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

Compatibility challenges occur on a disruptive hardware such as Intel SGX [128] or between distinctive system APIs such as Linux system calls and Windows API. Running an unmodified application, without recompilation or porting, requires a system API to be compatible across OS versions or across host OSes.

This thesis proposes using a library OS as a compatibility layer for running unmodified applications on host OSes and hardware

Dedication Page

This page is optional.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Operating systems simplify programming an application utilizing different hardware. A UNIX-style OS [147] encapsulates hardware resources using a system interface such as a system call able. Without a system interface, developers will have to program against hardware interfaces defined by manufacturers. Programming against bare hardware creates applications that are restricted to specific hardware. Operating systems allow application developers to program against a consistent, hardware-independent system interface, so that the applications can be portable across hardware configurations.

An application developed upon a system interface subjects to a different property as compatibility. **Compatibility** of an OS can be defined as the ability to reproduce a system interface which satisfies the hard-coded requirements of an application. Since a majority of applications are compiled into native code, usage of system interfaces is scattered around application binaries and does not easily change unless modifying source code. To accommodate application binaries, OS developers maintain a common goal as keeping system interfaces compatible across OS versions. Compatibility is also a goal for less widely-used OSes; for instance, FreeBSD emulates Linux system calls to reuse the more well-adopted Linux applications. For generality, OS developers tends to preserve every old system interfaces to maintain compatibility for exiting application binaries.

However, the trend of hardware development challenges the goal of maintaining OS compatibility. The majority of new hardware follow semantics of predecessors and requires no changes to existing system interfaces. However, more cutting-edge hardware tend to leak out of the typical abstractions encapsulated by OSes; one example is SGX (software guard extensions) [128] on recent Intel CPUs. SGX protects an application with integrity and confidentiality, without trusting

other system components such as OSes, hypervisors, and system software. Although an SGX application may still utilize system interfaces for OS functionality such as file systems and networking, the application does not assume the OS to be reliable. Therefore, SGX raises several compatibility issues to existing system interfaces, including the challenges of checking system interface results on an untrusted OS. Other examples can be found among research-type architectures, such as an asymmetric multi-processing architectures without inter-connected memory [57, 84], which challenges inter-process coordination. As more disruptive hardware may emerge in the future, OS developers should gradually promote new system interfaces which encapsulate latest hardware semantics; simultaneously, existing applications require timely solutions to resolve urgent compatibility issues on specific hardware.

Empirically, compatibility has caused struggles in OS development, especially when API changes are demanded. For instance, Linux and similar OSes introduce system calls such as `openat()` as a version of `open()` without TOCTOU (time-to-check-to-time-to-use) vulnerabilities. Unfortunately, to full replace the original `open()`, developers needs to modify every applications, otherwise Linux can never deprecate the unsafe version. At a larger scale, an early version of Windows Vista introduces a brand-new user interface API and file system, but ends up losing popularity due to compatibility-related complaints [164]. Because users prefer to use an application or a system that hasn't broke, earlier versions of OSes, such as Windows XP, remain popular even after end-of-service.

There are practical reasons for an OS to maintain compatibility for unmodified applications. The development of a commercial application requires a thorough process of testing and code inspection to ensure correctness and robustness. Modifying an application for new system interfaces can be a risk to stability. Moreover, third parties cannot port a proprietary software to new system interfaces even if they are motivated to do so. These dilemmas call for a solution to mitigate the compatibility issues for unmodified applications.

This thesis proposes building a compatibility layer which translates a legacy system interface to other alternative host interfaces, as a promising solution for supporting unmodified applications. OS developers are free to redesign system interfaces for adopting new hardware or addressing system challenges. A compatibility layer between the application and the OS can bridge the gap between interfaces. Take SGX for example; the compatibility challenges on SGX include

secure dynamic loading, redirecting system calls to host OSes, and inject security checks against untrusted system call results. The goal is to reduce the effort of porting a compatibility layer to any host platforms. A Linux compatibility layer potentially contains hundreds to thousands of system calls, with plenty of control options and corner cases [13]. This thesis presents a solution to building a rich-feature compatibility layer, in which majority of code does not have to be rewritten when porting to a new host interface.

Building such a compatibility layer can benefit from virtualizing a part of an OS, or more specifically, API components (e.g., system call table), to user space. Virtualization can preserve these OS components, using an intermediate interface exported by the host OS. For instance, a VM (virtual machine) carries an unmodified OS kernel as a full-stack compatibility layer. The intermediate interface for a VM is a virtual hardware interface, facilitated by hardware virtualization [143] such as Intel VT (virtualization technology) [175]. A VM can provide full compatibility by reusing an existing OS implementation.

An alternative to a VM is a **library OS** [31, 77, 125, 144], a user-space OS library loaded inside an application's address space. The purpose of a library OS is to reproduce the features and API of a system interface on top of a library, upon a generic host interface. Defining a host interface is for the simplicity of porting. The definition process is equivalent to finding a "pinch point" inside an OS, to partition out the high-level, host-independent OS components. A host interface must be relatively easy to develop, so that porting the library OS requires less effort than redeveloping a compatibility layer.

This thesis presents **Graphene**, a library OS for running unmodified applications upon innovative hardware and alternative system interfaces. Graphene reproduce the rich Linux system calls, for reusing a wide range of commercial software to benefit highly-customizable server and cloud environments. A host ABI (application binary interface) facilitates the host abstractions for the library OS, and is easily ported to different host system interfaces and hardware platforms. This thesis demonstrates the simplicity by porting the host ABI to two representative host examples, a Linux kernel and an isolated environment with SGX. Other ongoing ports include alternative kernels such as Windows, OSX, and FreeBSD, and research OSes such as L4 microkernels [112] and Barrelfish [45].

The library OS approach strikes a better balance between simplicity of porting and sufficiency of compatible OS functionality. A study of Linux system interface [173] show that system calls are not equally important to applications. Applications also subject to different popularity among users, as shown in installation statistics [174]. A portion of Linux system calls are strictly for administrative purposes, such as configuring Ethernet cards and rebooting the machines, and are exclusively used by system software such as `ifconfig` and `reboot`. As a result, a library OS can selectively implement system calls based on importance for applications with porting value.

Graphene primarily targets three types of applications: (1) server and cloud applications, such as Apache and Lighttpd; (2) command-line programs, such as Bash and GCC; (3) language runtimes, such as R, Python, and OpenJDK. Graphene implements sufficient system calls (145 out of 318 Linux system calls), upon a narrowed host ABI containing 39 functions.

Graphene inherits a part of the host ABI from Drawbridge [144], a previous library OS developed for reusing Windows applications. Drawbridge uses the library OS as a lightweight VM to run Windows desktop applications such as in a guest environment. Running a library OS as a partitioned OS component also requires less memory resource than a full VM, and thus improve the density of packing guests in a physical machine. Bascule [46] later adopts the design to implement single-process, Linux system calls. Finally, Haven [47] ports Drawbridge to SGX, to shield Windows application from untrusted host OSes. Graphene presents solution for running unmodified applications upon a variety of host OSes and hardware, and defines a host interface to be both easy to port and sufficient for developing a rich-feature library OS.

This thesis has several contributions over previous work. First, this thesis defines a host ABI which is easy to port on new host platform, by enumerating the porting effort, including translating host system interfaces and enforcing security checks. Second, this thesis demonstrates the development of a library OS using the host ABI, and presents emulation strategies for complex Linux features such as multi-process abstractions, with reasonable overheads and memory footprints. Third, this thesis presents a quantitative method of evaluating compatibility to prioritize system interface emulation in a library OS or a research prototype.

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.

1.1.1 Unmodified applications on SGX

SGX [128], or Software Guard Extensions, are a set of new instructions on Intel CPUs. The purpose of SGX is to protect application code from compromised OSes, hypervisors, and system software, with both integrity and confidentiality protection. The abstraction of SGX is an **enclaves**, an isolation environment where an application can securely utilize the CPU and memory of an untrusted host. Application code and data inside an enclave are both signed and encrypted inside the DRAM, and will be authenticated and decrypted when bringing into the last-level cache at cache miss. SGX also offers remote attestation of the integrity of an enclave and the CPU. SGX provides opportunities of delegating security-sensitive operations to untrusted hosts such as a public cloud or a client machine, without compromising on security.

Despite the benefits, the common expectation for utilizing SGX is that developers partition a piece of application code to run inside an enclave. The restriction is for both security and simplicity reasons. The initial code in an enclave must be statically compiled, with a security measurement to verify inside the CPU. Developers must also removing system calls and certain instructions such as `cpuid` and `rdtsc` from an application. SGX forbids these instructions for blocking dangerous behaviors that may jeopardize application security. For instance, system calls serviced by an untrusted OS may returns malicious results to applications. Also, SGX forbids `cpuid` and `rdtsc` because these instructions are easily intercepted or spoofed by an OS or a hypervisor. These restrictions cost developers porting effort to modify existing applications for SGX.

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 combine a modified C library with applications, to redirect system calls to the untrusted OS [40, 159]. The problem, however, is in checking the results of system

(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).

interfaces, because the OS is not trusted. Previous work [59] shows that checking untrusted system interface results can be subtle, because the existing system interfaces are not designed for an untrusted or compromised OS.

In summary, the existing porting models of SGX requires modifying application binaries, and injecting security checks for all the OS features used by an application. 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 a Python runtime, upon a trusted library OS.

1.1.2 Emulating multi-process abstractions

One characteristic of a UNIX program is the utilization of multi-process abstractions, such as `fork()`, `exec()`, and inter-process communication, to program self-contained sessions or commands. Especially, `fork()` is an unique feature of UNIX-style OSes, such as Linux and BSD, which clones a process with address space isolation between the parent and child. 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.

For programmability, Linux and similar OSes export several inter-process communication (IPC) mechanisms, each with unique use cases and semantics. The Linux IPC essentially combines the UNIX System V features, such as message queues and semaphores, and POSIX abstractions, such as signaling and namespaces, to present a wide range of options for programming. 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.

Monolithic OSes such as Linux generally implements IPC mechanisms as shared states in a coherent kernel space. Sharing kernel states, however, causes process sandboxing to be vulnerable in the kernel space. An isolated OS design tends to avoid sharing privileged OS states with other applications.

Moreover, not all architectures share the same assumption of having an inter-connected, coherent memory. Several recent architectures lack memory coherence in exchange of simplicity of implementation [57, 84]. Barrelfish [45] 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

Besides compatibility, Graphene also reduces the complexity of enforcing security isolation on applications, under different threat models. For instance, on a Linux host, the security implication of using Graphene is to isolate mutually-untrusting applications, similar to running each of these applications inside a OS sandbox. A SGX host, on the other hand, enforces a different threat model, where an enclave untrusted any OS components and applications running outside of the enclave. With two opposite threat models, Graphene shows how to simplify security isolation by separating API implementation from enforcing security policies.

Similar as the complexity of emulation, security isolation on a monolithic OS is also delicate and prone to mistake. the host ABI f Graphene simplifies the isolation of OS abstractions down to three host abstractions that are sharable among picoprocesses: files, network connecions,

and RPC streams. For each of these sharable host abstraction, Graphene enforces isolation policies using semantics commonly known and used by security experts; for instance, to specific file access rules, users provide a list of permitted files, similar to a profile for the AppArmor kernel module. For network connections, users specify firewall-like network rules, to permit an application to bind to a local network address, or to connect to a remote network address. Finally, for RPC streams, Graphene simply blocks any RPC streams which cross sandbox boundary. This thesis show that, by enforcing isolation rules on three host abstractions, the PAL ABI isolates the whole system call table inside each picoprocess.

For SGX, Graphene addresses a specific set of security isolation challenges. In addition to isolating mutually-untrusting applications, Graphene also isolate an SGX application from untrusted operating systems, hypervisor, or system software. Existing system APIs, such as system calls, expose a wide attack surface to an untrusted host, where an adversary can manipulate system API results to explore attack vectors [59]. Graphene simplifies the protection against random system API results which may or may not be malicious, by redefining a fixed-width enclave interface with security checks in mind. This thesis also enumerates the security checks for each enclave call, to verify the completeness of protection against untrusted host components.

1.3 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 [171], which emphasizes on security isolation, between mutually-untrusted applications. A later publication [27] 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. [173] presents the compatibility metrics for compatibility, with a study of the Linux API usage among Ubuntu users and applications.

1.4 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. The first part is a host ABI (application binary interface), called the PAL ABI, which encapsulates the abstractions needed from a host OS. On each host OS, a PAL (platform adaption layer) instantiates the PAL ABI, by exporting 39 PAL calls to its guests. The second part is a library OS which emulates a substantial subset of Linux system calls, in a dynamic library called `libLinux`. This chapter first introduces the PAL ABI and the principles behind its definition. Then, the chapter discusses the architecture of `libLinux`, possible approaches to approaches to implementing Linux functionality, and potential trade-offs during the development of a library OS.

2.1 The Host ABI

The Graphene architecture starts with defining a simple host ABI with OS abstractions essential to application execution, namely the PAL ABI. the PAL ABI separates low-level abstractions, such as hardware management and drivers, from implementing the API (application programming interfaces) which applications depend on. Exporting the PAL ABI on each host allows OS components such as the system call table and namespaces to move into the guest space as a library OS. As a result, OS developers can focus on porting the PAL ABI to new host OSes and hardware, instead of maintaining the backward compatibility of an API.

To define the PAL ABI, this thesis defines each **host** of the PAL ABI as an OS or a hypervisor that provides sufficient OS features to run a standalone application or virtual machine. An

example of a host is a monolithic kernel such as Linux, BSD, or Windows, which has defined a massive system API for programmability. Other example, such as an Intel SGX (Software Guard Extensions) enclaves [128], has more restricted OS functionality; for instance, SGX virtualizes the exception handlers inside each enclave, but provides no other OS functionality. The only way to obtain common OS functionality such as file systems or networking, besides introducing a library OS, is to request from a host OS outside the enclave through RPC (remote procedure call) [40, 59]. Due to compatibility challenges on SGX, this thesis use SGX as a representative example of a host with unusual assumptions (e.g., threat models) and restrictions than a monolithic kernel.

The PAL ABI also shares several characteristics with a virtual hardware interface which runs a virtual machine. A generic but backward-compatible virtual hardware interface, allows an unmodified OS kernel to run inside a virtual machine. For instance, a virtual hardware interface usually includes a virtual NIC (network interface controller), such as the virtualized E1000 interfaces available in VMware workstation or QEMU. The key difference between a virtual hardware interface and the PAL ABI is that the PAL ABI does not target reusing a whole, unmodified OS kernel as a guest. Instead, the PAL ABI focuses on more high-level abstractions such as files and network sockets to ensure portability on most host OSes. The concept of defining the PAL ABI with a customized guest OS (i.e., a library OS) running atop the PAL ABI is similar to para-virtualization. A para-virtualized VM defines hypercalls as interfaces between a guest OS and a hypervisor. Furthermore, the PAL ABI avoids duplication of OS components such as scheduler, page fault handler, file systems, and network stacks between the host and library OS. Conceptually, one can put a library OS and a VM on a spectrum: at one extreme, a virtual machine reuses a whole OS on a virtual hardware interface; at the other extreme, a Graphene library OS instance runs on the PAL ABI.

The following paragraphs discuss the key design principles of the PAL ABI, including simplicity, sufficiency for library OS development, and statelessness for migration.

Simplicity of porting. To avoid burdening OS developers, the PAL ABI must be simple to port on a host OS or hardware. The PAL ABI reduces porting efforts based on two strategies: first, the PAL ABI significantly reduces both the size and complexity of features that OS developers must implement on a host. Effectively, the PAL ABI avoids including similar APIs and infrequent corner

cases to simplify the porting effort. Second, the definition of the PAL ABI imitates the common system API the exists in most OSes, so that most calls can be mapped to existing system calls or system library functions on each host. The assumption that such a strategy is possible is based on the observation that similar OS functions, especially UNIX-style APIs, tend to commonly exist in most OSes. For instance, system calls like `read()` and `write()` exist on Linux, BSD, and POSIX API; Windows also has `ReadFile()` and `WriteFile()` with similar functionality and semantics. As the rest of this thesis proves, porting the PAL ABI tends to be straightforward on most OSes.

Sufficiency for library OS development. The PAL ABI defines the host abstraction available for a library OS to access host hardware abstractions. In order to develop a library OS with compatibility against a wide range of applications, the PAL ABI also has to sufficiently include a set of common OS abstractions which cannot be emulated in the guest space. For most OSes, common OS abstractions includes process creation, memory management, and I/O (typically, files and network connection) [71]. For each type of these abstractions, a monolithic kernel such as Linux tends to export multiple interfaces with similar functionality but different semantics. For instance, Linux and similar OSes include two system calls, `mmap()` and `brk()`, for memory allocation in a process. While `mmap()` allocates larger memory regions with page granularity, `brk()` simply grows a single, continuous heap space. A common practice in application development is that an application may design logics to switch among similar system calls, in case one of them is unavailable. This thesis show that, by adopting the semantics from one of these similar APIs or abstractions, a single call is sufficient to emulating other APIs or abstractions. For instance, the PAL ABI includes `VirtMemAlloc()` as a similar feature as `mmap()`, which is sufficient to emulating both `mmap()` and `brk()`.

The definition of the PAL ABI in Graphene is based on Drawbridge, a library OS for running single-process Windows applications in a lightweight, guest environment. The host ABI of Drawbridge including 36 functions, and is ported to multiple hosts, including Windows, Linux, Barrelfish, and SGX [46, 47, 144, 165]. Although running Windows and Linux applications may face a different set of challenges, the nature of their APIs is mostly similar, with a few exceptions. During the development of Graphene, developers found the occasions in which the host ABI of Drawbridge is not sufficient to address Linux-specific challenges, and decide to extend the PAL

ABI The Linux-specific extensions will be further discussed in Section 2.1.1 and Chapter 3.

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 that does not allow memory sharing (e.g., SGX). For a virtual machine, checkpointing and migration is based on snapshotting the guest states above a stateless virtual hardware interface. The PAL ABI shares the same property of statelessness. The statelessness of the PAL ABI guarantees any states in the hosts are temporary to the applications and library OS, and can be reproduced without checkpointing host states.

2.1.1 PAL call definitions

Table 2.1 enumerates 39 PAL calls defined in the PAL ABI: 25 PAL calls are inherited from the Drawbridge host ABI, including functions to managing I/O (e.g., `StreamOpen()`), memory allocation (e.g., `VirtMemAlloc()`), scheduling (e.g., `ThreadCreate()`), and several miscellaneous functions (e.g., `SystemTimeQuery()`). 14 PAL calls are added by Graphene, in order to implement Linux-specific features. For example, unlike Windows or OS X, 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 PAL ABI includes 5 PAL calls 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 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 improve performance, the PAL ABI defines an optional, bulk IPC feature 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 memory in the host.

Abstraction	Function Names	Description

Streams	`StreamOpen`
	`StreamRead`
	`StreamWrite`
	`StreamMap`
	`StreamFlush`
	`StreamSetLength`
	`ServerWaitforClient`
	`StreamAttrQuery`
	`StreamAttrQuerybyHandle`
	`StreamAttrSetbyHandle` †
		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`
	`VirtMemFree`
	`VirtMemProtect`
		Allocation, deallocation, and protection of a chunk of virtual memory.

Threads & scheduling	`ThreadCreate`
	`ThreadExit`
	`ThreadDelayExecution`
	`ThreadYieldExecution`
	`ThreadInterrupt` †
	`MutexCreate` †
	`MutexUnlock` †
	`SynchronizationEventCreate`
	`NotificationEventCreate`
	`EventSet`
	`EventClear`
	`StreamGetEvent` †
	`ObjectsWaitAny`
		Creation and termination of threads; Using scheduling primitives, including suspension, semaphores, events, and pollable IO events.

Processes	`ProcessCreate`
	`ProcessExit`
	`SandboxSetPolicy` †
		Creating or terminate a process with a library OS instance.

Miscellaneous	`SystemTimeQuery`
	`RandomBitsRead`
	`ExceptionSetHandler` †
	`ExceptionReturn` †
	`SegmentRegisterAccess` †
		Querying system time, and random number generation. Setting an exception handler, and returning from the handler.

Remote Procedure Call	`RpcSendHandle` † `RpcRecvHandle` † `PhysicalMemoryStore` † `PhysicalMemoryCommit` † `PhysicalMemoryMap` †	Sending opened stream handles or physical memory across processes.

Table 2.1: An overview of the PAL ABI of Graphene. The ones marked with the symbol † are introduced in the initial publication of Graphene [171] or later extended for this thesis. The rest are inherited from Drawbridge [144].

2.1.2 The PALs (Platform Adaption Layers)

On most OSes, OS developers implement the PAL ABI using options in the native system APIs. Specifically, OS developers will build a Platform Translation Layer (PAL) as a thin library to translate PAL calls to native system APIs serviced by the host OS. The effort of PAL development is per host OS, whereas the library OS implementation is reusable on every hosts. Based on the principle of porting simplicity, PAL development must be straightforward for average developers.

The development of Graphene experiments the porting of the PAL ABI on several representative host examples, such Windows, Linux, OS X, FreeBSD, and SGX with an untrusted Linux kernel. For most of these hosts, implementing the PAL ABI is straightforward because most OSes export a version of the similar POSIX-style API. If there are exceptions where porting the PAL ABI is challenging, the cause is likely to be the assumptions made by the developers of the host OS. For instance, 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 PAL ABI to Windows is to change memory mappings at the physical page level, but requires running the PAL in root permission.

Based on the experience in Graphene, it is hard to ensure the portability of the PAL ABI on every potential hosts. A host may simply lacks the functionality for implementing a PAL call. The assumption is, maintaining the compatibility of the PAL ABI poses a much less challenge than maintaining the whole system API. Besides, the library OS may flexibly switch among emulation strategies to compensate the absence of certain host abstraction. As an example, bulk IPC is optional in the PAL ABI since its first definition, due to the expectation that implementing the feature may not be feasible on some hosts. If bulk IPC is not available, the library OS can fall back to RPC-based IPC, with a reasonable amount of performance penalty. In the worst case, if there is no emulation strategies to compensate for the absence of a PAL call, user can predict the affected applications and avoid running these applications on specific hosts.

2.1.3 Security isolation

To target multi-tenant environments, such as cloud, Graphene ensures security isolation between mutually-untrusting applications running on the same host. The security isolation of Graphene is comparable to running each application in a complete geust OS, featuring a fully-isolated guest OS. Similar to the virtual hardware interface isolating each virtual machine, the PAL ABI also enforces security isolation between library OS instances.

On a host where the host OS is fully trusted, Graphene delegates the enforcement of security isolation to the host OS. The library OS and the application are mutually-trusted, as long as they are loaded in the same process. The host ABI also separates API implementation from 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 PAL ABI, including files, network sockets, and RPC streams. The host-level security isolation is orthogonal to API complexity. The security checks in the hosts are easily enforced, based on monitoring the references to host resources and rejecting authorized resource access.

A trusted host OS isolates an application a container including one or multiple mutually trusting processes, or a *sandbox*. 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.

Threat model. For most of the Graphene hosts (except for the SGX host), an application running inside Graphene fully trusts the local library OS instances as well as the host OS. For multiple processes inside a sandbox, library OS instances also have to trust each others. Applications or library OS instances running in a separate sandbox are not trusted and can adversary to the host OS and trusted applications, by exploiting vulnerabilities on the PAL ABI.

The threat model of Graphene on a SGX host (i.e., Graphene-SGX) is similar to running other host OSes, except that the applications does not trust 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 [59]. 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 exploitations.

2.2 Graphene overview

Graphene is a library OS designed for reusing unmodified applications. The library OSes map a target system API, such as the Linux system calls, to abstractions in the PAL ABI. A library OS is comparable to a partial, guest OS running in a virtual machine. However, compared with an actual virtual machine, 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 [125, 144]. Library OSes have also been proven useful for reusing applications on new hardware platforms, such as SGX enclaves [47].

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 [144] 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 abstractions, 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).

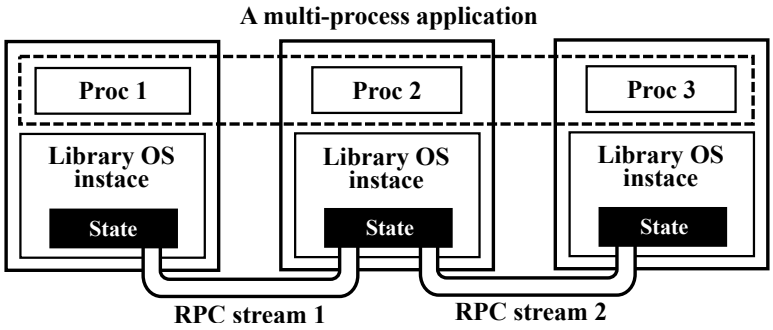


Figure 2.1: Multi-process support model of Graphene library OS. For each process of an application, a library OS instance will serve system calls and keep local OS states. States of multi-process abstractions are shared by coordinating over host-provided RPC streams, creating an illusion of running in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

2.2.1 The Graphene architecture

A library OS typically executes in either a paravirtual virtual machines [17, 125] or an OS process called a *picoprocess* [46, 144], with a restricted host ABI. Graphene executes within a picoprocess (Figure 2.2), which includes an *unmodified* application and its supporting libraries, which run alongside a library OS instance. The library OS is implemented over the PAL 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 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

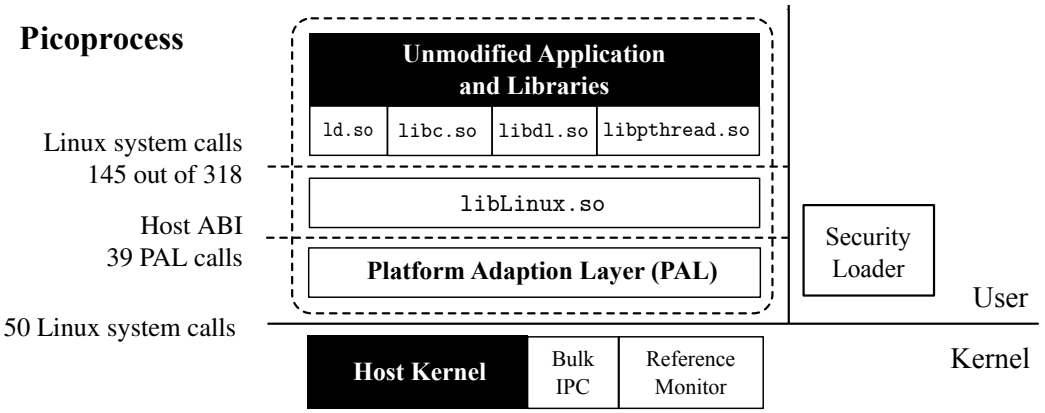


Figure 2.2: Building blocks of Graphene. Black components are unmodified. We modify the four lowest application libraries on Linux: `ld.so` (the ELF linker and loader), `libdl.so` (the dynamic library linker), `libc.so` (standard library C), and `libpthread.so` (standard threading library), that issue 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.

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

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 [48], 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 [28, 44, 61, 76, 107, 119, 120], 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 [81] or Mach-US [166], 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.

2.3 Summary

The Graphene design centers around building a para-virtualized layer (i.e., PAL) to reuse typical OS components, such as the system call table and namspaces in a library OS. 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 39 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 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.

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 [79], 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.

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.

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 [125], 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 ABI

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 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.

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.

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.

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.

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

- `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 nonblocking 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 connection. 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 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 [46]. 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.

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);
```

Introduced by Graphene, `StreamAttrSetByHandle()` is a PAL call for changing the attributes of a file or an I/O stream at the host level. `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 this `STREAM_ATTRS` data structure is to determine which stream attributes should be exposed to the guest. especially the attributes for a network stream (i.e., flags included in `network_option`), , 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.

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).

```
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 [175], which virtualizes page fault handlers and page table management to the guest.

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.

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 performance, 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.

`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.

```
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.

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 [46].

```
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 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.

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.

```
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 recreated 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 [124], 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), 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 the development of a practical, feature-rich library OS based on the PAL ABI, for reusing unmodified Linux applications. The major challenge in building the library OS, or `libLinux`, is to recreate the features of the Linux system interface, including system calls and namespaces on the PAL ABI. The development of `libLinux`, primarily focuses on two criterion: compatibility, the richness of Linux features and API, and performance, affected by the emulation strategies over the PAL ABI. This chapter shows how `libLinux` strikes a balance between compatibility and performance.

4.1 The `libLinux` architecture

The library OS of Graphene, or `libLinux`, is a single library that resides beneath a Linux application, for exporting compatible Linux features and API. `libLinux` guarantees reuse of an unmodified Linux application upon the PAL ABI, regardless of host limitations or distinctions. An unmodified Linux application assumes the existence of a Linux kernel or equivalent, with OS-specific features and characteristics, or **Linux personality**. `libLinux` reproduces the Linux personality, to act as a guest-level Linux kernel. Graphene develops `libLinux` as an ELF dynamic library (i.e., `libLinux.so`), and the PAL dynamically loads `libLinux`.

A key component of `libLinux` is a Linux system call table, which redirects system calls from a Linux application. A system call table is an important entry point to a Linux kernel. A system call table contains pointers to the kernel functions for each system call, indexed by the

system call numbers (e.g., NR_open or 10 on x86-64). Graphene moves the Linux system call table into libLinux, and develops system call handlers in the user space. Each system call handler emulates the semantics of a system call, based on either the specification described in the man pages [13], or the bug-for-bug behaviors observed in a real Linux kernel. Some system calls, such as rt_sigaction(), are partially documented in the man pages, and libLinux imitates a Linux kernel on implementing a poorly-documented system calls.

libLinux currently implements 145 system calls, and is sufficient to run a range of applications from servers to command-line programs or runtimes. For reference, a recent Linux kernel supports more than 300 system calls. A Linux kernel also contains a long tail of infrequently-used system calls. A study of the Linux system call usage [173] indicates that only 40 system calls are indispensable to every applications released in the Ubuntu official repositories. In the meantime, more than 100 system calls are used by only exactly one application, or none at all. The development of libLinux began with implementing 12 system calls, such as read() and open(), which are fundamental to running a "hello world" application, and gradually grows the system call count. Each time libLinux is tested against a new application, the number of system calls that need to be added has dropped. Graphene prioritizes the popular system calls, and leaves other system calls that are either unpopular or for administrative purposes such as rebooting or configuring network interfaces. libLinux demonstrates the sufficiency of implementing Linux system calls upon the PAL ABI, for a representative subset of applications, .

4.1.1 System call redirection

libLinux transparently redirects system calls from a Linux application. In a Linux kernel, a system call interrupt handler triggers the kernel operations whenever an application executes a "SYSCALL" or "INT $80" instruction. The interrupt handler switches the context of application, and then jumps to the kernel routines which serve the system calls. Because libLinux reuses unmodified Linux executables and libraries, it must redirect unmodified system call invocation to its system call table.

Normally, libLinux redirects system calls by modifying the C library (libc). Most Linux executables and libraries are linked against libc, and rely libc functions to access OS features in-

stead of invoking system calls directly. For example, compared with making the `read()` system call directly, more commonly an application uses libc's `stdio` functions, or calls the libc `read()` wrapper which internally runs `SYSCALL`. Unless configured otherwise, `libLinux` uses a modified **GNU C library (glibc)** [9], as the standard GNU libc of most Linux distributions, including Ubuntu. Graphene can be configured to use other libc variants, such as uClibc [23] and musl [15], if an application finds them sufficient.

Graphene modifies only 1,131 lines of the glibc code. glibc uses a platform-independent macro, `INLINE_SYSCALL()`, to invoke system calls. The macro `INLINE_SYSCALL()` contains assembly code that copies system call number and arguments to registers, and then uses `SYSCALL` to enter a Linux kernel. Graphene modifies `INLINE_SYSCALL()` to redirect a system call to an entry point of `libLinux` called `syscalldb()`. `syscalldb()` saves the current register state, similar to a context switch, and then calls the system call handler indicated by the system call number. For assembly code in glibc, Graphene replaces each `syscall` instruction with a dynamic call to `syscalldb()`, given the address of `syscalldb()` is dynamically determined. Figure 4.1 summarizes the mechanism of system call redirection.

Graphene modifies four glibc libraries: a runtime dynamic loader (`ld.so`), a core library (`libc.so`), a POSIX thread library (`libpthread`), and a dynamic loading library (`libdl`). Each of the Glibc libraries has separate purposes and features, and is mostly loaded on demand except `ld.so`. Graphene only modifies the glibc libraries which contains direct `SYSCALL` instructions. Other libraries, such as `libm.so`, only rely on existing libc functions, so Graphene leaves these libraries unmodified.

Hard-coded system calls. Static binaries, or some platform-dependent applications, contain hard-coded `SYSCALL` instructions which cannot be redirected by a modified libc. Application developers create a static binary with with hard-coded system calls by statically linking a local version of libc as part of the binary. It is also possible to program an application with assembly code that directly invokes system calls—usually in a language runtime (e.g., the go runtime) or a system software (e.g., busybox). Because a modified libc cannot redirect hard-coded system calls, the application switches context into the host kernel, causing security and compatibility breaches by exposing unauthorized or unsynchronized host OS states to the application.

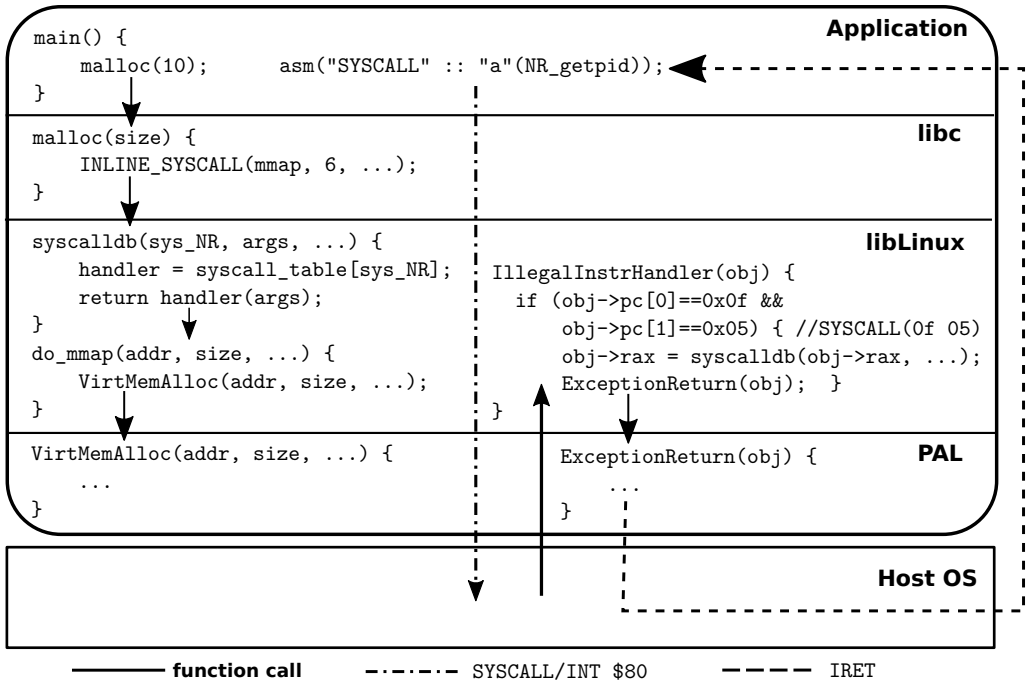


Figure 4.1: System call redirection for `libLinux`. In the normal case (the first instruction of `main()`), `malloc()` internally invokes `mmap()`, which is redirected to `syscalldb()` in `libLinux`.`libLinux` then invokes a PAL call, `VirtMemAlloc()`, to allocate host memory. The second instruction of `main()` invokes a direct system call, which is trapped by the host-level exception handler, and returned to `IllegalInstrHandler()` in `libLinux`.

`libLinux` needs support from a host OS to restrict direct system calls from an application. A Linux kernel allows an application to install a system call filter in the Berkeley Packet Filter (BPF) format, called a seccomp filter [152]. A seccomp filter can block or forward a system call based on the system call number, argument values, or the code address that invokes the system call. Graphene relies on the hosts to install a system call filter, or enforce architectural restriction, to detect direct system calls. For example, SGX already has a restriction that an enclave application cannot trigger any system call to switch context into the untrusted kernel. If the host detects a direct system call from an application, it can return a PAL exception to an exception handlers assigned by `libLinux`, using `ExceptionSetHandler()`. The exception handler can recover the system call number and arguments from the context saved at the system call, and forward the system call to the system call handler inside `libLinux`.

Using exceptions to forward direct system call is much slower than redirecting through a modified libc, due to the overhead of switching context between the application and the host kernel. To handle an exception from the host OS, the application at least switches context twice,

including both triggering the exception handler and returning to the original execution. To mitigate the overhead, `libLinux` can rewrite the hard-coded `SYSCALL` or `INT $80` inside each application binary during the run time. There can be two timings for rewriting the instructions: One is the timing when the runtime loader a new binary, and `libLinux` can perform a full scan of the binary to replace `SYSCALL` or `INT $80` with a jump to `libLinux`'s system call table. `libLinux` can also passively replace the instructions whenever the host detects a direct system call and triggers an exception handler. Graphene leaves binary rewriting as a future feature.

4.2 Resource management

`libLinux` depends on a host OS or hypervisor to manage hardware and privileged OS resources. The PAL ABI defines the abstractions managed by a host—from an I/O stream, a virtual memory area (VMAs), to a thread. These abstractions encapsulate the ubiquitously-installed hardware resources. Other host abstractions, such as a local RPC stream and a system clock, represents the low-level, privileged resources of a host OS. To run an application with `libLinux` as a guest, Graphene is able to drop the assumption of exporting or virtualizing any low-level resources, and program `libLinux` with the simplified, easy-to-implement host abstractions of the PAL ABI.

The role of `libLinux` in resource management is to allocate the host abstractions using the PAL ABI, as unambiguous requests for 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.

A language runtime, such as a Java virtual machine [10, 11, 30], or a Python [19] or Perl [18] runtime, has a similar role as `libLinux`. A language runtime commonly uses the ex-

isting system interface to request for resources needed by an application. For example, a language runtime may use the `mmap()` linuxapi to allocate a large heap, to assign chunks of the heap to an application. Therefore, the development of `libLinux` and the development of a language runtime share several challenges, including bridging the gap of resource allocation models between the guest and host, and influencing the host OS to efficiently assign hardware resources to applications.

`libLinux` reproduces the resource allocation models of Linux using the PAL ABI. To be portable across various hosts, the PAL ABI encapsulates the management of host resources. Graphene also simplifies the definition of the PAL ABI, to include a narrowed set of host abstractions that are necessary for a guest environment. A responsibility of `libLinux` is to implement different Linux models of allocating a resource. Take page management for example. Linux supports several way of memory allocations, including `mmap()` for allocating a fixed-size VMA, growing the stack of a process, and `brk()` for more fine-grained heap allocation. Since `libLinux` does not directly manage pages, it requires different emulation strategies to implement the allocation models needed by an application. 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.

Comparison with alternative approaches. Virtualization is one of the alternative approaches of guest-level resource management. A virtual machine often runs an unmodified OS kernel, which has control over the allocation of virtualized or dedicated hardware resources. To fully virtual hardware resources, a hypervisor can take one of the two common strategies. A strategy is to export a virtual hardware interface as an emulation of the physical hardware interface, in a hypervisor such as QEMU [20] or VMWare ESX [177]. A virtual hardware interface allows an unmodified OS kernel to directly manage virtualized hardware resources like physical hardware resources,

using a set of generic hardware drivers. Another strategy to leverage the hardware virtualization, such as IOMMU [55], to dedicate physical hardware resources to a virtual machine. Both of the virtualization strategies grant a virtual machine with more control over resource management than `libLinux` in Graphene.

Exokernel [77] adopts a library OS-like approach to export application-level system APIs, but grants each application the privilege to directly manage hardware resources. The rationale behind Exokernel is to bypass the complicated kernel logics for abstracting and multiplexing hardware resources, and to allow opportunities of domain-based optimization for each application. Exokernel enforces a security binding from machine resources to applications, so that each application can manage its own resources using an untrusted library OS. The similarity between the Exokernel and Graphene approaches is that they both delegate the protection and security isolation of hardware resources to the host kernel or hypervisor.

In terms of resource management, Exokernel and Graphene have made different decisions for the division of labour between the host and library OS. Exokernel prioritizes the efficiency of resource management for each application. To eliminate overhead of multiplexing resources, Exokernel exports the low-level hardware primitives, including physical memory, CPU, disks, TLB and page tables. Each library OS in Exokernel contains drivers to directly interfacing these hardware primitives, so that the choice of hardware is not longer transparent to an application. Graphene, on the other hand, prioritizes compatibility upon plenty of host OS and hardware platforms. Compared with the primitives exported by Exokernel, the PAL ABI of Graphene defines abstractions that are much more high-level and independent from the host OSes, such as files, virtual memory areas, and network sockets. Graphene sacrifices the application-specific opportunities for optimizing the resource management, but ensures the compatibility upon any hosts with the PAL ABI.

4.2.1 Virtual address space

A Linux application expects a contiguous, large virtual address space, to allocate a number of numerically-addressable memory regions. A program usually uses a libc allocator, requested by `malloc()` and `calloc()`, or a heap allocator of a managed language runtime, or reserves space on the current stack, to allocate fine-grained memory objects. To support application-level allocation,

an OS is responsible of maintaining a unique, consistent mapping between virtual memory areas (VMAs) and physical pages, and managing the virtual address space layout to prevent collision of VMAs. The Linux kernel, specifically, provides several ways of memory allocation, such as allocation by `mmap()` and `brk()`, and transparently growing a user stack downward. A application-level allocator may try several ways of requesting memory resources; for example, the glibc allocator uses both `brk()` and `mmap()` to allocate different sizes of memory objects. Applications depend on different memory allocation mechanisms of a Linux kernel, to dynamically allocate space for storing application data.

`libLinux` manages the virtual address space of each picoprocess. To emulate a Linux kernel, `libLinux` creates VMAs using two PAL calls: `VirtMemAlloc()` for creating an anonymous memory mapping, and `StreamMap()` for mapping a file into the virtual address space. Both PAL calls creates a page-aligned, fixed range in the virtual memory space, with the assumption that the host OS or hypervisor will assign a physical page to each virtual page being accessed, and fill the physical page with file content or zeros. `libLinux` does not assume a host to always implement demand paging. The only assumption that `libLinux` makes, when `VirtMemAlloc()` or `StreamMap()` returns successfully, is that the the application or `libLinux` is authorized to access any part of the created VMA, without causing a segmentation fault or memory protection fault. It is possible that a host may have statically assign physical pages to the whole VMA instead of gradually increasing the memory usage.

`libLinux` creates VMAs for two reasons. First, `libLinux` allocates memory regions on applications' request. `libLinux` also allocates memory for internal usages, such as maintaining the bookkeeping of OS states, and reserving space for buffering and caching. `libLinux` contains a *slab allocator* (for internal `malloc()`) and several object-caching memory allocators. For each abstraction, `libLinux` allocates a handle (e.g., a thread handle) using internal allocation functions. Therefore, the memory overhead of `libLinux` is primarily caused by allocating various types of handles for maintaining or caching OS states, and is roughly correlated with the abstractions used by the application.

`libLinux` maintains a list of VMAs allocated by either the application or `libLinux` itself. For each VMA, `libLinux` records the starting address, size, and the page protection (readable, writable, or executable). The VMA list traces the free space within the current virtual address

space. When an application allocates a VMA, libLinux queries the VMA list to search for a sufficient space. In another case, an application may specify the mapping address, and the VMA list can determine whether the address has overlapped with an existing VMA, to prevent corrupting the internal states of libLinux. According to the new VMA, libLinux uses VirtMemAlloc() or StreamMap() to create the mapping in the host OS. The VMA list also contains the mappings of PAL and the libLinux binary.

Whenever an application or glibc invokes a system call like mmap(), mprotect(), or munmap(), libLinux updates the VMA list to reflect the virtual address space layout created by the host. The basic design of a VMA list is a sorted, double-linked list of unique address ranges. Because Linux allows arbitrary allocation, protection, and deallocation at page granularity, libLinux often has to shrink or divide a VMA into smaller regions. libLinux tries to synchronize the virtual address space layout with the host OS, by tracing each memory allocation.

Different from a Linux kernel, libLinux does not isolate its internal states from the application data. libLinux shares a virtual address space with the application, and allows internal VMAs to interleave with memory mappings created by the application. In this design, an application does not have to context-switch into another virtual address space to enter libLinux. A consequence of the design is the possibility that an application will corrupt the states of libLinux, either accidentally or intentionally, by simply writing to arbitrary memory addresses. The threat model of Graphene does not assume libLinux to defend against applications because both libLinux and applications are untrusted by the host kernel.

Implementing brk(). brk() is a Linux system call for allocating memory space at the "program break", which defines the end of the executable's data segment. What brk() manages is a contiguous "brk region", which can be grown or shrunk by an application. Unlike mmap(), brk() allocates arbitrary-size memory regions, by simply moving the program break and returning the address to the application. The primary use of brk() in applications is to allocate small, unaligned memory objects, as a simple way of implementing malloc()-like behaviors.

libLinux implements brk() by dedicating a part of the virtual address space for the brk region. During the initialization, libLinux reserves an unpopulated memory space behind the executable's data segment, using VirtMemAlloc(). The size reserved for the brk region is de-

termined by user configurations. `libLinux` adjusts the end of the brk region within the reserved space whenever the application calls `brk()`, or `sbrk()`, a libc function which internally calls `brk()`. `libLinux` reserves the space for `brk()` to guarantee certain amount of memory resources for all the `brk()` calls, until the whole picoprocess is under memory pressure.

Address Space Layout Randomization (ASLR). `libLinux` implements Address Space Layout Randomization (ASLR) as a library OS feature. Linux randomizes the address space layout to defeat or at least delay a remote memory attack, such as a buffer overflow or a ROP (return-oriented programming) attack. A remote memory attack often depends on certain level of knowledge about the virtual address space layout of an application. For example, in order to launch an effective buffer overflow, an attacker tries to corrupt an on-stack pointer to make it points to security-sensitive data. With ASLR, a Linux kernel increases the unpredictability of memory mappings, so that a remote attacker is harder to pinpoint a memory target. To support ASLR, `libLinux` adds a random factor to the procedure of determining the addresses for allocating new VMAs. `libLinux` randomizes the results of both `mmap()` and `brk()`; for `brk()`, `libLinux` creates a random gap (up to 32MB) between the data segment and the brk region.

4.2.2 File systems

This section will discuss the implementation of file systems in `libLinux`, including a pass-through, sandboxed file system, the virtual file system layer for abstracting common file system operations, and other supported file system types.

4.2.2.1 A "chroot" file system

A Linux application depends on a list of indispensable resources within a hierarchical, POSIX file system. A POSIX file system is composed of a number of directories and files, including a root directory ("/") as the common ancestor. A POSIX application searches each file or directory in the file system by describing the *path* from the root directory to the target. An application either obtains a canonical or relative path from a user interface or configuration, or hard-codes the path in one of the application binaries. An application can heavily rely on the existence of specific paths

in a file system, such as `/tmp` (a default temporary directory) and `/bin` (a directory for system programs), as well as the POSIX file system features, such as directory listing and symbolic links.

`libLinux` creates a consistent, guest file system view containing the file dependencies of an application. A basic file system type in `libLinux` is a pass-through, sandboxed file system called a "**chroot (change root)**" file system. A chroot file system isolates a directory in the host file system, and maps such directory to a custom path inside `libLinux`. The mapping creates a restricted view for the application to access the files and directories inside the mounted host directory. A chroot file simply replaces the prefix of each searched path with the URI of the mounted host directory, and redirects the file operations to the host using the PAL ABI. For an application, each chroot file system has cherry-picked file resources in a host directory. The host reference monitor ensures that a chroot file system is sandboxed within the mounted host directory, so that any PAL calls can only access files and directories under the host directory, similar to a Linux program being `chroot()`'ed before running any untrusted execution.

For example, `libLinux` can mount a host directory "`file:/foo`" as a chroot file system under "`/bin`" in the guest file system. If the application search a path called "`/bin/bash`", `libLinux` will translate the path to "`file:/foo/bash`", and redirects access of `/bin/bash` to PAL calls for accessing `file:/foo/bash` in the host OS. Moreover, the host reference monitor enforces policies to prevent the untrusted application to escape the mapped directory, even if the application uses "dot-dot" to walk back to last level of directory; for example, `libLinux` cannot redirect a path `/bin/../etc/passwd` to `file:/etc/passwd`, because `file:/etc/passwd` does not belong to the chroot file system mounted at `/bin`. By mounting a chroot file system, `libLinux` creates a sandbox that disguises an unprivileged local directory (i.e., `/foo`) as a privileged system directory (i.e., `/bin`) in an application.

The implementation of common file operations in a chroot file system is mostly as straightforward as translating to one or few PAL calls. As previously stated, opening a file in a chroot file system simply requires calling `StreamOpen()` with the file URI translated from the requested path. If the chroot file system successfully opens the file in the host, it associates the returned PAL handle with a file descriptor, to translate common file system system calls such as `pread()` and `pwrite()` as PAL calls such as `StreamRead()` ad `StreamWrite()`, since the PAL ABI defines these two PAL calls to be positionless. For the more commonly-used `read()` and `write()`, the

chroot file system simply tracks the current offset of the file descriptor, and atomically retrieves and updates the offset in each system call. The batched `readv()` and `writev()` is translated to multiple `StreamRead()` ad `StreamWrite()` calls on the same file. Another two common system calls, `stat()` and `fstat()`, which retrieve the metadata of a file or a directory, need only one more step as translating the returned host-level stream attributes (i.e., `STREAM_ATTR`) to the POSIX data structure (i.e., `struct stat`).

The definition of the PAL ABI allows several opportunities of optimizing the latency of file system system call. Two common techniques being broadly used in `libLinux` are buffering and caching. To improve the latency of reading and writing a file, `libLinux` effectively buffers the content of multiple `read()` and `write()` system calls, until the application calls `fsync()` or the file offset exceeds the range of buffering. Buffering file changes potentially delay the timing of writing the data to physical disks, but `libLinux` accelerates the process by making the buffer a pass-through mapping of the file (using `StreamMap()`). For an application which performs lots of small, sequential reads or writes, or lots of small, random reads or writes with significant spatial locality, buffering the data can significantly improve the performance; evaluation shows that running GCC in Graphene to compile 0.7MLoC, on a Linux host, is only 1% slower than running on Linux. In terms of caching, `libLinux` contains a file system directory cache in the virtual file system, which will aggressively cache any directory information retrieved from the PAL ABI. The file system directory cache of `libLinux` also benefits other file systems, and the details will be further discussed in Section 4.2.2.3.

A chroot file system enforces container-style sandboxing of an application, but simultaneously allows sharing part of the file system tree with other applications and picoprocesses. Since `libLinux` supports mounting multiple chroot file systems in a picoprocess, Graphene users can configure a host to selectively export a few host directories containing the file resources in use. The security isolation of a single chroot file system is similar to the sandboxing of a Linux container [14], which restricts all the file operations of an application within a local file system tree unless the container is running on a stackable file system [3]. Graphene allows multiple applications to share a host directory, either read-only or with full access, and uses a host reference monitor to enforce AppArmor [37]-like, white-listed rules for isolating every file access. Graphene can share most of the system files and binaries, such as `/etc/hosts` and `/bin/bash`, without compromising

the security isolation of each application.

4.2.2.2 Guest-level file systems

Other than a pass-through file system, `libLinux` can use a different approach as implementing the file operations in a guest-level file system. A guest-level file system does not expose any host files and directories to applications; instead, a guest-level file system maintains its own file system states either in memory or in a raw format unknown by the host OS. A guest-level file system provides a different option for managing file resources in `libLinux`. Using a guest-level file system, `libLinux` potentially has more control over assigning physical resources to each file or directory.

One example of a guest-level file system is a pseudo file system, including the `proc`, `sys`, and `dev` file systems in Linux and similar OSes. A pseudo file system exports an extended system interface for accessing kernel states or raw devices. An application can use the `proc` and `sys` file systems to obtain information about processes as well as the whole kernel. The `proc` and `sys` file system have both redefined the common file operations such as `read()`, `write()`, and `readdir()`, for ad-hoc operations of accessing different types of process or kernel states. On the other hand, the `dev` file system exports both raw, physical devices and dummy, miscellaneous devices to an application. Examples of miscellaneous devices include `/dev/zero`, which outputs an infinite zero stream, and `/dev/urandom`, which outputs software-generated, pseudo-random bits. `libLinux` has implemented several critical entries of the `proc`, `sys`, and `dev` file systems, according to the command-line workloads targeted by Graphene.

Anther guest-level file system implemented in `libLinux` is a networked file system (NFS), which connects to a NFSv3 server running on either the local host or a remote machine. A networked file system provides another solution (besides a chroot file system) for `libLinux` to share file resources among applications or picoprocesses, by relying on a centralized NFS server to multiplex the file resources. A benefit of using a networked file system in `libLinux` is the natural support of a complete set of POSIX file system features. A networked file system does not depend on any local file resources, so all the file system features are implemented over a network connection. Therefore, the implementation of a networked file system is not restricted by the PAL calls defined for file access. However, the overhead of networking PAL calls can have significant impact

on the performance of a networked file system in `libLinux`, which can be much slower than an application-level NFS client on Linux (using `libnfs`).

Other guest-level file systems can potentially introduce a pre-formatted virtual disk into `libLinux`. Several popular file system formats, including EXT2, FAT, and NTFS, have been supported in either an application-level library (e.g., `libext2fs`) or a FUSE (Filesystem in userspace) driver (e.g., `NTFS-3G`). `libLinux` can potentially modify these libraries or FUSE drivers as guest-level file system drivers, to decompose a virtual disk. The drawback of using a pre-formated virtual disk is the difficulty of coordinating multiple picoprocesses that simultaneously access the same virtual disk. Since each single write to a file can involves writing to multiple physical blocks (the superblock, inodes, and data blocks), a guest-level file system driver must consistently coordinate multiple `libLinux` instances in order to share a virtual disk. Therefore, without inter-process coordination or a fully-distributed design, the usage of a pre-formated virtual disk in `libLinux` is most likely to be restricted to a single-process application.

4.2.2.3 A virtual file system

A POSIX file system defines a set of generic file system operations and primitives, making the underlying file system implementations transparent to most applications. An administrator can mount a file system at an arbitrary directory, to select among different solutions of managing file resources. When an application successfully opens a file under the mount point of a file system, a generic file descriptor is returned to represent the opened file resources and is fully independent from the underlying implementations. By presenting different resources as a generic primitive as file descriptors, an application can consistently use identical system calls, such as `read()` and `write()`, to invoke file and directory operations defined by the file system drivers. In a POSIX file system, an application can mostly be reused upon different file systems, as long as the required file resources are available in the chosen file systems.

Similar to a Linux kernel, `libLinux` includes an abstraction layer for exporting the generic operations and primitives of a POSIX file system, generally known as **a virtual file system**. A virtual file system defines a set of file and directory operations as the shared interface of every file system implementation. When an application opens or queries a target file resource, the virtual file system searches the file path among all mounted points, and then invokes the corresponding

operations implemented by the file system. In libLinux, each file system, such as a chroot file system or a pseudo file system, must provide a data structure to the virtual file system, containing function pointers referencing to all the file and directory operations implemented by the file system.

The virtual file system in libLinux enables several file system features and optimizations which indistinguishably benefit every file system. A few file operations, such as the batched readv() and writev(), can be emulated in the virtual file system using the basic file operations exported by the underlying file system. More importantly, the virtual file system in libLinux includes a local **directory cache**, as an optimization to the latency of searching a path in the guest file system tree, and retrieving file metadata. A directory cache is designed to reduce the frequency of executing the file operation of walking the file system data structures, by aggressively caching any directory information and file metadata returned from a file system implementation.

The directory cache in libLinux stores each searched path and its parent directories as directory entries using the spared picoprocess memory. The directory cache in libLinux has a similar architecture as the Linux file system directory cache. Each directory entry in the directory cache records the existence or nonexistence of a file system path, as well as the file attributes (e.g., file types, sizes, and permissions). If an application has given a path as a system call argument, libLinux first looks up in the directory cache to find any directory entries that matches with the given path. If a directory entry is already created for a path, libLinux can bypass the file system operations of querying the paths in the host file systems or other storage media. Since the directory cache in libLinux only caches paths searched by the local process, libLinux is likely to need less memory space for directory caching than a Linux kernel. The current implementation of libLinux never shrinks the directory cache until the picoprocess is terminated. Shrinking or freeing the directory cache space is a future work to libLinux.

The directory cache in libLinux contains several optimizations for reducing the latency of file-searching system calls. As one of the optimizations, the directory cache can confirm the existence of every prefixes of a canonical path by using one PAL call. Since libLinux does not maintain the mapping between each level of directory and the corresponding inode, libLinux simply needs to query the existence of each directory. When an application asks for a path and the path is not yet cached in the directory cache, libLinux only calls StreamAttrQuery() once to check the existence of the whole path, and uses the result to infer the existence of every parent

directories. For example, if libLinux successfully opens a file at /home/foo/bar, libLinux knows for sure that both /home and /home/foo exist in the file system. By reducing the amount of file system lookups for confirming path existence, libLinux reduces the number of PAL calls for searching in a chroot file system or a guest-level file system.

libLinux also applies several optimization techniques proposed by Tsai et al. [172], to improve the latency and frequency of cache hits in the directory cache. First, when searching a path inside the directory cache, libLinux uses an optimized algorithm to look up the canonicalized path all at once, instead of iteratively searching each path components. The optimized algorithm speeds up searching /home/foo/bar, from looking up /home, /home/foo, and /home/foo/bar in the directory cache, to directly searching an universal hash of the whole path. The optimization is based on the insight that libLinux has delegated the permission checks on each parent directory to the host OS. Another optimization is to aggressively create *negative* directory entries for paths that are known to be non-existent. Whenever an application has unlinked or moved a file, a negative directory entry can be created inside the directory cache, to prevent future system calls from calling StreamAttrQuery() to query the existence of the path.

4.2.3 Network sockets

libLinux supports three most common types of network sockets: TCP stream sockets, UDP datagram sockets, and UNIX-style domain sockets. A TCP or UDP socket is bounded with a host network interface, such as an Ethernet card or a loopback interface, whereas a domain socket is a local IPC (inter-process communication) abstraction similar to a FIFO (first-in-first-out) or a pipe. This thesis argues that other types of network sockets in Linux, such as raw packet sockets, is generally used by administrative programs. Most networked applications, including server-side and client-side applications, tends to treat a network socket as a contiguous I/O stream and lacks the incentive to create raw packet sockets or other types of network sockets; based on the intuition, the PAL ABI defines a network connection as an I/O stream, which encapsulates the composition and decomposition of network packets and omits platform-dependent features.

The Graphene architecture makes the network stack strictly a component of the host OS. The network stack inside an OS contains the implementation of various different network protocol

suites or families on top of the network interfaces. A virtualization solution generally moves or duplicates the network stack to a guest OS, and allows the guest OS to implement its own packet processing mechanisms, on a physical or virtual network interface. Several Linux network features or APIs assume the OS owns the network stack, and thus are challenging to implement in `libLinux` without any expansion to the PAL ABI. For example, an `ioctl()` opcode, `FIONREAD`, returns the number of bytes currently received on a network socket, including the packets queued inside the network stack. A use case of `recvfrom()` also allows an application to "peek" into the top of a network queue and retrieve the first few packets without draining the network queue. Since `libLinux` has no direct access to the network stack inside the host, it sometimes has to prefetch a network stream and buffer the incoming data inside the picoprocess. However, in normal cases, `libLinux` can implement `sendto()` and `recvfrom()` by directly passing the user buffers to `StreamWrite()` and `StreamRead()` and prevents the overhead of memory copy between user buffers and `libLinux`'s internal buffers.

4.2.4 Threads

A Linux multi-threaded application is generally programmed with the POSIX threads, or **pthreads**. The pthread library, or `libpthread`, is a user-space abstraction layer which creates schedulable tasks inside a Linux kernel or other OS kernel. Each pthread maps to a kernel thread, and `libpthread` maintains a descriptor (`pthread_t`) of each pthread, for signaling a pthread or blocking for the termination of a pthread. Moreover, `libpthread`contains several scheduling or synchronization primitives, including mutexes, semaphores, conditional variables, and barriars.

The creation of a pthread in `libpthread` is based on the `clone()` system call. When `clone()` is used to create a new thread, the application or `libpthread` assigns a preallocated space as the stack of the new thread, and a user function to start the thread execution. `libLinux` implement thread creation of `clone()` by calling `ThreadCreate()` in the PAL ABI. `ThreadCreate()` will start a new thread from a piece of trampoline code inside `libLinux`, which switches the stack pointer and jumps to the user function assigned as the starting function.

For each pthread, `libpthread` allocates an unique thread-local storage (TLS), which contains a **thread control block (TCB)** and thread-private variables. A TCB stores the private states

of a pthread, including at least a thread identifier. The definition of a TCB structure is up to the threading library; `libpthread` defines the TCB structure of a pthread as `pthread_t`, containing the linked list heads for queuing the pthread in a "join" queue or a wait queue for a synchronization primitive. On x86-64 Linux, `libpthread` locates the TCB of each pthread using using the FS segment register. The TCB is often followed by thread-private variables (variables defined with the `__thread` keyword) of the application and libraries, and both the TCB and thread-private variables are accessed by directly reading or writing at a specific offset from the FS segment register. Since accessing FS segment register is a privileged operation, an application can only set the TCB address by calling the `arch_prctl()` system call or passing the address as an argument to `clone()`. `libLinux` uses `SegmentRegisterSet()` in the host ABI to set the FS segment register in the implementation of `arch_prctl()` and `clone()`.

Besides creation of pthreads, `libpthread` also provides a collection of synchronization primitives, including mutexes, semaphores, read-write locks, conditional variables, and barriers. `libpthread` implements all these synchronization primitives based on futexes (accessed by Linux's `futex()` system call). A futex is a blocking and notification mechanism supported by a Linux kernel. The primary types of operations on a futex: waiting (`FUTEX_WAIT`) and signaling (`FUTEX_WAKE`). The `FUTEX_WAIT` operation blocks a thread until another thread updates a memory address and signals the blocking thread. The `FUTEX_WAKE` operation then allows a thread to signal one or multiple threads blocking for the memory update. Futexes provide a basic locking mechanism by combining two operations: checking a variable, and blocking until another thread updates the variable. Frank et al., 2002 [80] has shown the versatility of futexes to implementing user-level synchronization primitives. The same technique drives the implementation of synchronization primitives in `libpthread`.

For each futex, `libLinux` creates an event handle using `SynchronizationEventCreate()` in the PAL ABI. Whenever a `futex()` call checks a new word-aligned memory address for blocking, `libLinux` creates a host event and a wait queue to be mapped to the memory address. While a waiting `futex()` call blocks on a host event, another signaling `futex()` call can wake up as many blocking threads in the wait queueue as it wants, using `EventSet()`.

4.3 Multi-process applications

This section explains an efficiency implementation of the process creation mechanisms in `libLinux`, including `fork()` and `execve()`.

4.3.1 Forking a process

Implementing the UNIX-style, copy-on-write forking presents a particular challenge to the Graphene architecture. Using `ProcessCreate()` in the PAL ABI, each Graphene host creates a new picoprocess in a "clean" state, with an individual `libLinux` instance maintaining the OS resources and features for an application process. Forking a process involves cloning the state of running application and migrating all the resource handles inside `libLinux`, such as file descriptors, to the new picoprocess. Graphene drops the assumption that each of its hosts is capable of sharing physical pages between multiple applications or processes. Since `libLinux` cannot enable copy-on-write sharing between picoprocesses, `libLinux` needs an elaborate but efficient scheme for emulating the UNIX-style forking.

Without host support of copy-on-write sharing, `libLinux` emulates `fork()` by checkpointing and migrating the process states. When an application forks a process, the current `libLinux` instance holds a list of process resources that must be cloned for the new process. By checkpointing the process states, `libLinux` creates a snapshot of the current process, which is expected to the initial state of the new process, except a few minor changes. A process snapshot includes all allocated resources, such as VMA and file handles, and miscellaneous process states, such as signal handlers. After checkpointing, `libLinux` calls `ProcessCreate()` to create a new picoprocess in the host, and then migrates the process snapshot over the process handle as a RPC stream.

To fully emulate `fork()`, `libLinux` implements a checkpoint and migration scheme for duplicating the resource handles and application states between picoprocesses. For each type of resources, `libLinux` defines a function for decomposing a resource handle in a migratable form, and a function for reconstructing the resource handle inside another picoprocess. For example, for a VMA handle, `libLinux` checkpoints the address, size, initial flags, and page protection, and only if the VMA is accessed by the application and not backed by a file, `libLinux` copies the memory

data into the snapshot. For a file or network handle, `libLinux` runs a virtual file system operation, `vfs_checkout()`, to externalize the related states inside the file system implementations, but skip any temporary states such as buffers and directory cache entries. Finally, `libLinux` checkpoints the current thread handle, but modifies the handle snapshot with a new process ID.

The checkpoint and migration scheme of `libLinux` is comparable to VM migration by a hypervisor. When migrating a VM, a hypervisor has to copy the VM's guest physical memory to another host machine. A useful feature of a hypervisor is to migrate a live VM, and to implement the feature, the hypervisor needs hardware support for marking the dirty pages when it is copying the pages. Graphene also implements live migration of a picoprocess for `fork()` because of the general expectation that `fork()` should not halt the whole process. However, unlike live VM migration, Graphene chooses not copy the whole virtual address space of a picoprocess for three primary reasons. First, a checkpoint scheme that snapshots the whole picoprocess cannot differentiate temporary and permanent states inside a library OS. To improve I/O performance, `libLinux` tends to reserve virtual memory for caching and buffering, and `libLinux` can reduce migration time by skipping temporary states such as directory cache entries and I/O buffers. Second, by checkpointing handles individually, `libLinux` overwrites each handle and sanitizes sensitive states before sending the snapshot out to another picoprocess. Finally, the PAL ABI does not export any functionality for tracing the dirty pages during live migration, because the low-level hardware support needed is not available on a more restricted hardware like Intel SGX. For all the reasons above, `libLinux` only selectively checkpoints library OS states rather than snapshotting the whole picoprocess.

Migrating process states over a RPC stream adds a significant overhead to the latency of `fork()` in `libLinux`. To reduce the overhead, Graphene introduces a **bulk IPC** mechanism in the PAL ABI, to send large chunks of memory across picoprocesses. Using the bulk IPC mechanism, the sender (i.e., the parent) can request the host kernel to preserve the physical pages of application memory and snapshot data, and the receiver (i.e., the child) can map these physical pages to its own virtual address space. This bulk IPC mechanism is an efficiency way of sending pages out-of-band, while the parent process still uses a RPC stream to send control messages including the parameters of bulk IPC. Although the implementation is up to the host kernel, the bulk IPC mechanism should map the same physical pages in both parent and child, to minimize the memory copy in the host

kernel. The host kernel marks the physical pages copy-on-write in both picoprocesses, to ensure that the child receives a snapshot of the sent pages from the parent without sharing any future changes. The bulk IPC mechanism is optional in the PAL ABI, and `libLinux` can always fall back to sending process snapshots over RPC streams when the host fails to support bulk IPC.

Inheriting PAL handles. When a file handle to the child, `libLinux` sometimes needs to send the stored PAL handle, especially when the file handle represents a network socket or a deleted file. `libLinux` normally nullifies the PAL handle in the snapshot of a file handle since the PAL handle is only valid for the local PAL. However, if `libLinux` cannot recreate a PAL handle by calling `StreamOpen()` in the child picoprocess, `libLinux` needs host support to inherit the PAL handle from the parent. There are generally two conditions when the child process cannot recreate a PAL handle. First, a picoprocess cannot reopen a bounded network handle if another picoprocess still holds the local port. Second, the parent process may delete a file while holding a file descriptor to access the file content, generally as a way of detaching the file from the file system. If the file is deleted in the host file system, the child process cannot reopen the file using `StreamOpen()`.

`libLinux` uses two new PAL calls, `RPCSendHandle()` and `RPCRecvHandle()`, to send PAL handles out-of-band over a PRC stream. As `libLinux` walks through a file handle list for checkpointing, it marks the PAL handles that are network sockets or deleted files. If the parent deletes a file after migrating the file handle but before the child recreates the PAL handle, the child will either fail to reopen the file or accidentally open another file created afterward. `libLinux` can detect this corner case by coordinating the file system states across picoprocesses.

4.3.2 Process creation with `execve()`

Another Linux system call, `execve()`, creates a process with a separate executable and a clean memory state. The specification of `execve()` includes detaching the calling thread from a process and moving it to a brand-new virtual address space with the specified executable. As a common use case, a shell program (e.g., Bash) calls `execve()` after creating a thread using `vfork()`, to execute a shell command (e.g., `ls`) in a separate process, while the main process continues and waits for the shell command to finish. Linux uses the combination of `vfork()` and `execve()` as an equivalent of `spawn()` in the POSIX API, or `CreateProcess()` in the Windows API.

libLinux implements execve() by calling ProcessCreate() with the host URI of the executable, and selectively migrating process states to the new picoprocess. When the application calls execve() to run an executable, libLinux first has to identify the executable on the chroot file systems, to determine its host URI for creating a picoprocess. Although ProcessCreate() achieves the goal of creating a clean process with the target executable, execve() further specifies that the child must inherit the parent's credentials and file descriptors, except file descriptors opened with a CLOEXEC flag. libLinux uses the same checkpoint and migration scheme in fork() to selectively migrate handles and library OS states in execve(). The states migrated in execve() include the caller's thread handle, all the non-CLOEXEC file handles, program arguments and environment variables given by the application, and global OS states that are shared across libLinux instances (e.g., namespace information).

4.4 Coordinating guest OS states

A multi-process application executes on Graphene with the abstraction that all of its processes runs on a single OS. Each libLinux instance services system calls from its local state whenever possible. However, whenever a libLinux instance must share a library OS state with other instances, libLinux has to coordinate the state across picoprocesses via a RPC stream. Within a sandbox, multiple picoprocesses can securely coordinate shared states of multi-process abstractions, including process IDs, exit notification and signaling, System V IPC mechanisms (message queues and semaphores), shared file system states, and shared file descriptor states (Table 4.1). libLinux contains a coordination framework with several building blocks for implementing a shared multi-process abstraction.

As an example of balancing security isolation and coordination APIs, consider functionality that use the process ID namespace, such as UNIX signaling or exit notification (e.g., waitpid()). In Graphene, the process ID namespace, as well as signaling and related system calls, are implemented inside libLinux. A process can signal itself by having the library OS directly call the handler function. When picoprocesses are in the same sandbox, they coordinate to implement a consistent, shared process ID namespace, as well as to send and receive signals amongst

Abstraction	Shared State	Coordination Strategy
Fork	PID namespace	Batch allocations of PIDs, children generally created using local state at parent.
Signaling	PID mapping	Local signals call handler; remote signal delivery by RPC. Cache mapping of PID to picoprocess ID.
Exit notification	Process status	Exiting processes issue an RPC, or one synthesized if child becomes unavailable. The `wait` system call blocks until notification received by IPC helper.
`/proc/[pid]`	Process metadata	Read over RPC.
Message Queues	Key mapping Queued messages	Mappings managed by a leader, contents stored in various picoprocesses. When possible, send messages asynchronously, and migrate queues to the consumer.
Semaphores	Key mapping Semaphore count	Mappings managed by leader, migrate ownership to picoprocess most frequently acquiring the semaphore.
File System	File truncate sizes Deleted files FIFO & domain sockets	No coordination; completely relying on the PAL ABI; creating special files in the host to represent symbolic links.
Shared File Descriptors	Seek pointers	Mappings managed by parent, migrate ownership to picoprocess most frequently accessing the file descriptors.

Table 4.1: Multi-process abstractions implemented in Graphene, coordinated state, and implementation strategies.

themselves. Cross-process signals are implemented as RPCs over kernel-managed streams. When picoprocesses are in separate sandboxes, they do not share a PID namespace, and cannot send signals to each other. The reference monitor ensures that IPC abstractions, such as signaling, cannot escape a sandbox by preventing the creation of kernel-level streams across sandboxes.

A driving design insight is that the common case for coordination is among pairs of processes. Examples include a parent waiting on a child to exit, one process signaling another, or a single producer and single consumer sharing a message queue. Thus, Graphene optimizes for the common case of pairwise coordination, reducing the overhead of replicating data (see Section 4.4.3).

Although a straightforward implementation worked, tuning the performance was the most challenging aspect of the coordination framework. This section summarizes the lessons learned during the development of Graphene, from optimizing the coordination of various multi-process abstractions. This section then presents the design and driving insights of the coordination framework, followed by representative examples and a discussion of failure recovery.

4.4.1 Building blocks

The general problem underlying each of the coordinated library OS states is the coordination of **namespaces**. In other words, coordination between picoprocesses need a consistent mapping of names, such as process IDs or System V IPC resource keys, 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 coordinate shared abstractions over RPC.

`libLinux` creates an **IPC helper** thread within each picoprocess to respond to coordination messages from other picoprocesses. An IPC helper maintains a list of point-to-point RPC streams, and indefinitely waits for incoming messages. For each multi-process abstractions coordinated over RPC, `libLinux` defines a protocol for formating the header of each message, and determining the callback function for processing the message. GNU Hurd [81] has a similar helper thread to implement signaling among a process's parent and immediate children; Graphene generalizes this design to share a broader range of multi-process abstractions among any picoprocesses. An IPC helper serves remote messages and receive responses atomically, and is created in each picoprocess after the application spawned its first child process. 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 RPC messages.

Within a sandbox, all IPC helper threads exchange messages using a combination of a **broadcast stream** for global coordination, and **point-to-point** RPC streams for pairwise interactions, minimizing overhead for unrelated operations. The broadcast stream is created 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 RPC streams include the streams between parent and child processes established during `ProcessCreate()`, and RPC streams created through connecting to a RPC server identified by its URI. Because of the security isolation in the host, only picoprocesses in the same sandbox can connect to each other through RPC. If a picoprocess leaves a sandbox to create a new

one, its broadcast stream is shutdown and replaced with a new one, connected only between 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. One occasion of using the broadcast stream is **picoprocess ID allocation**. Because each picoprocess needs an unique ID to be recognized as a source or a destination of RPC messages, `libLinux` generates a random number as the ID of each picoprocess and confirms use the broadcast stream to confirm uniqueness. Another occasion of using the broadcast stream is **leader recovery**, which happens when a namespace leader unexpectedly crashes during coordination. For the implementation of leader recovery, see Section 4.4.2.

For each namespace (e.g., process IDs, System V IPC resource keys), `libLinux` elects one of the picoprocesses in a sandbox to serves as the **leader**. A leader is responsible for managing and memorizing the allocation of identifiers or resources in a namespace, in behave of all other picoprocesses. For a namespace like the process ID namespace, the leader subdivides the namespace for each picoprocess to reduce the RPC cost of allocation. For example, the leader might allocate 50 process IDs to a picoprocess which intends to clone a new thread or process. The picoprocess who receives the 50 process IDs becomes the **owner**, and can further assign the process IDs to children without 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 and exit notification. `libLinux` implements signals in various ways according to the causes of sigals. For signals triggered by hardware exceptions (e.g., `SIGSEGV`), `libLinux` uses the hardware exception upcalls in the PAL ABI. If a signal is sent from one of the processes for IPC purposes (e.g., `SIGUSR1`), `libLinux` exchanges RPC messages between picoprocesses to deliver the signal to the destination picoprocess. If a process signals itself, `libLinux` interrupts the targeted threads inside the process and uses internal data structures to call the appropriate user signal handler. `libLinux` implements all three of Linux's signaling namespaces: process, process group, and thread IDs. If a signal is sent for a process or a process group, every threads within the

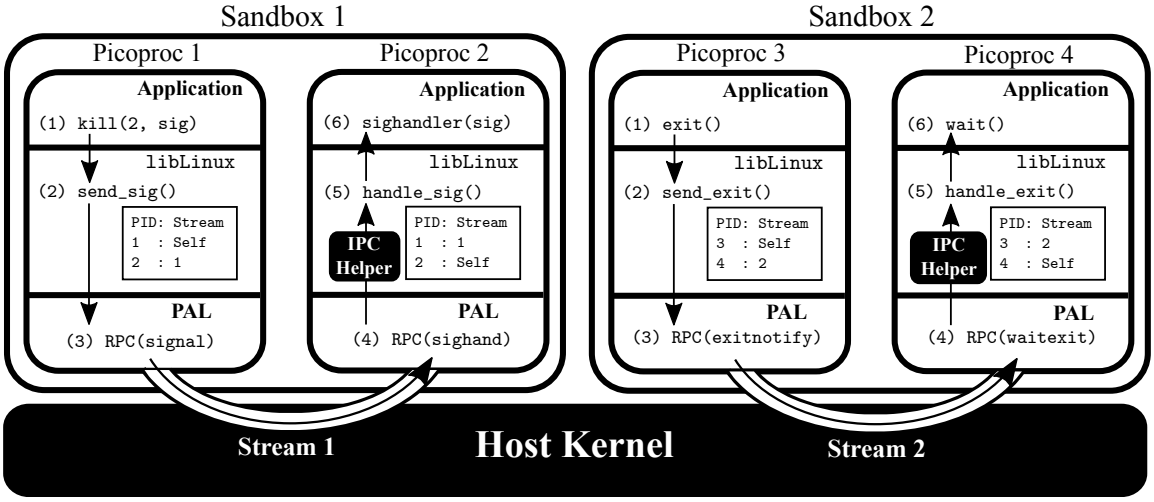


Figure 4.2: Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management. The location of each PID is tracked in `libLinux`; Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation of the `sigaction` interface. Picoprocess 4 waits on an `exitnotify` RPC from picoprocess 3 over stream 2.

process or the process group receives a copy of the signal, even if the threads belong to different picoprocesses.

Exit notification in Linux is based on the same mechanism as signaling. When a process exits, normally a `SIGCHLD` signal is delivered from the child process to its parent, to unblock the parent who might be waiting for exit notification using `wait()` or `waitpid()`. Exit notification is always coordinated over RPC streams between parent and child picoprocesses.

Figure 4.2 illustrates two sandboxes with picoprocesses collaborating to implement signaling and exit notification within their own process ID (PID) namespaces. Because process IDs and signals are library OS abstractions, picoprocesses in each sandbox can have overlapping process IDs, and cannot signal each other. The host reference monitor ensures that 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 4.2, a `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 calls `kill()`, the `SIGUSR1` handler will be called upon return. Also in Figure 4.2, 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 signaling semantics of `libLinux` closely match the Linux behavior, which delivers signals upon returning from a system call or an exception handler. Each process and thread have `sigaction` structures from the Linux source that implement the POSIX specification, including handler functions, as well as masking signals and reentrant behavior. `libLinux` does not modify libc's signal handling code. If an application has a signal pending for too long, e.g., the application is in a CPU-intensive loop, `libLinux` can use `ThreadInterrupt()` 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 a resource ID, which is arbitrarily allocated. Similar to process IDs, the leader divides the namespace of resource IDs among the picoprocesses, so that any picoprocess can allocate a resource ID from local state instead of involving the leader. Unlike the resouce IDs, System V IPC keys must be centrally managed by the leader, since an application might autonomously assign System V IPC keys to its processes. Global coordination is required to ensure that the same key maps to the same resource ID; the leader caches this information, but the owner of the resource ID makes the definitive decision about whether a ID mapping is still valid. A key which does not have a valid mapping can be assigned to a resource ID by any picoprocess to allocate a *private* IPC resource.

System V IPC message queues. In Graphene, the owner of a queue ID is responsible for storing the messages written to a System V IPC message queue. To ensure the serializability and consistency of all messages, any delivery and reception of messages must go through the owner. In the initial implementation of `libLinux`, sending or receiving messages remotely over a RPC stream orders of magnitude slower than accessing a local message queue. This observation 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 pi-

coprocesses 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 in the host, and the receiving process may regain the ownership of the message queue later from the leader and recover the serialized messages.

System V IPC semaphores. System V IPC semaphores follow a similar pattern to message queues. Each semaphore is owned by a picoprocess; a semaphore can be directly accessed by its owner as a local state, whereas other picoprocesses all have to access the semaphore through the owner over RPC. Since a semaphore shares the same performance pattern as a message queue, `libLinux` applies the same optimization of migrating the ownership of a semaphore to the picoprocess that most frequently acquires the semaphore. Another optimization of message queues, by making the updates asynchronous, does not apply to semaphores, because a participating picoprocess cannot proceed before successfully acquiring the semaphore. Most of the overhead in the Apache benchmark (see Section **??**) is attributable to semaphore overheads.

Shared file descriptors. The seek pointer of each file descriptor is implemented as a library OS abstraction; when reading or writing to a host file, the PAL ABI always obtains an absolute pointer from the beginning of the file. Although most applications do not share the seek pointer of an inherited file descriptor among processes, the `clone` system call can can be called with the `CLONE_FILES` flag and create a process which shares the whole file descriptor table with its parent. To share a file descriptor table among picoprocesses, one of picoprocesses (usually the oldest one) must be the leader of the file descriptor table to manage all mappings from file descriptors to the child picoprocess who owns the state of the file descriptors including the seek pointers. Every updates to a seek pointer must goes through the owner of the file descriptor (not the leader). The migration-based optimization for System V IPC message queues and semaphores is also effective for optimizing the performance of shared file descriptors.

Shared file system states. A chroot file system in `libLinux` is restricted by the PAL ABI to externalize any file system states. Other shared file system states are implemented as library OS abstractions, and have to be coordinated among picoprocesses. For example, a POSIX file system can contain special files such as a FIFO (first-in-first-out); a path bound to a UNIX domain socket; a symbolic link; or a file system lock. Implementation of these special files cannot completely depend on the PAL ABI, since the PAL ABI only supports regular files and directories.

A simple approach to coordinating file system states is to share a "dummy" host file. For example, `libLinux` can store the target of a symbolic link in a regular file on the chroot file system. For a FIFO, a bounded UNIX domain socket, or a file system lock, `libLinux` can store a mapping to the corresponding RPC stream, or to the picoprocess which owns the abstraction. By using the host file system as a less efficient but permanent coordination medium, `libLinux` can extend the coordination framework for sharing file system states.

Shared memory. The Graphene architecture does not currently permit shared memory among picoprocesses. This thesis expects that an extra PAL call and the existing support for System V IPC coordination 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 Graphene have avoided the use of shared memory in `libLinux`, both to maximize flexibility in placement of picoprocesses, potentially on an unconventional host (e.g., Intel SGX) or different physical machines. and to keep all coordination requests explicit. Shared memory may be useful to reduce latency for RPC messaging across picoprocesses on the same host.

Failure and disconnection tolerance. `libLinux` must tolerate disconnection between collaborating picoprocesses, either because of crashes or blocked RPC streams. In general, `libLinux` makes these disconnections isomorphic to a reasonable application behavior, although there may be some edge cases that cannot be made completely transparent to the application.

In the absence of crash recovery, 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, `libLinux`'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 pro-

cess will be affected. Making each `libLinux` instance resilient to arbitrary memory corruption of any picoprocess is left for future work.

Leader recovery. `libLinux` 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, `libLinux` sends out a message on the broadcast stream to volunteer for claiming the leadership. After a few rounds of competition, the winning picoprocess becomes the new leader and recover the namespace state by reading a namespace snapshot stored before the crash of the former leader or recollecting from other picoprocesses in the same sandbox.

When a picoprocess is moved to a new sandbox, `libLinux` will naturally detect the failure of leader because of blocked RPC. The sandboxed picoprocess will be the only candidate for leadership because the host has replaced the broadcast stream; as a result, the sandboxed picoprocess seamlessly transitions to new namespaces isolated from the previous 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 [61, 119]. We expect that Graphene performance could be improved on a microkernel with a more optimized IPC substrate, such as L4 [76, 107, 120]; 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 $\sim$2ms, but subsequent signals take only $\sim$55 μS.

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 Summary

This chapter demonstrates the implementation of a library OS, or `libLinux`, with a rich of Linux APIs and abstractions. Using the PAL ABI, `libLinux` faithfully reproduces the behavior of a Linux kernel, for both single-process and multi-process applications. In each process of an application, a `libLinux` instance serves as an intermediate layer between the application and the host, to manage and allocate host abstractions for a wide range of library OS abstractions. This thesis argues for the sufficiency of Linux APIs and abstractions supported by `libLinux`, based on the types of applications that are more likely to be ported across host platforms and the abstractions that these applications depend on.

`libLinux` achieves three goals. First, `libLinux` satisfies several resource management models and requirement, without duplicating or virtualizing the low-level components from the host OS or hypervisor. Although the PAL ABI has encapsulated the host resources, such as pages, CPUs, and I/O devices, `libLinux` introduces reasonable emulation and buffering to achieve the resource management model expected by the applications. Second, `libLinux` extends single-process abstractions to multiple `libLinux` instances collaborating to present a single OS view. To maximize the flexibility of placing the picoprocesses on different hosts, `libLinux` builds a coordination framework upon RPC messaging instead of shared memory. Third, `libLinux` identifies the performance overheads caused by coordinating over RPC, and designs several strategies of optimizing the coordination framework based on lessons learned in Graphene.

This chapter shows that, with reasonable amount of engineering effort, a library OS can be developed upon the previously-defined host ABI, with an extensive coverage of the Linux APIs and abstractions, and plenty of opportunities for performance optimizations.

Chapter 5

The Linux Host

This chapter uses Linux as an example of a Graphene host, to illustrate the implementation of the PAL ABI and security isolation. The usage of Graphene on a Linux host has two primary benefits. One benefit is to create a lightweight, VM-like, guest OS environment for running an application with a standalone OS view. The other benefit is to reduce the host kernel attack surface from an untrusted application, as the number of vulnerable kernel paths that can be triggered by the application. This chapter first demonstrates the feasibility of developing a Linux PAL prioritized for minimal Linux system call footprint, followed by a discussion of the security isolation mechanism.

5.1 Exporting the Host ABI

The initial design of the Linux PAL is based on an unmodified Linux kernel. By default, a Graphene picoprocess should run upon an off-the-shelf Linux kernel as an unprivileged, normal process, with a PAL loaded for exporting the PAL ABI. The Linux PAL demonstrates a minimal effort of implementing the PAL ABI on a single host, considering Linux is rich with APIs for programming all sorts of applications. Only two Graphene components on the host requires extension or modification of the Linux kernel: a bulk IPC kernel module and a trusted reference monitor.

On a Linux host, the majority of PAL calls are simply wrappers for similar Linux system calls, adding on less than 100 LoC on average for each PAL call. The most complex PAL calls on a Linux host are for exception handling, synchronization, and process creation, and each of these PAL calls requires multiple system calls and roughly 500–800 LoC in PAL. For example,

Component	Lines	(% Changed)
GNU Library C (`libc`, `ld`, `libdl`, `libpthread`)	606	0.07%
Linux Library OS (`libLinux`)	31,112	
Linux PAL	12,529	
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 develop the whole Graphene architecture on a Linux hosts. The application and other dynamically-loaded libraries are unmodified.

process creation (i.e., `ProcessCreate()`) requires both the `vfork()` and `execve()` system calls for creating a clean application instance, and would be more efficiently implemented inside the Linux kernel. Finally, the other major PAL components are an ELF loader (2 kLoC), Linux kernel and PAL headers (800 LoC), and internal support code providing functions like `malloc()` and `memcpy()` (2.3 kLoC). Table 5.1 lists the lines of code of each components of the Graphene architecture on a Linux host.

A PAL developer can use the Linux PAL to estimate the baseline of the PAL ABI development effort on a full-featured host OS . About half of the Linux PAL code turns out to be mostly generic to every host OSes and thus fully reusable for each PAL. The generic parts include the ELF loader, PAL headers, and internal support code, adding up to $\sim$6,000 LoC. The other half of the Linux PAL are host-dependent code containing mainly wrappers for Linux system calls. If the targeted host OS has exported a UNIX or POSIX-like API, porting the host-dependent code is mostly straightforward. For example, a follow-up experiment of developing a FreeBSD PAL finds most of the Linux PAL code to be highly portable.

5.1.1 Implementation details

The rest of this section will discuss a few PAL ABI abstractions that are particularly challenging on a Linux host. Similar challenges have been presented on other alternative host kernels such as FreeBSD, OS X, and Windows.

Bootstrapping a picoprocess. The Linux PAL works as a run-time loader to the Graphene library OS (`libLinux`). First, for the dynamic loading of `libLinux`, the Linux PAL contains an ELF loader, similar to the functionality of a libc loader (`ld.so`), to map the `libLinux` binary into

a picoprocess and resolve the addresses of PAL calls. Second, the Linux PAL constructs a process control block (PCB), providing information about the picoprocess and the host platform. For example, a member of the PCB exposes the basic CPU information (e.g., model name and number of cores) to `libLinux`, for implementing the `cpuinfo` entry of the `proc` file system as a library OS abstraction. Finally, the Linux PAL populates the stack with program augments and environment variables passed from the command line and switches to `libLinux`.

RPC streams. Three Linux abstractions are candidates for implementing RPC streams: pipes, UNIX domain socket, and loopback network sockets (bound at `127.0.0.1`). Creation of loopback network sockets is restricted by 65,535 ports which can be used to bind a socket on a network interface. The initial design of the Linux PAL uses pipes to implement RPC streams, but a later version switches to UNIX domain sockets. Although both pipes and UNIX domain sockets are viable options, different performance patterns are expected on these two Linux abstractions. The general expectation is that a pipe has much lower latency for sending small messages, whereas a UNIX domain socket has much higher throughput for sending large chunks of payloads. According to a micro-benchmark result of LMbench, on Linux 4.10 kernel, the latencies of sending one byte over a pipe and a UNIX domain socket are $\sim$2.2 μS and $\sim$3.5–4.5 μS, respectively. As for throughput, when sending a 10MB buffer, the bandwidth of a UNIX domain socket can reach $\sim$12 GB/s, whereas the bandwidth of a pipe is only $\sim$5 GB/s (for reference, the bandwidth of a loopback network socket is $\sim$7.5 GB/s), at less than half of the transfer rate of a UNIX domain socket. Based on the performance patterns described above, the latest design of the Linux PAL chooses UNIX domain socket for prioritizing the RPC throughput of sending large messages, particularly for migrating a snapshot of a forking process.

Exception handling. The Linux PAL can receive hardware exceptions (e.g., memory faults, illegal instructions, divide-by-zero) from an application, `libLinux`, or the PAL itself. A Linux kernel delivers all hardware exceptions to the user space as signals. If an exception is external to the Linux PAL (from an application or `libLinux`), the registered signal handler of the Linux PAL simply calls a guest exception handler assigned by `libLinux` using `ExceptionSetHandler()`. The Linux PAL also creates an exception object, either `malloc()`'ed or allocated from the stack, to pass the exception type and the interrupted context to the guest exception handler. Otherwise, if

an exception happens internally, the Linux PAL cannot deliver the exception to the guest exception handler because `libLinux` does not know how to recover from the exception (the execution inside the Linux PAL is transparent to `libLinux`). Unless an exception happens during the initialization, it must be triggered inside a PAL call made by `libLinux` or the application. To recover from an internal exception, the Linux PAL stores a piece of recovery information on stack at the entry of each PAL call. The PAL signal handler identifies the internal exception by comparing the faulting address to the mapping address of the Linux PAL, discovers the recovery information from the stack, rolls back the PAL call, and returns as a failed PAL call.

The PAL signal handler must avoid further triggering any hardware exceptions from the handler itself, or it can cause a **double fault**. Graphene ensures that the signal handler is carefully developed so that no memory faults or other exceptions can be caused by defects in the handler code. An unrecoverable case is corruption of a user stack, since the Linux kernel needs to dump the signal number and interrupted context on the stack before calling the PAL signal handler. The Linux PAL can avoid this case by assigning an alternative stack, or just kill the picoprocess when a double fault happens.

Process creation. Because the Linux PAL is designed as a runtime loader, a new, clean picoprocess can simply be created by calling `vfork()` and `execve()` with the Linux PAL as the executable. Within an application instance, the procedures for launching the first picoprocess and the consecutive ones are mostly identical, except a consecutive picoprocess is launched with a heritage of a global PAL data structure (containing a Graphene instance ID and a UNIX domain socket prefix), a broadcast stream, and an unnamed UNIX domain socket as a RPC stream to its parent. The Linux PAL keeps the other stream handles private to a picoprocess by marking the underlying file descriptors as `CLOEXEC` (close upon `execve()`). No page needs to be shared among picoprocesses.

A key challenge to implementing process creation in the Linux PAL is to reduce the overhead of initializing a new picoprocess. The elapsed time of process creation—from an existing `libLinux` instance calling `ProcessCreate()` to a new `libLinux` instance starting to initialize—contributes a major part of the `fork()` latency overhead in Graphene. There were several attempts of optimizing the process creation mechanism in the Linux PAL. The current design leverages `vfork()` and `execve()`, but caches the result of relocating symbols in the Linux PAL loader to

reduce consecutive picoprocess initialization time. A previous design uses `fork()` to snapshot a picoprocess with a fully-initialized PAL, and create picoprocesses ahead of time. The preforking design shows higher latency than the current design due to the IPC overhead for coordinating preforked picoprocesses.

Bulk IPC. The Linux PAL provides a `gipc` kernel module for transferring the physical pages of a large chunk of memory to another picoprocess. The `gipc` module exports a miscellaneous device, `/dec/gipc`, for committing physical pages from a picoprocess to an in-kernel store and mapping physical pages copy-on-write in another picoprocess. When `gipc` receives a request of committing a range of memory, it pins all the physical pages within the range, updates the page table to mark the pages copy-on-write, and awaits requests of mapping the pages to another picoprocess. Each physical page store has a limited number of slots for queuing physical pages, so a picoprocess can block during committing a range of memory if the physical page store is full.

The bulk IPC abstraction honors the sandbox boundary. An in-kernel physical page store cannot be shared across sandboxes. Any picoprocesses in a sandbox can access the same physical page store using an unique store ID; picoprocesses in different sandboxes can use the same store ID but will open different physical page stores.

5.2 Security isolation

Graphene separates OS features from security isolation. This section explains the Linux host design for isolating mutually untrusting applications, with a reduced attack surface for protecting Linux kernels. The discussion starts with the security guarantees and threat model, followed by the technical details of security isolation on a Linux host.

5.2.1 Goals and threat model

The security isolation model of Graphene ensures that mutually-untrusting applications cannot interfere with each other. A goal of Graphene is to provide security isolation with comparable strength as running applications in separate VMs. When running two unrelated applications on the

same machine, the security requirement of the OS involves not only blocking unauthorized access under normal circumstance, but also preventing an application from maliciously exploiting OS vulnerabilities to attack the other application. Because a modern OS, such as Linux or Windows, contains a rich of features and APIs, it is difficult to eliminate OS vulnerabilities or even just to verify whether an OS contains any vulnerabilities. A Linux container [14] does provide a separate OS view for each application, but still relies on the correctness of the whole Linux kernel to enforce security isolation. On the other hand, a VM or a library OS isolates the whole OS kernel or a part of the kernel in an unprivileged guest space for each application. The security isolation model prevents any vulnerabilities inside the VM or the library OS from compromising the host kernel and other applications.

Graphene enforces security isolation by separating backward-compatible OS features from security mechanisms. A Linux kernel exports a wide range of system calls, either as a legacy of previous kernels or as new programmability features. By implementing OS features in a library OS, Graphene reduces the attack surface of a Linux kernel to a small amount of system call corner cases. A reduced attack surface eliminates majority of execution paths inside a Linux kernel in which a malicious application can explore for vulnerabilities. The complexity of Linux features and APIs exported by a library OS is unrelated with the attack surface of the host kernel, unless the library OS asks for additional PAL calls. A Linux developer can even carve out a minimal Linux kernel with only the features needed by the Linux PAL, similar to shrinking a Linux kernel to a microkernel. Otherwise, Graphene depends on the host security mechanisms to restrict a library OS from accessing unauthorized system calls and resources upon an unmodified Linux kernel.

The Linux PAL installs a **system call filter** and a **reference monitor** for restricting the system calls, files, RPC streams, and network addresses accessed by a picoprocess. The Linux PAL requires 50 system calls in total for implementing both required and optional PAL calls. A system call filter, such as the Linux seccomp filter [152], can restrict the system call access of an application to only a small subset of all the system calls, with additional constraints on the parameters and optional flags permitted for each system call. A reference monitor further examines the arguments of permitted system calls to restrict the host resources accessed by an application, based on security policies configured in a manifest file [93]. The system call filter and the reference monitor significantly limit the ability of an untrusted Graphene picoprocess to interfere with the

rest of the system, preventing the risk of exposing any unknown vulnerabilities on a kernel path never exercised by the system call footprint of Graphene.

Graphene contributes a multi-process security model based on a **sandbox**, or a set of mutually-trusting picoprocesses running inside an isolated container. The reference monitor permits picoprocesses within the same sandbox to communicate over RPC streams, allowing the library OS to share and coordinate any states to create an unified OS view. If two picoprocesses belong to different sandboxes, the reference monitor will block any attempt of connecting RPC streams between the picoprocesses The access control over RPC streams enforces an all-or-nothing security isolation model: either two picoprocesses are in the same sandbox and share all the library OS states; or they are separated in two sandboxes and share nothing. Even though the library OS instance can span its state across multiple picoprocesses, a host kernel needs not to examine the accesses to shared library OS states, but still enforces security isolation between sandboxes.

Files and network addresses are the only host resources allowed to be shared across sandboxes, using well-studied, explicit rules. For sharing files, the reference monitor restricts the file access of a picoprocess within a few host file or directories, creating a restricted view of the local file system (close to Plan 9's unionized file system views [141]). The file rules in a manifest are similar to the policies of a **AppArmor profile** [37]; for each permitted file or directory, a developer specifies the URI prefix and the permitted access type, either as read-only or readable-writable. For sharing network addresses, the reference monitor restricts a picoprocess from connecting through a local address or connecting to a remote address, using **iptables-like firewall rules** [97]. Each network rule in a manifest specifies the local or remote IP address and port range that a picoprocess is permitted to bind or connect a network socket. The rules in a manifest file specify a minimal list of files and network addresses that a picoprocess needs to access, and are largely based on existing security policies (e.g., AppArmor profiles, firewall rules).

Threat model (details). When running on a normal Linux host (without SGX or other security hardware), Graphene assumes a trusted host kernel and reference monitor. All the components inside the kernel space, including the `gipc` kernel module for bulk IPC, and the reference monitor, are fully trusted by the other parts of the host kernel and the Graphene picoprocesses. On the other hand, the host Linux kernel does not trust the picoprocess, including the Linux PAL, a `libLinux`

instance, glibc, and the application. The system call filter and reference monitor initialized before an application starts running defend the whole host kernel from malicious system calls invoked by a picoprocess.

All the components running within a picoprocess, including the Linux PAL, the library OS (`libLinux`), glibc libraries, and the application, mutually trust each other. Without internal sandboxing, the Linux PAL or `libLinux` cannot protect its internal states or control flows from an application. Although some scenarios might require protecting the PAL or `libLinux` from the application, Graphene only restricts the adversary within a picoprocess; in other word, an adversary only compromises the library OS in the same picoprocess, but can never interfere the host kernel or other unrelated picoprocesses.

For a multi-process application, Graphene assumes that the picoprocesses running inside the same sandbox trust each other and that all untrusted code run in sandboxed picoprocesses. Graphene assumes the adversary can run arbitrary code inside one or multiple picoprocesseswithin a sandbox. The adversary can exploit any vulnerabilities in the library OS or IPC protocol, to propagate the attack to other picoprocesses. Graphene ensures that the adversary cannot interfere with any victim picoprocesses in a separate sandbox. A sandbox strictly isolates the coordination of `libLinux` instances; the reference monitor ensures that there is no writable intersection between sandboxes, so that the adversary cannot interfere with any victim picoprocesses.

Graphene reduces the attack surface of the host Linux kernel, but does not change the trusted computing base; however, reducing the effective system call table size of a picoprocess does facilitate adoption of a smaller host kernel. This thesis leaves the creation of a smaller host kernel for future work.

5.2.2 System call restriction

Graphene reduces the host ABI to 39 calls and the Linux system call footprint to 50 system calls. To reduce the effective attack surface to a Linux host, the Linux host restricts a picoprocess from accessing any system calls that are not part of the ordinary footprint of a Linux PAL. The system call restriction on Linux focuses on blocking most of the system calls that interferes with other

processes. The remaining permitted system calls with external effects are checked by the reference monitor (see Section 5.2.3).

Graphene restricts the host system calls using a seccomp filter [152], a feature introduced in Linux 2.6.12. A seccomp filter allows a Linux process to install an immutable Berkeley Packet Filter (BPF) program that specifies allowed system calls, as well as specifies the consequence of invoking certain system calls, such as creating a `ptrace` event or raising a `SIGSYS` signal. The BPF grammar is rich enough to filter scalar argument values, such as only permitting specific opcodes for `ioctl()`, as well as filter certain register values, such as blocking system calls from program counters (i.e., `RIP` register values) outside of the Linux PAL. The current seccomp filter installed by the Linux PAL contains 79 lines of straightforward BPF macros. Once a seccomp filter is installed in a process, the filter intermediates every system calls from the process and its future children, and guarantees the processes can never bypass the restriction. The Linux PAL uses `SIGSYS` signals to capture rejected system calls, and can either terminate the whole application or redirect the system call to `libLinux`. The consecutive steps of system call redirection are described in Section 4.1.1.

Developing a seccomp filter presents several technical challenges. First, a filter must restrict consecutive picoprocesses to install a new filter the reverts the system call restriction. Blocking the `prctl()` system call in a seccomp filter will prevent further installation of seccomp filters. Second, the BPF grammar can only filter certain values or ranges of a register. The filter needs to ensure that only the Linux PAL can invoke system calls; however, for satisfying the dynamic loading behavior of the PAL ABI, the Linux PAL is built as a shared library loaded at an address randomized by the Linux ASLR (Address Space Layout Randomization) feature. If a filter only permits a specific range of program counters, a child picoprocess will load the Linux PAL at another randomized address, and the inherited filter will restrict the child picoprocess to invoke any system calls. The Linux PAL introduces a small, initial loader loaded at a fixed address within each picoprocess and permitted to invoke system calls. Finally, a seccomp filter cannot check a string argument, such as a file path for `open()` or a network address for `bind()`. Checking a string argument requires involves reading user memory of unknown sizes and string comparison, and the BPF grammar only allows checking an argument arithmetically. Filtering permitted file paths and network addresses must rely on a trusted reference monitor (see Section 5.2.3).

The seccomp filter blocks unauthorized system calls from anywhere inside a picoprocess.

Even if none of the application binaries contains any `SYSCALL` or `INT $80` instruction, a piece of malicious application code can always bypass the Linux PAL to invoke unauthorized system calls. The application code can simply jump to a `SYSCALL` instruction inside the Linux PAL, or corrupt a returned address on the current stack to launch a ROP (return-oriented programming) attack. Even if the Linux PAL is hidden or isolated from the application, an adversary can always leverage a gadget, a byte sequence that resembles the target instruction, within an executable or a library. Therefore, the seccomp enforces both program-counter-based and argument-based restrictions to block unauthorized system calls from both the Linux PAL and the rest of picoprocess.

Security implications. Using an existing system call restriction mechanism like seccomp, Graphene limits the ability of an untrusted application to attack a Linux kernel. Ideally, since `libLinux` only requires the PAL ABI, Graphene can adopt a modified Linux kernel that only exports 39 PAL calls to each picoprocess. The seccomp filter instead isolates a picoprocess on an unmodified Linux kernel, with a reduced attack surface comparable to only exporting the PAL ABI. According to the principle of least privilege, each component or layer in a system should only be granted access to a minimal amount of resources or abstractions required for performing the expected tasks. The seccomp filter only permits a minimal amount of system calls with specific flags and opcodes required by the Linux PAL, so an untrusted application can only trigger a limited amount of execution paths inside the host Linux kernel. Graphene limits the ability of an untrusted application to explore known and unknown vulnerabilities on any kernel execution paths for servicing one of the blocked system calls.

Although a regular Linux process can also leverage a seccomp filter, Graphene makes a major contribution to reduce the system call footprint of any large-scale application to a fixed, small system call profile. Analysis shows that the system call footprint of a large-scale application such as Apache or MySQL can contain more than 100 system calls. Since `libLinux` has absorbed the Linux system call table, running Apache, MySQL, or any other application in Graphene leads to at most 50 host system calls. As a system running a wide range of applications can exposes a different partial view of the system call table to each application, Graphene has a static system call profile for all applications, allowing OS developers to focus on testing or analyzing a small portion of execution paths and corner cases of a Linux kernel. Sun et al. [168] proposes sandbox-

```
loader.exec = file:/usr/sbin/apache2        # allow loading executable
loader.preload_libs = file:/graphene/libLinux.so    # loading libLinux
fs.allow_ro.libc = file:/graphene/libc/     # loading modified libc
fs.allow_ro.mods = file:/usr/lib/apache2/modules/   # loading modules
fs.allow_ro.cond = file:/etc/apache2/       # reading configuration
fs.allow_rw.logs = file:/var/log/apache2/   # writing to logs
fs.allow_ro.http_docs = file:/var/www/      # reading website files
net.allow_bind.httpd = 0.0.0.0:80           # binding to local port 80
net.allow_conn.any = 0.0.0.0:1-65535        # allow any connection
```

Figure 5.1: A example of a manifest file, containing security rules for the reference monitor to permit accessing sharable resources. The manifest file is for running a Apache http server (without php and other language engines).

ing an uncertain, potentially-malicious application in Graphene with an unpredictable `libLinux` implementation.

Static binaries. Besides security purposes, a seccomp filter provides a compatibility feature for redirecting hard-coded system calls in a statically-linked application binary. Graphene leverages the seccomp filter to redirect these leaked system calls back to `libLinux`. The filter contains BPF rules to check if the program counters invoking the system calls are parts of the Linux PAL. The filter blocks system call invoked outside of the Linux PAL and delivers a `SIGSYS` signal to the PAL signal handler for redirecting the system calls to `libLinux`.

5.2.3 Reference monitor

The reference monitor on a Linux host checks the arguments of host system calls for referencing any sharable host resources. A host system call like `open()`, `connect()`, or `bind()` specifies a file system path or a network address for opening a file or network stream and cannot be filtered by a seccomp filter. The host kernel trusts the reference monitor to only permit a list of sharable resources in a picoprocess, based on rules in a manifest file. Once the reference monitor has permitted the creation of a file or network stream, consecutive operations on the stream such as reading or writing data can be trusted as long as being mediated by one of the permitted system calls.

The reference monitor enforces simple, white-listing rules based on security mechanisms already familiarized by users and developers. Figure 5.1 shows an example of resource access

rules in a manifest. First, a manifest lists the URI prefixes of permitted files or directories of an application, similar to an AppArmor profile. The executable (`loader.exec`) and the preloaded library OS binaries (`loader.preload_libs`) are permitted for read-only access by default. The reference monitor simply compares file URIs against each permitted URI prefix and checks the access types; unlike many existing security mechanisms in Linux and similar OSes, such as permission bits, Access Control Lists (ACLs), and SELinux labels, the reference monitor does not retrieve security policies from file metadata, but obtains the manifest from an out-of-band channel.

Manifest-based security simplifies the inspection, authentication, and population of security policies. An Android application is deployed with a similar manifest, listing the accessed files and other resources, which users approve when installing the application. Developers can authenticate a security policy by signing the content of a manifest. Moreover, to run an application, a user can choose among multiple manifest files with different levels of security privileges.

Network rules in a manifest are similar to **iptables firewall rules** for defending a server or a desktop machine. A network rule specifies a local or remote address that the application is permitted to bind or connect a network stream. A local or remote address can be an IPv4 or IPv6 address (possible to specify an "any" address, i.e., `0.0.0.0` or `[::1]`), combined with a specific port number or range. When an application creates a network stream, the reference monitor checks whether the local and remote addresses match one of the network rules.

The reference monitor on a Linux host is implemented as a Linux Security Module (LSM) extended from the existing AppArmor module. AppArmor is the default LSM of most Linux distributions, and a Linux kernel disallows multiple LSMs (e.g., AppArmor, SELinux) to be effective simultaneously. Graphene instruments a few security hooks of the AppArmor, to add checks for file system paths and network addresses. The security checks of the reference monitor are stackable with other host security mechanisms. For example, if a manifest lists a root-privileged file and the Graphene application runs in a unprivileged process, existing security checks in a Linux kernel still blocks the file access even though the reference monitor permits the access. The drawback of the implementation is that Graphene must run on a modified Linux kernel. Linux kernels do not support loading LSM as a dynamic kernel module. Graphene only replaces the AppArmor LSM in a Linux kernel; the rest of the Linux kernel remains unchanged.

A trusted security loader initializes the reference monitor when launching an application

in Graphene. When a user launches an application in Graphene from the command line, the first picoprocess begins in a new sandbox. The security loader reads the manifest file given by the user, and submits the sandbox rules to the reference monitor. The reference monitor exports a miscellaneous device called `/dev/graphene` for the security loader to submit sandbox rules using the `ioctl()` system call. Once the reference monitor starts a picoprocess in a sandbox, neither the first picoprocess nor any consecutive picoprocesses spawned in the sandbox can ever escape the sandbox or drop the restrictions on certain resources.

Alternative approaches. Other approaches can implement the reference monitor without modifying a Linux kernel, with a trade-off of performance or development simplicity. An approach is to implement the reference monitor as a trusted process receiving `ptrace` events from Graphene picoprocesses. Using the `ptrace()` system call, this reference monitor can retrieve user memory from the monitored picoprocesses, and block the system calls which request for unpermitted resources. Unfortunately, intercepting every system calls with `ptrace` events introduces significant overhead to PAL calls; thus, this approach is not ideal for isolating Graphene applications on a Linux host.

Another approach is to translate the resource rules in a manifest file to AppArmor or iptables rules. As explained in previous paragraph, the file and network rules in a manifest file are similar to the file lists in an AppArmor profile and the firewall rules enforced by iptables. Instead of implementing a Graphene-specific reference monitor, Graphene can convert a manifest file, either statically or dynamically, to security rules recognized by AppArmor and iptables. This approach requires no modification in a Linux kernel, and can benefit from existing optimizations of AppArmor and iptable. Graphene leaves the integration with AppArmor and iptables for future work.

Dynamic process-specific isolation. A child picoprocess may either inherit its parent's sandbox, or start in a new sandbox, by either specifying a flag to `ProcessCreate()` or calling the sandboxing PAL call, `SandboxSetPolicy()`. A new sandbox may obtain a subset of the original file system view, but can never request access to new regions of the host file system. If a child picoprocess voluntarily moves itself to a new sandbox using `SandboxSetPolicy()`, the Linux PAL issue another `ioctl()` call to `/dev/graphene` to dynamically detach the picoprocess from the

parent's sandbox and update sandbox rules. The reference monitor closes existing RPC streams and prevents RPC stream creation across sandboxes. When a process detaches from a sandbox, the reference monitor effectively splits the original sandbox by closing any RPC streams that could bridge the two sandboxes.

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. Section **??** describes an experiment that improves security isolation of Apache http server without sacrificing functionality.

5.3 Summary

The Linux PAL successfully leverages a limited subset of Linux system calls, to implement the whole PAL ABI for running a full-featured library OS. The PAL ABI separates the development of a host OS or hypervisor from the complexity of emulating a sufficiently-compatible Linux kernel. The chapter shows that most calls in the PAL ABI can be directly translated to similar system calls on a Linux host kernel. Only a few PAL calls, such as process creation and inter-thread synchronization, require additional attention for developing an efficient implementation strategy.

The Linux PAL also enforces robust security isolation between mutually-untrusting applications, by placing applications in separate, VM-like sandboxes. The security isolation on a Linux host is based on system call restriction using a seccomp filter, and a trusted reference monitor. Security isolation at the host interface restricts an untrusted application to explore vulnerable execution paths inside a Linux kernel. A seccomp filter enforce a fixed, minimal system call profile, regardless of bloated dependency of an application. The reference monitor follows simple, white-listed manifest rules listing all the authorized files and network addresses of an application, using well-known semantics such as AppArmor [37] or iptable-like firewall rules [97]. The reference

monitor can further enforce dynamic, process-specific isolation by splitting a sandbox to run a child picoprocess under more restricted resource permissions. Graphene on a Linux host can serve as a sandbox framework with a reduced attack surface upon the host kernel.

Chapter 6

The SGX Host

Intel SGX [128] shows a compelling example that an unmodified application fails to run inside a beneficial, new host environment. SGX is a hardware support for facilitating an application to defend against untrusted OSes, hypervisors, and infrastructures. Although SGX presents particular challenges for running an unmodified application, such as protecting the application from malicious host system calls, the Graphene architecture significantly reduces the complexity of resolving both compatibility and security restrictions of the SGX platform.

This chapter summarizes the development of a SGX framework using Graphene, for protecting unmodified Linux applications. This chapter starts with an overview of SGX-specific challenges for porting an application, followed by a comparison of approaches to shielding an application from an untrusted host [40, 47, 159]. This chapter then describes the design of Graphene-SGX, a SGX port of Graphene, which fits an unmodified application into the paradigm of SGX-ready applications, and customizes an interface to an untrusted OS for simplifying the security checks against malicious host system calls. This chapter shows that by checking a narrowed host interface, Graphene-SGX shields Linux system calls inside a `libLinux` instance.

6.1 Intel SGX overview

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

6.1.1 SGX (software guard extensions)

SGX [128] is a feature added in the Intel sixth-generation CPUs, as a hardware support for trusted execution environments (TEEs) [103, 117, 150, 169]. SGX introduces a number of essential hardware features that allow an application to protect itself from the host OS, hypervisor, BIOS, and other software. The security guarantees of SGX is particularly appealing in cloud computing, as users might not fully trust the cloud provider. Even if the whole cloud infrastructure is under on administrative domain, commodity operating systems have a long history of exposing security vulnerabilities to untrusted users, due to flaws in software and hardware [26, 38, 106, 122, 185]. Any sufficiently-sensitive applications would benefit from running on SGX to evade the consequences of a compromised OS kernel.

The primary abstraction of the SGX platform is an **enclave**, an isolated execution environment within the virtual address space of an application process. The features of an enclave include confidentiality and integrity protection: the code and data in an enclave memory region do not leave the CPU package unencrypted or unauthenticated; when memory contents are read back into the last-level cache, the CPU decrypts the contents, and checks the integrity of cache lines and the virtual-to-physical mapping. The memory encryption prevents a OS kernel, hypervisor, or even firmware from physically fetching the application secret from DRAMs; SGX can even survive a stronger attack at the hardware level, such as cold-boot attack [87], an attack based on removing DRAM from the memory bus at a low temperature and placing it in another machine. SGX also cryptographically measures the integrity of enclave code and data at start up, and is able to generate attestation to remote systems or enclaves to prove the integrity of a local enclave.

SGX enables the defense against a threat model where one only trusts the Intel CPUs and the code running in the enclaves. SGX protects applications from three different types of attacks on the same host, which are summarized in Figure 6.1. First, untrusted application code inside the same process but outside the enclave cannot access enclave memory or arbitrarily jump into enclave code. Second, OSes, hypervisors, and other system software cannot peek into enclaves from administrative domains; Third, other applications on the same host cannot exploit vulnerabilities in a OS kernel or system software to escalate privileges. Finally, off-chip hardware, such as buses, DRAM, and peripheral devices can be hijacked or replaced with malicious components, but can

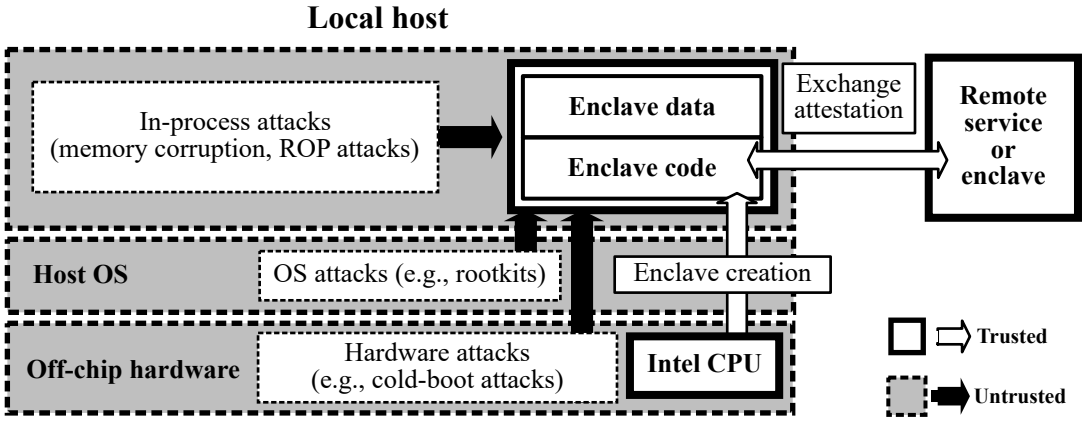


Figure 6.1: The threat model of SGX. SGX protects applications from three types of attacks: in-process attacks from outside of the enclave, attacks from OS or hypervisor, and attacks from off-chip hardware.

never steal or corrupt enclave secrets both encrypted and authenticated inside the memory. A SGX enclave can choose to trust a remote service or enclave and be trusted in return after performing a procedure of inter-platform attestation [35].

6.1.2 SGX frameworks

Despite the security benefits, 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 a piece of carefully-written shielding code in the enclave before being used by the application. The typical applications for processing security-sensitive data in a cloud environment include servers, language runtimes, and command-line programs, which rely on faithful emulation of Linux system call semantics, such as `mmap()` and `futex()`. Developers who wish to run these applications on SGX must either use a trusted, wrapper library that reproduces these semantics in an enclave, or replace large portion of application code unrelated to security. The extra effort for adapting existing application code into SGX causes delay to deployment of the technology; a number of security-sensitive applications can benefit from porting into SGX as soon as possible.

Related work shows concerns about the significant code changes to applications involved during porting to SGX. Although Haven [47] 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" [40]. Shinde et al. [159] argue that using a library OS, including libc, increases TCB size by two orders of magnitude over a thin API wrapper layer with shielding ability.

Graphene-SGX shows that a library OS can facilitate deployment of an unmodified application to SGX, granting immediate security benefits without crippling performance cost and full-blown TCB increase. The comparison between library OSes and thin shielding layers is inclusive in many ways. Besides the fact that Haven is evaluated upon a simulated hardware, Haven has a large TCB due to using Drawbridge [144], a full-featured library OS which recycles a significant portion of the Windows 7 source code. Graphene-SGX, on the other hand, shows performance overheads comparable to the range of overheads reported in SCONE. The authors of PANOPLY also notes that Graphene-SGX is 5-10% faster than PANOPLY [159]. 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 glibc, 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 [125].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 thin shielding layer has no significant impact on TCB size.

Besides running unmodified Linux binaries on SGX, Graphene-SGX also contributes a number of usability enhancements, including integrity support for dynamically-loaded libraries, enclave-level forking, and secure inter-process communication (IPC). Users need only configure features and cryptographically sign the configuration. Graphene-SGX is also useful as a tool to accelerate SGX research. Graphene-SGX does not subvert any opportunities of optimizing an application for SGX or partitioning application code outside of an enclave for further reducing TCB size. A number of SGX frameworks and security enhancements [109, 136, 153, 158] are complementary to Graphene-SGX.

6.1.3 Shielding complexity

A key question for developing a SGX framework is how much OS functionality should be pull into an enclave. A library OS and a thin shielding layer essentially make opposite decisions in protecting OS functionality on untrusted hosts. At one extreme, a library OS like Graphene-SGX and Haven pulls most application-supporting code of an OS into an enclave. On the other extreme, thin shielding layers, such as SCONE and PANOPLY, redirect an API layer (e.g., POSIX) or the system call table outside of an enclave and shield the application from malicious API or system call results.

The decisions of pulling OS functionality into enclaves impact the complexity of shielding an application from the untrusted host. The code and data inside of an enclave gain confidentiality and integrity protection from SGX; If an OS feature is kept outside of an enclave, the application or a wrapper layer in the enclave must design sufficient shielding code to check the results of the OS feature as part of the untrusted host. The concept of checking an untrusted OS feature is comparable to verifying the results of an outsourced database or program [42, 50, 192]. To make sure that outsourcing an operation is beneficial, the complexity of verifying an operation must be sufficiently lower than performing the operation. Some OS features are relatively verifiable; for example, a shielding layer can check the integrity of data sent to an untrusted storage using reasonably robust cryptographic functions. Other OS features, such a namespace, are less straightforward to verify; without a cryptographic protocol or a zero-knowledge proof, an API or system call wrapper needs to predict the correct results to check for integrity, and might end up emulating part of the operations like a library OS. Due to the complexity of shielding, leaving out OS features might result in adding a fair amount of shielding code, diminishing the benefit on TCB reduction.

A specific type of attacks called **Iago attacks** [59] threaten a framework that shields against an existing API layer or system call table in an untrusted domain. Checkoway and Shacham demonstrate several attacks to shielding systems like InkTag [91] and Overshadow [63], based on manipulating system call return values. A common feature of these shielding systems is the verification of a legacy API serviced by an untrusted kernel. Examples of Iago attacks include corrupting a protected stack by returning an address on stack to `mmap()`; forcing a replay attack on SSL (Secure Sockets Layers) protocol seeded by the returned values of `getpid()` and `time()`.

Applications often abuse system APIs for internal implementation, so an OS can explore vulnerabilities inside of either applications or a shielding system.

An important lesson learned from Iago attacks is that an existing API layer like POSIX or system call table is not suitable for the context of untrusting an operating system. The definition of POSIX API or Linux system calls assumes an untrusted client and has explicit semantics for checking against malicious inputs. On the other hand, these existing APIs do not specify how the clients should defend against an untrusted OS, leaving the design of proper defenses an exercise for application developers. Moreover, these existing APIs require an application to endow sensitive states to the operating system, making the API results more difficult to verify. For example, a Linux kernel associates a file descriptor with the current offset for accessing the file contents, whereas the application only specifies the file descriptor and a user buffer for `read()` or `write()`. While an OS can simply refuse to trust the inputs from an application, the same cannot be said for an self-protecting application or a shielding system without fully anticipating attacks from an OS.

Existing shielding layers, including SCONE [40] and PANOPLY [159], contribute shielding techniques for parts of the Linux system call table or POSIX API. In hindsight of the `mmap()` attack reported by Checkoway and Shacham [59], SCONE and PANOPLY prevent pointer-based Iago attacks by checking any memory addresses returned by the untrusted OS to be outside of the enclave memory, and the shielding layer will copy the memory contents into the enclave. SCONE also enforces the confidentiality and integrity of file contents and network payloads using cryptographic techniques to encrypt and authenticate the data, for any files and network connections marked by the users as security-sensitive. These techniques also apply to shielding applications in Graphene-SGX.

Library OSes like Haven [47] and Graphene-SGX provide an opportunity to redefine an API with the assumption of untrusting operating systems. A library OS absorbs API components or the system call table into an enclave, leaving a narrowed host interface which is much easier to defend than a bloated API layer. Both Haven and Graphene-SGX customize a host ABI for enclaves, or an **enclave ABI**, and treat the host OS as completely untrusted. Haven contains a proxy layer to redirect trustworthy OS services—besides services implemented inside the enclave—from a remote, trusted host. For example, Haven does not trust the host file system, and instead, loads a guest-level file system using an encrypted virtual disk provisioned from a remote host.

Graphene-SGX further defines an enclave ABI with shielding complexity in mind. Graphene-SGX adds a trusted enclave PAL below the the PAL ABI, to reduce the 39 PAL calls to merely 28 enclave calls (see Section **??**). Graphene-SGX defines each enclave call with clear strategies or semantics for checking the results. For each security-sensitive enclave call, Graphene-SGX accepts exactly one correct result, either cryptographically signed or provable with minimum bookkeeping or emulation. Among 28 enclave calls defined in Graphene-SGX, 18 calls are safe from Iago attacks; 8 calls are not security-senstive; 2 calls can be potentially blocked by the host (denial-of-the-service); only 2 calls are not yet shielded and currently left for future work .

Application code complexity. The motivating applications for SGX are small cryptographic functions like an encryption engine, or a simple network service running in the cloud. These applications are relatively small in the enclave, putting minimal demands on a shim layer. Even modestly complex applications, such as a R runtime and a simple analytics package, require dozens of system calls for providing wide-ranging OS functionality, including `fork()` and `execve()`. For these applications, there are several development options: first, application developers can modify the application to require less functionality of the runtume; second, a shielding layer can open and offer defenses for interfaces to the untrusted OS; finally, a library OS can pull more functionality into enclaves. This thesis argues that the best solution for ensuring an application to be secure in an enclave is up to the demand of the application. This chapter provides an efficient baseline for the approach of pulling functionality into a library OS, to run a wider range of unmodified applications on SGX. However, developers should be free to explore the other two approaches if application modification is possible.

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 and a SSL library in an enclave, but still allow client requests services outside the enclave. Similarly, a privilege-separated or multi-principal application might create a separate enclave for each privilege level. In general, Linux applications are more likely to be partitioned for privilege separation, especially for a set-effective-UID-to-root program that is escalated to root privileges from beginning.

The application partitioning techniques are application-specific, and often requires human intervention [121]. This thesis focuses on running an unmodified application as a whole. Parti-

tioning a complex application into multiple enclaves can be good for security, and should be encouraged given enough development time. 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 Security models

This section discusses the security models regarding running an unmodified application inside an enclave, including the threat model, user configurations for enclave shielding policies, and an inter-enclave coordination model for the support of multi-process applications.

6.2.1 Threat model

Graphene-SGX assumes a typical threat model for SGX enclaves, different from Graphene on a trusted Linux kernel or other hosts. An application only trusts the CPUs package and any code running inside the enclave, including the library OS (`libLinux`), a trusted PAL, libc, and any supporting libraries. All other 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. Graphene-SGX assumes any of these untrusted components to be potentially malicious, and constantly attempting to attack any known or unknown vulnerabilities of the trusted components. Graphene-SGX places supporting code outside of the enclave, as an untrusted PAL, which is needed only for liveness, but not safety.

The trust computing base (TCB) of Graphene-SGX also includes an architectural enclave, called `aesmd`, provided by Intel's SGX SDK [96]. `aesmd` is a privileged enclave authenticated by Intel's master signing key; `aesmd` receives attributes in the enclave signature of an application, and generates a token for approving enclave creation. Any framework that uses SGX for remote attestation must connect to `aesmd` to obtain a quote for proving that the enclave is running on an authentic Intel CPU, instead of a simulator. Graphene-SGX uses, but does not trust, the Intel SGX kernel driver, which mediates the creation of enclaves and swaps enclave pages. The current SGX hardware has a 128MB limitation on the total amount of physical memory shared among

all concurrent enclaves on the same host, and the Intel SGX kernel driver swaps enclave pages to storage, when one of the running enclaves demands more physical memory. Other than the `aesmd` enclave and the Intel SGX kernel driver, Graphene-SGX does not use or trust any other system software.

Graphene-SGX only handles the challenges of shielding an application from any attacks leveraging the vulnerabilities on the host interface (i.e., Iago attacks). Other application-specific security threats for SGX are beyond the scope of Graphene-SGX. For instance, an untrusted kernel can interrupt the enclave execution and refuse to schedule CPU resources to enclaves, causing deinal-of-service (DoS) in applications. Several work point out that an enclave can leak application secrets through side channels or controlled channels in cache architectures, memory access patterns, network traffics, and more. [62, 83, 86, 133, 180, 184, 186]. The techniques of thwarting or concealing side channels are application-specific and cannot be solely enforced in SGX frameworks or hardware. Several cryptographic function implementations have been known to be vulnerable to side channel attacks [187, 194]; for instance, users and developers, or Graphene-SGX itself, should avoid using one of the table-based AES libraries that are prone to memory access side channels, and switch to more secure, hardware-accelerated AES-NI [90].

6.2.2 User policy configuration

Despite that Graphene-SGX supports running an unmodified application on SGX, the user must make certain policy decisions regarding how Graphene-SGX should shield the application. The requirement is that the user must configure and sign the policy on a trusted host. A goal of Graphene-SGX is to balance policy expressiveness with usability, to minimize the cost of composing a policy and avoid mistakes. Without any user policy, Graphene-SGX creates a closed container where an application is not allowed to access any resources from the untrusted host.

As with Graphene on other hosts, Graphene-SGX reuses the **manifest** for user policy configuration of an application. In Graphene, a manifest specifies which resources an application is allowed to use, including a unioned, chroot file system and a set of iptables-style network rules. 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.

The same intention applies to Graphene-SGX, despite the difference of threat models.

In Graphene-SGX, a manifest is extended to protect an application from the untrusted host file system. Specifically, a manifest can specify secure hashes of trusted files that are integrity-sensitive and generally read-only, such as dynamic libraries, scripts, and configuration files. As part of opening a protected 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 default secure hash algorithm in Graphene-SGX is SHA-256, mainly for the generality in software signing, but other secure hash or signature algorithms are also viable options. Graphene-SGX includes a signing utility that hashes all trusted files and generates a signed manifest that can be used at run-time. The manifest can also specify files or directories that are not integrity-sensitive and can be accessed without being trusted. A manifest must explicitly specify all trusted or accessible files, and other unlisted files are considered potentially malicious.

The manifest also specifies certain resources be created at initialization time, including the number of threads, the maximum size of the enclave, and the starting virtual address of the enclave. Thus, Graphene-SGX extend the Graphene manifest syntax for specifying these options. Other security-sensitive options inherited from Graphene, such as filtering environment variables and enabling debug output, are also protected as part of the signed manifest.

6.2.3 Inter-enclave coordination

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, when an application calls `fork()`, Graphene-SGX creates a second enclave to run as a child process, and copies the parent enclave's contents over message passing. Graphene-SGX defines a group of coordinating enclaves an **enclave group**, similar to a sandbox of Graphene. Figure 6.2 shows an example of two mutually-untrusting enclave groups running on a host. Graphene-SGX supports all the Linux multi-process abstractions that Graphene has implemented in the user space, including `fork()`, `execve()`, signals, namespaces, shared file descriptors, and System V semaphores and message queues.

The implementation of multi-process abstractions in Graphene makes securing these abstractions easy in Graphene-SGX. Because all multi-process abstractions are implemented in en-

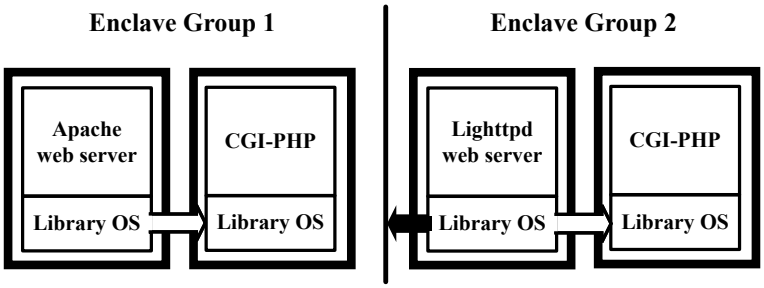


Figure 6.2: Two enclave groups, one running Apache and the other running Lighttpd, each creates a child enclave running CGI-PHP. Graphene-SGX distinguishes the child enclaves in different enclave groups.

claves and do not export shared states to the host OS, Graphene-SGX only has to adds two features for protecting multi-process abstractions. First, Graphene-SGX adds the ability for enclaves to authenticate each other via local attestation, and thereby establish a secured RPC channel, with messages both encrypted and signed. Second, Graphene-SGX provides a mechanism to securely fork into a new enclave, adding the child to the enclave group (see Section 6.3.3).

6.3 Shielding a library OS

This section discusses the shielding of a library OS in one or multiple enclaves, based on securing several features required by the host ABI, including dynamic linking, the PAL calls, and multi-process abstractions.

6.3.1 Shielding dynamic loading

To run unmodified Linux applications, Graphene-SGX implements dynamic loading and run-time linking with protection of binary integrity. In a major Linux distribution like Ubuntu, more than 99% of application binaries are dynamically linked against libraries [173]. Static linking is popular for SGX frameworks because it is easy to load and facilitates the use of hardware enclave measurements. Dynamic linking requires rooting trust in a dynamic loader, which must then measure the binaries. For Haven [47], the enclave measurement only verifies the integrity of Haven itself, and the same measurement applies to any application running on the same Haven loader.

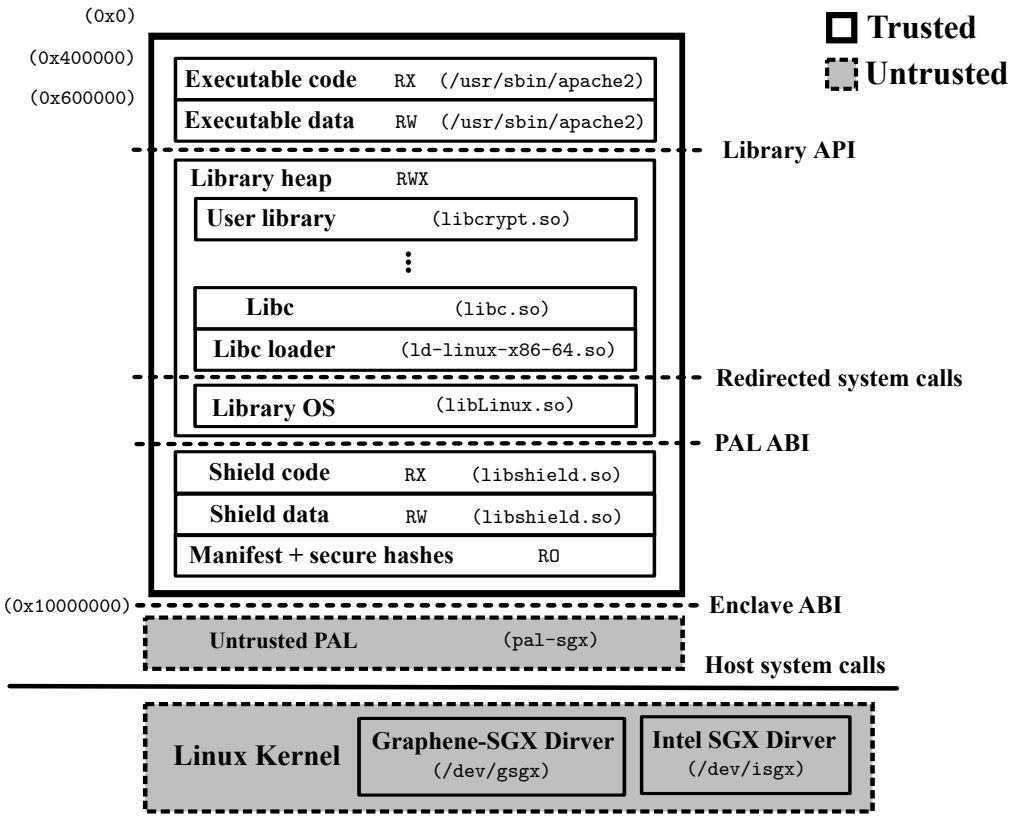


Figure 6.3: The Graphene-SGX architecture. The executable is position-dependent. The enclave includes an OS shield, a library OS, libc, and other user binaries.

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 PAL loader (`pal-sgx`), which calls the Intel's SDK 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 glibc libraries (`ld.so` and `libc.so`). After enclave initialization, the loader continues loading additional libraries, which are checked by the shielding libraries. If the secure hash does not match the manifest, the shield will refuse to load the libraries.

To reiterate, Graphene-SGX ensures the integrity of an application as follows. The Intel CPU verifies the measurement of the Graphene-SGX trusted PAL, an executable, and a manifest file. The trusted manifest includes secure hashes of all binaries dynamically loaded after en-

clave creation. This strategy does require trust in the Graphene-SGX, in-enclave boot-loading and shielding code to correctly verify and 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 [129] 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.

Graphene-SGX explicitly includes address zero in the enclave, as a net positive for security. Since Graphene-SGX does not make further strong claims regarding the presence of code that follows null pointers, including address zero is not strictly necessary. Graphene-SGX can still mark this address as unmapped in an enclave, preventing both trusted and untrusted code to access this address. Therefore, referencing a null pointer will still result in a page fault in the host. 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 [25], undermining the integrity of the enclave.

Relocation and resolution. Dynamic linking is not exactly a deterministic process. The loading order of user libraries may lead to different symbol resolution results. Some ELF binaries contains run-time linking functions (i.e., `IFUNC` functions), which can dynamically determine the target of symbols. ASLR (address space layout randomization), a feature implemented by `libLinux`, changes the base address of an relocatable binary in each execution. All these factors may affect the eventual result of dynamic loading to be different from what users or developers have expected.

Graphene-SGX puts the trust in `libLinux` and glibc loader (`ld.so`) to ensure the integrity of dynamic linking process. The shielding code verifies any inputs from the untrusted OS, including checking the integrity measurement of each binary, and filtering environment variables that may affect the linking result, such as `LD_PRELOAD` and `LD_LIBRARY_PATH`. Finally, for attestation, Graphene-SGX can generate a summary of the dynamic linking result, including the base address and global offset table (GOT) of each binary, to prove the integrity to a remote client.

6.3.2 Shielding the PAL ABI

For a single-process application, the Linux system calls are serviced by a library OS inside the enclave. Graphene-SGX reuses the same library OS used on other hosts, such as Linux, Windows, and FreeBSD, by including an in-enclave SGX PAL for exporting the PAL ABI. Within the 39 PAL calls defined in the PAL ABI, the SGX PAL focuses on exporting 35 calls that are required by `libLinux`. The remaining PAL calls are either pure optimizations (e.g., bulk IPC), or APIs for a different threat model (e.g., sandbox creation).

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 [59] demonstrate the subtlety of detecting semantic attacks via the Linux system calls, called Iago attacks. Projects such as Sego [111] go to significant lengths, including modifying the untrusted OS, to validate OS behavior on subtle and idiosyncratic system calls, such as `mmap()` or `getpid()`.

To reduce shielding complexity, Graphene-SGX further defines an enclave ABI which has simpler semantics than the PAL ABI and contains only 28 enclave calls to reach out to the untrusted OS. The challenge in shielding an enclave interface is carefully defining the expected behavior of

Classes	**Safe**	Benign	DoS	Unsafe
Entering enclaves & threads	**2**	0	0	0
Cloning enclaves & threads	**2**	0	0	0
File & directory access	**3**	0	0	2
Thread exits	**1**	0	0	0
Network & RPC streams	**6**	1	0	0
Scheduling	**0**	1	1	0
Stream handles	**2**	2	1	0
Mapping untrusted memory	**1**	1	0	0
Miscellaneous	**1**	1	0	0
Total	**18**	6	2	2

Table 6.1: An overview of 28 enclave calls of Graphene-SGX, including 18 *safe* calls (host behavior can be checked); 6 *benign* calls (no harmful effects); 2 *DoS* calls (may cause denial-of-service); and 2 *unsafe* calls (potentially attacked by the host).

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, Graphene-SGX 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, the enclave ABI 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 needed for verification. After copying chunks of the file into the enclave, but before use, the SGX PAL hashes the contents and checks against the manifest. The enclave ABI limits the possible return values of each enclave call to one predictable answer, and thus reduces the space that the untrusted 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 `libLinux`, and do not need shielding from the untrusted OS.

Table 6.1 lists the 28 enclave calls of Graphene-SGX, organized by the risk, and Table 6.2 further specifies the outputs, inputs, and checking strategies of the enclave calls. This thesis categorizes 18 enclave calls as *safe* because the responses from the untrusted OS are easily checked in the enclave. Graphene-SGX checks these safe enclave calls based on three strategies. The first strategy is to blocking out all inputs from the untrusted OS. For instance, when the enclave creates a new thread using `CLONE_THREAD()`, a pre-allocated enclave thread is waken up and takes no input from outside of the enclave. The second strategy is to define the input semantics to be as predictable as possible for checking. An example of a predictable call is `MAP_UNTRUSTED()`,

Classes	Enclave calls	Outputs	Inputs	*Risks*	Checking strategies / threats
Entering enclaves & threads	START_ENCLAVE		args,envp, rpc_fd	*safe*	Filter args & envp based on manifest; local attestation for RPC
	START_THREAD			*safe*	All thread start at clean state
Cloning enclaves & threads	CLONE_ENCLAVE	exec,manifest	rpc_fd	*safe*	Local attestation for child enclave measurement and RPC
	CLONE_THREAD			*safe*	Thread parameters stored in enclave; start a clean thread
File & directory access	FILE_OPEN	path	fd	*safe*	Check if listed in the manifest
	FILE_TRUNC	fd,size		*safe*	Update the secure hash
	FILE_ATTRS	fd	attrs	*unsafe*	File attributes need to be signed in advance (future work)
	DIR_LIST	fd	dir_list	*unsafe*	Directory contents need to be signed in advance (future work)
Thread exits	EXIT_THREAD			*safe*	Clean up state before exit; the thread can be reused, but will never return to the former state.
Network & RPC streams	SOCK_LISTEN	addr	fd	*safe*	Establish a TLS/SSL connection in application level or PAL
	SOCK_ACCEPT	fd	newfd	*safe*	
	SOCK_CONNECT	addr	fd	*safe*	
	SOCK_SEND	fd,data,size		*safe*	Contents secured by TLS/SSL in application level or PAL
	SOCK_RECV	fd	data,size	*safe*	
	SOCK_SETOPT	fd,option		*benign*	Only as hints to the host
	SOCK_SHUTDOWN	fd,access		*safe*	Send "close notify" over a TLS/SSL connection
Schedul-ing	YIELD	tid		*benign*	Only as hints to the host
	FUTEX	addr,op		*DoS*	Calls may prematurely return or never return; the host may corrupt futex values (addr is outside the enclave)
Stream handles	HANDLE_CLOSE	fd		*benign*	Only as hints to the host
	HANDLE_POLL	event_array	polled	*DoS*	The host may not deliver events
	HANDLE_SEND	fd,send_fds		*safe*	Handle contents and session keys sent over secured RPC
	HANDLE_RECV	fd	recv_fds	*safe*	
	HANDLE_FLUSH	fd		*benign*	Only as hints to the host
Untrusted memory	MAP_UNTRUST	fd,off,size	addr	*safe*	addr must be outside the enclave; secure hashes verified before use
	FREE_UNTRUST	addr,size		*safe*	Freeing untrusted memory cannot corrupt the enclave
Miscella-neous	SYSTIME		timestamp	*safe*	Ensure monotonic increase; retrieve timestamps from remote servers if accuracy is necessary
	SLEEP	sleep_msec	remaining	*benign*	remaining time $\leq$ sleep time

Table 6.2: Specifications of 28 enclave calls, including the outputs, inputs, risks (safe, benign, DoS, or unsafe), and strategies for checking the responses from the untrusted OS.

which simply maps a file outside the enclave. The third strategy is to establish cryptographic techniques for checking data integrity. For instance, after mapping a file with `MAP_UNTRUSTED()`, the SGX PAL copies the file contents into the enclaves, generates a secure hash, and matches with the manifests. Using the same strategy, a TLS/SSL connection can be establish either inside the application or PAL, to check the results of accessing network and RPC streams, with enclave calls like `SOCK_SEND()`, `SOCK_RECV()`, and `SOCK_SHUTDOWN()`.

Other 6 enclave calls 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 enclave call is `STREAM_FLUSH()`, which requests that any data buffered inside the host OS to be flushed out to a network or a disk. Cryptographic integrity checks on a file or network communication can detect when this operation is ignored by untrusted software. Another example is `YIELD()`, an enclave call for requesting the untrusted OS to schedule CPU resources. The result of `YIELD()` does not affect the integrity of an application because it simply serves as a hint to the untrusted OS scheduler.

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 `HANDLE_POLL()`. In the example of `HANDLE_POLL()`, a blocking synchronization call may never return, violating liveness but not safety. A malicious OS could cause a futex call to return prematurely or corrupt the futex value; 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. For `HANDLE_POLL()`, the untrusted OS may never deliver any stream events into an enclave. Denial-of-the-service attacks on these enclave calls are less of a security threat than integrity attacks, due to the assumption that the untrusted OS controls all the hardware resources.

Finally, only two enclave calls, namely `FILE_ATTRS()` and `DIR_LIST()`, are *unsafe*, because Graphene-SGX currently do not protect integrity of file attributes or directory lists. Checks for these two calls would require signing the file attributes or directory lists on a trusted host. Other existing work like Inktag [91] also demonstrate the integrity checks for file attributes. Graphene-SGX leaves the checks for these two enclave calls for future work.

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 the Merkle trees of file contents. The SGX PAL contains a page allocator for servicing `VirtMemAlloc()` calls inside the enclave. Once the SGX PAL has exhausted the reserved heap, no more pages can be assigned to the library OS or the application. The restriction of enclave memory is temporary, since SGX version 2 will add instructions for adding empty pages to enclaves in run time.

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. Since the number of threads in an enclave is restricted by the space allocated for thread control sections (TCSs), SGX version 2 will support dynamic thread creation alone with dynamic paging. The current version of Graphene-SGX requires users to specify how the maximum amount of threads the application needs inside the manifest.

This choice impacts 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 [136] 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.

SGX virtualizes the FS and GS registers, which allows Graphene-SGX to assign the in-enclave address of thread-local storage. Graphene-SGX sets the values of FS and GS registers

using the `WRFSGSBASE` instruction, and requires no extra enclave call to the untrusted OS.

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.

The untrusted OS may deliberately trigger memory faults, by modifying the page tables. For instance, controlled channel attacks [186] manipulate the page tables to trigger page faults on every branching points in an SGX application and observe the control flow. The overhead for delivering memory faults may also be a problem for an application that uses exception behavior for correctness, such as deliberately causing page faults on an address as a synchronization mechanism. Direct exception delivery within an enclave is an opportunity to improve performance and security in future generations of SGX, as designed in Sanctum [66]. T-SGX [158] also shows an example of delivering a page fault back to the enclave, if the page fault is triggered within a transaction created by Intel's Transaction Synchronization Extensions (TSX).

By handling exceptions inside the enclave, Graphene-SGX can 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. SGX also traps `SYSCALL` or `INT $80` inside an enclave; thus, Graphene-SGX redirects the system calls inside a static binary to `libLinux`.

6.3.3 Shielding multi-process applications

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 IPC mechanisms, using message passing instead of shared memory. Thus, Graphene-SGX implements multi-process abstractions in enclaves without

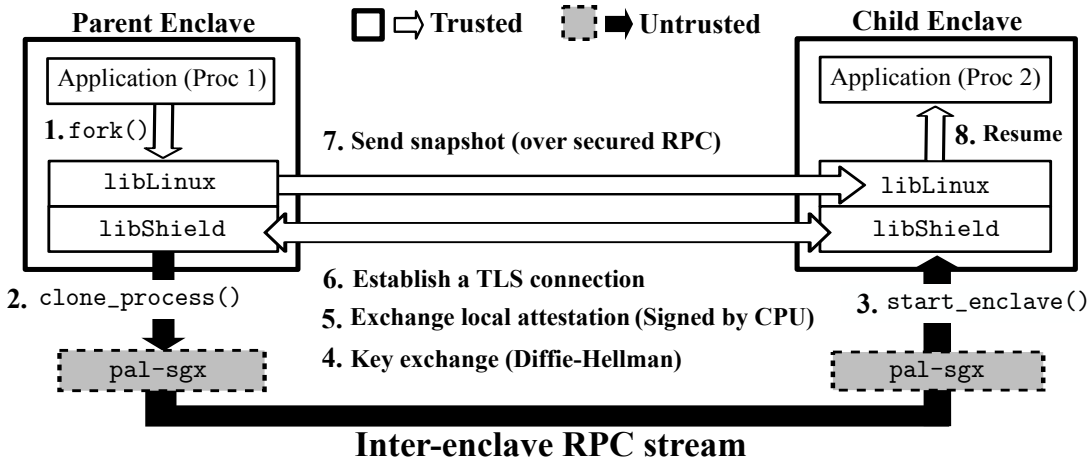


Figure 6.4: Process creation in Graphene-SGX. Numbers show the order of operations. When a process forks, Graphene-SGX creates a new, clean enclave on the untrusted host. Then the two enclaves exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process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

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 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 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 server may 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 applications 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

This chapter describes Graphene-SGX, a port of the Graphene library OS on the security-centered, Intel SGX platforms. SGX facilitates the protection of applications against the whole untrusted system stack ranging from off-chip hardware to the OS, but imposes several restrictions to running unmodified applications. Graphene-SGX removes the restrictions by servicing Linux system calls inside an in-enclave library OS instance and defining an enclave interface with explicit checks for responses from the untrusted OS. Compared with other thin, shielding layers [40, 159], Graphene-SGX shields a large range of Linux system calls, without extending the host ABI that an enclave needs to check.

Graphene-SGX shows the feasibility of protecting an unmodified application and library OS on an untrusted OS, using three shielding techniques. First, Graphene-SGX shields the dynamic loading process by generating a unique cryptographic measurement that verifies all the bi-

naries of one application. Second, Graphene-SGX further defines a simple enclave interface below the PAL ABI to shield an enclave from a series of subtle semantic attacks launched by the untrusted OS, known as Iago attacks [59]. For most of the enclave calls that reach out to the untrusted OS, Graphene-SGX either restricts the possible responses from the OS to one predictable answer, or ensures that deliberate failures of the OS are benign to the application. Finally, Graphene-SGX spans an multi-process application into multiple mutually-trusting, cooperative enclaves. To shield the inter-enclave collaboration from the untrusted OS, Graphene-SGX establishes mutual trust between the enclaves using local attestation of Intel CPUs, and negotiates TLS-secured, inter-enclave RPC streams for message passing.

Chapter 7

Performance Evaluation

This chapter evaluates the performance overheads of Graphene on two different host ABI implementations; one is an unmodified Linux kernel, and the other is inside of an SGX enclave running on an untrusted Linux host. The evaluation targets the following four aspects: (1) translation, isolation, and shielding costs of the host ABI and startup time; (2) emulation overheads of the library OS on system calls latency and throughput; (3) end-to-end performance of sample applications; (4) resource costs, including both memory footprint and CPU occupancy. The evaluation compares experiment results between Graphene, Graphene-SGX, and native processes running on an unmodified, generic Linux kernel, to show the cost of leveraging a library OS for compatibility and security isolation.

Experimental setup. All experiment results are collected on a Dell Optiplex 790 Small-Form Desktop, with a 4-core 3.2GHz Intel Core i5-6500 CPU (no hyper-threading, with 256KB I-Cache and D-Cache, 1MB L2 Cache, and 6MB L3 Cache), two 4GB DIMM 1600MHz DDR3 RAMs (8GB in total), and a Seagate 512GB, 7200 RPM SATA disk formatted as EXT4. The Intel CPUs are configured with SGX (Software Guard Extensions, requires BIOS update) and EPT (Extended Page table) enabled, and 128MB enclave page cache (EPC) size. To prevent fluctuated benchmarking results, Turbo Boost, SpeedStep, and CPU idle states are disabled for all experiments. All networked servers are evaluated over an 1Gbps Ethernet card connected to a dedicated local network.

All experiments are evaluated upon Graphene v0.4[1]. The host OS for evaluating the Linux

[1]Graphene is released at `https://github.com/oscarlab/graphene`

and SGX ports is Ubuntu 16.04.4 LTS Server with Linux kernel 4.10, which is also the baseline for comparison. All test programs and applications are dynamically linked with a modified glibc 2.19. The experiments for Graphene-SGX use the Intel SGX Linux SDK [96] and driver [95] v1.9.

7.1 The PAL ABI performance

The section evaluates the performance of the PAL ABI on each host. As a baseline, the latency or throughput of a PAL call is always bound by the performance of the underlying host system interface. For instance, the Linux PAL implements `StreamRead()` using the `read()` system call, and thus introduces the same performance patterns to the PAL call. However, the performance of the PAL ABI and the underlying host system interface is unlikely to be equivalent, due to the cost of translating between the two system interfaces when they have different semantics. Moreover, a host like SGX imposes compatibility challenges to applications, which the PAL must address. Part of the overheads on the SGX PAL subject to enclave exits, since enclave code cannot invoke system calls directly. The SGX PAL also suffer memory access overheads when bringing memory into the EPC (enclave page cache) or decrypting memory on a last-level cache miss.

More significant overheads on a few PAL calls contribute to security checks or enforcements, to protect applications inside host-specific threat models. For example, the threat model on a Linux host focuses on the attacks between mutually-untrusting applications via system interfaces; therefore, the security checks on the Linux PAL restrict the sharing of host resources and block system calls that are not required by the Linux PAL. In another threat model, with the SGX enclave, security checks for each PAL call protects the application and library OS against malicious inputs from an untrusted OS, using either cryptographic techniques or semantic checks. On the SGX PAL, the latency of a PAL call may be dominated by security checks, especially the ones based on cryptographic operations.

The evaluation in this section is based on micro-benchmark programs similar to LMbench 2.5 [131]. For each PAL calls, the evaluation also shows the breakdowns of its latency or throughput, by benchmarking the PALs both with and without the security mechanisms, such as the seccomp filter and reference monitor on the Linux PAL, as well as testing under different imple-

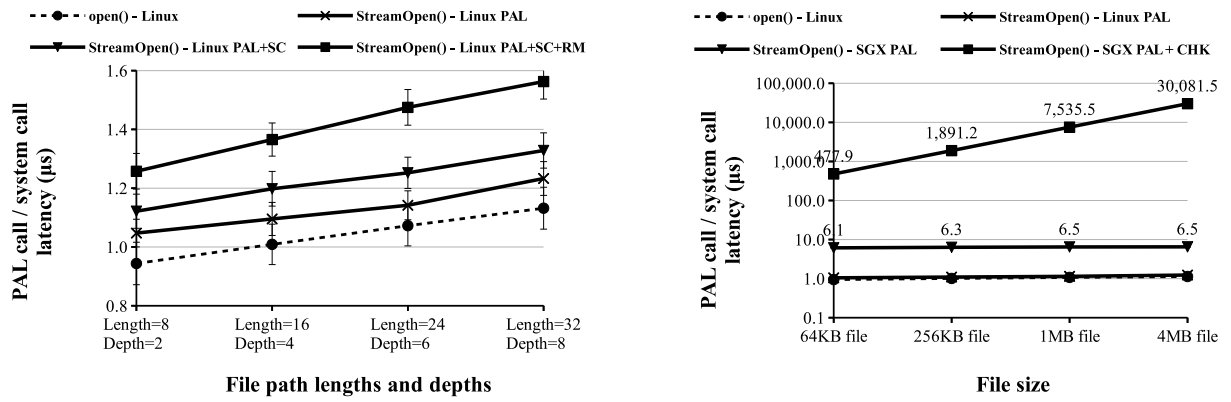
mentation strategies. The evaluation focuses on PAL calls that are especially sensitive for the performance of the Graphene library OS.

7.1.1 Stream I/O

This section separate the evaluation of stream I/O in the PAL ABI into two categories: (1) file system operations and (2) I/O operations on a network socket or a RPC stream. The evaluation for file system operations primarily measures the latency of retrieving file metadata from the storage or a host kernel file system directory cache, as well as the latency of sequential reads or writes inside a regular file. The evaluation for other I/O streams, such as a network socket or a RPC stream, then focuses on measuring the latency or bandwidth of sending and receiving messages across picoprocesses or applications.

Opening a file. Similar to `open()` in a Linux process, the latency of `StreamOpen()` to open a host file is correlated with the length and depth of file paths request, as shown in Figure 7.1 (a). The experiment measures the latency of repeatedly opening a file, which does not include the latency of retrieving the file attributes from the disk. Without disk I/O cost, the latency of `open()` in a native Linux process is dominated by lookup time inside the Linux file system directory cache, and is proportional to the number of components in the path [172]. The latency of `StreamOpen()` on the Linux PAL includes the latency of `open()`, with additional cost for translating a URI to the corresponding file path. The benchmark result shows this overhead to be around 6–10%. The seccomp filter, with the BPF JIT (Just-in-time) optimization, adds an additional $\sim$0.9 μs, or 7–10% overhead to the latency. Finally, enabling the reference monitor adds 14-21% overhead. The overhead of reference monitor contributes to comparing the file path against the sandbox rules inside the kernel module, and thus is correlated with path length.

Opening a file on the SGX PAL imposes significant overheads for verifying the integrity of file contents. Figure 7.1 (b) shows the latency of `StreamOpen()` inside of an SGX enclave, versus the latency on the Linux PAL and in a native Linux process. Without any security checks to shield the guest from the untrusted OS, the latency of `StreamOpen()` is dominated by the overhead of exiting the enclave and copying the argument, such as the file paths, out of the enclave. The overheads of unshielded `StreamOpen()` is 4.7–5.5$\times$, or $\sim$5 μs. If a file is shielded with integrity



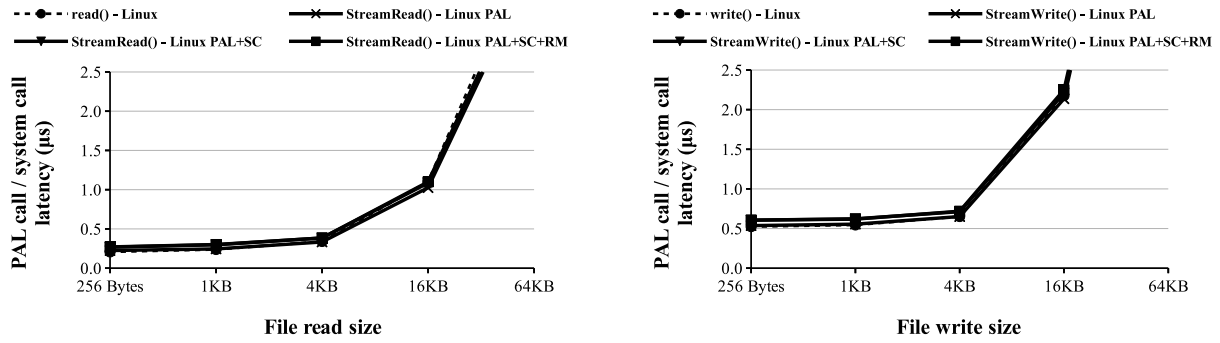
(a) Linux vs. Linux PAL **(b) Linux vs. Linux PAL vs. SGX PAL**

Figure 7.1: Latency of `StreamOpen()` on the Linux PAL and SGX PAL, versus `open()` on Linux. Lower is better. Figure (a) compares `StreamOpen()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**), against `open()` on Linux. Figure (b) compares `StreamOpen()` on a SGX PAL, with and without integrity checks (**+CHK**), against the Linux PAL and `open()` on Linux.

protection, `StreamOpen()` will verify the checksum of the whole file against the manifest, and generate a Merkel Tree of file chunk hashes for optimizing the latency of following `StreamRead()` or `StreamMap()`. The overhead of enforcing the integrity check is correlated with the file size, and dominated by the time of calculating a SHA256 hash of the file. For a 4MB file, the latency of `StreamOpen()` can be up to $\sim$30 ms.

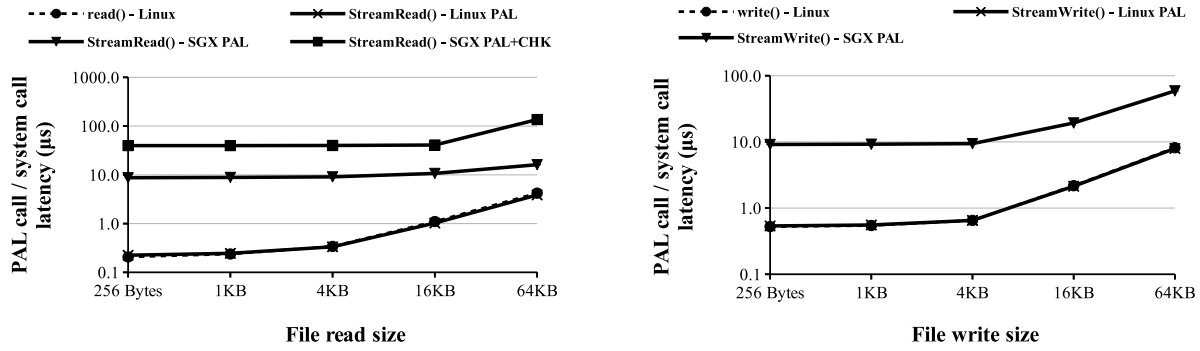
File reads and writes. The latency of file reads and writes on the Linux PAL is close to `read()` and `write()` in a Linux process. Figure 7.2 (a) and (b) compare the latency of sequential reads and writes on the Linux PAL and Linux, and show almost no overheads on the Linux PAL. The seccomp filter adds a fixed overhead around 0.06–0.09 ms, which is marginal to the overall latency. Enabling the reference monitor has nearly no overheads, since the reference monitor only checks file paths at `StreamOpen()`.

On the SGX PAL, as shown in Figure 7.3 (a) and (b), both sequential reads and writes have significant overheads over the latency on Linux or the Linux PAL. The overheads contribute to: (1) copying the contents between the enclave and the untrusted PAL; (2) cryptographic operations for integrity checks. Without any integrity checks, the cost of exiting the enclave and copying the contents across the enclave boundary is 8–12 μs for reads and 8–50 μs for writes. If the SGX PAL checks the integrity of file contents at file reads, the latency is bounded by the cost of copying 16KB blocks into the enclave and calculating the secure hashes to compare with the



(a) Sequential read **(b) Sequential write**

Figure 7.2: Latency of sequential `StreamRead()` and `StreamWrite()` on the Linux PAL, versus `read()` and `write()` on Linux. Lower is better. Figure (a) and (b) respectively compares `StreamRead()` and `StreamWrite()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**), against `read()` and `write()` on Linux.



(a) Sequential read **(b) Sequential write**

Figure 7.3: Latency of sequential `StreamRead()` and `StreamWrite()` on the SGX PAL, versus the Linux PAL and Linux. Lower is better. Figure (a) and (b) respectively compares `StreamRead()` and `StreamWrite()` on the SGX PAL, with and without integrity checks (**+CHK**) and reference monitor (**+RM**), against the Linux PAL and `read()` and `write()` on Linux. The current design does not support integrity checks for `StreamWrite()`.

Merkle tree. Reducing the hashing size from 16KB to even smaller block will improve the latency of small reads, but also will increase the size of Merkle tree. The experiment does not measure the overhead of integrity protection on file writes because the feature is not yet implemented in the SGX PAL.

TCP and UDP sockets. The Linux PAL imposes ∼18% and ∼30% overheads on the latency of TCP and UDP sockets (bound on localhost), respectively, as shown in Figure 7.4 (a). The overheads specifically contribute to cost of translating to `sendmsg()` and `recvmsg()` in the Linux host, which requires passing a data structure containing buffer pointer, size, and a socket address

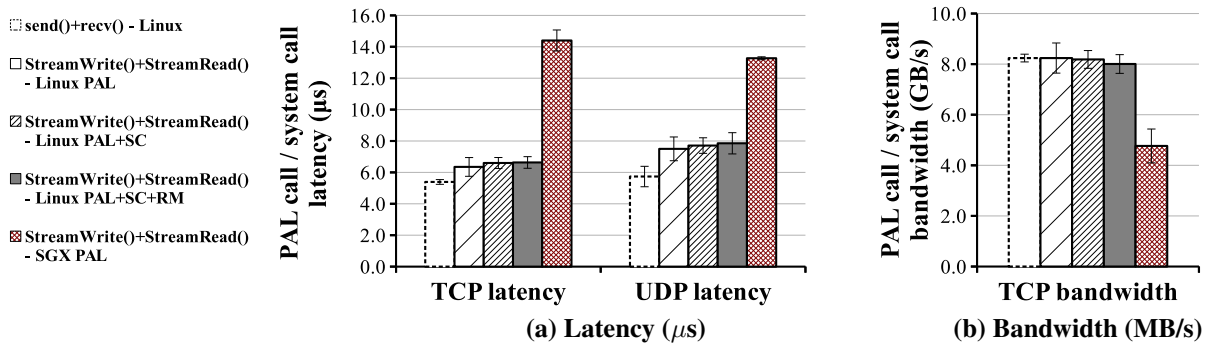


Figure 7.4: (a) Latency of sending a short message over TCP and UDP sockets (lower is better), and (b) bandwidth of sending large data over TCP (higher is better). The comparison is between (1) `recv()` and `send()` on Linux; (2) `StreamRead()` and `StreamWrite()` on a Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same PAL calls on the SGX PAL, without data protection.

for UDP messaging. The UDP socket address is also checked by the reference monitor if enabled, adding extra overheads to the PAL calls. Figure 7.4 (b) also shows the bandwidth of TCP sockets for sending 64KB messages locally, which has only 4% overheads on the Linux PAL. In addition, both the seccomp filter and reference monitor cause less than 1% overheads on TCP bandwidth.

TCP and UDP sockets on the SGX PAL also have significant overheads compared to the Linux PAL, due to the cost of enclave interface. As shown in both Figure 7.4 (a) and (b), the overheads of the SGX PAL on TCP and UDP latency are $\sim$167% and $\sim$131%, respectively; the overhead on TCP bandwidth also reaches $\sim$79%. Note that the current design of the SGX PAL does not protect the messages sent or received over a TCP or UDP socket. The decision is based on previous work [40], which concludes that inline encryption and authentication inside applications is more efficient than shielding at the enclave interface or the system interface. Graphene-SGX makes the assumption that most networked applications have adopted SSL/TLS to protect the confidentiality and integrity of network payloads.

RPC latency and bandwidth. Due to the choice of underlying abstraction, both the latency and bandwidth of a RPC stream on the Linux PAL is close to a UNIX domain socket on native Linux kernel. Figure 7.5 (a) compares the latency of sending one-byte messages over a local RPC stream, a pipe, and a UNIX domain socket (the last two are both in a native Linux process). The results show that the latency of a UNIX domain socket is about twice as slow as a pipe, and close to the latency of a RPC stream on the Linux PAL, with or without the seccomp filter and reference

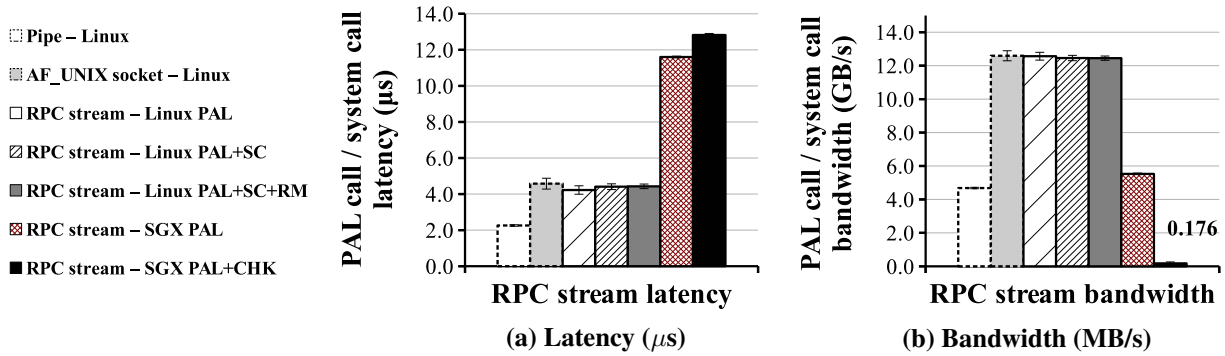


Figure 7.5: (a) Latency of sending a short message over RPC (lower is better), and (b) bandwidth of sending large data (higher is better). The comparison is between (1) `read()` and `write()` over a pipe or an AF_UNIX socket on Linux; (2) `StreamRead()` and `StreamWrite()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same PAL calls on the SGX PAL, with and without data protection (**+CHK**).

monitor. Figure 7.5 (b) also shows the bandwidth of messaging over a RPC stream, a UNIX domain socket, and a pipe, and first two reach bandwidth more than double of the pipe bandwidth. Both the seccomp filter and reference monitor introduce marginal overheads (less than 5%) to RPC latency and bandwidth. Note that this performance pattern is specific to kernels after 4.2; a zero-copy design for UNIX domain socket is adopted in Linux 4.2, and makes a UNIX domain socket almost equally fast as the bulk IPC abstraction in the Linux PAL (see Section 7.1.4).

For the SGX PAL, the fundamental cost of enclave exits and copying message contents, without protecting the message contents, is $\sim$154% to the latency, or $\sim$127% to the bandwidth. Unlike network sockets, applications generally assume transferring data over a local pipe or FIFO to be secure on a trusted host, and do not enforce protection like SSL/TLS. For each RPC stream, the SGX PAL establishes a TLS connection using a 256-bits AES-GCM algorithm, which both authenticates and encrypts the message contents. The AES-GCM algorithm is accelerated by the Intel AES-NI instructions, which exist on all SGX-enabled CPUs. With the hardware acceleration, when sending one-byte messages, the overhead on RPC latency is $\sim$10% compared to the UNIX domain socket; furthermore, RPC bandwidth of sending large messages is reduced by $\sim$31$\times$, at $\sim$0.176 GB/s. Switching to a more efficient cryptographic algorithm or library may improve the efficiency of RPC streams, and such experiments are left for future work.

Summary. According to the evaluation, the overheads on accessing a file or an I/O stream with one of the PAL design contribute to translation cost of the PAL ABI semantics and the reference

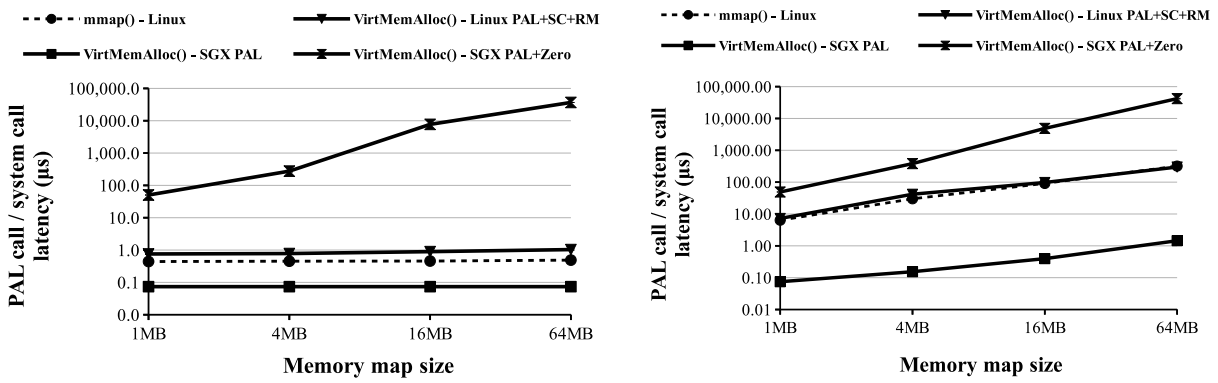
monitor. Because a file or an I/O stream is shareable among picoprocesses, the reference monitor must check the access, at least at opening the file or the I/O stream.

Security checks, either in the host kernel or inside an enclave, often contribute to non-trivial overheads on the PAL calls for accessing I/O streams. The cost of security checks varies between different threat models. For the Linux PAL, the cost includes the overhead of enabling a seccomp filter, and the cost of checking file paths and network addresses inside of an reference monitor. With the JIT (Just-in-time) optimization, the overheads of seccomp filter is generally less than 10%. The overheads of security monitors can range from 0–21%, but only impact PAL calls which accept an URI as argument (e.g., `StreamOpen()` and `StreamAttrQuery()`).

The SGX PAL further adopts several cryptographic techniques for protecting the confidentiality and integrity of I/O streams, and cryptographic operations tend to cause significant overheads. Verifying the file contents, either at first open of the file or consequential file reads, causes 500–24,000× overheads on `StreamOpen()` or 25–150× overheads on `StreamRead()`. Authenticating and encrypting a RPC stream with a hardware-accelerated AES-GCM algorithm causes up to 335% overheads on latency in comparison with the underlying UNIX domain sockets, or up uo $\sim$20 $\times$ overhead on bandwidth.

7.1.2 Page management

This section evaluates the performance of virtual page allocation and deallocation using the PAL ABI. The latency of creating or destroying a virtual memory area may differ among PAL implementations due to different memory management models. The Linux PAL, for instance, has a memory management model similar to Linux, backed by demand paging. The SGX PAL, on the other hand, has to preallocate all pages at loading time. Although the SGX driver still swaps enclave pages, the latency of page-in and page-out can be up to 40,000 cycles [136], and the current SGX hardware does not allow dynamically adding pages to an enclave. As a result, the SGX PAL includes an in-enclave memory allocator which manages unused enclave pages and assigns the pages to memory allocation requests. To evaluate the differences of memory management models, the experiments include two scenarios: (1) simply allocating and deallocating virtual pages; (2)



(a) allocation + deallocation **(b) allocation + memory access + deallocation**

Figure 7.6: Latency of (a) allocating and deallocating a range of virtual pages, and (b) the same operations with writing to each page after allocation. Lower is better. The comparison is between (1) `mmap()` and `munmap()` on Linux; (2) `VirtMemAlloc()` and `VirtMemFree()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same PAL calls on the SGX PAL, with and without zeroing the pages before use (**+Zero**).

allocating virtual pages, accessing the beginning of every pages, and then deallocating the pages. For each scenario, the benchmark tests with different memory mapping sizes.

Figure 7.6 (a) shows the latency of simple allocation and deallocation. Linux and the Linux PAL show similar latency for allocating and deallocating any numbers of virtual pages, as the cost of updating the page table in the host kernel. For the Linux PAL, the latency of `VirtMemAlloc()` and `VirtMemFree()` is 50–80% higher than `mmap()` and `munmap()` on Linux (the overheads of seccomp filter and reference monitor is negligible). For the SGX PAL, the latency of allocation and deallocation is much lower, at $\sim$10% of the latency of `mmap()` and `munmap()`. However, since SGX doesn't guarantee the initial state of unmeasured enclave pages, the SGX PAL needs to zero all pages before passing to the guest. Accessing the pages after allocation may cause cache misses and enclave page swapping, when the allocation size is larger than the size of last-level cache. The result shows that, if the SGX PAL zeros pages after allocation, the latency of allocation and deallocation can increase by up to several orders of magnitude.

If the latency includes accessing every pages in between allocation and deallocation, as shown in Figure 7.6 (b), the overall latency (including allocating, access, and deallocating pages) would naturally be proportional to the allocation size. The benchmark results show almost no difference between the latency on the Linux PAL and in a native Linux process. Because the benchmark only access the first words of every pages, the latency on the SGX PAL without page zeroing

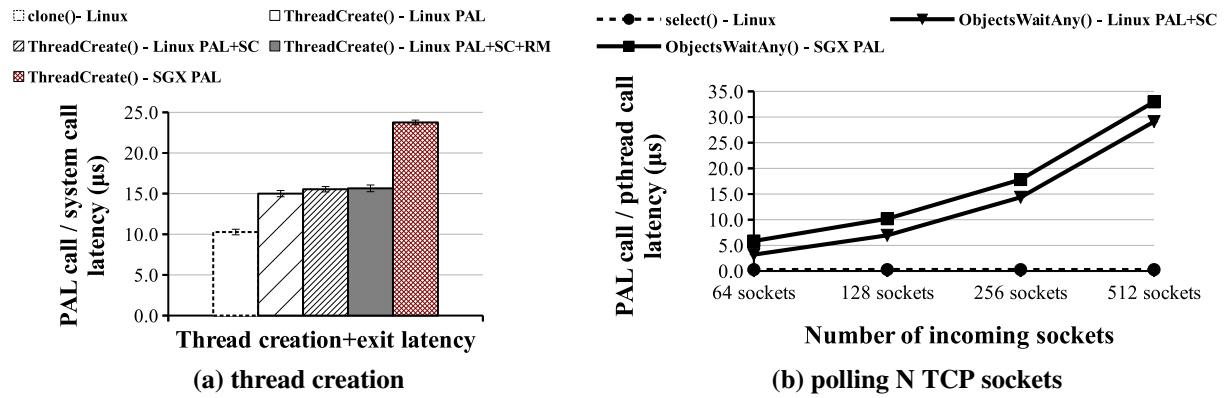


Figure 7.7: (a) Thread creation latency and (b) latency of polling a number of TCP sockets. Lower is better. The comparison is between (1) `clone()` and `select()` on Linux; (2) `ThreadCreate()` and `ObjectsWaitAny()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same PAL calls on the SGX PAL.

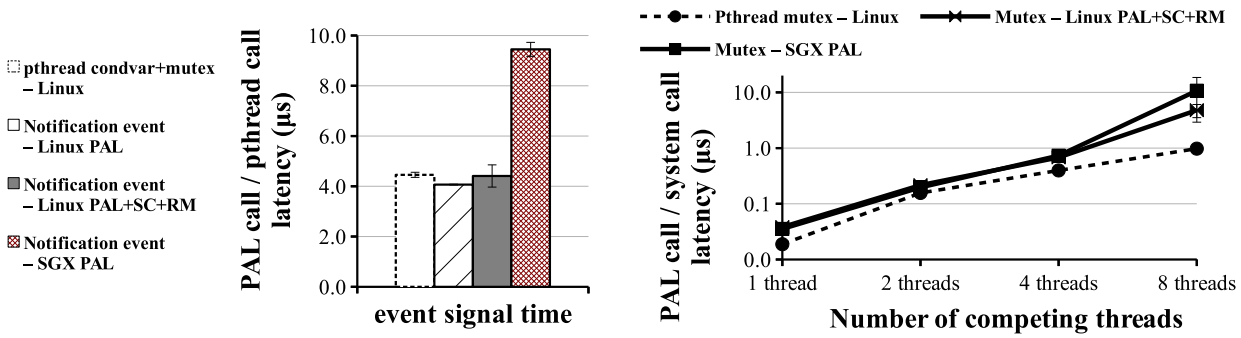
is still lower than the Linux PAL. If the SGX PAL zeros the pages after allocation, accessing the first words of every pages causes no additional overheads on the overall latency.

7.1.3 Scheduling

This section evaluates several PAL calls for scheduling multiple threads in a picoprocess, to create a new thread, poll an I/O stream (a TCP socket for instance), and use a synchronization primitive, such as a notification event or a mutex.

Thread creation. Figure 7.7 (a) shows the latency of thread creation and exit on the Linux and SGX PALs, in comparison with `clone()` (with `CLONE_VM`) and `exit()` in a native Linux process. The latency on the Linux PAL is ∼46% higher than Linux, and the overhead mainly contributes to allocating an initial stack for the new thread. The seccomp filter and reference monitor have small impact (less than 5%) on the latency of thread creation and exit.

The latency of thread creation on the SGX PAL further includes the cost of attaching a preallocated enclave thread. Inside an enclave, the number of concurrent threads is bound by the number of TCSes (thread control structure). To create a new thread, the SGX PAL has to walk the list of TCSes to find an unused enclave thread. As a result, the latency of thread creation and exit on the SGX PAL is ∼131% slower than Linux.



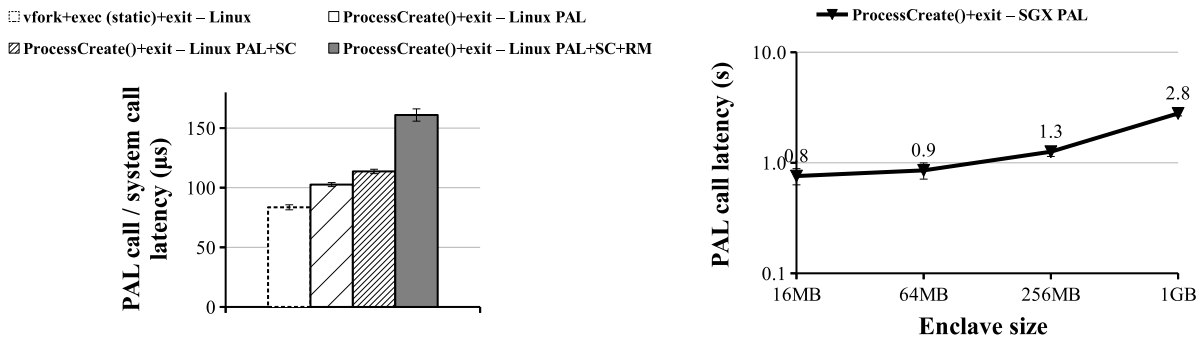
(a) signal an event **(b) competing a mutex among N threads**

Figure 7.8: Latency of (a) signaling an event and (b) competing a mutex among N threads (N: 1 to 8). Lower is better. The comparison is between (1) pthread condition variables and mutexes on Linux; (2) Notification events and mutexes on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same abstractions on the SGX PAL.

Polling TCP sockets. The latency of polling an array of TCP sockets for incoming traffic is proportional to number of TCP sockets. Figure 7.7 (b) compares the latency of `ObjectsWaitAny()` on 64 to 512 TCP sockets with `select()` in a native Linux process. The benchmark result shows that both the Linux and SGX PALs have significant overheads on polling TCP sockets, at up to 29.1μs and 31.8μs, respectively, for polling 512 sockets. The overheads mostly contribute to scanning the PAL handle arrays and retrieving the file descriptors for polling. Although not shown in Figure 7.7 (b), the overheads of the seccomp filter and reference monitor on the Linux PAL are negligible. The SGX PAL further adds a fixed cost to `ObjectsWaitAny()`, at $\sim$2.7μs, for exiting the enclave to poll the file descriptors.

Events and mutexes. Figure 7.8 (a) shows the latency of signaling notification events on the Linux and SGX PALs. For a native Linux process, the pthread library provide primitives similar to notification events in the PAL ABI, using a combination of conditional variables and mutexes. In fact, the latency of event signaling on the Linux PAL is slightly lower than updating a pthread conditional variable, with extra a $\sim$10% overhead if the seccomp filter and reference monitor is enabled. For the SGX PAL, the overhead on signaling an event The overhead on the SGX PAL is $\sim$130%, and mostly contributes to the cost of exiting the enclave to call `futex()`.

The latency of acquiring and releasing a mutex, as shown in Figure 7.8 (b), is not scalable when multiple threads access the same mutex. Within a single thread, acquiring and releasing a PAL mutex requires no `futex()` calls but simply updating a counter in the mutex handle. If the



(a) process creation+exit **(b) SGX enclave creation+termination**

Figure 7.9: Latency of creating (a) a clean process on the Linux PAL, and (b) an enclave on the SGX PAL, in respect of different enclave sizes. The comparison is between (1) a combination of `vfork()` and `exec()`'ing a minimal static program on Linux; (2) `ProcessCreate()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the same PAL call on the SGX PAL.

thread number is increased to two, the latency of acquiring and releasing a PAL mutex may still be low because the Linux PAL always spin for a few rounds to check if other thread has released the mutex. Afterward, if the mutex is still locked, the Linux PAL will block voluntarily, by calling `futex()` with `FUTEX_WAIT`. With eight threads trying to acquire the same mutex, the latency of acquring and releasing the mutex is up to $\sim$10μs, or 286% upon the latency of a pthread mutex.

7.1.4 Multi-process abstractions

This section evaluates the performance of two multi-process abstractions in the PAL ABI: process creation and bulk IPC.

Process creation. Process creation is a relatively expensive operation in either one of the PAL implementation. Instead of adopting copy-on-write style, fork-like semantics, the PAL ABI only creates clean processes to avoid the requirement of page sharing. Therefore, the latency of creating a process, on either the Linux or SGX PAL, includes the cost of initializing the PAL binary in a new process or enclave. The following benchmarks evaluate the slowdown of process creation in the PAL ABI compared with process creation on a Linux kernel using `vfork()` and `execve()`. For accuracy, the benchmarks measure the wall time from creating a new process to receiving the exit notification of the process. The binary being loaded by `execve()` is a statically compiled program which returns immediately.

According to the results shown in Figure 7.9 (a), the latency of process creation and exit on the Linux PAL is ∼22% slower than the latency of similar operations on a Linux kernel. The initialization time of the Linux PAL contributes a large portion to this overhead, besides the minor cost of establishing the RPC stream between the parent and child processes (using an unnamed UNIX domain socket). The Linux PAL with the seccomp filter installed subjects to the same overheads, except in order to force the PAL binary always loaded at the same address, a separate security loader starts in each new process at a low, static address to load the PAL binary afterwards. The overhead with the seccomp filter installed is ∼26%, including the system call slowdown and the extra cost of reloading the PAL loader in each new process. If the reference monitor is enabled, the reference monitor must attach the new process to the sandbox inside the Linux kernel, and always check that the new process only loads the security loader instead of any other binaries. The overhead with both the seccomp filter and the reference monitor is ∼93%.

For the SGX PAL, the latency of creating a new enclave for each child process is significantly higher than creating a normal picoprocess. Figure 7.9 (b) shows that the latency of enclave creation is related, but not proportional, to the enclave size configured by users. To create a 16MB enclave, which is the minimum size to load a functioning Graphene process, takes ∼0.8s. If the enclave size is set at 1GB, the creation time can be up to ∼2.8s, and will cause significant slowdown to the guest if the guest frequently spawns processes. Although some techniques such as preallocating empty enclave may reduce the enclave creation time, this thesis leaves such experiment as future work. Furthermore, the SGX version 2 hardware with support dynamic memory management, and eliminate the need of preserving large heap space in enclave

Bulk IPC vs RPC streams. The PAL ABI introduces a bulk IPC feature to improve the communication between processes. Bulk IPC is an optional feature, and is assumed to have significantly higher bandwidth than a regular RPC stream by sharing the pages as copy-on-write in another process. The evaluation results in Figure 7.10 show that the benefit of using bulk IPC over a RPC stream subjects to different host Linux kernel versions. On a Linux kernel earlier than 4.2, such as the 3.19 kernel evaluated in Figure 7.10, the RPC stream has much lower bandwidth (rough32%) than Linux kernel 4.10. The reason of the slowdown is due to the introduction of a zero-copy design of UNIX domain sockets in 4.2. The zero-copy design allows a UNIX domain socket to

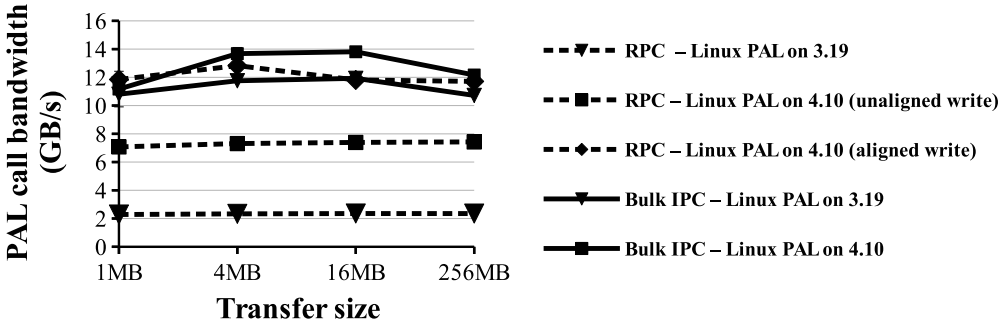


Figure 7.10: Bandwidth of sending large messages over (a) RPC streams and (b) Bulk IPC channels. The messages are sent in different sizes (1MB to 256MB), and either aligned or unaligned with the page boundary. Higher is better. Both abstractions are benchmarked on Linux kernel 3.19 and 4.10 as the hosts. The impact of the seccomp filter or reference monitor is marginal (less than 1%).

share the physical pages of a page-aligned buffer with the destination process, creating the same effect as the bulk IPC feature in the PAL ABI. Therefore, using the bulk IPC is only beneficial on a Linux kernel older than 4.2.

7.1.5 Exception handling

This section evaluates the performance of handling an exception from the host OS or hardware, including installing an exception handler, interrupting a running thread with `ThreadInterrupt()` (raising an `INTERRUPT` exception in the target thread), and catching a hardware protection fault (writing to a read-only address).

Figure 7.11 (a) shows the latency of installing an exception handler in either the Linux or SGX PAL, compared with `sigaction()` in a Linux process. The result shows that regardless the PAL is running in a regular picoprocess or an enclave, installing an exception handling in the PAL is much lower than install a signal handler on Linux, because the PAL call does not have to take any system call or enclave exits.

Figure 7.11 (b) shows the latency of interrupting a running thread and raising an exception in the target thread, compared with delivering a `SIGUSR1` signal in a Linux process. Thread interruption on the Linux PAL is slightly slower than delivering a `SIGUSR1` signal, by $\sim$25% without seccomp filter and reference monitor, or $\sim$32% with seccomp filter and reference monitor. On the SGX PAL, the latency of thread interrupt is much higher, due to the cost of extra thread exits.

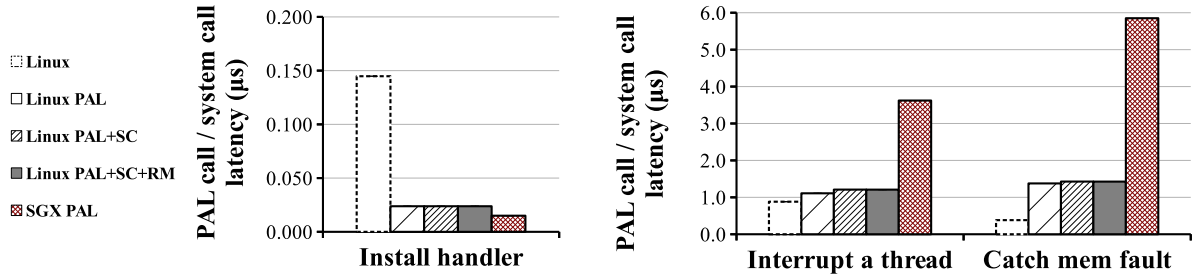


Figure 7.11: Latency of (a) installing an exception handler; (b) interrupting a running thread with signals (on Linux) or `ThreadInterrupt()` on the PALs; (c) catching a memory protection fault. Lower is better. The comparison is between (1) signals on Linux; (2) the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**); (3) the SGX PAL.

Interrupting a thread running on the SGX PAL is $\sim$3.1$\times$ shower than delivering a `SIGUSR1` signal in a Linux process.

Figure 7.11 (b) shows the latency of catching a memory protection fault, compared with handling a `SIGSEGV` signal in a Linux process. On the Linux, handling a hardware fault is $\sim$2.6$\times$ slower than handling a `SIGSEGV` signal on Linux; such an overhead is primarily caused by delivering the exception information to the guest, including the faulting register contexts. Figure 7.11 (b) also shows that catching a hardware fault inside an enclave is even more expensive than thread interruption, by nearly 60%. The reason of such a dramatic difference is that interrupting a thread only causes the interrupt thread to exit and enter the enclave once, whereas handling a hardware fault requires the faulting thread to exit and enter the enclave twice, in order to reset the in-enclave buffer for preserving the faulting contexts.

7.2 Library OS overheads

This section evaluates the cost of emulating Linux system calls inside of the Graphene library OS, or `libLinux`. The evaluation is based on benchmark results of Linux system calls inside a picoprocess, since the Linux system call table is the main target that `libLinux` emulates as a guest. The Linux system calls performance on `libLinux` is affected by the emulation strategies or the underlying host abstractions (i.e., PAL calls) that `libLinux` selects to implement system calls. This section shows the emulation overhead of `libLinux` by comparing the benchmark results of

Linux system calls in a picoprocess with the native system call performance in a Linux process and the related PAL calls, as evaluated in Section 7.1. Finally, the evaluation shows that with the right emulation strategies, `libLinux` can achieve acceptable performance at the Linux system call level, which also contributes as an important factor to application performance.

7.2.1 Single-process system calls

Table 7.1 lists a representative sample of tests from LMbench 2.5 benchmark suite [131] (extended with additional experiments). Each row reports a mean and 95% confidence interval, assuming the benchmark results are normally distributed; to improve the precision, the number of iterations in each test is increased to at least a thousand times, which effectively lower the variance in most tests. Although assuming a normal distribution may not be realistic for most benchmark results, the error is likely to be marginal with the very low variance observed in the tests. Besides, to measure the marginal cost of the seccomp filter and reference monitor on a Linux host, the experiments include the cases both with and without the seccomp filter and reference monitor.

The evaluation results categorize the system calls emulated inside `libLinux` as three types. The first type of system calls is completely serviced inside `libLinux`; the evaluation results show that these system calls are even faster than native, because `libLinux` does not switch context into the kernal space to service the system calls. For instance, a null system call (i.e., `getppid()`) and installing a signal handler with `sigaction()` are up to three time as fast as the native performance. The second type of system call requires translation to a native host system calls.

7.2.2 Process creation

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

	System call latency (μs), +/- Confidence Interval, % Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	μs	+/-	μs	+/-	*%O*	μs	+/-	*%O*	μs	+/-	*%O*
simple syscall	0.045	0.000	0.015	0.000	*-75*	0.015	0.000	*-75*	0.015	0.000	*-75*
static syscall	0.045	0.000	Not supported			1.155	0.000	*2,775*	5.800	0.001	*14,400*
file read	0.116	0.000	0.120	0.000	*0*	0.120	0.000	*0*	0.115	0.000	*0*
file write	0.078	0.000	0.118	0.000	*50*	0.118	0.000	*50*	0.112	0.000	*38*
file stat	0.399	0.000	1.154	0.000	*188*	1.164	0.000	*190*	1.144	0.000	*185*
file fstat	0.120	0.000	0.193	0.001	*58*	0.192	0.000	*58*	0.190	0.000	*58*
file open/close	0.976	0.063	2.680	0.008	*173*	3.025	0.008	*208*	17.152	0.016	*1,650*
select file (100)	0.965	0.000	2.511	0.001	*159*	2.505	0.000	*159*	2.505	0.000	*159*
select file (250)	1.983	0.000	5.941	0.002	*200*	5.939	0.001	*200*	5.953	0.002	*201*
select file (500)	3.800	0.001	11.739	0.007	*209*	11.567	0.002	*204*	11.589	0.002	*205*
select tcp (100)	2.148	0.000	2.828	0.001	*32*	2.926	0.001	*36*	8.039	0.002	*274*
select tcp (250)	4.977	0.001	6.254	0.001	*26*	6.361	0.003	*28*	11.577	0.002	*133*
select tcp (500)	9.767	0.002	11.899	0.010	*22*	11.985	0.002	*23*	17.695	0.164	*81*
sighandler install	0.146	0.000	0.113	0.000	*-27*	0.113	0.000	*-27*	0.110	0.000	*-27*
`SIGUSR1`	0.895	0.000	0.189	0.000	*-79*	0.187	0.000	*-79*	0.178	0.000	*-80*
`SIGSEGV`	0.379	0.000	1.526	0.000	*303*	1.575	0.000	*316*	6.117	0.000	*1,511*
pipe write/read	2.412	0.290	4.440	0.464	*84*	4.491	0.138	*86*	12.895	0.398	*435*
AF_UNIX	4.386	0.293	5.587	0.048	*27*	5.824	0.044	*33*	12.425	0.050	*183*
UDP socket (local)	6.300	0.708	9.451	0.235	*50*	9.938	0.219	*58*	17.538	0.703	*178*
TCP socket (local)	7.217	0.505	9.422	0.203	*30*	10.075	0.209	*40*	17.925	0.001	*148*

	System call bandwidth (MB/s), +/- Confidence Interval, % Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	MB/s	+/-	MB/s	+/-	*%O*	MB/s	+/-	*%O*	MB/s	+/-	*%O*
TCP socket (local)	7,465	38	7,011	52	*6*	6,932	64	*8*	4,242	2	*76*
AF_UNIX	12,643	419	12,179	85	*4*	12,173	306	*4*	5,333	103	*137*
pipe write/read	4,791	295	12,262	204	*-61*	12,151	130	*-61*	5,290	48	*-9*

	System call throughput (operations/s), +/- Confidence Interval, % Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	ops/s	+/-	ops/s	+/-	*%O*	ops/s	+/-	*%O*	ops/s	+/-	*%O*
file create (0KB)	151,819	734	122,526	343	*24*	116,195	205	*31*	40,471	248	*275*
file delete (0KB)	247,750	1,048	133,397	424	*86*	120,683	138	*105*	37,706	127	*557*
file create (4KB)	154,318	21	83,880	201	*84*	73,797	993	*109*	21,989	37	*602*
file delete (4KB)	250,097	461	109,782	504	*128*	101,480	480	*146*	35,355	14	*607*
file create (10KB)	102,749	90	64,693	134	*59*	62,891	72	*63*	18,194	6	*465*
file delete (10KB)	186,029	458	93,833	232	*98*	89,493	129	*108*	33,368	94	*458*

Table 7.1: System call benchmark results based on LMbench 2.5. Comparison is among (1) native Linux processes, (2) Graphene picoprocesses on Linux host, both without and with JIT-optimized SECCOMP filter (**+SC**) and reference monitor (**+RM**), and (3) Graphene in SGX enclaves. System call latency is in microseconds, and lower is better. System call bandwidth and throughput are in megabytes per second and operations per second, respectively, and higher is better. Overheads are relative to Linux 4.10; negative overheads indicate improved performance.

	System call latency (μs), +/- Confidence Interval, % Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	μs	+/-	μs	+/-	*%O*	μs	+/-	*%O*	s	+/-	*O*
fork/exit	193	6	1,004	22	*421*	1,050	25	*444*	1.227 s	0.069 s	*6,360* ×
double fork/exit	562	18	2,241	74	*299*	2,325	28	*314*	2.423 s	0.086 s	*4,314* ×
vfork/exit	456	10	1,305	288	*186*	1,352	239	*197*	1.156 s	0.056 s	*2,536* ×
fork/execve	610	28	1,470	12	*141*	1,570	14	*158*	1.282 s	0.067 s	*2,102* ×
double fork/execve	979	23	2,837	176	*190*	2,932	63	*199*	2.311 s	0.107 s	*2,359* ×

Table 7.2: Benchmark results of various combinations of `fork()`, `vfork()`, and `execve()`, based on LMbench 2.5. Comparison is among (1) native Linux processes, (2) Graphene picoprocesses on Linux host, both without and with JIT-optimized SECCOMP filter (**+SC**) and reference monitor (**+RM**), and (3) Graphene in SGX enclaves. Latency is in microseconds, except for Graphene-SGX, which is orders-of-magnitude slower. Lower latency is better. Overheads are relative to Linux 4.10; negative overheads indicate improved performance.

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 7.2), 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.

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.

Figure **??**(c) shows the overhead of forking a process. As described in Section 6.3.3, the latency of `fork()` in Graphene-SGX is affected by three factors: creation of a new enclave, local attestation of the integrity, and duplicating the process state over an encrypted RPC stream. Combining these factors, `fork()` is one of the most expensive calls in Graphene-SGX. The default enclave size is 256MB. Our evaluation shows that the latency of forking a process is around 0.8s (16MB process) to 2.7s (128MB process), but can be more expensive if the parent process uses more memory. The trend matches the performance of Graphene without the bulk IPC optimization.

We also measure the overhead of isolating a Graphene picoprocess inside the reference 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`,

Test		Linux		Graphene+SC+RM		
		μS	+/-	μS	+/-	
msgget	in-process	3320	0.7	2823	0.3	-15 %
(create)	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 7.3: 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.

result in substantial overheads relative to Graphene. An efficient implementation of an environment similar to FreeBSD jails [167] would make all reference monitoring overheads negligible.

7.2.3 Multi-process abstractions

Table 7.3 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 [130]. 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.

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

Table 7.4: 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.

7.3 Application performance

7.3.1 Process Migration and Application 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 7.4 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.

7.3.2 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`—§7.3.3), 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) [39]. 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.

7.3.3 Application performance

Table 7.6 lists execution time of our *unmodified* application benchmarks (detailed in §7.3.2). 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 [24], 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`).

	Execution time (s), +/- Conf. Interval, % Overhead										
GCC/make	**Linux**		**in KVM**			**Graphene + SC + RM**			**Graphene + SGX**		
gcc (helloworld)	0.020	.000	0.022	.000	7 %	0.023	.000	12 %	6.097	.012	298 ×
gcc (.7MLoC)	21.64	.00	23.37	.02	8 %	21.88	.00	1 %	75.41	.03	248 %
make bzip2	2.332	.000	2.448	.000	5 %	2.407	.000	3 %	not supported yet		
make -j4 bzip2	0.888	.000	0.967	.000	9 %	0.923	.004	4 %	not supported yet		
make libLinux	4.832	.000	5.037	.000	4 %	5.112	.001	6 %	not supported yet		
make -j4 libLinux	1.361	.000	1.413	.000	4 %	1.459	.000	7 %	not supported yet		
Ap. Bnch	Avg Throughput (MB/s), +/- Conf. Interval, % Overhead										
Apache	**Linux**		**in in KVM**			**Graphene + SC + RM**			**Graphene + SGX**		
25 conc	6.282	.005	5.327	.031	18 %	4.586	.002	36 %	2.312	.192	171 %
50 conc	6.305	.002	5.420	.060	16 %	4.555	.012	38 %	2.752	.463	129 %
100 conc	6.347	.010	5.152	.036	22 %	4.572	.009	39 %	2.488	.781	155 %
Lighttpd	**Linux**		**in KVM**			**Graphene + SC + RM**			**Graphene + SGX**		
25 conc	6.66	.01	6.46	.03	3 %	5.65	.00	18 %	2.124	.072	171 %
50 conc	6.65	.13	6.41	.02	4 %	4.79	.00	39 %	2.437	.107	171 %
100 conc	6.69	.01	6.39	.03	5 %	4.56	.01	47 %	2.417	.235	171 %
	Execution Time (s), +/- Conf. Interval, % Overhead										
bash	**Linux**		**in KVM**			**Graphene + RM**			**Graphene-SGX**		
Unix utils	0.87	.00	1.10	.01	26 %	2.01	.00	134 %	not supported yet		
Unixbench	0.55	.00	0.55	.00	0 %	1.49	.00	192 %	not supported yet		

Table 7.5: 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**), and (3) Graphene in SGX enclaves.

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

		Execution time (s), +/- Confidence Interval, % Overhead								
Test		**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
		s	+/-	s	+/-	*%O*	s	+/-	*%O*	s	+/-	*%O*
GCC	helloworld.c (5LoC)											
	gzip.c (5kLoC)											
	oggenc.c (50kLoC)											
	gcc.c (0.5MLoC)											

		Execution time (s), +/- Confidence Interval, % Overhead								
Test		**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
		s	+/-	s	+/-	*%O*	s	+/-	*%O*	s	+/-	*%O*
Python	helloworld											
	fibonacci											
	objects											
	http download											
R	matrix mult.											
	sorting											
	linear regr.											
	FFT											
	eigen											
	fibonacci											
	hilbert											

		Server bandwidth (MB/s), +/- Confidence Interval, % Overhead								
Test		**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
		MB/s	+/-	MB/s	+/-	*%O*	MB/s	+/-	*%O*	MB/s	+/-	*%O*
Lighttpd	25 conc											
(25-thread)	50 conc											
	100 conc											
NGINX	25 conc											
(event-driven)	50 conc											
	100 conc											
Apache	25 conc											
(25-thread)	50 conc											
	100 conc											
Apache	25 conc											
(5-proc)	50 conc											
	100 conc											

Table 7.6: 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**).

and Apache on KVM are primarily attributable to bridged networking.

Table 7.6 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.

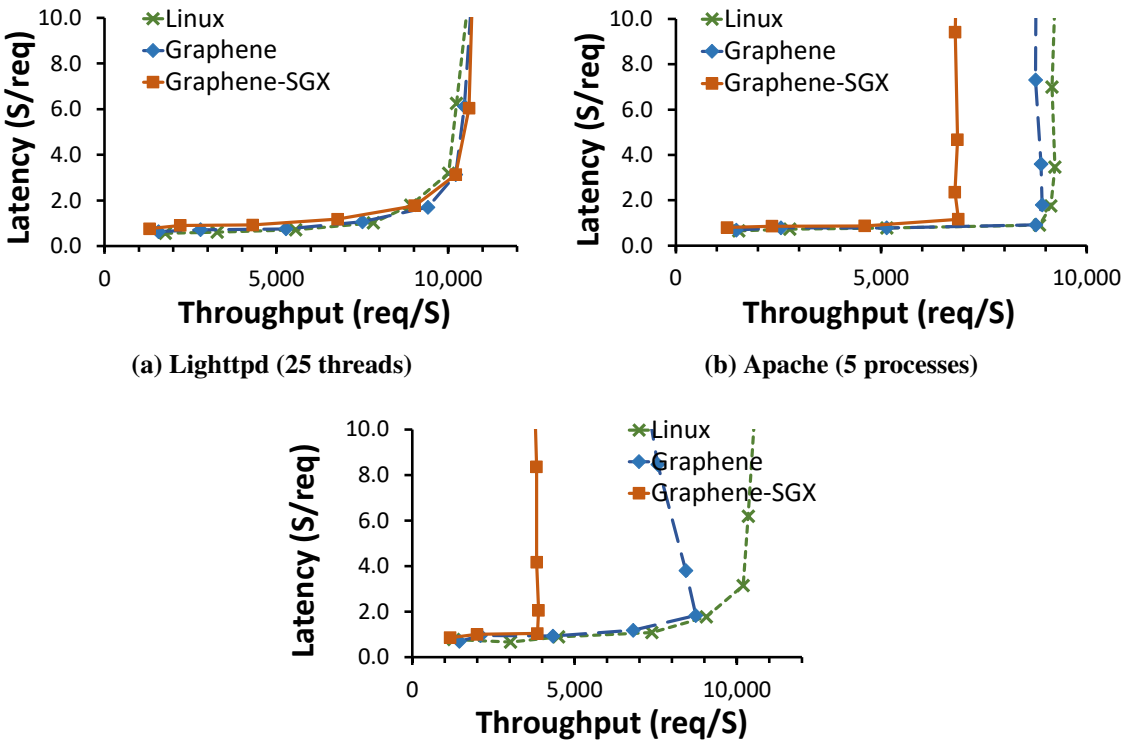
We exercise `GCC/make` with inputs of varying sizes: bzip2 (v1.0.6, 5KLoC, 13 files), Graphene's `libLinux` (31 KLoC, 78 files) and GCC (v3.5.0, 551 KLoC, collected as a single source file). We benchmark Apache (4 preforked workers) and Lighttpd (4 threads) with ApacheBench, which issues 25, 50, and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

We exercised Bash with 300 iterations of the Unixbench benchmark [24], 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.

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** [1] (v2.4.18), and **NGINX** [16] (v1.10). For each workload, we use ApacheBench [2] 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 7.12 shows the throughput versus latency of these server applications in Graphene-SGX, Graphene and Linux. Each workload is discussed below.



(a) Lighttpd (25 threads)

(b) Apache (5 processes)

(c) NGINX (event-driven)

Figure 7.12: 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.

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 [1] 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

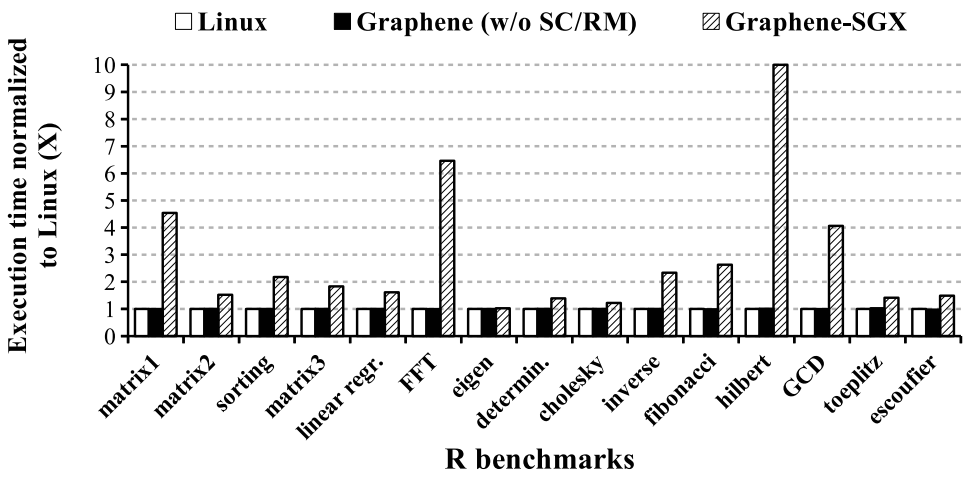
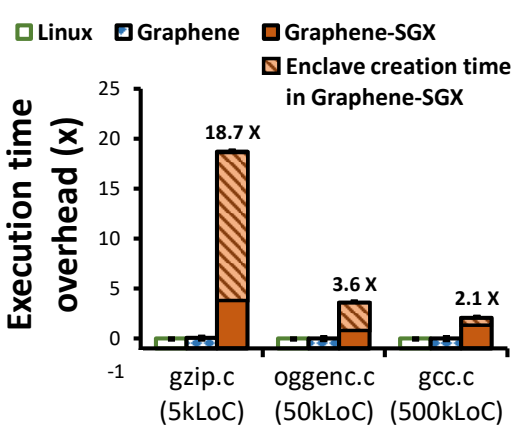


Figure 7.13: Performance overhead on desktop applications, including latency of R, execution time of GCC compilation, download time with CURL. The evaluation compares native Linux, Graphene, and Graphene-SGX.

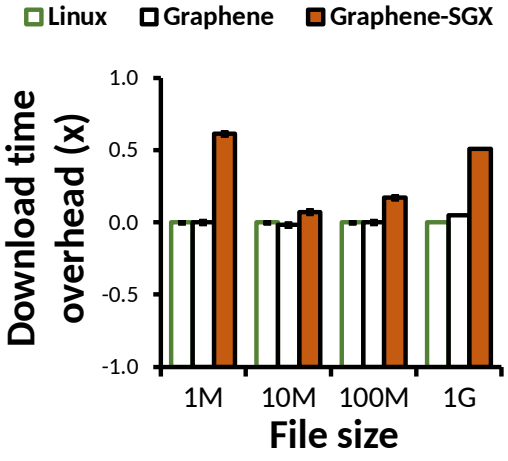
workload, the response time on Graphene-SGX is 12–35% higher than Linux, due to the overhead of coordination across enclaves over encrypted RPC streams. The peak throughput achieved by Apache running in Graphene-SGX is 26% lower than running in Linux. In this workload, most of the overheads are SGX-specific, such as exiting enclaves when accessing the RPC, as non-SGX Graphene has only 2–8% overhead compared to Linux.

NGINX [16] 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 [136] to reduce exits would help this workload; in future work, we will improve asynchronous I/O in Graphene-SGX.

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 sta-



(b) GCC



(c) CURL

Figure 7.14: 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.

tistical computing [22]; **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 μs [171], whereas starting an empty Linux VM takes 10.3s and starting a Linux (LXC) container takes 200 ms [29].

On SGX, the enclave creation time is relatively higher, ranging from 0.5s (a 256MB enclave) to 5s (a 2G enclave), which is a fixed cost that any application framework will have to pay to run on SGX. Enclave creation time is determined by the latency of the hardware and the Intel kernel driver, and is primarily a function of the size of the enclave, which is specified at creation time because it affects the enclave signature. For non-server workloads that create multiple processes during execution, such as GCC in Figure 7.14, the enclave creation contributes a significant portion to the execution time overheads, illustrated as a stacked bar.

R [22] 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. [21], 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 [8]. Each GCC execution typically creates five processes, and we run each process in a 256MB enclave by default. For a small workload like compiling `gzip.c` (5 kLoC), running in Graphene-SGX (4.1s) is 18.7× slower than Linux (0.2s). The bulk of this time is spent in enclave creation, taking 3.0s in total, while the whole execution inside the enclaves, including initialization of the library OS and OS shield, takes only 1.1s, or 4.2× overhead. For larger workloads like `oggenc.c` (50 kLoC) and `gcc.c` (500 kLoC), the overhead of Graphene-SGX is less significant. For `gcc.c` (500 kLoC), we have to enlarge one of the enclaves (`cc1`) to 2GB, but running on Graphene-SGX (53.1s) is only 2.1× slower than Linux (17.2s), and 7.1s is spent on enclave creation. The overhead of non-SGX Graphene on GCC is marginal.

CURL [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 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.

7.4 Summary

Chapter 8

Compatibility Measurement

This chapter evaluates the compatibility of Graphene and other Linux system call emulation layers such as L4Linux. In general, OS developers struggle to evaluate the impact of an API change that affects backward-compatibility, primarily because of a lack of metrics. This chapter proposes new metrics for measuring partial compatibility of a system, ot the impact to compatibility by changing or deprecating an API. Based on the compatibility metrics, OS developers can strategies API implementation to maximize The experiment also contributes a large data sets and analysis of how system APIs are used in practice.

8.1 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 (§9.2). 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 [69, 174]. 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.

8.1.1 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.

8.1.2 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.

8.2 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 obfuscating itself, such as by jumping

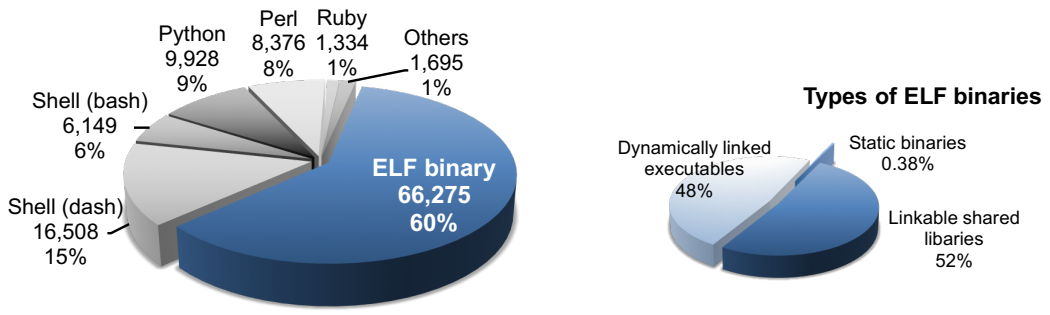


Figure 8.1: Percentage of ELF binaries and applications written in interpreted languages among all executables in the Ubuntu/Debian Linux repository, categorized by interpreters. ELF binaries include static binaries, shared libraries and dynamically-linked executables. Interpreters are detected by *shebangs*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

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.

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.

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.

8.2.1 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.

8.3 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 completeness against the APIs exported by GNU libc 2.21.

8.3.1 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 [72], L4Linux [89], FreeBSD emulation layer [73], and Graphene library OS [171]. 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

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.)

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 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%.

8.3.2 Standard C libraries

This study also uses weighted completeness to evaluate the compatibility of several libc variants — eglibc [6], uClibc [23], musl [15] 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

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()`).

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 performs 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).

8.4 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 both applications and users (i.e., weighted completeness).

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.

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

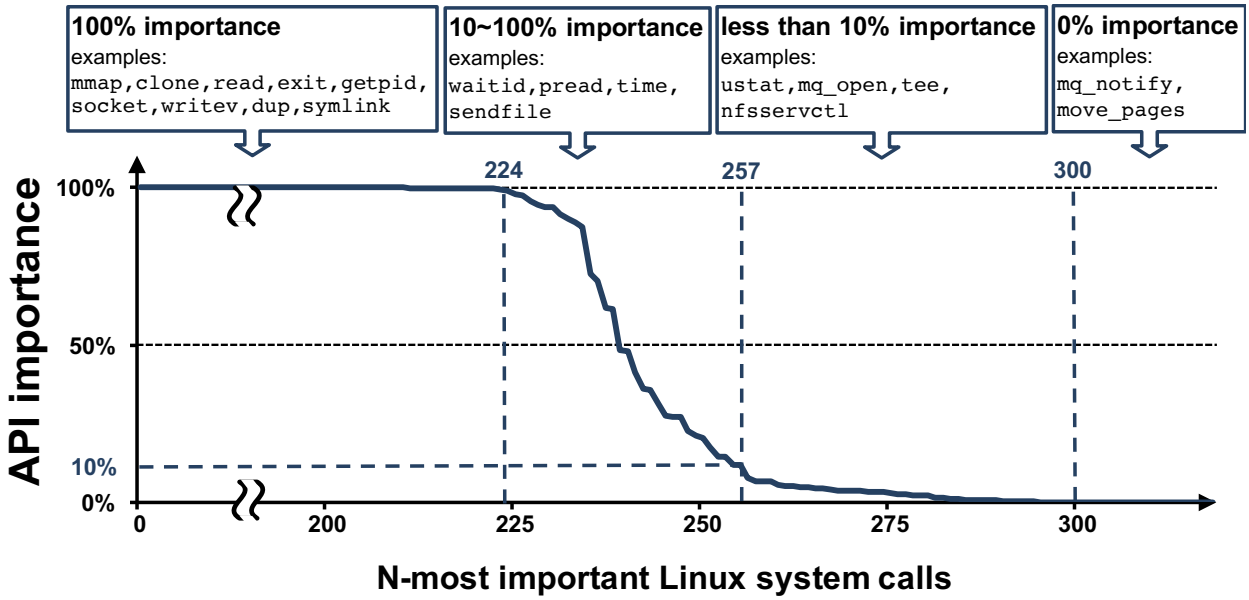


Figure 9.1: The trend of API importance as N-most important system calls among total 320 system calls of Ubuntu Linux 15.04 with Linux kernel 3.19. . Higher is more important; 100% indicates all installations include software that make the system call.

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`.

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 in-

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.

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.

stance 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 Ubun-

tu/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 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 [49]. 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.edu/api-compat-study`.

A goal of weighted completeness is to help guide the process of developing new system prototypes. Section 9.1 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%.

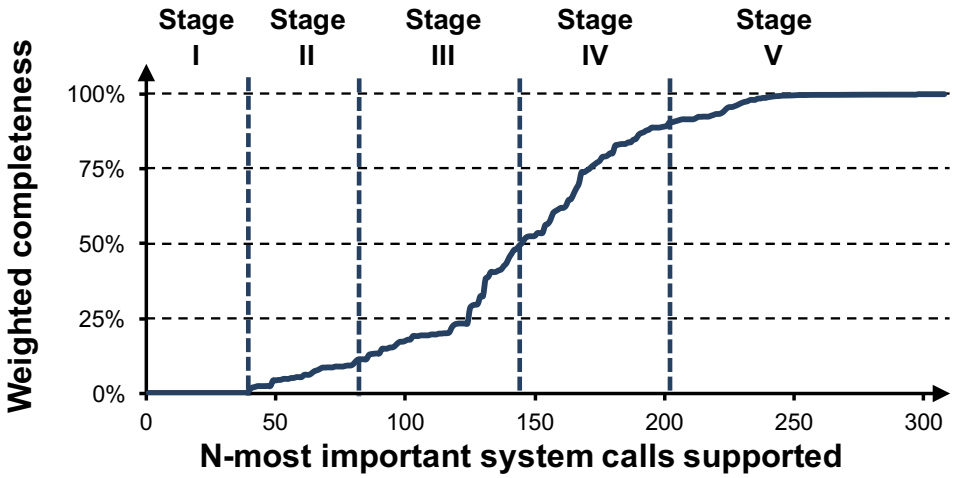


Figure 9.2: Accumulated weighted completeness when N top-ranked system calls are implemented in the OS. Higher is more compliant.

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 Vectored system call opcodes

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.

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

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.

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 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

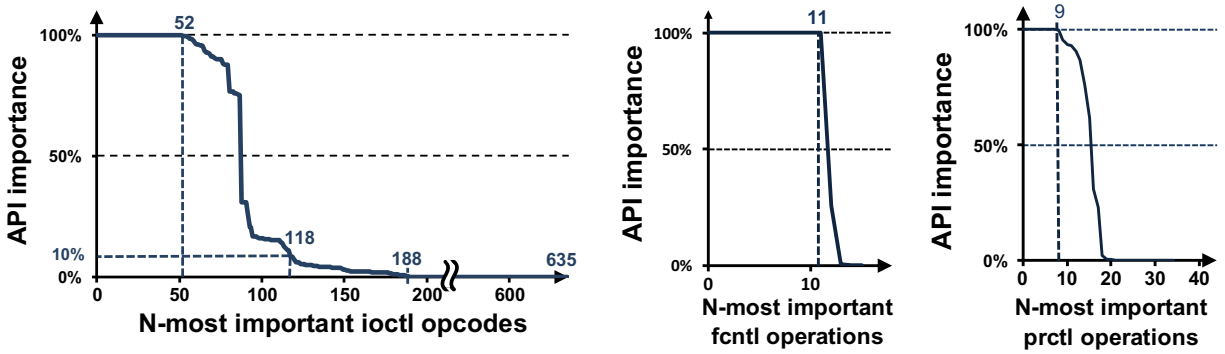


Figure 9.3: Ranking of API importance among ioctl, `fcntl` and `prctl` opcodes. Higher is more important; 100% indicates all installations include software that request the operations.

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. 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 func-

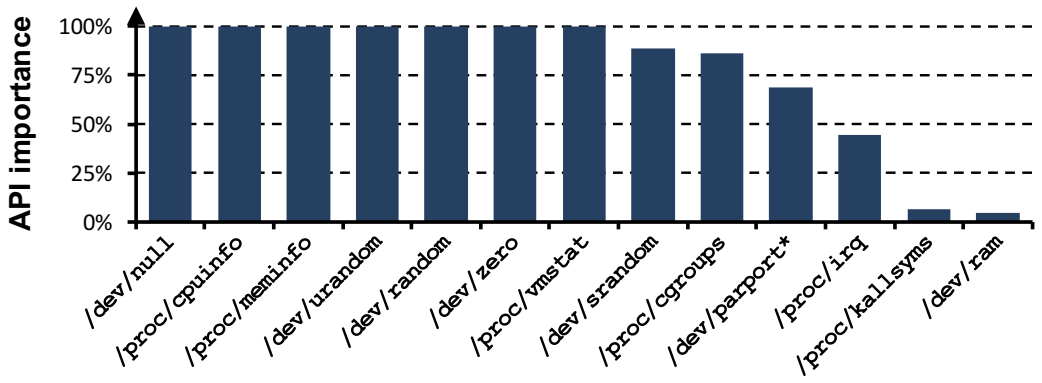


Figure 9.4: API importance distribution over files under `/dev` and `/proc`. Higher is more important; 100% indicates all installations include software that accesses the file.

tionality 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 [48]. 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 [99]. For files used by a single application, an abstraction like a fine-grained capability [156] 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 APIs

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 [9] exports APIs for using locks and condition variables, which internally use the subtle `futex` system call [80].

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) [154], 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,

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.

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 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. [60] 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 [53, 56, 182], 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 [170]. 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.

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.

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

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 Work

The chapter discusses related work to differentiate the contribution of Graphene over previous research, especially in the areas compatibility and security isolation frameworks based on library OSes or virtualization. This chapter also compares the implementation techniques inside of the library OS, such as PRC-based coordination, with related work that take similar techniques. Finally, this chapter summarizes the existing shielding frameworks and development tools for SGX, and innovative SGX applications.

10.1 Library OSes and virtualization

Previous library OSes. Previous library OSes [17, 46, 74, 125, 144] focus on running single-process applications in a picoprocess or a unikernel for verious reasons including compatibility and security isolation. Bascule [46] 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 [17].

A number of recent projects have provided a minimal, isolated environment for web applications to download and execute native code [74, 92, 132, 179, 188]. 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 tailoring hardware management heuristics to individual application needs [31, 36, 65, 101, 115], 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. [101] 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 [72] (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 [43], and, unlike a library OS, does not deduplicate functionality.

Virtualization. Recent library OSes, including Graphene, search for a better division of labor between the host kernel and guests. Paravirtualized VMs attempt to move away from modeling specific hardware designs in software toward a more virtualization-friendly hardware model [43, 75, 183]. Library OSes can be viewed as extreme paravirtualization—attempting to find the most ideal interface between guest and host.

Distributed coordination. Distributed operating systems,such as LOCUS [78, 178], Amoeba [64, 134] and Athena [58] 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 [141], 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 [45] 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 [163] 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 [45] 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 [118]. Although this model was abandoned as cumbersome for general-purpose access control [76], 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 [113] 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 [161], containers [51], and Solaris Zones [145], 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 [104]. FreeBSD jails apply a similar approach to augment an isolated `chroot` environment with other isolated namespaces, including the network and hostname [167]. Similarly, Zap [137] 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 [155] 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 Trusted execution

Protection against untrusted OSes. Protecting applications from untrusted OSes predates hardware support. Virtual Ghost [67] 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 [126], MUSHI [191], SeCage [123], InkTag [91], and Sego [110] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koeberl et al. [108], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [116] built a 2-way sandbox for x86 by

separating the Native Client (NaCl) [189] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [127] to protect the piece of application logic from the untrusted OS. Jang et al. [100] build a secure channel to authenticate the application in the Untrusted area isolated by the ARM TrustZone technology. Song et al. [162] extend each memory unit with an additional tag to enforce fine-grained isolation at machine word granularity in the HDFI system.

Trusted execution hardware. XOM [117] 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 [169] 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++ [52] 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) [103], 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 [66] 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 shielding frameworks. SGX shielding frameworks, including Haven [47], SCONE [40], PANOPLY [159], and Graphene-SGX, enforce end-to-end isolation to legacy applications without partitioning. A SGX shielding system preserves the trusted computing base (TCB) of an application, and further increases it with a shielding layer to defend against the untrusted OSes. By avoiding application partitioning, a shielding system minimizes the effort of reprogramming the applications for enclave execution, often with recompilation or packaging the binaries in an encrypted enclave. In the following paragraphs, we compare the current shielding systems with the Graphene-SGX approach.

Haven [47] uses a library OS called Drawbridge in each enclave to shield a single-process *Windows* application from the untrusted host OS. Haven absorbs the implementation of system APIs (i.e., Win32 APIs) from the host OS, and exports a narrow enclave interface on which untrusted inputs are carefully filtered to defend against the Iago-type attacks. Adding a library OS to each enclave causes a bloat of TCB—for Haven, the size of a library OS binary and shielding layer is $\sim$200MB. Haven has to establish the trust and integrity in all these binaries loaded into an enclave. Except that the shielding layer is a part of the enclave since its creation, Haven enforces the integrity of both the library OS and the isolated application, by storing all binaries on an encrypted virtual disk and relying a remote, trusted server to provision the key for decryption. Haven builds a trusted path from a remote server to local cloud machines, to securely bootstrap application execution inside the enclaves.

SCONE [40] isolates Linux micro-services in enclaves as a container-like environment. After a brief attempt of building a library OS like Haven, SCONE chooses a different approach of shielding the system API usage in applications, by designing shielding strategies based on each API. SCONE stacks the application on top of file-system and network shielding libraries, and extends a standard library C (musl [15]) to securely exit the enclave for system calls. Within the SGX-aware libc, SCONE carefully filters the inputs from the host system calls, as the defend against known Iago attacks. For instance, SCONE ensures that pointers given to and returned by a host system call will point to addresses outside the enclave, to prevent the host OS to manipulate pointers and cause memory corruption in the enclave. SCONE also authenticates or encrypts file or network streams based on configurations given by the developers.

PANOPLY [159] further reduces the TCB of a shielding system over the SCONE approach, by excluding both a library OS and libc from enclaves. Instead, PANOPLY uses a shim layer shielding a portion of the POSIX API. The shim layer yields about 20 KLoC as its TCB (trusted computing base), which is much smaller than libc and/or a library OS. As PANOPLY delegates the libc functions outside the enclave, its shim library defends the supported POSIX API, including 91 *safe* functions and 163 *wild (unsafe)* functions. PANOPLY also supports multi-process API including `fork()`, `exec()`, signaling, and sharing untrusted memory with inline encryption. Compared to Graphene-SGX, PANOPLY has made some different design decisions in supporting multi-process API, including supporting fork by copying memory on-demand with statically determining memory access, and using secured messaging for inter-process negotiating instead of coordinating over an encrypted RPC stream.

SGX applications and development tools. Besides shielding systems, SGX has been used in specific applications or to address other security issues. VC3 [151] runs MapReduce jobs in SGX enclaves. Similarly, Brenner et al. [54] run cluster services in ZooKeeper in an enclave, and transparently encrypt data in transit between enclaves. Ryoan [94] sandboxes a piece of untrusted code in the enclave to process secret data while preventing the loaded code from leaking secret data. Opaque [193] 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 [157], as well as inter-domain routing in Tor [105].

Several improvements to SGX frameworks have been recently developed, which can be integrated with applications on Graphene-SGX. Eleos [136] reduces the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. SGXBOUND [109] is a software technique for bounds-checking with low memory overheads, to fit within limited EPC size. T-SGX [158] 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 [153] 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 [121] 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. [41] 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 [102] 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 [138]. They find architecture-specific subsystems have highest fault rate, followed by file systems. Harter et al. [88] 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 [82, 135, 149], to measure the popularity of application programming languages [33], to analyze dependencies between the packages [68], to identify trends in package sizes [32], the number of developers involved in developing and maintaining a package [148], and estimating the cost of development [34]. Jain et al. used popularity contest survey data to prioritize the implementation effort for new system security policies [98]. 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) [70] 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 [140]. 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 [142]. 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 [160]. 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 [190]. Static binary analysis has been used to automatically generate application-specific sandboxing policies [114]. Dynamic analysis has been used to compare system call sequences of two applications as an indicator of potential intellectual property theft [181], to identify opportunities to batch system calls [146], to model power consumption on mobile devices [139], and to repackage applications to run on different systems [85]. These projects answer very different questions than ours, but could, in principle, benefit from the resulting data set.

Chapter 11

Conclusion

Application developers pay varied efforts to port applications to new OSes or hardware in order to gain qualitative benefits. Application porting efforts range from recompilation to reimplementation, due to API distinction or host-specific restrictions. Existing library OSes [46, 47, 144] 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.

This thesis presents a library OS called Graphene [171], which supports unmodified 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, including 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) [172]. 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:

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

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 ran-

dom 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 approxi-

mated value of weighted completeness is:

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

References

[1] Apache HTTP server project. https://httpd.apache.org/, .

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

[3] Aufs. http://aufs.sourceforge.net/.

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

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

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

[11] 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.

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

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

[14] Linux containers. https://linuxcontainers.org/.

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

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

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

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

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

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

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

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

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

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

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

[26] CVE-2016-5195. https://nvd.nist.gov/vuln/detail/CVE-2016-5195, November 2016.

[27] 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. ISBN 978-1-931971-38-6.

[28] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. MACH: a new kernel foundation for UNIX development. Technical report, Carnegie Mellon University, 1986.

[29] Kavita Agarwal, Bhushan Jain, and Donald E. Porter. Containing the hype. In *Proceedings of the 6th Asia-Pacific Workshop on Systems*, APSys '15, 2015.

[30] Bowen Alpern, C Richard Attanasio, John J Barton, Michael G Burke, Perry Cheng, J-D Choi, Anthony Cocchi, Stephen J Fink, David Grove, Michael Hind, et al. The Jalapeno virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

[31] Glenn Ammons, Jonathan Appavoo, Maria Butrico, Dilma Da Silva, David Grove, Kiyokuni Kawachiya, Orran Krieger, Bryan Rosenburg, Eric Van Hensbergen, and Robert 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.

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

[33] Juan Jose Amor, Jesus M. Gonzalez-Barahona, Gregorio Robles, and Israel 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.

[34] Juan José Amor, Gregorio Robles, and Jesús M. González-Barahona. Measuring Woody: The size of Debian 3.0. *CoRR*, 2005.

[35] Ittai Anati, Shay Gueron, Simon P Johnson, and Vincent 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.

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

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

[38] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, December 2000.

[39] Andrea Archangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Linux Symposium*, pages 19–28, 2009.

[40] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Daniel O'Keeffe, Mark L. Stillwell, David Goltzsche, Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. SCONE: Secure linux containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Nov 2016.

[41] Vaggelis Atlidakis, Jeremy Andrus, Roxana Geambasu, Dimitris Mitropoulos, and Jason 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.

[42] Sumeet Bajaj and Radu Sion. Correctdb: Sql engine with practical query authentication. *Proceedings of the VLDB Endowment*, May 2013.

[43] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew 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. ISBN 1-58113-757-5. doi: http://doi.acm.org/10.1145/945445.945462.

[44] Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. Mach-1: An operating environment for large-scale multiprocessor applications. *Journal of IEEE SOFTWARE*, 2(4):65–67, jul 1985.

[45] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh 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.

[46] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2013.

[47] Andrew Baumann, Marcus Peinado, and Galen 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.

[48] Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos 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.

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

[50] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. In *Advances in Cryptology–CRYPTO 2013*, pages 90–108. Springer, 2013.

[51] Sukadev Bhattiprolu, Eric W. Biederman, Serge Hallyn, and Daniel Lezcano. Virtual servers and checkpoint/restart in mainstream Linux. *ACM Operating Systems Review*, 42:104–113, July 2008.

[52] Rick Boivie and Peter Williams. SecureBlue++: CPU support for secure executables. Technical report, IBM Research, 2013.

[53] Nikita Borisov, Rob Johnson, Naveen Sastry, and David Wagner. Fixing races for fun and profit: How to abuse atime. In *Proceedings of the USENIX Security Symposium*, pages 303–314, 2005.

[54] Stefan Brenner, Colin Wulf, and Rüdiger Kapitza. Running zookeeper coordination services in untrusted clouds. In *10th Workshop on Hot Topics in System Dependability (HotDep 14)*, 2014.

[55] Thomas Wolfgang 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.

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

[57] Calin Cascaval, José G Castanos, Luis Ceze, Monty Denneau, Manish Gupta, Derek Lieber, José E Moreira, Karin Strauss, and Henry 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.

[58] George A. Champine, Daniel E. Geer, Jr., and William N. Ruh. Project Athena as a Distributed Computer System. *IEEE Computer*, 23(9):40–51, September 1990.

[59] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call API is a bad untrusted RPC interface. *SIGPLAN Not.*, pages 253–264, March 2013. ISSN 0362-1340. URL http://doi.acm.org/10.1145/2499368.2451145.

[60] Hao Chen, David Wagner, and Drew Dean. Setuid demystified. In *Proceedings of the USENIX Security Symposium*, 2002.

[61] J. Bradley Chen and Brian 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.

[62] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting privileged side-channel attacks in shielded execution with déjà vu. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2017.

[63] Xiaoxin Chen, Tal Garfinkel, E. Christopher Lewis, Pratap Subrahmanyam, Carl A. Waldspurger, Dan Boneh, Jeffrey Dwoskin, and Dan R.K. Ports. Overshadow: A virtualization-

based approach to retrofitting protection in commodity operating systems. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 2–13, 2008.

[64] 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.

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

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

[67] John Criswell, Nathan Dautenhahn, and Vikram 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.

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

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

[70] 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.

[71] D.M. Dhamdhere. *Operating Systems: A Concept-based Approach*. McGraw-Hill, 2007. ISBN 9780071264365.

[72] Jeff Dike. *User Mode Linux*. Prentice Hall, 2006.

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

[74] John R. Douceur, Jeremy Elson, Jon Howell, and Jacob 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.

[75] Hideki Eiraku, Yasushi Shinjo, Calton Pu, Younggyun Koh, and Kazuhiko Kato. Fast networking with socket-outsourcing in hosted virtual machine environments. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 310–317, 2009.

[76] Kevin Elphinstone and Gernot 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.

[77] 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.

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

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

[80] Hubertus Franke, Rusty Russel, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast userlevel locking in Linux. In *Ottawa Linux Symposium*, 2002.

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

[82] Jesus M. Gonzalez-Barahona, Gregorio Robles, Martin Michlmayr, Juan José Amor, and Daniel M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 2009.

[83] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache attacks on intel sgx. In *Proceedings of the 10th European Workshop on Systems Security*, 2017.

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

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

[86] Marcus Hähnel, Weidong Cui, and Marcus Peinado. High-resolution side channels for untrusted operating systems. In *Proceedings of the USENIX Annual Technical Conference*, 2017.

[87] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold-boot attacks on encryption keys. *Commun. ACM*, pages 91–98.

[88] Tyler Harter, Chris Dragga, Michael Vaughn, Andrea C. Arpaci-Dusseau, and Remzi 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.

[89] Hermann Härtig, Michael Hohmuth, Jochen Liedtke, Jean Wolter, and Sebastian Schönberg. The performance of μ-kernel-based systems. *SIGOPS Operating System Review*, 1997.

[90] Gael Hofemeier and Robert Chesebrough. Introduction to Intel AES-NI and Intel Secure Key instructions. `https://software.intel.com/en-us/node/256280`, 2012.

[91] Owen S. Hofmann, Sangman Kim, Alan M. Dunn, Michael Z. Lee, and Emmett 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.

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

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

[94] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett 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.

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

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

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

[98] Bhushan Jain, Chia-Che Tsai, Jitin John, and Donald E. Porter. Practical techniques to obviate setuid-to-root binaries. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

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

[100] Jin Soo Jang, Sunjune Kong, Minsu Kim, Daegyeong Kim, and Brent Byunghoon Kang. Secret: Secure channel between rich execution environment and trusted execution environment. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2015.

[101] M. Frans Kaashoek, Dawson R. Engler, Gregory R. Ganger, Hector M. Briceño, Russell Hunt, David Mazières, Thomas Pinckney, Robert Grimm, John Jannotti, and Kenneth Mackenzie. Application performance and flexibility on exokernel systems. In *Proceedings*

of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP), pages 52–65, 1997.

[102] Asim Kadav and Michael 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.

[103] David Kaplan, Jeremy Powell, and Tom 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`.

[104] Michael Kerrisk. User namespaces progress. *Linux Weekly News*, 2012.

[105] Seongmin Kim, Youjung Shin, Jaehyung Ha, Taesoo Kim, and Dongsu 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.

[106] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceedings of the ACM IEEE International Symposium on Computer Architecture (ISCA)*, 2014.

[107] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. sel4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 207–220, 2009.

[108] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, page 10. ACM, 2014.

[109] Dmitrii Kuvaiskii, Oleksii Oleksenko, Sergei Arnautov, Bohdan Trach, Pramod Bhatotia,

Pascal Felber, and Christof Fetzer. SGXBOUNDS: Memory safety for shielded execution. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2017.

[110] Youngjin Kwon, Alan M. Dunn, Michael Z. Lee, Owen S. Hofmann, Yuanzhong Xu, and Emmett Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. *SIGOPS Oper. Syst. Rev.*

[111] Youngjin Kwon, Alan M Dunn, Michael Z Lee, Owen S Hofmann, Yuanzhong Xu, and Emmett 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.

[112] L4Family. The L4 microkernel family. available at `http://www.l4hq.org/`.

[113] Oren Laaden and Serge E. Hallyn. Linux-CR: Transparent application checkpoint-restart in Linux. In *Linux Symposium*, 2010.

[114] LapChung Lam and Tzi-cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Recent Advances in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2004.

[115] Ian Leslie, Derek Mcauley, Richard Black, Timothy Roscoe, Paul Barham, David Evers, Robin Fairbairns, and Eoin Hyden. The design and implementation of an operating system to support distributed multimedia applications. *IEEE JSAC*, pages 1280–1297, 1996.

[116] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker, and Will Drewry. Minibox: A two-way sandbox for x86 native code. In *Proceedings of the USENIX Annual Technical Conference*, pages 409–420, 2014.

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

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

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

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

[121] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rudiger Kapitza, Christof Fetzer, and Peter Pietzuch. Glamdring: Automatic application partitioning for Intel SGX. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2017.

[122] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level cache side-channel attacks are practical. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2015.

[123] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. Thwarting memory disclosure with efficient hypervisor-enforced intra-domain isolation. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[124] Robert Love. *Linux Kernel Development*. Addison-Wesley Professional, 3rd edition, 2010. ISBN 0672329468, 9780672329463.

[125] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon 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.

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

[127] Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 143–158, 2010.

[128] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, 2013. URL `http://doi.acm.org/10.1145/2487726.2488368`.

[129] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos Rozas. Intel software guard extensions (Intel SGX) support for dynamic memory management inside an enclave. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–9, New York, New York, USA, June 2016. ACM Press. ISBN 9781450347693. doi: 10.1145/2948618.2954331. URL `http://dl.acm.org/citation.cfm?doid=2948618.2954331`.

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

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

[132] James Mickens and Mohan 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.

[133] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. 2017.

[134] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990. ISSN 0018-9162.

[135] Raymond Nguyen and Ric 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.

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

[137] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason 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.

[138] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles 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.

[139] Abhinav Pathak, Y. Charlie Hu, Ming Zhang, Paramvir Bahl, and Yi-Min 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.

[140] 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.

[141] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990.

[142] 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.

[143] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, July 1974.

[144] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen 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.

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

[146] Mohan Rajagopalan, Saumya K Debray, Matti A Hiltunen, and Richard D Schlichting. System call clustering: A profile directed optimization technique. Technical report, The University of Arizona, May 2003.

[147] Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Communication ACM*, July 1974.

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

[149] Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, and Juan Jose 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.

[150] Nuno Santos, Krishna P. Gummadi, and Rodrigo Rodrigues. Towards trusted cloud computing. In *Proceedings of the USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

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

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

[153] Jaebaek Seo, Byounyoung Lee, Seongmin Kim, Ming-Wei Shih, Insik Shin, Dongsu Han, and Taesoo Kim. SGX-Shield: Enabling address space layout randomization for SGX programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[154] Hovav 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.

[155] Zhiyong Shan, Xin Wang, Tzi-cker Chiueh, and Xiaofeng Meng. Facilitating inter-application interactions for os-level virtualization. In *Proceedings of the International Conference on Virtual Execution Environments (VEE)*, pages 75–86, 2012.

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

[157] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada 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.

[158] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating controlled-channel attacks against enclave programs. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[159] Shweta Shinde, Dat Le Tien, Shruti Tople, and Prateek Saxena. PANOPLY: Low-TCB Linux applications with SGX enclaves. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

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

[161] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry 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.

[162] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. Hdfi: Hardware-assisted data-flow isolation. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2016.

[163] Xiang Song, Haibo Chen, Rong Chen, Yuanxuan Wang, and Binyu Zang. A case for scaling applications to many-core with OS clustering. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2011.

[164] Joel Spolsky. How microsoft lost the API war. June 2004.

[165] 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.

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

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

[168] Ruimin Sun, Donald E Porter, Matt Bishop, and Daniela Oliveira. The case for less predictable operating system behavior. In *Proceedings of the USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2015.

[169] TrustZone. Arm trustzone technology overview. `http://www.arm.com/products/processors/technologies/trustzone/index.php`.

[170] Dan Tsafrir, Tomer Hertz, David Wagner, and Dilma Da Silva. Portably preventing file race attacks with user-mode path resolution. Technical report, IBM Research Report, 2008.

[171] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and Security Isolation of Library OSes for Multi-Process Applications. In *Proceedings of the ACM European Conference on Computer Systems (EuroSys)*, 2014.

[172] Chia-Che Tsai, Yang Zhan, Jayashree Reddy, Yizheng Jiao, Tao Zhang, and Donald 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.

[173] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald 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. ISBN 978-1-4503-4240-7.

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

[175] 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.

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

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

[178] Bruce Walker, Gerald Popek, Robert English, Charles Kline, and Greg Thiel. The LOCUS distributed operating system. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 49–70, 1983.

[179] Helen J. Wang, Chris Grier, Alexander Moshchuk, Samuel T. King, Piali Choudhury, and Herman Venter. The multi-principal OS construction of the Gazelle web browser. In *Proceedings of the USENIX Security Symposium*, pages 417–432, 2009.

[180] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A. Gunter. Leaky cauldron on the dark land: Understanding memory side-channel hazards in SGX. 2017.

[181] Xinran Wang, Yoon-Chan Jhi, Sencun Zhu, and Peng 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.

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

[183] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and performance in the Denali isolation kernel. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[184] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. Stacco: Differentially analyzing side-channel traces for detecting SSL/TLS vulnerabilities in secure enclaves. 2017.

[185] Wen Xu, Juanru Li, Junliang Shu, Wenbo Yang, Tianyi Xie, Yuanyuan Zhang, and Dawu Gu. From collision to exploitation: Unleashing use-after-free vulnerabilities in linux kernel. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[186] Yuanzhong Xu, Weidong Cui, and Marcus 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.

[187] Yuval Yarom, Daniel Genkin, and Nadia Heninger. Cachebleed: a timing attack on openssl constant-time rsa. *Journal of Cryptographic Engineering*, 7(2):99–112, 2017.

[188] Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2009.

[189] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas 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.

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

[191] Ning Zhang, Ming Li, Wenjing Lou, and Y Thomas Hou. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *Military Communications Conference, 2012-MILCOM 2012*, pages 1–6. IEEE, 2012.

[192] Yupeng Zhang, Jonathan Katz, and Charalampos Papamanthou. Integridb: Verifiable sql for outsourced databases. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2015.

[193] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2017.

[194] YongBin Zhou and DengGuo Feng. Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing. *IACR Cryptology ePrint Archive*, 2005:388, 2005.