g., fork), or both.

The term library OS also refers to an older generation of research focused on tailor-

ing hardware management heuristics to individual application needs [29, 34, 60, 93, 107], whereas newer library OSes, including Graphene, focus on providing application compatibility across different hosts without dragging along an entire legacy OS. A common thread among all libOSes is moving functionality from the kernel into applications and reducing the TCB size or attack surface. Kaashoek et al. [93] identify multi-processing as a problem for an Exokernel libOS, and implemented some shared OS abstractions. The Exokernel's sharing designs rely on shared memory rather than byte streams, and would not work on recent libOSes, nor will they facilitate dynamically sandboxing two processes.

User Mode Linux [68] (UML) executes a Linux kernel inside a process by replacing architecture-specific code with code that uses Linux host system calls. UML is best described as an alternative approach to paravirtualization [41], and, unlike a library OS, does not deduplicate functionality.

**Distributed coordination APIs.** Distributed operating systems, such as LOCUS [73, 167], Amoeba [61, 124] and Athena [56] required a consistent namespace for process identifiers and other IPC abstractions across physical machines. Like microkernels, these systems generally centralize all management in a single, system-wide service. Rote adoption of a central name service does not meet our goals of security isolation and host independence.

Several aspects of the Graphene host kernel ABI are similar to the Plan 9 design [131], including the unioned view of the host file system and the inter-picoprocess byte stream. Plan 9 demonstrates how to implement this host kernel ABI, whereas Graphene uses a similar ABI to encapsulate multi-process coordination in the libOS.

Barrelfish [43] argues that multi-core scaling is best served by replicating shared kernel abstractions at every core, and using message passing to coordinate updates at each replica, as opposed to using cache coherence to update a shared data structure. Barrelfish is a new OS; in contrast, Cerberus [154] applies similar ideas to coordinate abstractions across multiple Linux VMs running on Xen. In order for a library OS to provide multi-process abstractions, Graphene must solve some similar problems, but innovates by replicating additional classes of coordination abstractions, such as System V IPC, and facilitates dynamic

sandboxing. The focus of this paper is not on multi-core scalability, but on security isolation and compatibility with legacy, multi-process applications. That said, we expect that systems like Barrelfish [43] could leverage our implementation techniques to efficiently construct higher-level OS abstractions, such as System V IPC and signals.

L3 introduced a "clans and chiefs" model of IPC redirection, in which IPC to a non-sibling process was validated by a the parent ("chief") before a message could leave the clan [110]. Although this model was abandoned as cumbersome for general-purpose access control [71], the Graphene sandbox design observes that a stricter variation is a natural fit for security isolation among multi-process applications.

Cerberus focuses on replicating lower-level state, such as process address spaces which Graphene leaves in the host kernel. As a result, the performance characteristics are different. Although this comparison is rough, we replicated their test of ping-ponging 1000 `SIGUSR1` signals and compare the ratio to their reported data, albeit with different hardware and our baseline kernel is newer (3.2 vs 2.6.18). When signals are sent inside of a single guest on Graphene, they are *faster* by 79%, whereas performance drops by a 5.5–18× on Cerberus. When passing signals across coordinating guests both approaches are competitive: Graphene's cross-process signal delivery is 4.6× slower than native, whereas Cerberus ranges from 3.3–11.3× slower, depending on the hardware.

**Migration and security isolation.** Researchers have added checkpoint and migration support to Linux [105] by serializing kernel data structures to a file and reloading them later. This introduces several challenges, including security concerns of loading data structures into the OS kernel from a potentially untrusted source. In contrast, Graphene checkpoint/restore requires little more than a guest memory dump.

OS-based virtualization, such as Linux VServer [152], containers [49], and Solaris Zones [136], implement security isolation by maintaining multiple copies of kernel data structures, such as the process tree, in the host kernel's address space. In order to facilitate sandboxing, Linux has added support for launching single processes with isolated views of namespaces, including process IDs and network interfaces [96]. FreeBSD jails apply a sim-

ilar approach to augment an isolated `chroot` environment with other isolated namespaces, including the network and hostname [158]. Similarly, Zap [127] migrates groups of process, called a Pod, which includes a thin layer virtualizing system resource names. In these approaches, all guests must use the same OS API, and the host kernel still exposes hundreds of system calls to all guests. Library OSes move these data structures into the guest, enabling a range of personalities to run on a single guest and limiting the attack surface of the host.

Shuttle [146] permits selective violations of strict isolation to communicate with host services under OS-based virtualization. For example, collaborating applications may communicate using the Windows Common Object Model (COM); Shuttle develops a model to permit access to the host COM service. Rather than attempting to secure host services, Graphene moves these services out of the host and into collaborating guests.

## 10.2 SGX and Isolated Execution

**Protection against untrusted OSes.** Protecting applications from untrusted OSes predates hardware support. Virtual Ghost [63] 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 [118], MUSHI [176], SeCage [114], InkTag [82], and Sego [102] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koberl et al. [100], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [108] built a 2-way sandbox for x86 by separating the Native Client (NaCl) [173] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [117] to protect the piece of application logic from the untrusted OS. Jang et al. [92] build a secure channel to authenticate the application in the Untrusted area isolated by the ARM TrustZone technology. Song et al. [153] extend each memory unit with an additional tag to enforce fine-grained isolation at machine word granularity in the HDFI system.

Issue 1.2.c: add discussion of Haven, SCONE, Panoply

**Trusted execution hardware.** XOM [109] 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.

Besides SGX, other hardware features have been introduced in recent years to enforce isolation for trusted execution. TrustZone [**?** ] 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++ [50] 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.

AMD is introducing a feature in future chips called SEV (Secure Encrypted Virtualization) [95], 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.

Sanctum [62] 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 [38, 45, 150], SGX has been used in specific applications or to address other security issues. VC3 [142] runs MapReduce jobs in SGX enclaves. Similarly, Brenner et al. [52] run cluster services in ZooKeeper in an enclave, and transparently encrypt data in transit between enclaves.

Ryoan [85] sandboxes a piece of untrusted code in the enclave to process secret data while preventing the loaded code from leaking secret data. Opaque [177] 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 [148], as well as inter-domain routing in Tor [98].

Several improvements to SGX frameworks have been recently developed, which can be integrated with applications on Graphene-SGX. Eleos [126] reduces the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. SGXBOUND [101] is a software technique for bounds-checking with low memory overheads, to fit within limited EPC size. T-SGX [149] 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 [144] 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 [113] uses data-flow analysis at compile-time, to automatically determine the partition boundary in an application.

## 10.3 System API Studies

**API usage study.** Concurrent with our work, Atlidakis et al. [39] conducted a similar study of POSIX. A particular focus of the POSIX study is measuring fragmentation across different POSIX-compliant OSes (Android, OS X, and Ubuntu), as well as identifying points where higher-level frameworks are driving this fragmentation, such as the lack of a ubiquitous abstraction for graphics. Both studies identify long tails of unused or lightly-used functionality in OS APIs. The POSIX study factors in dynamic tracing, which can yield performance insights; our study uses installation metrics, which can yield insights about the impact of incompatibilities end-users. Our paper contributes complimentary insights, such as a metric and incremental path for completeness of an emulation layer, as well as analysis of the importance of less commonly-analyzed APIs, such as pseudo-files

under `/proc`.

A number of previous studies have investigated how other portions of the Operating System interact, often at large scale. Kadav and Swift [94] studied the effective API the Linux kernel exports to device drivers, as well as device driver interaction with Linux—complementary to our study of how applications interact with the kernel or core libraries. Palix et al. study faults in all subsystems of the Linux kernel and identify the most fault-prone subsystems [128]. They find architecture-specific subsystems have highest fault rate, followed by file systems. Harter et al. [80] studied the interaction of a set of Mac OS X applications with the file system APIs—identifying a number of surprising I/O patterns. Our approach is complementary to these studies, with a focus on overall API usage across an entire Linux distribution.

**Application statistics.** A number of previous studies have drawn inferences about user and developer behavior using Debian and Ubuntu package metadata and popularity contest statistics. Debian packages have been analyzed to study the evolution of the software itself [77, 125, 141], to measure the popularity of application programming languages [30], to analyze dependencies between the packages [64], to identify trends in package sizes [32], the number of developers involved in developing and maintaining a package [140], and estimating the cost of development [31]. Jain et al. used popularity contest survey data to prioritize the implementation effort for new system security policies [89]. This study is unique in using this information to infer the relative importance of of system APIs to end users, based on frequency of application installation.

A number of previous projects develop techniques or tools to identify software incompatibilities, with the goal of avoiding subtle errors during integration of software components. The Linux Standard Base (LSB) [66] predicts whether an application can run on a given distribution based on the symbols imported by the application from system libraries. Other researchers have studied application compatibility across different versions of same library, creating rules for library developers to maintain the compatibility across versions [130]. Previous projects have also developed tools to verify backward

compatibility of libraries, based on checking for any changes in library variable type definitions and function signatures [132]. Another variation of compatibility looks at integrating independently-developed components of a larger software project; solutions examine various attributes of the components' source code, such as recursive functions and strong coupling of different classes [151]. In these studies, compatibility is a binary property, reflecting a focus on correctness. Moreover, these studies are focused on the interface between the application and the libraries or distribution ecosystem. In contrast, this paper proposes a metric for relative completeness of a prototype system.

**Static analysis.** Identifying the system call footprint of an application is useful for a number of reasons; our work contributes data from studying trends in system API usage in a large set of application software. The system call footprint of an application can be extracted by static or dynamic analysis. The trade-off is that dynamic analysis is easier to implement quickly, but the results are input-dependent. Binary static analysis, as this paper uses, can be thwarted by obfuscated binaries, which can confuse the disassembler [175]. Static binary analysis has been used to automatically generate application-specific sandboxing policies [106]. Dynamic analysis has been used to compare system call sequences of two applications as an indicator of potential intellectual property theft [169], to identify opportunities to batch system calls [138], to model power consumption on mobile devices [129], and to repackage applications to run on different systems [79]. These projects answer very different questions than ours, but could, in principle, benefit from the resulting data set.

# Chapter 11

# Conclusion

Due to the diversity of system platforms, developers pay varied efforts to port applications from a system to another, in order to gain qualitative benefits. The porting effort for an application can range from recompilation of the binaries to fundamental re-implementation. Essentially, the difficulty of porting is determined by the distinction between the *personalities* of platforms — either they are Windows, UNIX, library OSes, etc. Existing library OSes [44, 45, 134] provide the personalities of monolithic kernels, such as Windows or Linux, within a single picoprocess. The single-process abstractions, such as accessing unshared files or creating in-process threads, can be wrapped inside a library OS; however, multi-process abstractions, on the contrast, require multiple picoprocesses to collaboratively provide a unified OS views.

We design a library OS called Graphene [160], which supports legacy Linux, multi-process applications. In Graphene, the idiosyncratic, Linux multi-process abstractions — including forking, signals, System V IPC, file descritor sharing, exit notification, etc — are coordinated across picoprocesses over simple, pipe-like RPC streams on the host. The RPC-based, distributed implementation of OS abstractions can be isolated by simply sandboxing the RPC streams. The beneficial features of Graphene, including isolation among mutually untrusting applications, migration of system state, and platform independence, are comparable to virtualization, but at a lower resource cost. Especially, with platform independence, Graphene can extend the legacy support for Linux applications onto other platforms, includ-

ing isolated execution platforms like Intel SGX enclaves. A Graphene picoprocess isolated in an enclave can seal the execution of a legacy application, in an environment immune to attacks from host kernels and hardware peripherals.

Besides supporting whole applications, we explore opportunity for porting applications to a more fine-grained, partitioned model using enclaves, where an application can be split into isolated, selectively trusted components. In particular, applications developed in managed languages, such as Java, will encounter obstacles when being ported into enclaves due to limitations of Intel SGX. We propose a framework that automatically split a Java application into cleanly partitioned enclaves, to isolate sensitive execution from untrusted components and hosts.

In practice, developers, including us, struggle to prioritize the implementation of system APIs and abstraction, because what they believe to be more important is inevitably skewed toward their preferred workloads. Alternatively, we suggest a more fractal measurement for estimating how system APIs (e.g., Linux system calls) are used in applications, weighted by the application popularity. The study reveals that all system APIs are not equally important for emulating, and by prioritizing API emulation developers can plan an optimal path to maintain the broadest application support. According to the measurement, by merely adding two important but missing system calls to Graphene, the fraction of applications that can plausibly use the system will grow from 0.42% to 21.1%.

At the high level, the principles for developing OS personalities can be vastly distinct among different specifications, and often entangled with the implementation of security mechanisms and performance optimizations. Similar challenges can be observed in legacy, monolithic kernels. We demonstrate a case of an performance-centric, heavily engineered component, the Linux file system directory cache. The directory cache is designed as an optimization for path lookup, yet it interleaves searching path components with permission checks (e.g., searching prefixes) and file system features (e.g., resolving symbolic links), causing suboptimal latency when the cache is warm (no cache misses) [162]. A fast path to improve the hit latency will decouple searching in the directory cache from other

operations, by caching the results of prefix checking, symbolic links, etc, in the kernel data structures. In conclusion, this thesis seeks systematic and generalizable solutions, for mitigating the limitations on fulfilling legacy application requirements in innovative system designs (e.g., library OSes, enclaves, file system fast paths).

# Appendix A

# Formal Definitions

## A.1 API Importance

A system installation (inst) is a set of packages installed ($\{\texttt{pkg}_1, \texttt{pkg}_2, ..., \texttt{pkg}_k \in \texttt{Pkg}_{\texttt{all}}\}$). For each package (pkg) that can be installed by the installer, we analyze every executable included in the package ($\texttt{pkg} = \{\texttt{exe}_1, \texttt{exe}_2, ..., \texttt{exe}_j\}$), and generate the API footprint of the package as:

$$\texttt{Footprint}_{\texttt{pkg}} = \{\texttt{api} \in \texttt{API}_{\texttt{all}} \mid \exists \texttt{exe} \in \texttt{pkg}, \texttt{exe} \text{ has usage of } \texttt{api} \quad \}$$

The API importance is calculated as the probability that any installation includes at least one package that uses an API; i.e., the API belongs to the footprint of at least one package. Using Ubuntu/Debian Linux's package installation statistics, one can calculate the probability that a specific package is installed as:

$$Pr\{\texttt{pkg} \in \texttt{Inst}\} = \frac{\text{\# of installations including pkg}}{\text{total \# of installations}}$$

Assuming the packages that use an API are $\texttt{Dependents}_{\texttt{api}} = \{\texttt{pkg} | \texttt{api} \in \texttt{Footprint}_{\texttt{pkg}}\}$. API importance is the probability that at least one package from $\texttt{Dependents}_{\texttt{api}}$ is installed

on a random installation, which is calculated as follows:

$$\begin{aligned}
&\texttt{Importance}(\texttt{api}) = Pr\{\texttt{Dependent}_{\texttt{api}} \bigcap \texttt{Inst} \neq \emptyset\} \\
&= 1 - Pr\{\forall \texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}, \texttt{pkg} \notin \texttt{Inst}\} \\
&= 1 - \prod_{\texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}} Pr\{\texttt{pkg} \notin \texttt{Inst}\} \\
&= 1 - \prod_{\texttt{pkg} \in \texttt{Dependent}_{\texttt{api}}} (1 - \frac{\text{\# of installations including pkg}}{\text{total \# of installations}})
\end{aligned}$$

## A.2 Weighted Completeness

Weighted completeness is used to evaluate the relative compatibility on a system that supports a set of APIs ($\texttt{API}_{\texttt{Supported}}$). For a package on the system, we define it as supported if if every API that the package uses is in the supported API set. In other words, a package is supported if it is a member of the following set:

$$\texttt{Pkg}_{\texttt{Supported}} = \{\texttt{pkg} | \texttt{Footprint}_{\texttt{pkg}} \subseteq \texttt{API}_{\texttt{Supported}}\}$$

Using weighted completeness, one can estimate the fraction of packages in an installation that end-users can expect a target system to support. For any installation that is an arbitrary subset of available packages ($\texttt{Inst} = \{\texttt{pkg}_1, \texttt{pkg}_2, ..., \texttt{pkg}_k\} \subseteq \texttt{Pkg}_{\texttt{all}}$), weighted completeness is the expected value of the fraction in any installation (Inst) that overlaps with the supported packages ($\texttt{Pkg}_{\texttt{Supported}}$):

$$\texttt{WeightedCompleteness}(\texttt{API}_{\texttt{Supported}}) = E(\frac{|\texttt{Pkg}_{\texttt{Supported}} \bigcap \texttt{Inst}|}{|\texttt{Inst}|})$$

where $E(\texttt{X})$ is the expected value of X.

Because we do not know which packages are installed together, except in the presence of explicit dependencies, we assume package installation events are independent.

Thus, the approximated value of weighted completeness is:

$$\frac{E(|\texttt{Pkg}_{\texttt{Supported}} \bigcap \texttt{Inst}|)}{E(|\texttt{Inst}|)} \sim \frac{\sum_{\texttt{pkg} \in \texttt{Pkg}_{\texttt{Supported}}} (\frac{\text{\# of installations including pkg}}{\text{total \# of installations}})}{\sum_{\texttt{pkg} \in \texttt{Pkg}_{\texttt{all}}} (\frac{\text{\# of installations including pkg}}{\text{total \# of installations}})}$$

## 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] Byte unixbench. http://code.google.com/p/byte-unixbench/.

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

[5] *diet libc*: A libc optimized for small size. https://www.fefe.de/dietlibc.

[6] The embedded GNU Libc. http://www.eglibc.org.

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

[8] The GNU C library. http://www.gnu.org/software/libc.

[9] HotSpot Java Virtual Machine. http://openjdk.java.net/groups/hotspot/.

[10] IBM J9 Java Virtual Machine. https://www.ibm.com/support/knowledgecenter/en/SSYKE2_7.0.0/com.ibm.java.lnx.70.doc/user/java_jvm.html.

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

[12] Lighttpd. https://www.lighttpd.net/.

[13] Linux man pages – section 2: system calls. available at https://linux.die.net/man/2/.

[14] *musl* libc. http://www.musl-libc.org.

[15] NGINX. https://www.nginx.com/.

[16] OSv. available at http://osv.io.

[17] Perl. https://www.perl.org/.

[18] Python. https://www.python.org/.

[19] QEMU. https://www.qemu.org/.

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

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

[22] *uClibc*. https://www.uclibc.org.

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

[24] Linux kernel security vulnerabilities. http://www.cvedetails.com/, 2013.

[25] Graphene-sgx: A practical library os for unmodified applications on sgx. In *Proceedings of the USENIX Annual Technical Conference*, pages 645–658, Santa Clara, CA, 2017. USENIX Association.

[26] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. MACH: a new kernel foundation for UNIX development. Technical report, Carnegie Mellon University, 1986.

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

[28] B. Alpern, C. R. Attanasio, J. J. Barton, M. G. Burke, P. Cheng, J.-D. Choi, A. Cocchi, S. J. Fink, D. Grove, M. Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[29] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a JVM in a virtualized execution environment. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 44–54, 2007.

[30] J. J. Amor, J. M. Gonzalez-Barahona, G. Robles, and I. Herraiz. Measuring Libre software using Debian 3.1 (sarge) as a case study: Preliminary results. *UPGRADE - The European Journal for the Informatics Professional*, (3), June 2005.

[31] J. J. Amor, G. Robles, and J. M. González-Barahona. Measuring Woody: The size of Debian 3.0. *CoRR*, 2005.

[32] J. J. Amor, G. Robles, J. M. González-Barahona, and I. Herraiz. From pigs to stripes: A travel through Debian. In *Proceedings of the DebConf5 (Debian Annual Developers Meeting)*, July 2005.

[33] 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.

[34] T. Anderson. The case for application-specific operating systems. In *Workshop on Workstation Operating Systems*, 1992.

[35] AppArmor. `http://wiki.apparmor.net/`.

[36] J. Appavoo, M. A. Auslander, D. Da Silva, D. Edelsohn, O. Krieger, M. Ostrowski, B. S. Rosenburg, R. W. Wisniewski, and J. Xenidis. Providing a Linux API on the

scalable K42 kernel. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, 2003.

[37] A. Archangeli, I. Eidus, and C. Wright. Increasing memory density by using KSM. In *Linux Symposium*, pages 19–28, 2009.

[38] 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.

[39] V. Atlidakis, J. Andrus, R. Geambasu, D. Mitropoulos, and J. Nieh. POSIX abstractions in modern operating systems: The old, the new, and the missing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2016.

[40] A. Aviram, S.-C. Weng, S. Hu, and B. Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[41] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 164–177, New York, NY, USA, 2003. ACM.

[42] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An operating environment for large-scale multiprocessor applications. *Journal of IEEE SOFTWARE*, 2(4):65–67, jul 1985.

[43] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 29–44, 2009.

[44] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[45] 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.

[46] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 335–348, 2012.

[47] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference*, pages 41–46, 2005.

[48] T. Bergan, N. Hunt, L. Ceze, and S. D. Gribble. Deterministic process groups in dos. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

[49] S. Bhattiprolu, E. W. Biederman, S. Hallyn, and D. Lezcano. Virtual servers and checkpoint/restart in mainstream Linux. *ACM Operating Systems Review*, 42:104–113, July 2008.

[50] R. Boivie and P. Williams. SecureBlue++: CPU support for secure executables. Technical report, IBM Research, 2013.

[51] N. Borisov, R. Johnson, N. Sastry, and D. Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the USENIX Security Symposium*, pages 303–314, 2005.

[52] 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.

[53] T. W. Burger. Intel virtualization technology for directed I/O (VT-d): Enhancing Intel platforms for efficient virtualization of I/O devices. `http://software.intel.com/en-us/articles/intel-virtualization-technology-for-directed-io-vt-d-enhancing-intel-platforms-for-efficient-virtualization-of-io-devices/`, February 2009.

[54] X. Cai, Y. Gui, and R. Johnson. Exploiting unix file-system races via algorithmic complexity attacks. *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 27–41, 2009.

[55] C. Cascaval, J. G. Castanos, L. Ceze, M. Denneau, M. Gupta, D. Lieber, J. E. Moreira, K. Strauss, and H. S. Warren. Evaluation of a multithreaded architecture for cellular computing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2002.

[56] G. A. Champine, D. E. Geer, Jr., and W. N. Ruh. Project Athena as a Distributed Computer System. *IEEE Computer*, 23(9):40–51, September 1990.

[57] 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.

[58] H. Chen, D. Wagner, and D. Dean. Setuid demystified. In *Proceedings of the USENIX Security Symposium*, 2002.

[59] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 120–133, 1993.

[60] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1994.

[61] D. R. Cheriton and T. P. Mann. Decentralizing a global naming service for improved performance and fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 7(2):147–183, May 1989.

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

[63] 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.

[64] O. F. de Sousa, M. A. de Menezes, and T. J. P. Penna. Analysis of the package dependency on Debian GNU/Linux. *Journal of Computational Interdisciplinary Sciences*, pages 127–133, March 2009.

[65] Debian popularity contest. `http://popcon.debian.org`.

[66] S. Denis. Linux distributions and applications analysis during linux standard base development. *Proceedings of the Spring/Summer Young Researchers. Colloquium on Software Engineering*, 2, 2008.

[67] D. Dhamdhere. *Operating Systems: A Concept-based Approach*. McGraw-Hill, 2007.

[68] J. Dike. *User Mode Linux*. Prentice Hall, 2006.

[69] R. Divacky. Linux emulation in FreeBSD. `http://www.freebsd.org/doc/en/articles/linux-emulation`.

[70] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2008.

[71] K. Elphinstone and G. Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2013.

[72] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 251–266, 1995.

[73] B. D. Fleisch. Distributed System V IPC in LOCUS: a design and implementation retrospective. *SIGCOMM Comput. Commun. Rev.*, 16(3):386–396, August 1986.

[74] L. foundation. Tool interface standard (tis) portable formats specification, version 1.2—executable and linking format (elf) specification. Technical report, May 1995.

[75] H. Franke, R. Russel, and M. Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.

[76] Free Software Foundation. GNU Hurd. `http://www.gnu.org/software/hurd/hurd.html`.

[77] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 2009.

[78] M. Gschwind. The cell broadband engine: Exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, June 2007.

[79] P. J. Guo and D. Engler. CDE: Using system call interposition to automatically create portable software packages. In *Proceedings of the USENIX Annual Technical Conference*, 2011.

[80] T. Harter, C. Dragga, M. Vaughn, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. A file is not a file: Understanding the i/o behavior of apple desktop applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 71–83, 2011.

[81] H. Härtig, M. Hohmuth, J. Liedtke, J. Wolter, and S. Schönberg. The performance of $\mu$-kernel-based systems. *SIGOPS Operating System Review*, 1997.

[82] 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.

[83] J. Howell, B. Parno, and J. R. Douceur. How to run POSIX apps in a minimal picoprocess. In *Proceedings of the USENIX Annual Technical Conference*, pages 321–332, 2013.

[84] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *SIGOPS Oper. Syst. Rev.*, 41(2), 2007.

[85] 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.

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

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

[88] iptables man page. `http://linux.die.net/man/8/iptables`.

[89] B. Jain, C.-C. Tsai, J. John, and D. E. Porter. Practical techniques to obviate setuid-to-root binaries. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

[90] 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.

[91] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2012.

[92] 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.

[93] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 52–65, 1997.

[94] A. Kadav and M. M. Swift. Understanding modern device drivers. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 87–98, 2012.

[95] 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`.

[96] M. Kerrisk. User namespaces progress. *Linux Weekly News*, 2012.

[97] 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.

[98] 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.

[99] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[100] 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.

[101] 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)*, 2017.

[102] 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.*

[103] 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.

[104] The L4 microkernel family. available at `http://www.l4hq.org/`.

[105] O. Laaden and S. E. Hallyn. Linux-CR: Transparent application checkpoint-restart in Linux. In *Linux Symposium*, 2010.

[106] L. Lam and T.-c. Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In E. Jonsson, A. Valdes, and M. Almgren, editors, *Recent Advances*

*in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2004.

[107] I. Leslie, D. Mcauley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, pages 1280–1297, 1996.

[108] 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.

[109] 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.

[110] J. Liedtke. Clans & chiefs. In *Architektur von Rechensystemen, 12. GI/ITG-Fachtagung*, pages 294–305, 1992.

[111] J. Liedtke. Improving IPC by kernel design. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 175–188, 1993.

[112] J. Liedtke. On micro-kernel construction. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 237–250, 1995.

[113] 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 *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2017.

[114] 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.

[115] R. Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010.

[116] 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.

[117] 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.

[118] 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.

[119] 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.

[120] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013.

[121] P. E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy Update Techniques in Operating System Kernels*. PhD thesis, 2004.

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

[123] J. Mickens and M. Dhawan. Atlantis: robust, extensible execution environments for web applications. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 217–231, 2011.

[124] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[125] R. Nguyen and R. Holt. Life and death of software packages: An evolutionary study of debian. In *Proceedings of the 2012 Conference of the Center for Advanced Studies on Collaborative Research*, CASCON '12, pages 192–204, 2012.

[126] 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.

[127] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 361–376, 2002.

[128] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller. Faults in Linux: Ten years later. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 305–318, 2011.

[129] A. Pathak, Y. C. Hu, M. Zhang, P. Bahl, and Y.-M. Wang. Fine-grained power modeling for smartphones using system call tracing. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, pages 153–168. ACM, 2011.

[130] S. Pavel and S. Denis. Binary compatibility of shared libraries implemented in C++ on GNU/Linux systems. *Proceedings of the Spring/Summer Young Researchers. Colloquium on Software Engineering*, 3, 2009.

[131] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[132] A. Ponomarenko and V. Rubanov. Automatic backward compatibility analysis of software component binary interfaces. In *IEEE International Conference on Computer Science and Automation Engineering (CSAE)*, volume 3, pages 167–173, June 2011.

[133] G. J. Popek and R. P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, July 1974.

[134] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 291–304, 2011.

[135] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009.

[136] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *Proceedings of the Large Installation System Administration Conference (LISA)*, pages 241–254, 2004.

[137] 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.

[138] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. System call clustering: A profile directed optimization technique. Technical report, The University of Arizona, May 2003.

[139] D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communication ACM*, July 1974.

[140] G. Robles and J. M. González-Barahona. From toy story to toy history: A deep analysis of Debian GNU/Linux, 2003.

[141] G. Robles, J. M. Gonzalez-Barahona, M. Michlmayr, and J. J. Amor. Mining large software compilations over time: Another perspective of software evolution. In *Proceedings of the International Workshop on Mining Software Repositories*, MSR, pages 3–9, 2006.

[142] 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.

[143] SECure COMPuting with filters (seccomp). `https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt`. Accessed on 3/12/2016.

[144] 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.

[145] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 552–561, Oct 2007.

[146] Z. Shan, X. Wang, T.-c. Chiueh, and X. Meng. Facilitating inter-application interactions for os-level virtualization. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 75–86, 2012.

[147] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: A fast capability system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 1999.

[148] 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.

[149] 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.

[150] 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.

[151] H. Singh and A. Kaur. Component compatibility in component based development. *International Journal of Computer Science and Mobile Computing*, 3:535–541, 06 2014.

[152] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2007.

[153] 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.

[154] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[155] J. Spolsky. How microsoft lost the API war. June 2004.

[156] SQL Server Team. SQL Server on Linux: How? Introduction. `https://blogs.technet.microsoft.com/dataplatforminsider/2016/12/16/sql-server-on-linux-how-introduction/`, December 2016.

[157] J. M. Stevenson and D. P. Julin. Mach-US: UNIX on generic OS object servers. In *USENIX Technical Conference*, 1995.

[158] M. Stokely and C. Lee. The FreeBSD handbook, 3rd edition, vol 1: Users's guide, 2003.

[159] D. Tsafrir, T. Hertz, D. Wagner, and D. D. Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.

[160] 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.

[161] 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.

[162] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter. How to Get More Value from your File System Directory Cache. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2015.

[163] Ubuntu popularity contest. `http://popcon.ubuntu.com`.

[164] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kagi, F. H. Leung, and L. Smith. Intel virtualization technology. *Computer*, May 2005.

[165] `wait4` man page. `http://linux.die.net/man/2/wait4`.

[166] C. A. Waldspurger. Memory resource management in vmware esx server. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[167] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 49–70, 1983.

[168] H. J. Wang, C. Grier, A. Moshchuk, S. T. King, P. Choudhury, and H. Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*, pages 417–432, 2009.

[169] X. Wang, Y.-C. Jhi, S. Zhu, and P. Liu. Detecting software theft via system call based birthmarks. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 149–158, 2009.

[170] J. Wei and C. Pu. TOCTTOU vulnerabilities in UNIX-style file systems: An anatomical study. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, 2005.

[171] 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.

[172] 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)*, 2009.

[173] 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.

[174] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 19–19, 2006.

[175] M. Zhang and R. Sekar. Control flow integrity for cots binaries. In *Proceedings of the USENIX Security Symposium*, pages 337–352, 2013.

[176] 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.

[177] 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.