A Library Operating System for Compatibility

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

December 2017

Abstract of the Dissertation

A Library Operating System for Compatibility

by

Chia-Che Tsai

For the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

2017

Compatibility challenges occur when sharing an application across system interfaces such as Linux and Windows APIs, or reusing an application on a disruptive hardware such as Intel SGX [122]. Existing approaches require either exhaustively porting applications to new APIs or preserving all previous APIs for backward compatibility. Since both approaches are time-consuming, developers urgently need a solution to the compatibility issues on innovative OSes or hardware, to promptly benefit average users. This thesis demonstrates a library OS approach for reusing unmodified applications on a new OS or hardware. The approach starts with defining a host ABI which is simple to port and also sufficiently contains essential OS abstractions such as file and page management. The host ABI divides the development of a compatibility layer into API emulation in a library OS and encapsulating host distinction with a PAL (platform adaption layer).

This thesis presents the Graphene library OS, which demonstrates the simplicity and sufficiency of its host ABI by enumerating host abstractions used for emulating Linux system calls and the related porting efforts. For instance, Graphene emulates multi-process abstractions using two host abstractions: creating a new process, and a simple RPC stream. Leveraging a distributed coordination model, multiple Graphene instances across processes collaboratively present a united OS view to an application. Two main porting targets of Graphene, Linux, and SGX, each present challenges to enforcing security isolation. On a Linux host, Graphene isolates mutually-untrusting applications. On SGX, Graphene protects a security-sensitive application against an untrusted OS. From a security perspective, Graphene restricts the attack surface through system interfaces and simplifies security checks against unexpected exploitations. Finally, this thesis presents quantitative measurements to evaluate the partial compatibility of OS prototypes and importance of APIs, to help to prioritize API porting.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to express my sincere, deepest gratitude to my advisor, Donald E. Porter. During the past six and half years, Don has stood by me at every up and down, giving me every bit of his support. He has indulged me with incredible patience, and invested enormous time and energy to guide me finishing this dissertation. Moreover, he has very kindly provided advice and recommendation to help me pursue my academic career. He has always been an inspiration and a role model to me. I cannot thank him enough for all the influence he has on me.

I would also like to thank my committee, R. Sekar, Michael Ferdman, and Timothy Roscoe. Their suggestions are inspirational and insightful. I especially appreciate the committee for having waited patiently for fifteen months after the thesis proposal, allowing me to take my pace and finish the thesis writing.

It is also an honor to work alongside so many amazing collaborators. I want to thank all of my friends in the OSCAR Lab: William Jannen, Bhushan Jain, Amogh Akshintala, Yang Zhan, Tao Zhang, and Yizheng Jiao. Each of them has offered me precious feedback and suggestion to projects in my graduate career. Also thanks for putting up with me all these years.

Special thanks go to Mona Vij, who has been both a collaborator and a good friend to me for years. She has contributed significantly to the design and promotion of Graphene-SGX. I am deeply grateful for the opportunity of working in Intel Lab with her, an opportunity that changed the fate of both me and the Graphene library OS. Mona has also helped me tremendously in pursuing my academic career.

The Graphene library OS is a combined effort of numerous contributors. I want to thank all the graduate students and collaborators who have participated in the project: Kumar Saurabh Arora, Nehal Bandi, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, Gurpreet Chadha, Manikantan Subramanian, Naveen Kalaskar, Thomas Mathew, Anchal Agarwal, Amit Arya, Arjun Bora, Sourabh Yerfule, Ujwala Tulshigiri, Imran Brown, Daljeet Singh Chhabra, Shahzeb Nihalahmed Patel, and Sridarshan Shetty. The project wouldn't have been successful without them.

Chapter 1

Introduction

Operating systems simplify programming of an application to utilize hardware with different interfaces and features. A UNIX-style OS [142] encapsulates hardware distinction using system interfaces such as the system call table. Without system interfaces, developers have to program against hardware interfaces defined by manufacturers. Programming against hardware interfaces restricts applications 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.

In general, OS developers preserve old system interfaces to maintain compatibility for existing application binaries. This thesis defines compatibility of an OS as the ability to reproduce a system interface with the requirements of applications. Many developers deploy applications as native binaries, with the use of system interfaces hard-coded inside the binary code. To accommodate unmodified application binaries, OS developers share a common goal of keeping system interfaces consistent across OS versions, or backward-compatible. Compatibility is also a goal for less widely-used OSes, such as FreeBSD, to emulate a more popular system interface, such as the Linux system call table, to reuse more well-adopted applications.

Unfortunately, the recent trend of hardware development has challenged the feasibility of maintaining the compatibility of a system interface. Although most new hardware shares the interfaces and semantics with their predecessors, more cutting-edge hardware tend to leak out of the common hardware abstractions encapsulated by an OS. One example is SGX (software guard extensions) [122] 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 security challenges such as validating untrusted system call results [55]. Other examples exist among research hardware, such as asymmetric multi-processing architectures without inter-connected memory [53, 79], which challenges inter-process coordination. As more disruptive hardware may emerge in the future, applications need an immediate solution to these critical compatibility issues.

Empirically, compatibility has caused struggles in OS development, especially when developers demand API changes. 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 fully replace the original `open()`, developers need to modify every application; otherwise, Linux developers can never retire 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 [158]. Because users may prioritize compatibility over the adoption of new technology, early stable versions of the OS, such as Windows XP, remain popular even after end-of-service (EOL) [17].

There are practical reasons to maintain compatibility for unmodified applications in a commercial OS. The development of a commercial application includes a long process of thoroughly testing and examining the code to ensure correctness and safety. Modifying an application for a new system interface can introduce new risks to stability and extra cost to restart the debugging cycle. Moreover, third parties may have no access to source code of proprietary software even if they are motivated to port the applications. All these dilemmas call for a universal solution to mitigating compatibility issues on reusing unmodified applications.

Compatibility issues alienate users from system interface alternatives that grant access to new hardware or introduce security or performance benefits. This thesis proposes a general approach to building a compatibility layer for translation of a legacy system interface. A compatibility layer between an application and an OS can bridge the gap between system interfaces. Take SGX for example; compatibility challenges on SGX include secure dynamic loading, redirecting system calls to host OSes, and security checks against untrusted system calls. The approach reduces the effort of porting a compatibility layer to any host OSes and hardware, by eliminating the cost of reimplementing various complex system APIs on each host.This thesis presents a solution to building a rich-feature compatibility layer, without the tremendous cost of updating applications to adjust for a new system interface.

This thesis presents **Graphene**, a library OS for running unmodified applications on innovative hardware and alternative system interfaces. A library OS [31, 72, 119, 138] is a library which emulates OS features and APIs in an application process. This thesis focuses on building a library OS using a simple host interface defined for portability. Defining such a host interface is equivalent to finding a "pinch point" inside an OS; the host interface partitions OS components above this boundary, such as system calls and namespaces, into the library OS to make the components reusable across host OSes and hardware. This approach simplifies the porting effort per host as exporting the host interface using a PAL (platform adaption layer).

Graphene reproduces a rich of Linux system calls for a wide range of commercial applications in server or cloud environments. Graphene primarily targets three types of applications. The first is a server or cloud application, such as Apache or Memcached. The second is a command-line program used in a UNIX environment, such as a shell program or a compiler. The last is a language runtime, such as R, Python, or OpenJDK, which heavily deploys system interfaces for language features such as dynamic loading and garbage collection. Graphene implements 145 out of 318 Linux system calls.

Graphene defines a host ABI (application binary interface) which contains only essential host abstractions that are easy to port on different host OSes. This thesis demonstrates the simplicity by experimenting the porting on two sample hosts—a native Linux kernel and an SGX enclave on an untrusted Linux host. The choices of host targets cover two extreme cases of compatibility support with opposite security models. This thesis enumerates the development effort of translating and securing the host ABI on each host, as demonstration of simplicity. Other ongoing ports of the host ABI include alternative OS kernels such as Windows, OS X, and FreeBSD, and research OSes such as L4 microkernels [106] and Barrelfish [44].

Building a library OS is similar to virtualizing a part of an OS. A virtualization solution reuses OS components upon an intermediate interface such as a virtual hardware interface. A virtual machine (VM) usually carries an unmodified, full-fledged OS kernel to reproduce the whole system stack that implements a system interface. Although a VM provides full compatibility for existing applications, it requires a virtualizable architecture [137] or assistance from hardware virtualization such as VT (virtualization technology) [169] to mitigate software virtualization overheads.

Compared to the alternatives, the library OS approach strikes a better balance between simplicity of porting and sufficiency of compatible OS functionality. A study of Linux system interface [166] show that system calls are not equally important to applications. Applications also subject to different popularity among users, as shown in installation statistics [168]. 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 inherits a part of the host ABI from Drawbridge [138], a library OS developed for Windows applications. Drawbridge uses the library OS as a lightweight VM to run Windows desktop applications in a guest environment. Using a library OS also improves the density of packing the guests on a physical host due to significantly lower memory footprint than a VM. Bascule [45] later adopts the design to implement single-process system calls in Linux. Haven [46] further ports Drawbridge to SGX, to shield Windows application from untrusted host OSes. Graphene presents solution to running unmodified applications upon a variety of host OSes and hardware, with a host interface defined for both simplicity of porting and sufficiency for developing a rich Linux-compatible library OS.

The contributions of this thesis over previous work are as follows. 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 system interface for existing applications is challenging, to motivate the approach of the Graphene library OS.

1.1.1 Unmodified Applications on SGX

SGX [122], 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 **enclave**, which isolates and protects an application on an untrusted host to harvest the CPU and memory resources. Application code and data inside an enclave are both signed and encrypted inside the DRAM when leaving the CPU package. SGX also offers remote attestation of the integrity of an enclave and the CPU. SGX provides opportunities for delegating security-sensitive operations to an untrusted public cloud or client machine.

Despite the benefits, the typical expectation for SGX is that developers need to port a piece of application code to run inside an enclave. The restriction is for both security and simplicity since each enclave needs a static code image with security measurement signed by users. Developers also have to remove system calls and instructions such as `cpuid` and `rdtsc` since they compromise security without further enforcement. For instance, system calls serviced by an untrusted OS may return malicious results to exploit semantic flaws in an application caused by being unaware of the attack vectors. SGX forbids `cpuid` and `rdtsc` because these instructions are easily intercepted or spoofed by an OS or a hypervisor. These restrictions introduce porting efforts to existing applications on SGX.

The threat model of SGX distrusts OS services, except APIs that operate entirely inside an enclave (e.g., `malloc()` and `memcpy()`). The absence of trusted OS services is an issue for porting any application to SGX. Existing solutions combine a modified C library with applications, to redirect system calls to the untrusted OS [39, 152]. The problem, however, is in checking the results of system interfaces, because the OS is not trusted. Previous work [55] shows that checking

(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).

untrusted system interface results can be subtle because the existing system interfaces are not designed for an untrusted or compromised OS.

In summary, the previous 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 introducing a library OS into enclaves restricts the interaction with an untrusted OS to only OS services which have clear semantics for checking. By implementing the host ABI inside an enclave, users can quickly port 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. In particular, `fork()` is a 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. A unique process identifier (PID) known by all processes determines the destination of a signal. 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 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 private 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 for simplicity of implementation [53, 79]. Barrelfish [44] 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

As a bonus benefit, Graphene 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 an OS sandbox. An 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 to the complexity of emulation, security isolation inside a monolithic OS is delicate and prone to vulnerabilities. The host ABI of Graphene simplifies the isolation of OS abstractions down to three host abstractions that are sharable among picoprocesses: files, network connections, 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 allow an application to bind to a local network address or to connect to a remote network address. Finally, for RPC streams, Graphene blocks merely any RPC streams which cross sandbox boundary. This thesis shows 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 protects an SGX application from untrusted operating systems, hypervisor, or system software. Existing system APIs, such as system calls, expose an extensive attack surface to an untrusted host, where an adversary can manipulate system API results to explore attack vectors [55]. 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 of 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 adapt to various host platforms, as a compatibility layer for applications. Graphene overcomes the OS and hardware restrictions 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. This thesis also bases the reasoning on a metric which evaluates the partial compatibility of a system interface. Graphene prioritizes relatively indispensable system calls over administration-specific 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 [164], which emphasizes on security isolation, between mutually-untrusted applications. A later publication [27] focuses on porting the host ABI to Intel SGX and demonstrates security benefits over using a thin API redirection layer. [166] presents the compatibility metrics for compatibility, with a study of the Linux API usage among Ubuntu users and applications.

1.4 Organization

The organization of the rest of this thesis as follows: Chapter 2 describes the overview of Graphene and design principles. Chapter 3 formally defines the host ABI and provides a specification of a PAL. Chapter 4 discusses the library OS in details. Chapter 5 describes the PAL on Linux, as an example of implementing the host ABI and isolating library OS instances. Chapter 6 discusses SGX-specific challenges to application porting and PAL implementation in an enclave. Chapter 7 evaluates the performance. Chapter 8 presents a quantitative metric for compatibility, to evaluate the completeness of Linux functionality. Chapter 9 discusses the related work. Chapter 10 concludes the thesis.

Chapter 2

System Overview

This chapter gives the overview of Graphene. This thesis divides the development of Graphene into two parts. The first part is a host interface which encapsulates any host-level abstractions, using a PAL (platform adaption layer). The second part is a library OS which emulates a substantial subset of Linux system calls on the host interface. This chapter first introduces the host interface and its design principles, followed by the discussion of the library OS emulation strategies and design trade-offs.

2.1 The PAL ABI

The development of Graphene starts with defining a simple host ABI (application binary interface) called the PAL ABI, containing only OS abstractions essential to target applications. The PAL ABI separates the implementation of an existing system API (application programming interfaces), which determines the compatibility against applications, from hardware abstraction features, such as file systems, network stacks, and device drivers. Graphene moves the system API components to a library OS in the userspace and reimplements the functionality using the PAL ABI. To port Graphene to a new host OS or hardware, OS developers only have to implement the PAL ABI on the target host system API, instead of paying a tremendous cost to translate the whole system API specification. Figure 2.1 illustrates the porting process of Graphene.

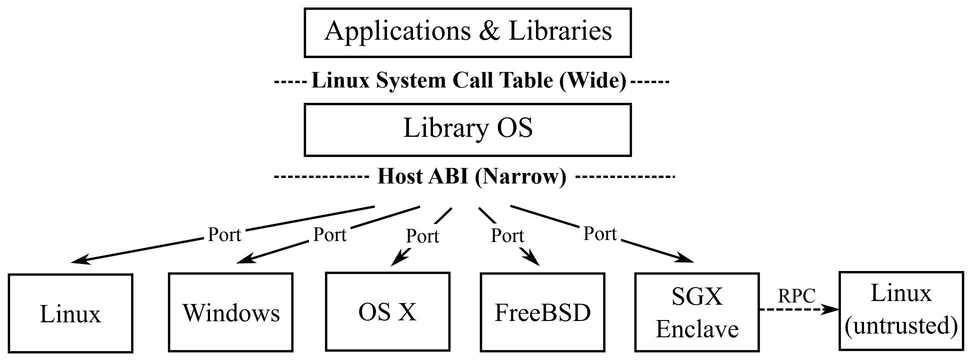


Figure 2.1: Porting model of Graphene.

2.1.1 Platform Adaption Layers

For each host OS or hardware, Graphene uses a thin library called a **platform translation layer (PAL)** to translate among host interfaces. The main purpose of a PAL is to mitigate the semantic gap between the PAL ABI and native host system APIs. By implementing a PAL on a new host OS or hardware, users can reuse the same library OS to run the same collection of unmodified Linux applications.

Graphene currently contains PAL implementations for several popular OSes, including Linux, Windows, OS X, and FreeBSD. Most of these OSes provide a POSIX-like system API similar to the PAL ABI. Due to the similarity, translating most of the PAL ABI to one of the system APIs are straightforward for average OS developers. The PAL ABI is also much smaller than the actual POSIX API, making it extremely portable.

A part of the PAL ABI may be challenging to port on an OS, due to unexpected system assumptions made by the OS. For instance, Windows does not support fine-grained memory deallocation for de-privileged applications. To implement system calls like `munmap()` and `mprotect()`, Graphene needs host ABIs to deallocate or protect virtual memory pages at page granularity. A few host abstractions such as a bulk IPC feature are optional to the host ABI; if a host OS does not support these abstractions, the library OS must fall back to alternatives.

2.1.2 Definitions and Design Principles

Graphene defines 40 calls in the PAL ABI (also called PAL calls), with a set of host abstractions sufficient for library OS development. This thesis defines a **host** of the PAL ABI to be an OS or hypervisor which contains enough OS functionality for running a standalone application or virtual machine. Most of the host targets in Graphene are monolithic OSes, including Linux, Windows, OS X, and FreeBSD. A monolithic OS usually contains a massive amount of system APIs, which is sufficient for implementing the PAL ABI.

A special example of a host is an SGX (Software Guard Extensions) enclaves [122], which restricts OS functionality for security reasons. The restrictions on SGX are results of a strong threat model which distrusts any OS features except ones that are virtualized by the CPUs or migrated into enclaves. The only way to obtain any missing OS features such as storage or networking is to request through RPC (remote procedure call). Requesting untrusted OS services through RPC also introduces new security threats that application developers tend not to anticipate [39, 55]. Due to all the compatibility and security challenges discussed above, this thesis uses SGX as a representative example of a host with unusual assumptions (e.g., threat models) and restrictions compared to a monolithic OS.

The PAL ABI shares several characteristics with a virtual hardware interface exported by a hypervisor. A generic, backward-compatible virtual hardware interface allows an unmodified OS kernel to run inside a virtual machine as on the bare metal. 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 contains higher-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. To compare a VM and a library OS on a spectrum, a VM reuses a whole OS on a wide, backward-compatible virtual hardware interface whereas a library OS implements only system API components on a simplified host ABI.

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

Porting Simplicity. To reduce porting efforts, Graphene defines the PAL ABI using two strategies: first, Graphene significantly reduces both the size and complexity of host OS features that OS developers have to implement. Effectively, Graphene avoids duplicated OS features and handling rare corner cases on the PAL ABI. Second, the definition of the PAL ABI imitates common system APIs in a POSIX-like monolithic OS, to directly translate most calls to a few similar host abstractions. For instance, the stream APIs in the PAL ABI, such as `StreamRead()` and `StreamWrite()` are similar to system calls like `read()` and `write()` exist on Linux, BSD, and POSIX API, or `ReadFile()` and `WriteFile()` on Windows. As the rest of this thesis proves, porting the PAL ABI is straightforward on most monolithic OSes.

Sufficiency for Library OS Development. To develop a library OS with compatibility against a wide range of applications, the PAL ABI contain any OS abstractions that the library OS cannot easily emulate. For most hosts, the host OS abstractions include process creation, memory management, and I/O (typically, files and network connection) [66]. For each type of abstractions, a monolithic OS may define several variants of system APIs with similar functionality. For instance, Linux provides two system calls, `mmap()` and `brk()`, both for memory allocation in a process. `mmap()` allocates larger memory regions with page granularity, whereas `brk()` simply grows a single, continuous heap space for more fine-grained allocation. Many applications such as GCC [7] switch among system API variants in case one of them is unavailable on certain OS distributions. This thesis shows that, by adopting only the semantics of one of these similar APIs or abstractions, the host OS can stay simple with the library OS emulating the rest of APIs. For instance, the PAL ABI includes `VirtMemAlloc()` as a similar feature as `mmap()`, which is sufficient to emulating both `mmap()` and `brk()`.

Graphene defines the PAL ABI partially based on Drawbridge, a library OS for single-process Windows applications. The host ABI of Drawbridge contains 36 functions, and several works have ported the host ABI to different hosts, including Windows, Linux, Barrelfish, and SGX [45, 46, 138, 159]. 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. Section 2.1.3 and Chapter 3 will further discuss the Linux-specific extensions of the PAL ABI.

Migration. The Graphene library OS shares several features of VMs, including checkpointing and migrating a running application. Migrating a process is also the key to emulating copy-on-write forking, on a host without physical memory sharing (e.g., SGX). A hypervisor checkpoints and migrates a VM by snapshotting the VM states above a stateless virtual hardware interface. The PAL ABI is also defined to be statelessness, by ensuring any states in the hosts to be temporary and reproducible to the applications and library OS.

2.1.3 PAL Calls Definitions

Table 2.1 lists the 40 calls defined in the PAL ABI: 25 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 calls are added by Graphene, to implement Linux-specific features. For example, unlike Windows or OS X, Linux 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. On Windows or OS X, the x86-specific segment registers are mostly ignored, and even frequently reset to eliminate attack vectors. Graphene discovers these abstractions as a necessity for implementing a rich Linux library OS.

Graphene introduces five calls for remote procedure call (RPC) between library OS instances in a multi-process application. Graphene simplifies porting multi-process abstractions on each host OS to implementing RPCs. The basic RPC abstraction is a pipe-like RPC stream for message passing between processes. To improve performance, the PAL ABI defines an optional, bulk IPC abstraction to send large chunks of virtual memory across processes.

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 [164] or later extended for this thesis. The rest are inherited from Drawbridge [138].

2.1.4 Host-Enforced Security Isolation

To target multi-tenant environments, Graphene enforces strong security isolation between mutually-untrusting applications running on the same host. The security isolation of Graphene is comparable to running each application in a VM or a container. Just as a virtual hardware interface isolating each VM, the PAL ABI also enforces security isolation between library OS instances.

On a trusted host OS, Graphene delegates security isolation as a host-level feature. The library OS and the application must mutually trust each other, due to lack of internal privilege separation in a process. On each host, a reference monitor enforces security isolation policies, by access control on OS abstractions sharable among processes, including files, network sockets, and RPC streams. Separating security isolation from API implementation simplifies security checks for applications that only require complete protection from other tenants.

In Graphene, one or multiple processes of the same application run in a **sandbox**. Multiple library OS instances coordinate in a sandbox to present a unified OS view to the application. The design simplifies the enforcement of security isolation for multi-process abstractions. Graphene uses the reference monitor to block RPC streams across the sandbox boundary, stopping applications in different sandboxes from accessing multi-process OS states. The current design focuses on security isolation, although we do expect to extend the design for more sophisticated policies in the future.

Threat Model. For most hosts, application trusts the host OSes as well a library OS instances in the same processes. For multiple processes inside a sandbox, the library OSes in these processes also trust each other. Applications or library OSes are not trusted by the host OSes or processes outside of the sandboxes. Applications and library OSes can become the adversary to the host OS, by exploiting vulnerabilities on the PAL ABI.

The threat model of Graphene on SGX contains the adversary from other hosts but excludes the host OS, hypervisor, and any hardware except the CPU from its trusted computing base (TCB). An untrusted OS or hypervisor potentially has lots of opportunities to invade applications or VMs, using Iago attacks [55]. The challenges of porting Graphene to SGX is not limited to resolving the compatibility issues of enclaves but also defending applications and library OSes against untrusted host OSes.

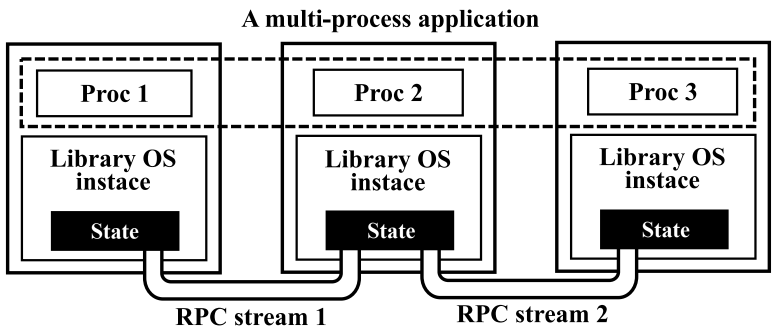


Figure 2.2: Multi-process support model of Graphene library OS. For each process of an application, a library OS instance will serve system calls and keep local OS states. States of multi-process abstractions are shared by coordinating over host-provided RPC streams, creating an illusion of running in single OS for the application.

2.2 The Library OS

This section gives the overview of Graphene, a library OS for reusing unmodified Linux applications on 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, the library OS design of Graphene and previous work [119, 138] eliminates duplicated features between the guest to the host kernel, such as the CPU scheduler or file system drivers, and thus reduces the memory footprint.

A principal drawback for prior library OSes is the inability to support 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. 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 [138] cannot simulate process forking because copy-on-write memory sharing is not a universal OS feature.

In Graphene, multiple library OS instances collaboratively implement Linux abstractions, but present single, shared OS view to the application. Graphene instances coordinate states using message passing over RPC streams. With a distributed POSIX implementation, Graphene can

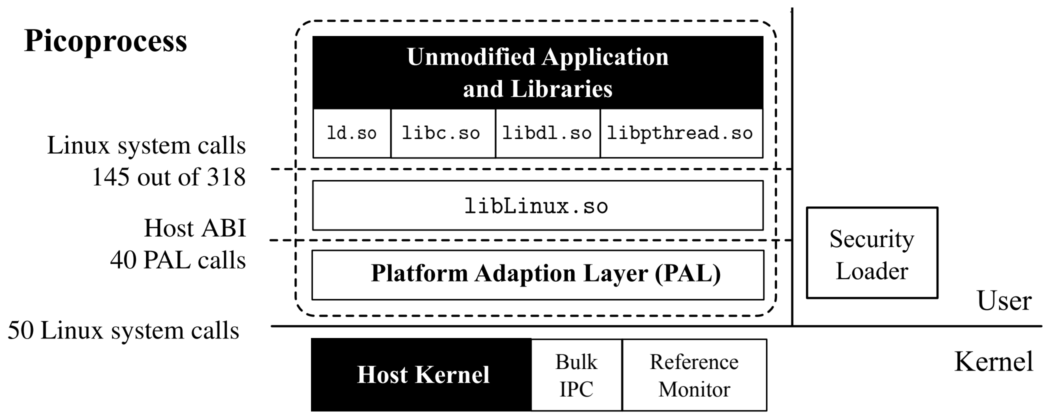


Figure 2.3: Building blocks of Graphene. Black components are unmodified. We modify the four lowest application libraries on Linux: `ld.so` (the ELF linker and loader), `libdl.so` (the dynamic library linker), `libc.so` (standard library C), and `libpthread.so` (standard threading library), that issue 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.

create an illusion of running in a single OS for multiple processes in an application.

2.2.1 The Architecture

A library OS typically executes in either a para-virtualized VM [18, 119] or an OS process called a *picoprocess* [45, 138], with a restricted host ABI. Graphene executes within a picoprocess (Figure 2.3), which includes an unmodified application and its supporting libraries, which run alongside a library OS instance. The Graphene library OS is implemented over the PAL ABI designed to expose very generic abstractions that are easy to port on any host OS.

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()`) while the stack grows down until they meet in the middle. In contrast, the host ABI provides only minimal abstractions for allocating, deallocating, and protecting regions of virtual memory. This clean division of labor encapsulates idiosyncratic abstractions in the library OS.

At a high level, a library OS scoops the layer just below the system call table out of the

OS kernel and refactors the 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 PAL ABI minimizes the amount of application state in the kernel, facilitating migration. A library OS instance can programmatically read and modify its OS state, copy the state to another instance, and the remote instance can load a copy of this state into the OS—analogous to hardware registers. A picoprocess may not modify another picoprocesses' OS states.

2.2.2 Multi-Process Abstractions

A key design feature of UNIX is that users compose simple utilities to create more significant applications. Thus, it is unsurprising that many popular applications are multi-process—an essential feature missing from previous library OSes. 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 just return a fixed value to each application. This single-process design is isolated, but the library OS cannot run a shell script, which requires forking and executing multiple binaries, signaling, waiting, and other PID-based abstractions.

There are two primary design options for multi-process abstractions in library OSes: (1) implement processes and scheduling in the library OS; (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.

Multi-process abstractions inside the library OS also possibly benefit from hardware MMU virtualization, similar to the model explored by Dune [47]. However, this design 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 OSes in multiple picoprocesses collaborate to implement shared abstractions. Graphene supports a rich of Linux multi-process abstractions including copy-on-write forking, `execve()`, 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 pipe-like RPC stream, and B's instance then calls the appropriate signal handler. The host OS is unaware of the implementation of multi-process abstractions, as well as security isolation of the corresponding states.

The Graphene library OS is also capable of gracefully handling disconnection from other library OSes, facilitating dynamic application sandboxing. RPC streams may disconnect 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 disconnect RPC streams from the parent by the reference monitor, the library OS will interpret the event as if the other process terminated, close any open pipes, and deliver exit notifications.

Comparison with Microkernels. The building blocks of Graphene are very similar to the system abstractions of a microkernel [28, 43, 56, 71, 101, 113, 114], except a microkernel often has an even narrower, more restricted interface than the host ABI. A multi-server microkernel system, such as GNU Hurd [76] or Mach-US [160], implements Unix abstractions across a set of daemons that are shared by all processes in the system. Graphene, on the other hand, implements system abstractions as a library in the application's address space and coordinates library state among picoprocesses to implement shared abstractions. Graphene guarantees isolation equivalent to running an application on a dedicated VM; it is similar to implementing the security isolation model 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., platform adaption layer) to reuse OS components, such as the system call table and namespaces in a library OS. Graphene defines a host ABI, as a new boundary between the OS and userspace. The host ABI is designed to be simple enough to port on a new host (containing 40 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 that occur in OS development, such as defending applications inside of an SGX enclave.

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 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 PAL ABI as a key component of the Graphene architecture. The PAL ABI acts as a boundary between the host and the library OS, and defines several UNIX-like features. This thesis defines the PAL ABI primarily based on two criteria: *simplicity*, to bound the development effort per host and *sufficiency*, to encapsulate enough host abstractions for library OS development. This chapter also explains the rationale behind the definitions.

3.1 PAL Calling Convention

The PAL ABI partially adopts the x86-64 Linux convention. A PAL contains a simple run-time loader that can load the library OS as an ELF (executable and linkable format) binary [74], similar to `ld.so` for loading a user library. Inheriting the 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 simplifying debugging with GDB since GDB 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 OS X applications follow different calling conventions and binary formats from Linux applications. On each host, a PAL acts as a simple ELF loader. On Windows and OS X, the PALs are developed as application binaries recognized by the host OSes, but contain ELF loading code for linking Linux application binaries and the library OS. The PALs also resolve calling convention inconsistency such as placement of function arguments in registers or on stacks.

Error Codes. For clarity, a PAL call only returns two types of results: a non-zero number or pointer if the call succeeds, or zero if it fails. Unlike the Linux convention, the host ABI does not return negative values as error codes (e.g., `-EINVAL`). For applications, interpreting negative return values from system APIs causes obscure corner cases that applications may easily miss. For instance, error codes of failed `read()` system calls may be misinterpreted as negative input sizes, causing bugs in applications. Instead, a PAL delivers the failure of a PAL call as an exception, so that the library OS can assign a handler to capture the failure. The design avoids confusing semantics as interpreting negative return values.

Dynamic Linking vs Static Linking. Graphene dynamically links an application, library OS, and the PAL inside of a user process. Dynamic linking ensures complete reuse of an unmodified application, as well as an unmodified library OS. Graphene allows the binaries of application, library OS, and PAL to be deployed individually to the users. The dynamic linking of applications is a prerequisite to running most Linux applications without modification or recompilation.

There are cases where users prefer static linking on a host (e.g., SGX) and consider recompilation acceptable. Compiling an application, library OS, and PAL into a single binary is useful for reducing the memory footprint in unikernels [119], by compiling out unnecessary code and data segments. 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 and left for future work.

3.2 The PAL ABI

This section defines the PAL ABI, as a developer's guide to porting. This section describes the usage of each PAL call, followed by justifications of simplicity and sufficiency.

3.2.1 Stream I/O

An OS typically supports three types of I/O: (1) **storage**, for externalizing data to a permanent store; (2) **network**, for exchanging data with another machine over the internet; (3) **RPC** (remote procedure call), for connecting concurrent applications or processes. A host OS must contain these I/O abstractions and manages the related I/O devices such as storage and network devices. It is also important to share these I/O abstractions among multiple processes of an application.

The I/O abstractions are simple byte streams. Byte streams send or receive data over I/O devices or in-kernel queues as continuous byte sequences. On a storage device, a byte stream is logically stored as a sequential file, but physically divided into blocks. On a network device, the hardware sends and receives packets for a byte stream, using identification with an IP address and a port number. An RPC stream can be simply a FIFO (first-in-first-out), which applications or processes use to exchange messages. Similar to the API of a UNIX-style OS, which treats "everything as a file descriptor" [141], the PAL ABI encapsulates different types of I/O devices through unified APIs such as reading and writing. Byte streams simplify managing various types of I/O in the library OS.

The PAL ABI identifies I/O streams by URIs (unified resource identifier). A URI is a unique name to describe an I/O stream, including a prefix to identify the I/O type and the information for locating an I/O stream on the related I/O device. The prefix of URI can be one of the following keywords: "`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 instance, a file URI identifies a file by the path in the host file system. A network URI identifies the IP address and port number of a network connection. The URIs standardize identification of I/O resources on host OSes.

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

3.2.1.1 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 or creates an I/O stream based on the URI given by `stream_uri`. 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 I/O devices. If `StreamOpen()` succeeds, it returns a **stream handle**. The library OS stores the stream handle as a reference to the opened I/O stream. A stream handle is an opaque pointer, and the library OS only references the handle as an identifier instead of trying to interpret the handle 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 parameters of `StreamOpen()` specify the options for opening an I/O stream:

- `access_flags` specifies access mode of the I/O stream, to 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. The access modes specify the basic permissions for the library OS to access the opened stream. The access flags are checked by the host OS, based on security policies. For example, the library OS can only append data to a file opened with the `APPEND` mode.
- `share_flags` specifies permissions to share a regular file (ignored for other types of I/O streams) with other applications. `share_flags` can be a combination of six different values: `OWNER_R`, `OWNER_W`, and `OWNER_X` represent the permissions to be read, written, and executed by the creator of the file; `OTHER_R`, `OTHER_W`, and `OTHER_X` represent the permissions to be read, written, and executed by everyone else.
- `create_flags` specifies the semantics of file creation when the file is nonexistent on the host file system. With `TRY_CREATE`, `StreamOpen()` creates the file only when the file is nonexistent. If `ALWAYS_CREATE`, the PAL call 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 the URIs, `StreamOpen()` can create two types of I/O streams: A byte stream and a **server handle** to receive remote connections. A server handle cannot be directly read or written but can be given to `ServerWaitForClient()` to block until the next client connection. The reason that the PAL ABI must support I/O servers is that receiving remote connections requires controls at the TCP/IP layer and allocating resources in the network stack, and thus cannot be emulated in the library OS 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]` accesses a regular file on the host file system. The file is identified by a path, containing directory names from the file system root (/) or current working directory (CWD). The current working directory is the location where the PAL starts and does not change during execution. There could be security risks that the target of a relative path may be ambiguous, especially if the path starts with a "dot-dot" (i.e., walking back a directory). Therefore, a reference monitor should always canonicalize a relative path before checking against security policies.
- `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 exist 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. The address of a TCP or UDP server can be either IPv4 or IPv6, with a port number smaller than 65536.
- `pipe.srv:[name]` or `pipe:[name]` create a named RPC server or a connection to an RPC server. The name of an 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, compared to using a network stream locally. An RPC stream is supposed to have lower latency than a network stream.

`StreamOpen()` is easy to port on most host OSes because it limits the types of I/O streams to three basic and common forms: files, TCP or UDP network sockets, and RPC streams. The semantics of StreamOpen() belong to a subset of POSIX's open() semantics, which coincides with Windows's OpenFile() semantics, with parameter options available in both host OSes. URIs are also ubiquitously recognized and easily translated to host-specific identifiers.

This thesis argues that StreamOpen() supports sufficient I/O abstractions for emulating Linux I/O features that most server or command-line applications need. Low-level abstractions such as storage blocks or raw sockets are mostly only important to administration-type applications such as fsck and ipconfig. Other Linux I/O features, such as asynchronous I/O and close-on-execve(), are emulatable in the library OS.

3.2.1.2 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 migrating stream handles. The PAL calls only read or write at absolute offsets from the beginning of an opened file, and do not rely on states stored in the host OSes as the file cursors. Stateless file access allows migrating a library OS to another process or host without migrating the host OS states. All host OS states associated with an I/O stream is only meaningful to the host and can always be recreated by the library OS.

The PAL calls do not support asynchronous I/O, or peeking into network or RPC buffers. Asynchronous I/O and buffer peeking are essential OS features to many applications and the library OS can emulate these features using StreamRead(), StreamWrite() and other PAL calls (e.g., ObjectsWaitAny()). The library OS can maintain in-memory buffers to store data prematurely received from an I/O stream, to service asynchronous I/O and buffer peeking. Chapter 4 further discusses these features in detail.

The fact that all three types of I/O streams supported by `StreamOpen()` are simply byte streams justifies the portability of the PAL calls. `StreamRead()` and `StreamWrite()` cover all access types on the I/O streams, using synchronous semantics available in most host OSes. `StreamRead()` and `StreamWrite()` also cover both random and sequential access to a regular file and transferring over a TCP or UDP socket. Other I/O operations (e.g., asynchronous I/O) can mostly be emulated in the library OS using the two PAL calls.

Alternatives. An alternative strategy is to define asynchronous I/O in the host ABI instead of synchronous I/O. Although asynchronous I/O may not be universally portable, a library OS can easily emulate synchronous I/O with asynchronous I/O using a thread to poll I/O events. Asynchronous I/O potentially has more predictable semantics, because the library OS can explicitly tell which PAL calls will be blocking. This strategy is later taken by Bascule [45]. Graphene chooses synchronous I/O in the PAL ABI to prioritize portability, but will explore alternative designs in the future.

3.2.1.3 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 library OS can specify an address in memory to map the file, or a null address (i.e., zero) to map at any address decided by the host OS. `expect_addr`, `offset`, and `size` must be aligned to allocation granularity decided by the host OS (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 mapped address; otherwise, the PAL call returns a null pointer.

The PAL ABI defines `StreamMap()` for two reasons. First, memory-mapped I/O is suitable for file access in certain applications, and cannot be fully emulated using `StreamRead()` and `StreamWrite()`. An application may choose memory-mapped I/O for efficiency, especially for smaller, frequent file reads and writes. Second, memory-mapped I/O is asynchronous by its nature. An OS is supposed to lazily flush the data written to a file-backed memory mapping. The feature is difficult to emulate without an efficient way to mark recently-updated pages (e.g., using page table dirty bits).

Although `StreamMap()` allows multiple processes to map the same file into memory, it is hard to guarantee coherent file sharing on every host OSes. To support memory-mapped I/O, a complete implementation of the PAL ABI must include coherent file sharing. For some host target where coherent file sharing is impossible, such as an SGX enclave, the PAL ABI implementation must be considered incomplete. Potentially the library OS can implement a remote memory access protocol, with significant overheads to intercept memory access and trace unflushed contents. For most monolithic host OSes, `StreamMap()` with coherent file sharing is easy to implement using APIs like `mmap()` (POSIX) and `CreateFileMapping()` (Windows).

`StreamMap()` is sufficient for most use cases of memory-mapped I/O. The library OS can further manage file-backed memory using PAL calls for page management (e.g., `VirtMemProtect()` and `VirtMemFree()`). A few hardware features, such as huge pages and data execution protection (DEP), will require extensions to the PAL ABI though. Fortunately, these hardware features are mostly considered optional in applications and rarely dictate usability.

3.2.1.4 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 byte stream. Before any connection arrives, `ServerWaitforClient()` blocks indefinitely. 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. Other than adjusting backlog sizes, ServerWaitForClient() covers most of the server-specific behaviors and is easy to port on most host OSes.

3.2.1.5 File and Stream Attributes

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

StreamAttrQuerybyHandle() and StreamAttrQuery() query the attributes of an I/O stream, and fill in a data structure as STREAM_ATTRS. The only difference between the two PAL calls is that StreamAttrQuerybyHandle() queries an opened stream handle whereas StreamAttrQuery() queries a URI without opening the I/O stream in advance. StreamAttrQuery() is convenient for querying stream attributes when the library OS does not plan 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 microseconds) before the other end abruptly disconnects the stream. 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 for configuring a network stream. Currently, network_options accepts the following generic options: KEEPALIVE (enabling keep-alive messages), TCP_NODELAY (no delay in sending small data), and TCP_QUICKACK (no delay in sending ACK responses).

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

Introduced by Graphene, StreamAttrSetByHandle() can configure the attributes of a file or an I/O stream in the host OS. StreamAttrSetByHandle() accepts an updated STREAM_ATTRS data structure, which contains new attributes to assign to the I/O stream.

It is a dilemma to decide which stream attributes to define in STREAM_ATTRS. especially for a network socket. A network socket in a monolithic OS often provides several options to fine-tune the behavior of network stacks and drivers. Exposing these options on the PAL ABI allows the library OS to emulate network socket APIs more completely. However, extending the PAL ABI with network socket options also compromises portability on host OSes that do not provide equivalence of these features. Eventually, the library OS should not expect every attributes defined in STREAM_ATTRS to be configurable on every host OSes.

```
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 the file. StreamSetLength() can explicitly request the host to expand a file to an appropriate length so that sequential 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.

Potentially `StreamSetLength()` can be absorbed by `StreamAttrSetByHandle()`. Currently, the PAL call is preserved as a legacy from previous versions of the PAL ABI and as an optimization for file operations which need to frequently update file sizes (`StreamAttrSetByHandle()` is much slower than `StreamSetLength()`).

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()` opens a file URI that points to a directory, such as "`file:/usr/bin`", it returns a stream handle which allows consecutive `StreamRead()` calls to read the file list as a byte stream. The stream handle returns 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

The PAL ABI manages page resources using an abstraction as a **virtual memory area (VMA)**. A VMA is an aligned, non-overlapping region in a process. The PAL ABI can creates two different types of VMAs: a file-backed VMA, created by `StreamMap()`, and an anonymous VMA, created by another PAL call called `VirtMemAlloc()`.

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

`VirtMemAlloc()` creates an anonymous VMA. When `VirtMemAlloc()` is given an expected address, the host OS must allocate memory at the exact address, or it should return failure. If no address is given (`expect_addr` is NULL), `VirtMemAlloc()` can create the VMA at wherever the host OS sees fit, as long as the VMA does not overlap with any VMAs previously allocated. 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 to StreamMap() but without WRITE_COPY). If VirtMemAlloc() succeeds, it returns the starting address of the created VMA, which the library OS 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 three PAL calls for allocating, freeing, and protecting virtual pages are generally portable on POSIX-style host OSes. Applications usually rely on similar coarse-grained page management features in the host OS to implement user-level fine-grained object allocation, such as malloc() or class instantiation in managed language runtimes.

However, the real challenge to porting the PAL ABI is to accommodate different allocation granularities among host OSes. A POSIX-style OS often assumes dynamic allocation with page granularity (normally with four-kilobyte pages); the assumption is deeply ingrained in the page fault handler sand page table management inside an OS like Linux or BSD; the page management component in an OS is usually 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 [169]; VT provides a nested page table to virtualizes page table management and page fault handling to the library OS.

3.2.3 CPU Scheduling

The host ABI for CPU scheduling includes two abstractions: thread creation and scheduling primitives for inter-thread synchronization and coordination. Threading in the host OS requires a CPU scheduler to dynamically assign a non-blocking thread to an idle CPU core until next epoch for scheduling. The host OS usually implements one or several scheduling algorithms and also defines APIs for applications to configure scheduler parameters. Scheduling algorithms and APIs are mostly idiosyncratic to each host OS and hardly portable on every host OSes.

As a compromise, Graphene focuses on defining host ABIs for CPU scheduling features essential to application usability. For instance, applications may depend on multiple threads to execute concurrently, either on different CPU cores or on the same CPU core with a time-sharing model. Scheduling algorithms in the host OSes must satisfy certain criteria, such as fairness, throughput, and reasonable CPU utilization. As long as the host OSes have chosen a general-purpose, maturely-implemented scheduling algorithm, the PAL ABI can omit features for configuring scheduler parameters.

3.2.3.1 Creating or Terminating a Thread

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

`ThreadCreate()` creates a thread available for scheduling in the host OS. The parameters specify the initial state of a new thread, including the function to start thread execution and a parameter to the function. As soon as `ThreadCreate()` successfully returns, the caller thread and the new thread can be both scheduled by the host OS. `ThreadCreate()` returns a thread handle to reference the new thread in the caller.

To improve portability, Graphene simplifies the definition of `ThreadCreate()` in several ways. First, `ThreadCreate()` does not accept an additional parameter to specify the initial stack. The simplification helps to port `ThreadCreate()` on a host where a new thread cannot start on an arbitrarily-assigned stack. For instance, an SGX enclave statically defines the stack address of each thread to prevent the host OS from manipulating enclave thread execution. On most host OSes, a new thread created by `ThreadCreate()` starts on a fixed-size stack, but the library OS can easily swap the stack with a much larger one. Second, `ThreadCreate()` takes no creation options except a starting function and a parameter. Every thread created by `ThreadCreate()` should look identical to the host OS, to keep the abstraction portable on various host options.

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

ThreadExit() terminates the current thread. 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 the current thread, including the initial stack.

3.2.3.2 Scheduling a Thread

The PAL ABI defines several calls to interrupt a thread, either blocking or running or to allow a running thread to give up CPU resources voluntarily. The purpose of these scheduling APIs is to prevent a thread from busily waiting for a specific condition, such as setting a variable to a specific value or the arrival of a certain time in the future. Busy-waiting wastes CPU cycles, and can potentially block application execution if the host OS has no enough CPU cores to schedule each thread or does not implement a time-slicing scheduling algorithm (e.g., round-robin).

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

ThreadDelay() and ThreadYield() suspend the current thread for rescheduling in the host OS. ThreadDelay() suspends the current thread for the given timespan (delay_microsec, in microseconds). If the thread is suspended successfully and rescheduled after the expiration of the specified period, ThreadDelay() returns zero after resuming thread execution. If the thread is rescheduled prematurely, due to interruption of other threads, ThreadDelay() returns the remaining time in microseconds.

ThreadYield() simply yields the current execution and requests for rescheduling the current thread in the host OS. By calling ThreadYield(), a thread can requests for rescheduling when it expects to wait for certain conditions. When a thread calls ThreadYield(), the host scheduler will suspend the current time slice of the thread, and rerun the scheduling algorithm to select a runnable thread (can be the same thread if there is no other competitor).

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

Graphene introduces ThreadInterrupt() as a PAL call for interrupting a thread and forcing the thread to enter an exception handler immediately. There are two reasons for defining

`ThreadInterrupt()`. First, `ThreadInterrupt()` can interrupt a suspended thread, and force the thread to resume execution immediately. Second, `ThreadInterrupt()` can interrupt a running thread from an infinite waiting loop. Without `ThreadInterrupt()`, a running thread can only detect events at a certain "checkpoint" in the library OS.

The three abstractions defined for suspension and rescheduling commonly exist on most host OSes, including Linux, Windows, and OS X. System APIs similar to `ThreadDelay()` and `ThreadYield()` exist in most OSes, with slight but mitigable definition differences. On a POSIX-compliant OS, a PAL can implement `ThreadInterrupt()` using a user signal (e.g., `SIGUSR1`); or on other OSes such as Windows, similar inter-thread communication mechanisms exist for similar reasons.

Scheduler Parameters. The PAL ABI currently contains no APIs for the library OS to configure scheduler parameters in the host OS. Linux and other OSes allow applications to configure scheduling parameters, such as scheduling priorities and policies, to improve CPU utilization. For simplicity, the PAL ABI delegates scheduling to the host scheduler, and only allows host-level configuration for scheduler parameters. As a result, the library OS cannot emulate any Linux system APIs for configuring scheduler parameters, such as `sched_setparam()`.

Luckily, scheduler parameters does not impact most applications targeted by Graphene. In general, applications can progress without setting scheduling priorities or policies, but may suffer poor performance. A rare exception is when an application set the CPU affinity of two collaborating threads to ensure concurrent execution. Between a producer thread and a consumer thread, failing to schedule the threads on individual CPU cores may cause the threads to deadlock. Consider the following scenario: the consumer thread A may busily wait for the producer thread B to deliver a new job, but never yield the CPU to allow thread B to proceed its execution. 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 must be smaller than the total number of CPU cores (specified in the PAL control block).

3.2.3.3 Scheduling Primitives

The PAL ABI defines two scheduling primitives for synchronization between threads: mutually-exclusive (mutex) locking and event waiting. These scheduling primitives improve user-space synchronization implemented by atomic instructions or compare-and-swap (CAS). The primitives prevent a thread from spinning on a CPU core until the state of a lock or an event is atomically changed, by suspending the thread in the host OS.

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

MutexCreate() creates a handle for 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 events. Any thread can use EventSet() to signal an event. Signaling a synchronization event wakes up exactly one waiting thread to continue its execution. A synchronization event coordinates threads that cooperate as producers and consumers; a producer thread can signal exactly one blocking consumer at a time. On the other hand, a notification event stays signaled until another thread manually resets the event using EventClear(). A notification event notifies the occurrence of a one-time event, such as the start or termination of execution. ObjectsWaitAny() is also used to wait for event signaling.

The definition of the two scheduling primitives covers two typical types of synchronization behaviors. A mutex enforces atomicity of a critical execution section. An event enforces dependency relationship between threads. Both primitives are easy to port on most host OSes; the host

ABI directly adopts the definition from the Windows API and can be easily implemented on Linux or similar OSes using futexes or POSIX thread (pthread) APIs.

3.2.3.4 Waiting for Scheduling Events

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

`ObjectsWaitAny()` blocks the current thread for specific events listed in a handle array (specified by `handle_array` and `handle_num`). A common usage of `ObjectsWaitAny()` is to block on a scheduling primitive, such as a mutex lock or a notification event. If a certain event happens on one of the listed handles `ObjectsWaitAny()` resumes thread execution and returns the handle to the caller. `ObjectsWaitAny()` can only block on exactly one mutex lock or event but can wait for multiple I/O events. When waiting on I/O events, `ObjectsWaitAny()` blocks until one of the listed stream handles receives incoming data or connections or encounters failures such as I/O stream shutdown.

`ObjectsWaitAny()` takes a `timeout` argument to prevent waiting for an event indefinitely. If the timeout expires before any event occurs, `ObjectsWaitAny()` stops blocking and returns no handle.

`ObjectsWaitAny()` can poll multiple stream handles until an I/O event occurs such as receiving inbound data or sudden failure. Unlike a mutex lock or an event object, a stream handle can trigger 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 [45].

```
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.

3.2.3.5 Thread-Local Storage

On some OSes, such as Linux and Windows, applications require a thread-local storage (TLS) to store thread-private variables or maintain a thread control block (TCB). On x86-64, TLS is often referenced by one of the FS and GS segment registers, to improve the performance of accessing any variable in the TLS. As a convention of Linux, many Linux application executables contain hard-coded access to TLS using the FS register. Since setting the value of the FS/GS registers is a privileged operations, the PAL ABI requires a call to enter the host kernel and set the registers for referencing TLS.

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

The PAL 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. On success, the PAL call returns the current value of FS/GS register.

Unfortunately, the portability of `SegmentRegisterAccess()` depends on the choice of host OSes. Linux and similar OSes allow setting FS/GS register in the userspace due to heavy usage of the FS register in the standard C library. However, in other OSes, especially Windows and OS X, 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. OS X's xnu kernel considers FS/GS registers to be of no concrete use. These OS kernels periodically reset the FS/GS registers to mitigate any user attempt of changing them. If a host OS fails to implement `SegmentRegisterAccess()`, the library OS may have to develop workarounds such as binary translation to virtualize TLS access.

3.2.4 Processes

The PAL ABI creates clean, brand-new processes for multi-process applications. A process, or a **picoprocess** in the perspective of the library OS, contains a new PAL instance, a new library OS, and the application specified to the PAL ABI. The PAL ABI is designed to simplify porting a multi-process applications, by dropping the assumption of coherent memory sharing across processes. Therefore, the PAL ABI chooses a completely different process creation model from the typical copy-on-write forking model of UNIX-style OSes.

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

`ProcessCreate()` creates a clean process to load an application executable specified by `application_uri`. `ProcessCreate()` also allows specifying a manifest file (`manifest_uri`) for user policy configuration, as well as command-line arguments (`args`) passed to the new process. `ProcessCreate()` is equivalent to relaunching the specified application in Graphene, except two distinctions: (1) `ProcessCreate()` returns a process handle to its caller; (2) a process created by `ProcessCreate()` naturally belongs to the same *sandbox* as its parent. Section 3.2.5 will discuss the sandbox abstraction in detail.

3.2.4.1 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 instance, most host OSes bound network connections with the processes that first accept the connections, and only allow sharing connections through inheriting file descriptors 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 PAL ABI introduces `RpcSendHandle()` and `RpcRecvHandle()` for sharing I/O stream handles over an RPC stream (a process handle also an RPC stream). `RpcSendHandle()` migrates the host state of a stream handle (`cargo`) over another RPC stream. `RpcSendHandle()` 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.

3.2.4.2 Bulk IPC (Physical Memory Store)

The PAL ABI introduces an optional bulk IPC feature, as an alternative to RPC streams. 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 process 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 an 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 in each sandbox so that unrelated processes 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 process closes the corresponding store handles, and no data remains 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.

Sandboxes delegate enforcement of security isolation to the host OSes. 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 PAL 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, the PAL ABI also contains miscellaneous features, such as exception handling and querying system times. Some of the miscellaneous features, such as exception handling, are specifically introduced for implementing Linux functionality in the library OS. This section lists these miscellaneous abstractions in the PAL ABI.

3.2.6.1 Exception Handling

The exception handling in the PAL 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 opaque 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 thread in the same process. If `ExceptionSetHandler()` is given a null pointer as the handler, it cancels any handler previously assigned to the exception. If the library OS does not assign a handler, the default behavior of handling the exception is to kill the whole process.

3.2.6.2 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 [118], combined with the Time Stamp Counter (TSC), a CPU counter tracking the number of cycles since the system reset. `SystemTimeQuery()` exports a reliable time source to the guest, based on the calculation of any arbitrary time sources used in the host.

3.2.6.3 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 PAL ABI consists of a sufficient set of simple, UNIX-like OS features. The goal of defining the PAL ABI is to ensure host abstractions that are easy to port on various host OS but also sufficient for developing a library OS with rich features. Individual PAL call, either manages a common hardware resource, such as memory pages or CPUs, or encapsulates a host OS feature, 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 challenging to port on certain host OSes (e.g., Windows) are optional and require workarounds in the library OS.

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 main 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 Implementing the Library OS

The library OS of Graphene, or `libLinux`, is a single library that resides beneath a Linux application to reproduce compatible Linux features and APIs. `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 behaviors observed in a running Linux kernel.

`libLinux` currently implements 145 system calls, 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 [166] indicates that only 40 system calls are indispensable to every application 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. Graphene also 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

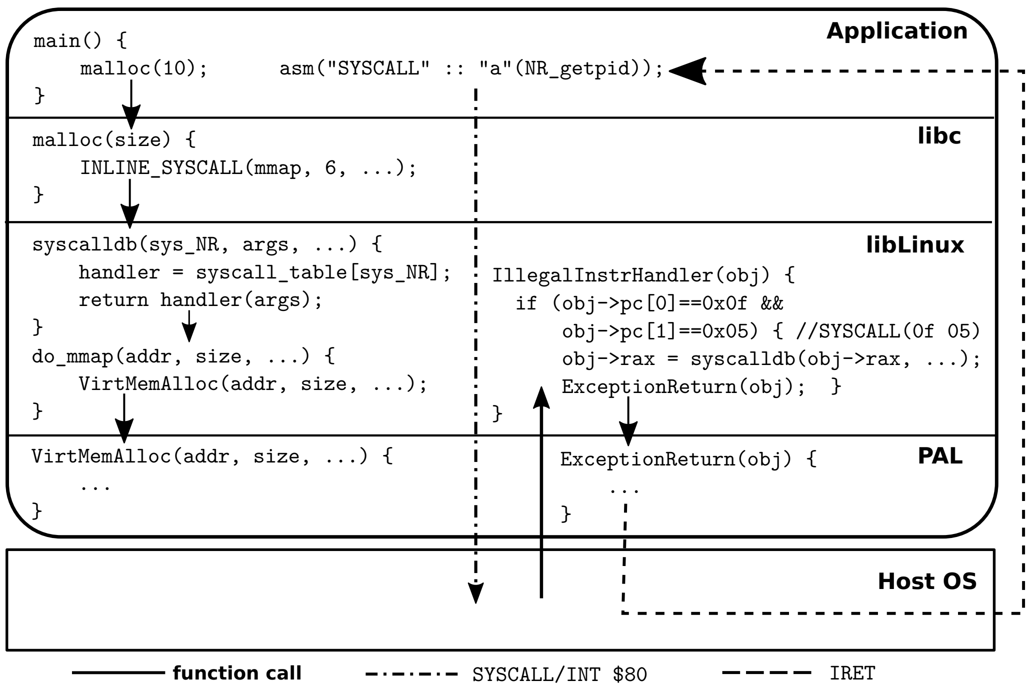
`libLinux` transparently redirects system calls from a Linux application. In a Linux kernel, a system call handler triggers the kernel operations whenever an application executes a "`SYSCALL`" or "`INT $80`" instruction. The interrupt handler switches the application context and jumps to the kernel routines that service the requested 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 rely on libc functions to access OS features instead 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], for application binaries from any GNU Linux distributions, including Ubuntu [167]. Graphene can be configured to use other libc variants, such as uClibc [24] and musl [15] if an application finds them sufficient.

Graphene modifies only 943 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. 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 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.



```
main() {
    malloc(10);      asm("SYSCALL" :: "a"(NR_getpid));
}
```

```
malloc(size) {
    INLINE_SYSCALL(mmap, 6, ...);
}
```

```
syscalldb(sys_NR, args, ...) {
    handler = syscall_table[sys_NR];
    return handler(args);
}
do_mmap(addr, size, ...) {
    VirtMemAlloc(addr, size, ...);
}
```

```
IllegalInstrHandler(obj) {
  if (obj->pc[0]==0x0f &&
      obj->pc[1]==0x05) { //SYSCALL(0f 05)
      obj->rax = syscalldb(obj->rax, ...);
      ExceptionReturn(obj);  }
}
```

```
VirtMemAlloc(addr, size, ...) {
    ...
}
```

```
ExceptionReturn(obj) {
    ...
}
```

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 [147]. 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 an architectural restriction to detect direct system calls. For example, an SGX enclave restricts an application to invoke any system call by triggering an "illegal instruction" exception. If the host detects a direct system call, the PAL delivers an exception to the handler assigned by `libLinux`. 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. When handling an exception, the application at least switches context twice, including both triggering the exception handler and returning to the original execution. A mitigation for the overhead is to rewrite the hard-coded `SYSCALL` or `INT $80` inside each application binary during the loading. `libLinux` can also passively replace the instructions whenever the host detects a direct system call and triggers an exception handler. Graphene leaves system call interception by binary rewriting as a future feature to explore in `libLinux`.

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 an RPC stream, represents the low-level, privileged resources of a host OS. Graphene drops the prerequisite of virtualizing any other low-level resources to run `libLinux` as a ubiquitous compatibility layer.

`libLinux`'s role in resource management is to allocate host abstractions as requests for host-managed resources. For instance, `libLinux` allocates virtual pages using the PAL ABI to request for physical pages in the host OS. Such a library OS design operates on the faith that the host OS will assign physical pages to VMAs with both fairness aad efficiency. Unless the allocation exceeds user quotas or other host-level 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 [20] or Perl [19] runtime, has a similar role as `libLinux`. A language runtime commonly uses existing system APIs to request for resources needed by an application. For example, a language runtime may use `mmap()` 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 assign hardware resources to applications efficiently.

`libLinux` reproduces the Linux resource allocation models. Take page management for example. Linux supports several ways of memory allocations, including `mmap()` for allocating a fixed-size VMA, stack allocation, and `brk()` for more fine-grained heap allocation. Since `libLinux` does not directly manage physical pages, it requires different emulation strategies to implement the allocation models expected by an application. A strategy is to "overallocate" certain host abstractions. The purpose of overallocation is to keep the flexibility of adjusting the host resources afterward from the library OS.

The Comparison with Alternative Approaches. Virtualization allows guest OSes to directly manage hardware resources. A virtual machine often runs an unmodified OS kernel, containing drivers to manage virtualized or dedicated resources. To fully virtual hardware resources, a hypervisor can either emulate a virtual hardware interface, such as QEMU [21], or leverage hardware virtualization, such as IOMMU [52]. Both virtualization strategies grant a virtual machine with direct control over hardware resource management.

Exokernel [72] 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 provide opportunities for domain-based optimization in 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.

Regarding resource management, Exokernel and Graphene have made different decisions for the division of labor between the host and library OS. Exokernel prioritizes the efficiency of resource management for each application. To eliminate the 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. 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 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.

4.2.1.1 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 4.2.1.3.

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.

4.2.1.2 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.

4.2.1.3 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. [165], 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.

4.2.2 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 normally used by administration-type programs. Most networked applications, including server-side and client-side applications, tends to treat a network socket as a contiguous I/O stream. 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 network protocol suites or families on top of the network interfaces. A virtualization solution moves or duplicates the network stack to a guest OS, and allows the guest OS to implement its 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.

4.2.3 Virtual Address Space

A Linux application expects a contiguous, sufficient virtual address space to allocate memory regions for storing code or data. 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. An OS is responsible of maintaining a unique, consistent mapping between virtual memory areas (VMAs) and physical pages, and preventing collision of VMAs. The Linux kernel, specifically, provides several ways of allocating pages, such as allocation by `mmap()` and `brk()`, and transparently growing a user stack downward. Applications depend on different memory allocation mechanisms of a Linux kernel.

`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.

Application Data Segments. `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 determined 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.

4.2.4 Threads

Linux application developers usually use the POSIX threads, or **pthreads** to program multi-threaded applications. 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, libpthreadcontains several scheduling or synchronization primitives, such as mutexes, conditional variables, and barriers.

The creation of a pthread in libpthread uses 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 a unique thread-local storage (TLS), which contains a **thread control block** and thread-private variables. A thread control block stores the private states of a pthread, including at least a thread identifier, followed by thread-private variables (variables defined with the `__thread` keyword) of the application and libraries. Both the thread control block 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 TLS 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 `arch_prctl()` and `clone()`.

Besides thread creation, `libpthread` also provides a collection of synchronization primitives, including mutexes, 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 [75] have shown the versatility of futexes in 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 queue as it wants, using `EventSet()`.

4.3 Multi-Process Applications

This section explains an 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 Graphene. `ProcessCreate()` in the PAL ABI creates a new process 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 process. Graphene drops the assumption that each of its hosts can share 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 to copy to 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 an RPC stream.

To fully emulate `fork()`, `libLinux` implements a checkpointing 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 checkpointing 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 checkpointing 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 tracking 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 an 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 Executables

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 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 an 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 accesses 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

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.

amongst themselves. `libLinux` implements inter-process signaling using RPC messaging. 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 for 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 processes needs 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 cases, 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 formatting the header of each message and determining the callback function for processing the message. GNU Hurd [76] 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. For avoiding 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. A PAL creates the broadcast stream the process 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 an RPC server identified by its URI. Because of the security isolation in the host, only processes in the same sandbox can connect to each other through RPC. If a process leaves a sandbox to create a new one, its broadcast stream is shutdown and replaced with a new one, connected only between a parent process 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 **process ID allocation**. Because each process 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 process 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 processes 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 processes. For a namespace like the process ID namespace, the leader subdivides the namespace for each process to reduce the RPC cost of allocation. For example, the leader might allocate 50 process IDs to a process which intends to clone a new thread or process. The process who receives 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 signals. 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 to a process or a process group, every thread within the

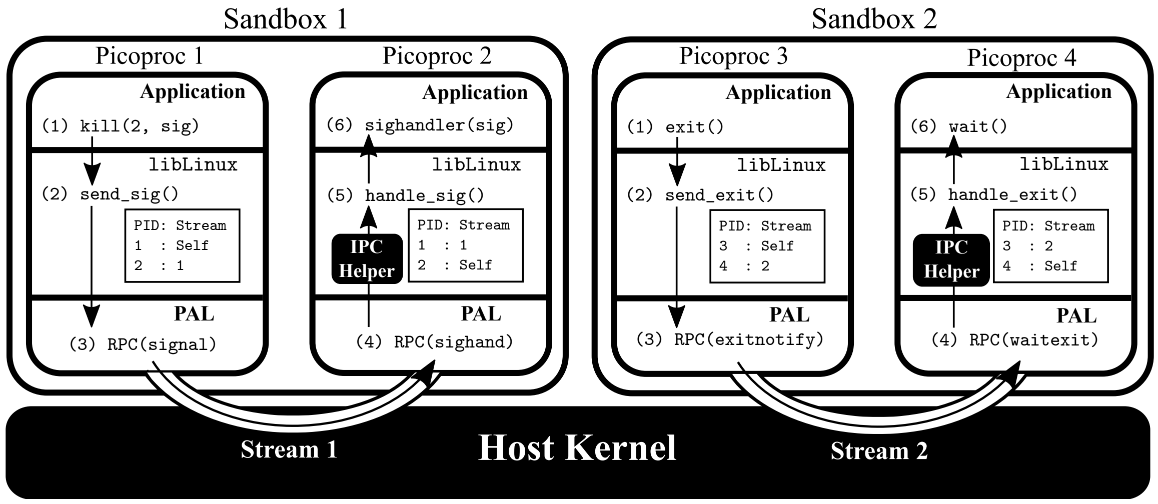


Figure 4.2: Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management. The location of each PID is tracked in `libLinux`; Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation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

Linux also delivers exit notifications as signals. 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()`. `libLinux` exchanges exit notifications between parent and child picoprocesses over RPC streams.

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 processes, so that any process can allocate a resource ID from local state instead of involving the leader. Unlike the resource 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 an ID mapping is still valid. A key which does not have a valid mapping can be assigned to a resource ID by any process 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, delivery and reception of messages must go via a central owner. In the initial implementation of `libLinux`, sending or receiving messages remotely over an 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 picoprocesses that previously accessed the queue. If a pending message was sent concurrently with the deletion notification (i.e., there is an application-level race condition), the message is treated as if it were sent after the deletion and thus dropped. The second optimization migrates queue ownership from the producer to the consumer, which must read queue contents synchronously.

Because non-concurrent processes can share a message queue, our implementation also uses a common file naming scheme to serialize message queues to disk. If a picoprocess which owns a message queue exits, any pending messages are serialized to a file 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 7.3) 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 process 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 [56, 113]. Running on a microkernel can improve the performance of Graphene with a more optimized IPC substrate [71, 101, 114]. Currently, Graphene takes 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 messaging 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 process it maps, and establish a point-to-point stream to the process. 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. 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. 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.

Prioritize Pairwise Coordination Within a Sandbox. 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 RPCs and designs several strategies foroptimizing the coordination framework based on lessons learned in Graphene.

Chapter 5

The Linux Host

This chapter uses Linux as an example to illustrate the porting effort of the PAL ABI. 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 an isolated view of OS states. 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 security isolation.

5.1 Exporting the Host ABI

The Linux PAL uses an unmodified Linux kernel as the host OS. By default, a Graphene picoprocess should run on 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 host-level components require 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`)	398	0.03%
Linux Library OS (`libLinux`)	33,833	
Linux PAL	12,640	
SGX PAL (described in Chapter 6)	28,166	
Reference monitor bootstrapper	1,446	
Linux kernel reference monitor module (`/dev/graphene`)	1,473	
Linux kernel IPC module (`/dev/gipc`)	943	

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).

Developing a Linux PAL requires significantly less effort than developing a Linux library OS (see Table 5.1). 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 ∼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 exist on other host OSes 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 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. The 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. On the Linux PAL, a new, clean process 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 process and the consecutive ones are mostly identical, except a consecutive process 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 process by marking the underlying file descriptors as `CLOEXEC` (close upon `execve()`). No page needs to be shared among processes.

A key challenge to implementing process creation in the Linux PAL is to reduce the overhead of initializing a new process. 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 process with a fully-initialized PAL, and create processes ahead of time. The preforking design shows higher latency than the current design due to the IPC overhead for coordinating preforked processes.

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 Security Models

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 [147], 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 [88]. 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 [135]). 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** [92]. 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 40 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 interfere with other processes. The reference monitor checks the remaining permitted system calls with external effects, such as `open()` (see Section 5.2.3).

Graphene restricts the host system calls using a seccomp filter [147], 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 process installs a seccomp filter, 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`. Section 4.1.1 lists the consecutive steps of system call redirection.

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 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 a 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 40 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 effort 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. [162] proposes sandboxing an uncertain, potentially-malicious application in Graphene with an unpredictable `libLinux` implementation.

```
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).

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 I/O operations such as reading or writing data can be trusted 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 Linux PAL uses a Linux kernel module as the reference monitor. Upon installation of the kernel module, a special device `/dev/graphene` is available for a Graphene picoprocess to issue system call requests through `ioctl()` calls. The seccomp filter ensures the Graphene picoprocess only calls system calls like `open()` and `bind()` through the reference monitor interface. The security checks of the reference monitor are stackable with other host security mechanisms. For example, if a manifest lists a root-privileged file to be accessed by an unprivileged process, existing security checks in a Linux kernel still blocks the file access even though the reference monitor permits the access.

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. 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 processes. Using the `ptrace()` system call, this reference monitor can retrieve user memory from the monitored process, 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 on a Linux host.

Another approach is to translate the isolation rules to an AppArmor profile or iptables rules. Since file and network rules of Graphene are similar to file rules of AppArmor and firewall rules of iptables, Graphene can convert a manifest file, either statically or dynamically, to security policies recognized by AppArmor and iptables. This approach will not require a Graphene-specific reference monitor installed as a Linux kernel module. 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. 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.

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 Linux PAL translates most of the PAL ABI to similar Linux system calls. 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 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 enforces a fixed 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 [92]. 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 [122] shows a compelling example where an unmodified application fails to run inside a beneficial, new host environment. SGX provides an opportunity of running trusted applications with a strong threat model where OSes, hypervisors, and peripheral devices can be malicious. SGX presents challenges to running an unmodified application, including shielding the application from malicious host system calls. Graphene significantly reduces the complexity of resolving both compatibility and security issues for running unmodified applications on SGX.

This chapter summarizes the development of an SGX framework using Graphene to protect unmodified Linux applications from the untrusted host OS. This chapter starts with the overview of SGX-specific challenges for porting an application, followed by a comparison of approaches to shielding an application from an untrusted host [39, 46, 152]. This chapter then describes the design of Graphene-SGX, an SGX port of Graphene; Graphene-SGX fits dynamically-linked, unmodified applications into the paradigm of SGX-ready applications, and customizes an interface to an untrusted OS to simplify security checks against malicious host system calls.

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 [122] is a feature added in the Intel sixth-generation CPUs, as a hardware support for trusted execution environments (TEEs) [97, 111, 145, 163]. 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 are particularly appealing in cloud computing, as users might not fully trust the cloud provider. Even if the whole cloud infrastructure is under the 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, 100, 116, 177]. 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. The memory encryption prevents an 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 attacks [82], 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 can 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, as 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 an 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 never steal

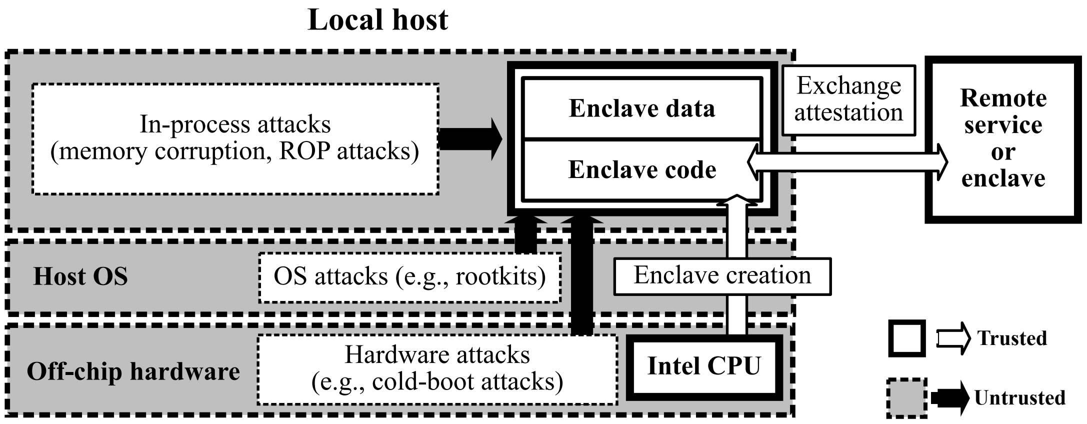

Local host
In-process attacks (memory corruption, ROP attacks)
Enclave data
Enclave code
Exchange attestation
Remote service or enclave
Host OS
OS attacks (e.g., rootkits)
Enclave creation
Off-chip hardware
Hardware attacks (e.g., cold-boot attacks)
Intel CPU
Trusted
Untrusted

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.

or corrupt enclave secrets both encrypted and authenticated inside the memory. An 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 the semantics in an enclave, or partitions application code unrelated to security. The extra effort for adapting existing application code into SGX can delay deployment of the technology; some 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 in porting to SGX. Although Haven [46] 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" [39]. Shinde et al. [152] 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. Besides the fact that Haven is evaluated upon a simulated hardware, Haven has a large TCB from adopting native Windows 7 code [138]. 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 [152]. 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 [119], 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 several 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 [103, 130, 148, 151] are complementary to Graphene-SGX.

6.1.3 Shielding Complexity

A key question for developing an SGX framework is how much OS functionality to 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, library OSes like Graphene-SGX and Haven pull 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 decision to pull OS functionality into enclaves impacts 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 [41, 48, 184]. 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 just like a library OS.

Specifically, **Iago attacks** [55] threaten a framework that shields against an existing API layer or system call table in an untrusted domain. Checkoway and Shacham demonstrate several attacks against shielding systems like InkTag [86] and Overshadow [58], 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 the stack from `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 that 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 a self-protecting application or a shielding system without fully anticipating attacks from an OS.

Existing shielding layers, including SCONE [39] and PANOPLY [152], 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 [55], 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 [46] 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 PAL ABI, to reduce the 40 PAL calls to merely 28 enclave calls (see Section 6.3.2). 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-sensitive; 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 runtime; 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 [115]. This thesis focuses on running an unmodified application as a whole. Partitioning 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 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 will try to exploit 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 [91]. `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 93.5MB 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 denial-of-service (DoS) in applications. Several works 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 [57, 78, 81, 127, 173, 176, 178]. 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 [179, 186]; 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 [85].

6.2.2 User Policy Configuration

Despite the fact 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 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 extends 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 as 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 enclaves and do not export shared states to the host OS, Graphene-SGX only has to add two features

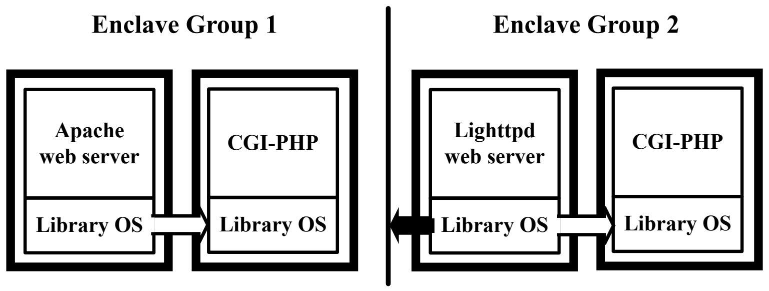


Figure 6.2: Two enclave groups, one running Apache and the other running Lighttpd, each creates a child enclave running CGI-PHP. Graphene-SGX distinguishes the child enclaves in different enclave groups.

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 [166]. 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 [46], the enclave measurement only verifies the integrity of Haven itself, and the same measurement applies to any application running on the same Haven loader.

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

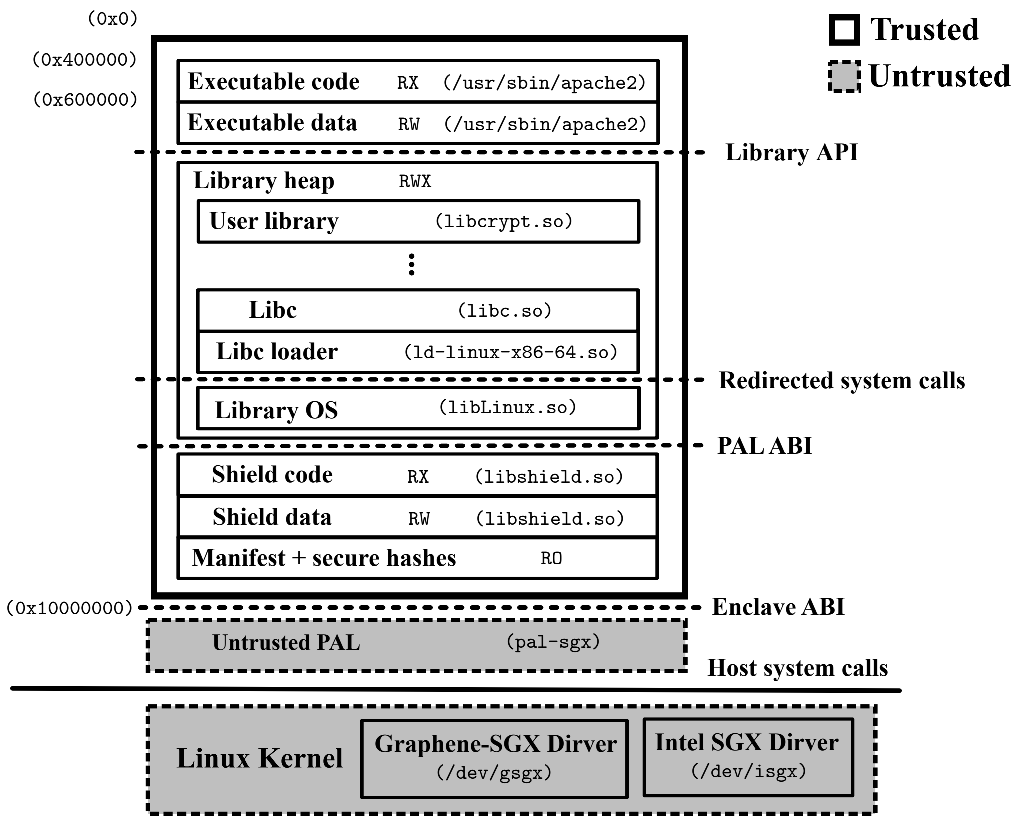

(0x0)
(0x400000)
(0x600000)
Executable code RX (/usr/sbin/apache2)
Executable data RW (/usr/sbin/apache2)
Library API
Library heap RWX
User library (libcrypt.so)
Libc (libc.so)
Libc loader (ld-linux-x86-64.so)
Redirected system calls
Library OS (libLinux.so)
PAL ABI
Shield code RX (libshield.so)
Shield data RW (libshield.so)
Manifest + secure hashes RO
(0x10000000)
Enclave ABI
Untrusted PAL (pal-sgx)
Host system calls
Linux Kernel
Graphene-SGX Dirver (/dev/gsgx)
Intel SGX Dirver (/dev/isgx)
Trusted
Untrusted

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.

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 enclave 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 page 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. 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 [123] 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 contain 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 a 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 the 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 40 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 Shacham [55] demonstrate the subtlety of detecting semantic attacks via the Linux system calls, called Iago attacks. Projects such as Sego [105] 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 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

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 (with checkable semantics); 6 *benign* calls (no harmful effects); 2 *DoS* calls (may cause denial-of-service); and 2 *unsafe* calls (potentially attacked by the host).

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()`, 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

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.

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 does 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 [86] 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 [130] 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 [178] 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 [61]. T-SGX [151] 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 major library OS changes. This subsection explains how Graphene-SGX protects multi-processing abstractions from an untrusted OS.

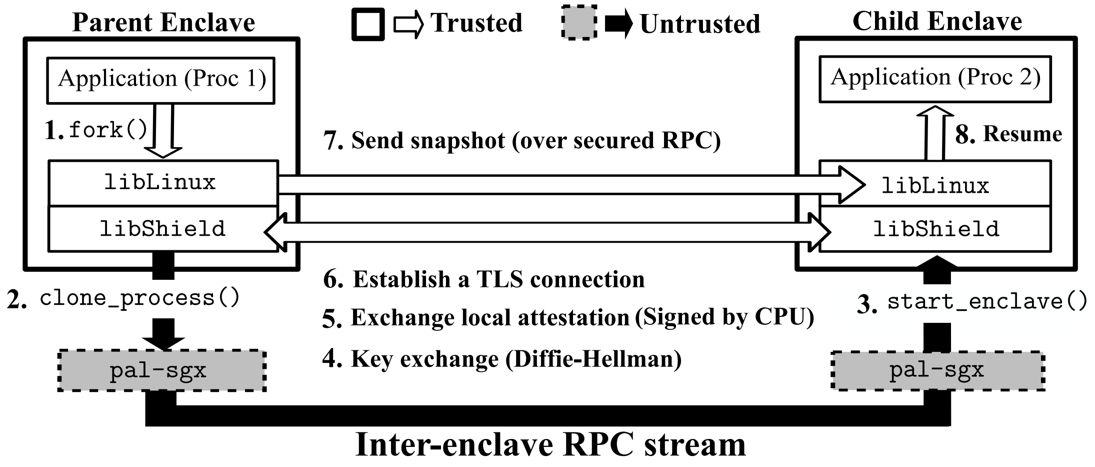


Figure 6.4: Process creation in Graphene-SGX. Numbers show the order of operations. When a process forks, Graphene-SGX creates a new, clean enclave on the untrusted host. Then the two enclaves exchange an encryption key, validates the CPU-generated attestation of each other, and migrates the parent process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

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.

Process Creation with New Executables. `execve()`, a system call which starts a process with a specific executable, offers a different way from `fork()` to create processes. 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 on 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 [39, 152], 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 binaries 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 [55]. 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 a 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 of Graphene. The evaluation includes 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 call latency and throughput; (3) end-to-end performance of sample applications; (4) resource costs, including both memory footprint and CPU occupancy. The evaluation compares the performance on Linux and SGX, two host OS examples where Graphene has been fully ported. Thus, this chapter exemplifies the potential cost of leveraging a library OS for compatibility and security isolation.

Experimental Setup. The evaluation is based on Graphene v0.4[1]. All experiment results are collected on a Dell Optiplex 790 Small-Form Desktop, with a 4-core 3.2GHz Intel Core i5-6500 CPU without hyper-threading two 4GB DIMM 1600MHz DDR3 RAMs (8GB in total), and a Seagate 512GB, 7200 RPM SATA disk formatted as EXT4. SGX is enabled on the CPU with 93.5MB EPC (enclave page cache). To prevent fluctuation in experiment results, Turbo Boost, SpeedStep, and CPU idle states are disabled for all experiments. All networked servers are evaluated over 1Gbps Ethernet cards connected to a dedicated local network, except that some micro-benchmarks are evaluated over a local loopback device (i.e., localhost).

The host OS 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. Graphene-SGX uses the Intel SGX Linux SDK [91] and driver [90] v1.9.

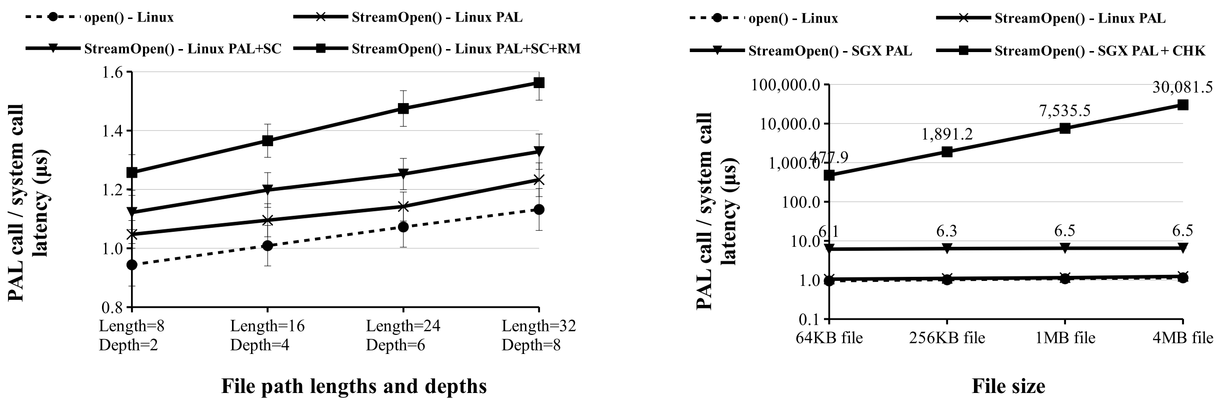
[1]Graphene is released at `https://github.com/oscarlab/graphene`

7.1 The PAL ABI Performance

The section evaluates the performance of the PAL ABI, by benchmarking either the latency or throughput of PAL calls. As the implementation of PAL calls largely depends on the underlying host system interfaces, the PAL ABI is likely to share the same performance patterns as the host OSes, with various amount of overheads. A logical source of these overheads is the translation between PAL calls and a host system interface, given the gap between the semantics of the two interfaces. Other overheads include the costs of enclave exits or VM exits and indirect performance impacts caused by switching contexts between the host OSes and the library OS instances. For SGX, the latency of entering and exiting enclave can be up to $\sim$7,000 cycles, and enclave exits also cause TLB flushes and last-level cache pollution, which both downgrade enclave performance [130]. Enclave executions also suffer memory overheads for swapping memory into the EPC (enclave page cache) or decrypting memory at last-level cache misses.

Security checks also impact the PAL ABI performance in several cases. For most hosts, Graphene isolates mutually-untrusting applications from attacking each other or the host OS, by restricting the attack surface which consists of shared host abstractions and system interfaces. For SGX, Graphene further assumes a stronger threat model, which distrusts any OSes, hypervisors and off-chip hardware. Security checks for SGX require detecting potentially-malicious inputs from an untrusted OS, including using cryptographic techniques to encrypt or authenticate data. This chapter shows that security checks on a stronger threat model like SGX tends to impose high overheads on many PAL calls.

The experiments in this section are based on a ported version of LMbench 2.5 [125] using the PAL ABI. For each PAL call, the evaluation compares the performance, either as latency or throughput, among the PALs for Linux and SGX hosts and a native Linux kernel, to measure the overheads of PALs. For each host target, the evaluation tests both with and without security checks or enforcements, such as seccomp filter and reference monitor on Linux host, or checking untrusted inputs in an enclave. This section also includes a detailed analysis of performance overheads impacted by design decisions in PAL implementation.



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

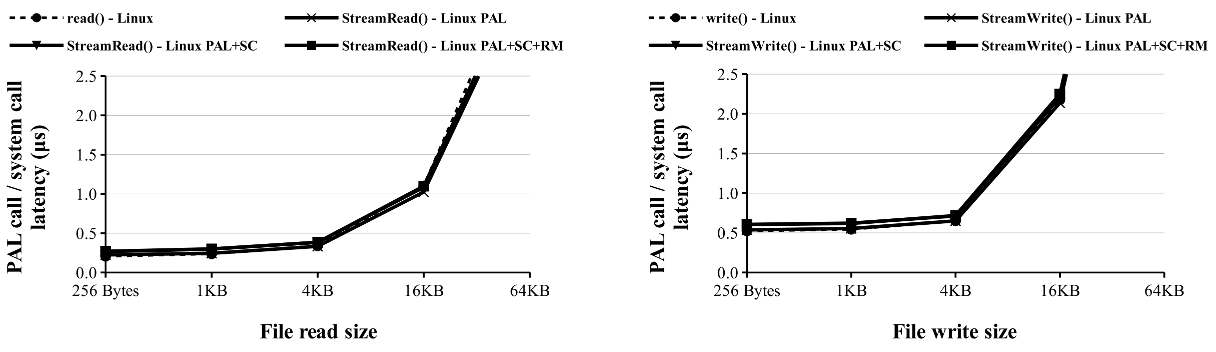
Figure 7.1: Latency of `StreamOpen()` on the Linux PAL and SGX PAL, versus `open()` on Linux. Lower is better. Figure (a) compares `StreamOpen()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor (**+RM**), against `open()` on Linux. Figure (b) compares `StreamOpen()`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

7.1.1 Stream I/O

This section evaluates the performance of file operations, network sockets, and RPC streams. To be accurate, the evaluation separates the benchmarking strategies for different types of I/O streams. For file operations, the evaluation measures the performance of opening, reading, and writing a regular host file through PAL calls or native host system calls. For network sockets or RPC streams, the evaluation measures both latency and bandwidth of transferring data over the I/O streams.

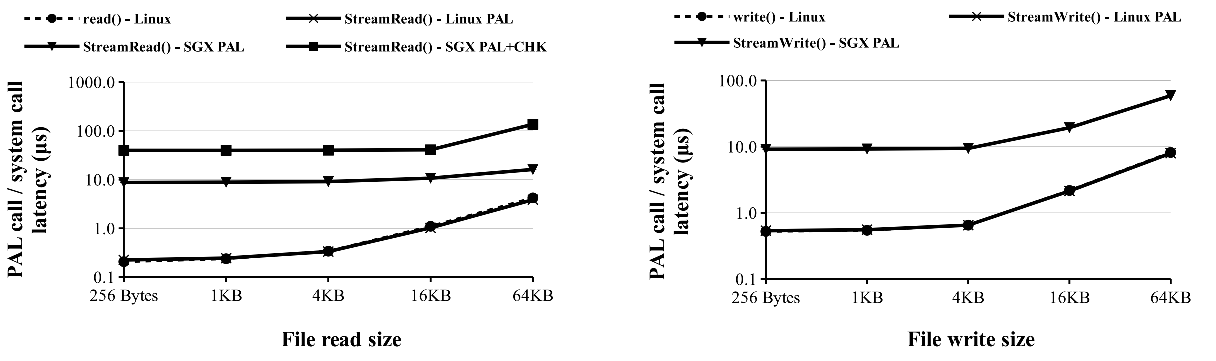
Opening a File. File system operations such as opening a file are often subject to slowdowns caused by security checks and system interface translation. As shown in Figure 7.1(a), host system call translation, system call restriction (using the seccomp filter), and reference monitoring each contribute a nontrivial portion to the file opening overheads, adding up to 21–31%. Figure 7.1(a) also shows a correlation between file opening latency and path lengths, since both searching file records and checking against reference policies requires path comparison.

File integrity protection induces orders-of-magnitude higher overheads than the translation costs. For reference, exiting an enclave and copying memory for opening a file in the host OS take a fixed overhead at $\sim$5 μs. The cost of integrity protection includes calculating a secure hash of the file (by SHA-256 in the current implementation) and generating a Merkle tree of hash values for verifying consequential reads and writes. As shown in Figure 7.1 (b), the cost is correlated



(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()`.

with file size, as ~470μs for a 64KB file or ~30 ms for a 4MB file, even with the Intel hardware acceleration. Although different cryptographic algorithms or implementations may improve the performance, it appears to be difficult to obtain significant overhead reduction. A possible optimization is to offload the generation of Merkle trees to the signing phase so that the PAL can avoid verifying the whole file at file opening.

File Reads and Writes. File reads and writes show very different performance patterns between a Linux host and an SGX enclave. With a trusted host OS, the PAL calls for reading and writing a file have similar latency as the underlying `read()` and `write()` system calls (see Figure 7.2).

The impact of seccomp filter and reference monitoring on reads and writes are nearly negligible, especially since the reference monitor does not check each read and write operations.

If the host OS is untrusted and the guest runs inside an enclave, reading a file suffers significant overheads due to: (1) copying file contents between enclaves and the untrusted hosts; (2) cryptographic operations for checking the integrity of file contents. As shown in Figure 7.3, the cost of enclave exits and memory copy is fixed at 8–12 μs for reads and 8–50 μs for writes, but the cryptographic checks can be much more expensive depending on the size of reads and writes. The current SGX PAL assumes file contents to be checked in 16KB chunks, so any file reads smaller than 16KB are as expensive as reading an aligned 16KB chunk. Reducing the checking size will improve the latency of small reads, but increases the memory overhead for storing the whole Merkle tree. The current SGX PAL does not protect the integrity or confidentiality of file writes, so the evaluation only tests file writes without cryptographic checks.

Network and RPC Streams. On the Linux PAL, overheads on network and RPC streams are either minor or negligible, because most operations require no security checks. The observation is backed by Figure 7.4 (a) and Figure 7.5 (a), which shows the highest overheads on a TCP socket, a UDP socket, and a RPC stream, as measuring the latency of sending a single-byte message, are $\sim$18%, $\sim$30%, and nearly 0%, respectively, compared to the underlying host abstractions. The overheads are likely to be less significant with larger workloads since the translation between the PAL ABI and host system calls tends to be a fixed cost. As shown in Figure 7.4 (b) and Figure 7.5 (b), the bandwidth for sending large messages over a TCP socket and an RPC stream suffer less than 5% overheads on the Linux PAL.

For SGX, reading or writing on a network or RPC stream suffers similar overheads as file reads and writes, due to the same requirement of enclave exits and memory copies. For TCP, UDP, and RPC streams, the benchmark results show overheads up to 167%, for simply exiting enclaves and invoking host Linux system calls. To improve the performance, one can adopt an exit-less, asynchronous enclave interface similar to the designs in SCONE [39] and Eleos [130]. Graphene leaves the exploration of these techniques as future work.

Note that the current SGX PAL design does not protect the integrity or confidentiality of messages sent or received over TCP or UDP sockets. Previous work [39] makes the similar con-

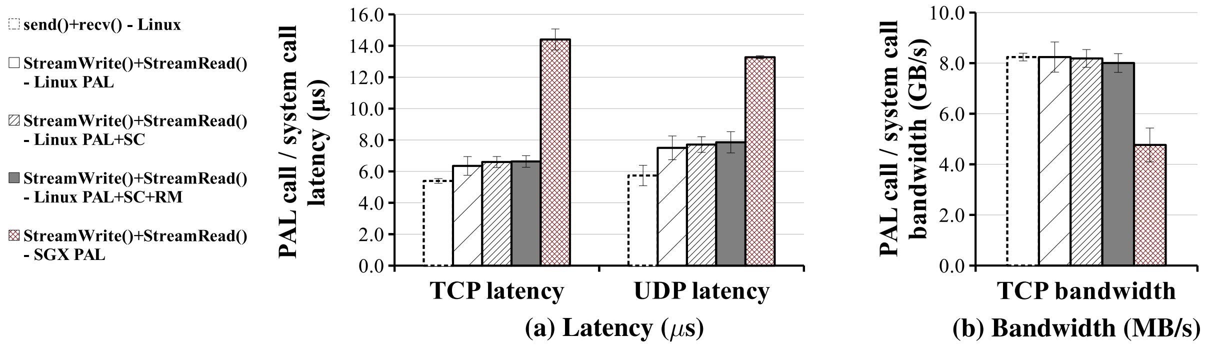


Figure 7.4: (a) Latency of sending a short message over TCP and UDP sockets (lower is better), and (b) bandwidth of sending large data over TCP (higher is better). The comparison is between (1) `recv()` and `send()` on Linux; (2) `StreamRead()` and `StreamWrite()` on a Linux PAL, with and without a seccomp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

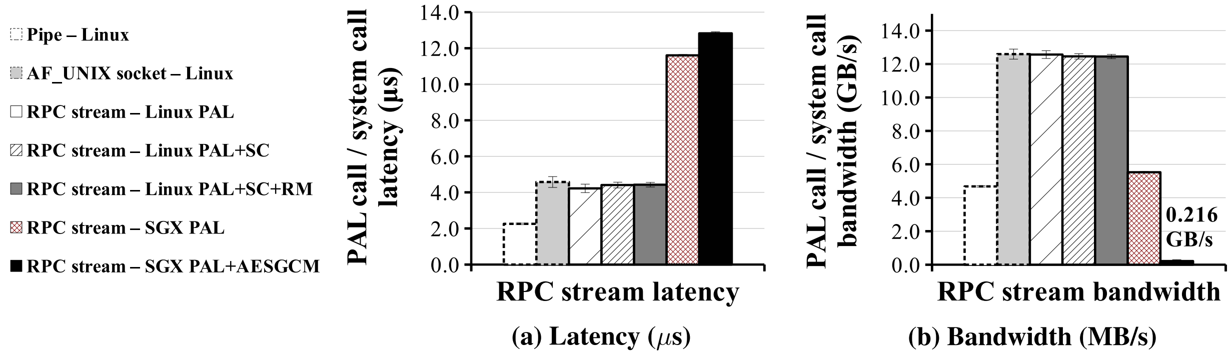
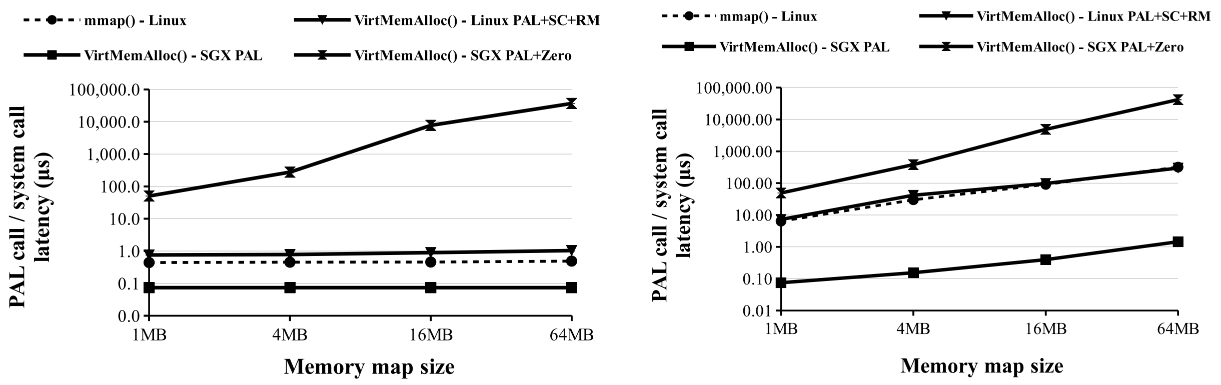


Figure 7.5: (a) Latency of sending a short message over RPC (lower is better), and (b) bandwidth of sending large data (higher is better). The comparison is between (1) `read()` and `write()` over a pipe or an AF_UNIX socket on Linux; (2) `StreamRead()` and `StreamWrite()` on the Linux PAL, with and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

clusion, based on the observation that SSL/TLS encryption and authentication inside applications are quite prevalent and much more efficient than enforcement at enclave exits.

For RPC streams, the SGX PAL must protect the integrity and confidentiality of messages since applications are not aware of security threats from the untrusted OS. The SGX PAL establishes a TLS connection on every RPC streams, bootstrapped by the local attestation of SGX hardware. As a result, the performance of RPC streams on the SGX PAL is bound by the performance of cryptographic algorithm used by the TLS connections, which is AES-GCM in the current implementation. Figure 7.5 shows that, even with the Intel hardware acceleration (i.e., AES-NI), the overheads on PRC streams are still quite significant; the cryptographic operations have only a



(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**).

small impact on single-byte messages (∼10%), but the bandwidth downgrades from ∼5.5 GB/s to ∼216 MB/s when sending 64KB messages.

7.1.2 Page Management

Single-process abstractions, such as page management, tend to have lower overheads by avoiding complicated security checks. In Figure 7.6, the overheads of memory allocation and deallocation using `VirtMemAlloc()` and `VirtMemFree()` over the native `mmap()` and `munmap()` are almost negligible (less than 5%), even if the guest has accessed the allocated pages. Graphene expects the same performance on any host OS with dynamic page management.

The current SGX PAL follows a static page management model, due to the restriction of SGX hardware. For each enclave, the SGX PAL preallocates a heap space at loading time, and cannot update the enclave page layout after the enclave starts. Although the untrusted OS still offers page-in and page-out when an enclave requires more pages than the EPC size (93.5MB in the current setup), the SGX PAL implements internal page management to bypass the untrusted OS. The latency of allocating virtual pages, however, is dominated by the cost of zeroing pages before returning to the guest. Due to the memory access overheads in enclaves, zeroing pages that are larger than the last-level cache size (4MB on an Intel i5-6500 CPU) can be as expensive as 600-

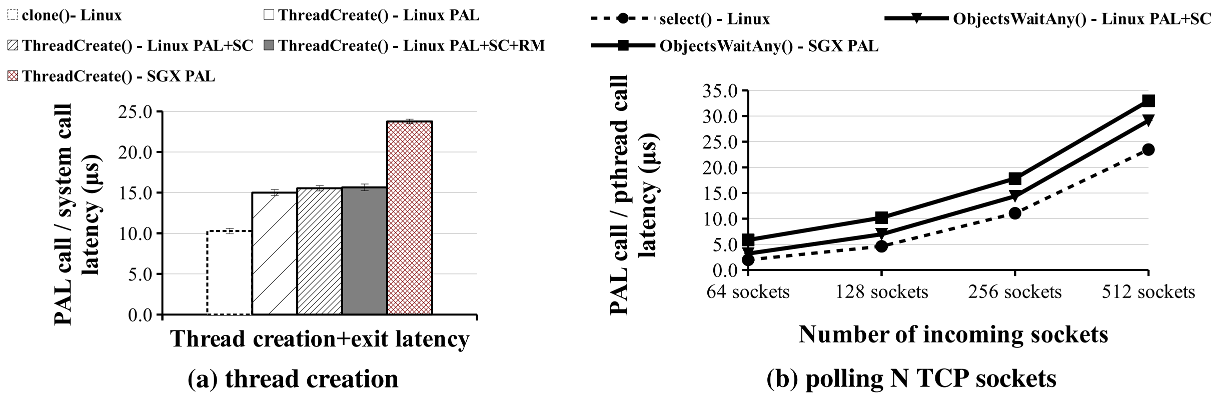


Figure 7.7: (a) Thread creation latency and (b) latency of polling a number of TCP sockets. Lower is better. The comparison is between (1) `clone()` and `select()` on Linux; (2) `ThreadCreate()` and `ObjectsWaitAny()` on the Linux PAL, with and without a seccomp filter (**+SC**) and reference monitor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

75,000× to original `mmap()` and `munmap()` latency. The SGX version 2 hardware can potentially improve the overheads by dynamically removing and adding pages to an enclave. Without the memory zeroing cost, page allocation and deallocation on the SGX PAL is only ∼16% slower than `mmap()` and `munmap()`, since the SGX PAL for simply updating the mappings inside an enclave.

7.1.3 Scheduling

This section evaluates the performance of thread creation, polling multiple TCP sockets, and synchronization primitives, such as notification events and mutexes.

Thread Creation. Also classified as a single-process operation, thread creation on Linux PAL expects little impact from security checks or enforcements. The implementation effort for process creation mostly focuses on keeping the semantics of the PAL call—`ThreadCreate()`—as simple as possible. For instance, a new thread created by `ThreadCreate()` always starts on a pre-allocated stack which the guest needs not to assign. As shown in Figure 7.7 (a), the overhead on the Linux PAL to create and terminate a thread is ∼46%, most of which results from stack allocation. The overheads of seccomp filter and reference monitor are negligible (less than 5%). The latency on the SGX PAL is slightly more expensive, mostly due to populating an unused TCS (thread control structure) inside the enclave. Thread creation on SGX does not accept any inputs from the untrusted OS, and thus requires no security checks against potential Iago attacks.

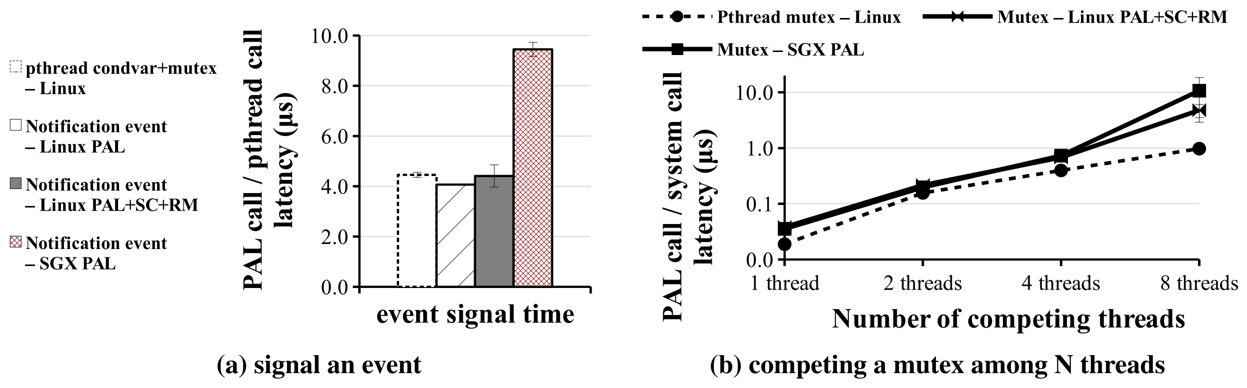
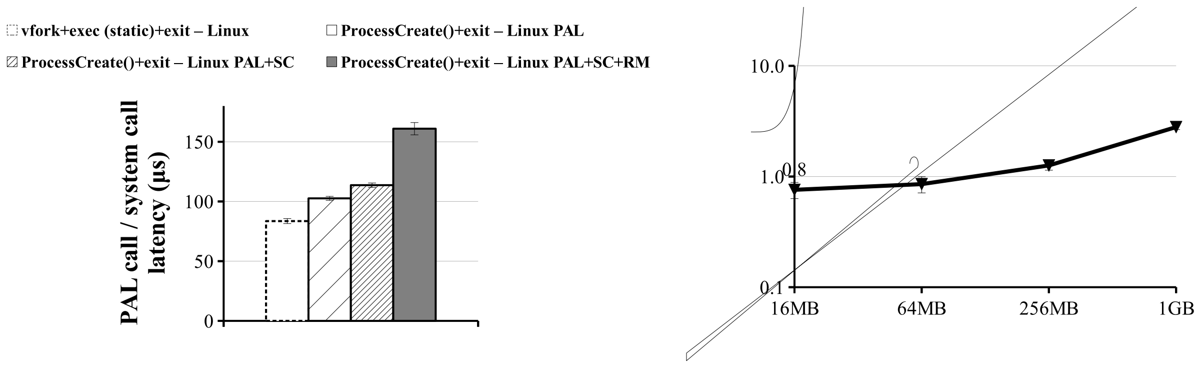


Figure 7.8: Latency of (a) signaling an event and (b) competing a mutex among N threads (N: 1 to 8). Lower is better. The comparison is between (1) pthread condition variables and mutexes on Linux; (2) Notification events and mutexes on the Linux PAL, with and without a seccomp filter (**+SC**) and reference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

Polling TCP Sockets. Polling TCP sockets or any stream handles is one of the operations that suffers more translation costs. Mapping file, TCP, UDP, or RPC handles to host file descriptors requires reading the contents of handles, and thus causes overheads proportional to numbers of handles. Figure 7.7 (b) compares the latency of `ObjectsWaitAny()` on 64 to 512 TCP sockets with `select()`, and shows that the overhead of system interface translation is 24–60% on the Linux PAL. For the SGX PAL, polling the same amount of TCP sockets requires more time for enclave exits and copying a bitmap of file descriptors out of the enclave. The overhead on the SGX PAL is 13–80% compared to `select()`.

Events and Mutexes. The Linux PAL and SGX PAL implement notification events, synchronization events, and mutexes with atomic or compare-and-exchange instructions, but use futexes in the host OS to free the CPUs when blocking on other threads. The implementation is similar to pthread primitives such as `pthread_cond` and `pthread_mutex`; therefore, similar performance is expected on these primitives, with less than 10% overhead on the Linux PAL. (see Figure 7.8). For the SGX PAL, these primitives are much more expensive ($\sim$130–250%) if the PAL calls require exiting enclaves for futexes.



(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()` with 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.

7.1.4 Multi-Process Abstractions

This section evaluates two multi-process abstractions in the PAL ABI: process creation and bulk IPC channels for zero-copy data transfer between processes.

Process Creation. Process creation is one of the most expensive PAL calls on both the Linux PAL and SGX PAL. Instead of adopting the copy-on-write, fork-like semantics of process creation from Linux and similar OSes, the PAL ABI creates clean, new processes without sharing any process memory. The semantics are similar to `spawn()` in POSIX, or the combination of `vfork()` and `execve()` in Linux.

In general, process creation on the PAL ABI expects the following three types of performance overheads: (1) initialization of the PAL binary in the new process; (2) creating a trustworthy RPC stream among parent and child processes; (3) adding the new process to the trusted group of its parent. Each of these factors impacts the latency of process creation in a non-trivial way. As measured in Figure 7.9 (a), the overhead on process creation is up to 93% on the Linux PAL, with both seccomp filter and reference monitoring. Among the overhead, 22% is for PAL and RPC stream initialization. 36% is for enabling the seccomp filter in the child process, including executing a separate binary which then loads the PAL binary at the same address, to support program-counter-based system call filtering. Finally, adding the new process to the same sandbox in the reference monitor causes 55% overhead on the latency.

Process Creation on SGX. The SGX PAL isolates each process inside a separate enclave and uses secured RPC streams for coordination between parent and child processes. As a part of the process creation, the SGX PAL initializes an empty enclave signed for the target executable and establishes a TLS connection over an RPC stream. The process creation procedure includes several defenses against potential attacks from the host OS, including impersonating one of the enclaves and launching man-in-the-middle attacks on RPCs [152]. The SGX PAL defends these attacks using the local attestation feature of SGX hardware and several cryptographic techniques.

Although establishing the TLS connections slows down process creation, the largest portion of process creation overheads on the SGX PAL turns out to be enclave creation. Figure 7.9 (b) evaluates the cost of enclave creation by gradually increasing the enclave size of each new process. The results show that the process creation time on the SGX PAL is largely correlated with enclave sizes; creating a process in a 1GB enclave takes $\sim$2.8s, or 26,000$\times$ to the process creation time on the Linux PAL. Although process creation is normally considered an expensive operation in most OSes, the extremely high overheads on the SGX PAL can be a huge impact, especially for applications which frequently create new processes, such as shell applications and makefiles.

Several optimizations are possible for improving process creation on the SGX PAL. One optimization is to create enclaves in advance or to recycle enclaves after process termination. Moreover, the SGX2 hardware [123] introduces new instructions for inserting empty pages into an enclave after initialization, and therefore allows new processes to start in small enclaves.

Bulk IPC vs RPC Streams. The PAL ABI introduces an optional, bulk IPC feature for improving RPC messaging. The assumption around the abstraction is that it has higher bandwidth than an RPC stream by sharing pages as copy-on-write across processes. However, the assumption is not always valid, since RPC streams can be as efficient as bulk IPC on some host OSes. The concern is backed by the benchmark results in Figure 7.10, which show that page-aligned RPC messaging on Linux kernels later than 4.2 is comparably efficient as using the bulk IPC abstractions. The reason is an optimization introduced on Linux 4.2 to implement zero-copy transfers over UNIX sockets [157]. Graphene still considers the bulk IPC abstraction beneficial for older Linux kernels or other host OSes.

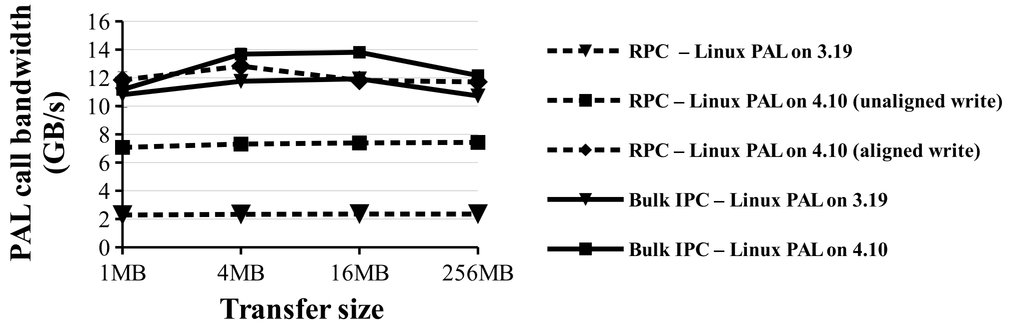


Figure 7.10: Bandwidth of sending large messages over (a) RPC streams and (b) Bulk IPC channels. The messages are sent in different sizes (1MB to 256MB), and either aligned or unaligned with the page boundary. Higher is better. Both abstractions are benchmarked on Linux kernel 3.19 and 4.10 as the hosts.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

7.1.5 Exception Handling

Exception handling on the PAL ABI includes installing upcall functions as handlers to exceptions, and delivery of exceptions to the installed handler. Handler installation is relatively fast on almost every PAL implementation. For the Linux PAL and SGX PAL, handler installation is at least 6× faster than calling `sigaction()` on native Linux (Figure 7.11), because the PAL call only has to update a function pointer in the process. However, exception delivery tends to be a much more expensive operation, due to implementing various corner cases upon restrictions of host OSes or hardware.

Both the Linux PAL and SGX PAL implement an elaborate exception delivery scheme to handle various corner cases where exceptions can happen. First, the PAL ABI defines several types of exceptions, which are all delivered through a unified interface. The deliverable exceptions include hardware exceptions such as memory faults, arithmetic errors (divide-by-zero), and illegal instructions, and OS-triggered exceptions such as system calls forbidden by the seccomp filter. On the Linux PAL, most exceptions are delivered as signals. However, on the SGX PAL, hardware exceptions are delivered directly into enclaves, with the interrupted register contexts dumped into a designated area called SSA (state save area). Moreover, exceptions can happen either in the guests (i.e., applications and library OSes) or inside the PAL code or a PAL exception handler. To gracefully handle these corner cases, both the Linux PAL and SGX PAL examines the saved register values to determine the reasons and code addresses that raise the exceptions.

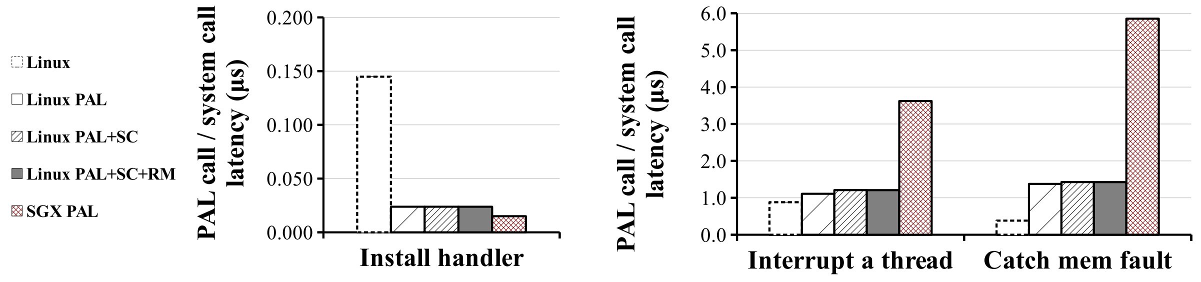


Figure 7.11: Latency of (a) installing an exception handler; (b) interrupting a running thread with signals (on Linux) or `ThreadInterrupt()` on the PALs; (c) catching a memory protection fault. Lower is better. The comparison is between (1) signals on Linux; (2) the Linux PAL, with and without a seccomp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

Figure 7.11 shows the overheads on exception delivery on the two PAL implementations, based on two corner cases: (1) interrupting the current thread (similar to sending `SIGUSR1`); (2) delivering a memory protection fault from the application. For thread interruption, because the exception happen in the middle of a PAL call, the overhead is only 37% on the Linux PAL. However, the latency of the Linux PAL to deliver a memory fault from the application is up to 273% slower than the similar operation on native Linux; the primary reason of the significant slowdown is to deliver the exception information from the host OS, including the faulting address and interrupted register contexts.

For the SGX PAL, exception delivery is dominated by enclave exits. Although the SGX PAL does not trust the untrusted OS to deliver exceptions, SGX forces enclave execution to stop and exit to the untrusted OS at exception, causing significant overheads on exception handling. Moreover, to handle a hardware exception such as memory fault or illegal instruction, the SGX PAL has to effectively exit an enclave twice, one for handing the exception and the other for freeing the limited SSA (state save area). On the SGX PAL, the overhead on delivering a memory fault is nearly twice as the overhead on thread interruption.

7.2 Library OS Performance

This section evaluates the performance of Linux system calls serviced by the Graphene library OS, or `libLinux`. The evaluation is based on an extended version of LMbench 2.5 [125]. Each benchmark result reports a mean and 95% confidence interval, based on the assumption that the results of each test are normally distributed. To measure the marginal cost of host security checks, the experiments test both with and without the seccomp filter and reference monitor on the Linux host. The experiments on the SGX host are based on a default security policy, which enforces security checks on most inputs from the untrusted host OS, but does not automatically encrypt or authenticate data files or network traffics.

The evaluation results categorize system calls as three types. The first type of system calls can be completely serviced inside `libLinux`; in this case, the latency of system calls in a picoprocess is likely to be close to native or even be more optimized. For these system calls, `libLinux` emulates the semantics in the guest space, gaining performance benefits by avoiding host system calls or expensive enclave exits. For instance, `getppid()`, which only returns an OS state (parent process ID) from `libLinux`, is 67% faster in Graphene than in a native Linux process, regardless of which host target the library OS is running on.

The second type of system calls requires always interacting with the host OS via the PAL ABI, or enclave calls on SGX. For example, to implement `open()`, `libLinux` needs to call `DkStreamOpen()` to open a file handle in the host OS. The overheads on emulating these system calls tend to be more significant than others, as including translation costs and overheads of host security mechanisms. Especially if emulating a system call requires multiple PAL calls or enclave calls, the overheads multiply as the time spent on accessing host system calls increases.

The third type of system calls are also implemented with the involvement of PAL calls, but can benefit from optimizations inside library OS, using techniques such as PAL call batching, I/O buffering, and directory caching. For these system calls, the overheads roughly depend on the number of PAL calls that `libLinux` can prevent for each instance of system calls since `libLinux` cannot reduce the slowdown on each PAL call. For instance, `libLinux` buffers file contents for system calls like `read()` and `write()` by mapping parts of the file inside picoprocesses. Since `libLinux` can batch the effect of multiple small reads or writes at nearby locations in a file, the operations can be up to $\sim$45% faster than in a native Linux 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
`getppid`	0.045	.000	0.015	.000	-67	0.015	.000	-67	0.015	.000	-67
`getppid` (direct)	0.045	.000	Not supported			1.155	.000	2,467	5.800	.001	12,789
`open /dev/zero`	0.997	.072	1.247	.000	25	1.256	.000	26	1.207	.004	21
`stat /dev/zero`	0.362	.000	0.466	.000	29	0.467	.000	29	0.451	.000	25
`fstat /dev/zero`	0.117	.000	0.111	.000	-5	0.111	.000	-5	0.107	.000	-9
`read /dev/zero`	0.116	.000	0.121	.000	4	0.121	.000	4	0.115	.000	-1
`write /dev/zero`	0.077	.000	0.116	.000	51	0.116	.000	51	0.112	.000	45
install sigaction	0.146	.000	0.113	.000	-23	0.113	.000	-23	0.110	.000	-25
send `SIGUSR1`	0.895	.000	0.189	.000	-79	0.187	.000	-79	0.178	.000	-80
catch `SIGSEGV`	0.379	.000	1.526	.000	303	1.575	.000	316	6.117	.000	1,514

Table 7.1: Single-process system call performance based on LMbench. The comparison is among (1) native Linux processes; (2) Graphene on Linux host, both without and with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. System call latency is in microseconds, and lower is better. Overheads are relative to Linux; negative overheads indicate improvement.

7.2.1 Single-Process System Calls

System calls directly serviced inside `libLinux` tend to have a close-to-native performance or even gain significant speed-up. Table 7.1 lists several system call samples that share the characteristic, including `getppid()`, `sigaction()`, and common file operations on a pseudo device such as `/dev/zero`. The benchmark results on these system calls show at most 20–80% speed-up compared to the native performance. File operations on `/dev/zero` have slightly higher overheads, especially for `open()` and `stat()` which suffer the cost of directory cache lookup. Even if a host implements the PAL ABI with significant overheads, such as in an SGX enclave, these system calls have a similar performance because no PAL call is involved.

The other two benchmark results in Table 7.1 show significant overheads on delivering hardware and OS-triggered exceptions. One example is the overheads on system calls directly made from a statically compiled binary using `SYSCALL` or `INT $80` instructions. Using host-specific system call restriction, such as a seccomp filter on a Linux host, Graphene can capture these system calls and redirect them back to `libLinux` as an exception. Base overhead on a direct system call is up to 24$\times$ on the Linux host, or 128$\times$ on SGX. Another result shows the overhead of sending a `SIGSEGV` signal to be around 3$\times$ and 15$\times$ for Linux and SGX hosts, respectively.

7.2.2 File Systems

File system performance in `libLinux` is subject to several optimizations including directory caching and buffering. To reduce the impact of PAL call latency, a key to the chroot file system implementation is to lower the average number of expensive PAL calls needed for emulating each system call. For instance, the directory cache in `libLinux` stores path existence and metadata in spare picoprocess memory, to skip the redundant cost of querying the host file system when accessing the same path in the future. Table 7.2 lists the latency of `open()` and `stat()` for repeatedly opening or querying the same path. Directory caching reduces the overheads on `stat()` to 35–41% regardless of the hosts. The latency of `open()` also benefits from directory caching, but the cost of opening the file in the host OSes overshadows the optimization, causing 187–237% overheads on the Linux host with the seccomp filter and reference monitor, or 15.2–16.5× in an enclave.

For `read()` and `write()`, the latency in Graphene depends on the buffering strategy in `libLinux`. The experiments are based on a strategy which buffers reads and writes smaller than 4KB (not including 4KB) using a 16KB buffer directly mapped from the file. In Table 7.2, buffered reads and writes (256 bytes and 1KB) on Linux host are 22–92% and -45–8% slower than native, respectively. The latency of unbuffered reads and writes (4KB and 16KB), is closer to native when running on a Linux host, but suffers significant overheads (10–29×) in an enclave due to copying file contents.

Table 7.2 also lists the throughputs of creating and deleting a large number of files, measured in operations per second. Among all the benchmark results, deletion throughputs tend to have much higher overheads than creation throughputs. Compared to running on the Linux host, both file creation and deletion from an enclave suffer significantly higher overheads (2.7–6×).

Test	System call latency (μs), +/- Confidence Interval, % Overhead										
	Linux 4.10		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	μs	+/-	μs	+/-	%O	μs	+/-	%O	μs	+/-	%O
`open` (depth=2,len= 8)	0.947	.072	2.337	.013	147	2.719	.007	187	16.600	.007	1,653
`open` (depth=4,len=16)	1.011	.074	2.627	.009	160	2.922	.009	189	17.168	.016	1,598
`open` (depth=8,len=32)	1.131	.074	3.360	.000	197	3.812	.007	237	18.415	.016	1,528
`stat` (depth=2,len= 8)	0.361	.000	0.502	.000	39	0.499	.000	38	0.487	.000	35
`stat` (depth=4,len=16)	0.420	.000	0.585	.000	39	0.584	.000	39	0.571	.001	36
`stat` (depth=8,len=32)	0.553	.000	0.780	.000	41	0.780	.000	41	0.767	.000	39
`fstat` (any length)	0.120	.000	0.193	.000	61	0.193	.000	61	0.187	.000	56
`read` (256B)	0.207	.072	0.252	.000	22	0.255	.000	23	0.342	.000	65
`read` (1KB)	0.227	.072	0.435	.000	92	0.434	.000	91	0.805	.001	255
`read` (4KB)	0.315	.072	0.545	.001	73	0.607	.000	93	9.545	.006	2,930
`read` (16KB)	1.022	.072	1.308	.000	28	1.356	.000	33	11.437	.022	1,019
`write` (256B)	0.515	.002	0.285	.000	-45	0.287	.000	-44	0.490	.000	-5
`write` (1KB)	0.535	.001	0.575	.000	7	0.580	.000	8	1.420	.002	165
`write` (4KB)	0.618	.000	0.856	.002	39	0.909	.002	47	9.784	.006	1,483
`write` (16KB)	2.034	.000	2.303	.013	13	2.356	.001	16	19.730	.021	870

(Note: `libLinux` only buffers reads and writes smaller than 4KB; default buffer size is 16KB.)

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

Table 7.2: File-related system call performance based on LMbench. The comparison is among (1) native Linux processes; (2) Graphene on Linux host, both without and with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. System call latency is in microseconds, and lower is better. System call throughput is in operations per second, and higher is better. Overheads are relative to Linux; negative overheads indicate improvement.

	System call latency (μs), +/- Confidence Interval, % Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	μs	+/-	μs	+/-	%O	μs	+/-	%O	μs	+/-	%O
pipe	2.412	.290	4.440	.464	84	4.491	.138	86	13.364	.000	454
UNIX socket	4.386	.293	5.587	.048	27	5.824	.044	33	14.431	.000	229
UDP socket	6.300	.708	9.451	.235	50	9.938	.219	58	17.538	.703	178
TCP socket	7.217	.505	9.422	.203	31	10.075	.209	40	17.925	.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
pipe	4,791	295	12,262	204	-61	12,151	130	-61	218	0	2,098
UNIX socket	12,643	419	12,179	85	4	12,173	306	4	218	0	5,700
TCP socket	7,465	38	7,011	52	6	6,932	64	8	4,242	2	76

Table 7.3: Network socket and pipe performance based on LMbench. The comparison is among (1) native Linux processes; (2) Graphene on Linux host, both without and with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. System call latency is in microseconds, and lower is better. System call bandwidth is in megabytes per second, and higher is better. Overheads are relative to Linux; negative overheads indicate improvement.

7.2.3 Network Sockets and Pipes

The PAL ABI performance largely determines the performance of network socket and pipe emulation in `libLinux`. In general, `libLinux` does not implement a network or pipe stack, neither does `libLinux` buffer data read or written to a network socket or a pipe. Table 7.3 shows similar benchmark results as unbuffered file reads and writes evaluated in the previous section; the overheads on a network socket or a pipe are up to 33–86% on the Linux host, or 148–454% in an SGX enclave, when measuring the latency of sending single-byte messages over the abstractions.

7.2.4 Virtual Memory

The latency of virtual memory allocation and deallocation in `libLinux` contains both the cost of updating the internal list of VMAs (virtual memory areas) in `libLinux` and the latency of corresponding PAL calls, (`VirtMemAlloc()` and `VirtMemFree()`. Section 7.1.2 reports the latency of `VirtMemAlloc()` and `VirtMemFree()` on the Linux PAL to be close to the native `mmap()` and `munmap()` latency. Table 7.4 shows a significant 116–126% overheads on the Linux host, primarily as the cost of VMA list updating. For Graphene-SGX, the latency of virtual memory allocation

		System call latency (μs), +/- Confidence Interval, %/× Overhead							
Test		**Linux 4.10**		**Graphene+SC+RM**			**Graphene-SGX**		
		μs	+/-	μs	+/-	O	μs	+/-	O
mmap+free	(1MB)	0.854	.011	1.848	.001	116.393 %	53	0	61 ×
mmap+free	(4MB)	0.872	.055	1.889	.069	116.628 %	260	20	319 ×
mmap+free	(16MB)	0.872	.056	1.973	.006	126.261 %	7,762	30	8,901 ×
mmap+write+free	(1MB)	7.930	.052	8.692	.379	9.609 %	53	0	6 ×
mmap+write+free	(4MB)	33.440	11.567	29.716	10.140	-11.136 %	264	10	7 ×
mmap+write+free	(16MB)	101.416	11.547	92.449	9.137	-8.842 %	7,738	33	75 ×
brk+free	(1KB)	0.445	.002	0.146	.000	-67 %	0.136	.000	-69 %
brk+free	(4KB)	0.446	.003	0.146	.000	-67 %	0.136	.000	-70 %
brk+free	(16KB)	0.444	.004	0.146	.000	-67 %	0.136	.000	-69 %
brk+write+free	(1KB)	1.783	.059	0.167	.000	-91 %	0.160	.000	-91 %
brk+write+free	(4KB)	1.924	.266	0.197	.000	-90 %	0.190	.000	-90 %
brk+write+free	(16KB)	4.195	.033	0.317	.000	-92 %	0.311	.000	-93 %

Table 7.4: `mmap()` and `brk()` latency. The comparison is among (1) native Linux processes; (2) Graphene on Linux host, with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. System call latency is in microseconds, and lower is better. Overheads are relative to Linux; negative overheads indicate improvement.

and deallocation will be dominated by the PAL calls, especially since the SGX PAL has to zero allocated pages before returning to `libLinux`.

As an opposite example, allocation and deallocation with `brk()` are significantly faster inside `libLinux` than native, because `libLinux` encapsulates most of the operations. The emulation of `brk()` in `libLinux` preallocates a large enough heap space for consequential allocations, so `brk()` can be mostly serviced in `libLinux` without making frequent PAL calls. As a result, either on the Linux host or the SGX host, the latency on `brk()` is up to 70% faster than native.

7.2.5 Threading

Thread creation in `libLinux` is subject to additional overheads including allocating thread control blocks (TCB) and synchronization between the caller of `clone()` and new thread. To emulate a simple `clone()` system call, the overhead on the Linux host is ~84%. Within the overhead, 50% is contributed by the PAL call, `ThreadCreate()`, and the rest counts toward TCB allocation and synchronization. If `libLinux` runs inside an enclave, the overhead on `clone()` is much more significant, reaching ~10×. Most of the cost on the SGX host is caused by syn-

	System call latency (μs), +/- Confidence Interval, %/× Overhead										
Test	**Linux 4.10**		**Graphene**			**Graphene+SC+RM**			**Graphene-SGX**		
	μs	+/-	μs	+/-	O	μs	+/-	O	μs	+/-	O
`clone+exit`	10.486	.316	19.218	.456	83 %	19.325	.463	84 %	120.297	18.987	10 ×
`pthread_create+join`	8.456	.138	23.622	1.386	179 %	23.884	1.841	182 %	168.981	26.777	19 ×
select files (64 fds)	0.742	.005	144	12	193 ×	149	11	200 ×	461	0	620 ×
select files (128 fds)	1.276	.001	427	0	333 ×	440	0	344 ×	1,119	0	876 ×
select files (256 fds)	2.287	.001	1,486	5	649 ×	1,539	0	672 ×	2,977	1	1,301 ×
select TCP (64 fds)	1.975	.001	45	0	22 ×	51	0	25 ×	371	17	187 ×
select TCP (128 fds)	4.609	.001	104	0	22 ×	117	0	24 ×	855	32	185 ×
select TCP (256 fds)	11.050	.005	272	8	24 ×	288	2	25 ×	1,518	10	136 ×
pthread_cond/mutex	4.244	.644	6.384	.354	50 %	6.711	.000	58 %	11.981	.062	182 %
pthread_mutex (1 thread)	0.019	.000	0.019	.000	0 %	0.019	.000	0 %	0.019	.000	0 %
pthread_mutex (2 threads)	0.167	.008	0.154	.004	-8 %	0.156	.005	-7 %	0.159	.003	-5 %
pthread_mutex (4 threads)	0.386	.011	0.316	.009	-18 %	0.316	.004	-18 %	0.353	.005	-9 %
pthread_mutex (8 threads)	0.793	.016	0.758	.031	-4 %	0.802	.042	1 %	2.203	.501	178 %

Table 7.5: Performance of threading and scheduling operations, including cloning a thread, polling file descriptors, and synchronization primitives (signaling a conditional variable, acquiring a mutex). The comparison is among (1) native Linux processes; (2) Graphene on Linux host, both without and with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. System call latency is in microseconds, and lower is better. Overheads are relative to Linux; negative overheads indicate improvement.

chronization since the overhead on `ThreadCreate()` is only 2×. If comparing the latency of `pthread_create()+pthread_join()`, the overheads mostly double on both the Linux host and the SGX host, primarily because of extra synchronization cost through emulated futexes used by the pthread library for exit notifications.

Polling file descriptors using `select()` is one of the most expensive system calls emulated by `libLinux`. For instance, to poll 256 file descriptors, Table 7.5 shows up to 672× overheads on the Linux host, or 1,301× inside an enclave. The reason for the high latency is due to the definition of the PAL call, `ObjectsWaitAny()`, which only returns one handle at a time. Therefore, to poll 256 file descriptors, `libLinux` has to call `ObjectsWaitAny()` up to 256 times, to confirm the readiness of all polled file descriptors. A possible optimization is to redefining the PAL call to allow returning multiple handles, which remains as future work.

The synchronization primitives offered by the pthread library, including conditional variables and mutexes, are largely affected by the performance of futexes emulated by `libLinux`, especially when one thread has to block for the others. As one example, the latency of signaling a blocking thread through a condition variable is 50% slower on the Linux host, or 182% slower in

	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	420	1,050	25	444	1.227 s	.069 s	6,357 ×
double fork/exit	562	18	2,241	74	299	2,325	28	314	2.423 s	.086 s	4,310 ×
vfork/exit	456	10	1,305	288	186	1,352	239	196	1.156 s	.056 s	2,534 ×
fork/execve	610	28	1,470	12	141	1,570	14	157	1.282 s	.067 s	2,101 ×
double fork/execve	979	23	2,837	176	190	2,932	63	199	2.311 s	.107 s	2,360 ×

Table 7.6: Process creation latency. The comparison is among (1) native Linux processes; (2) Graphene on Linux host, both without and with seccomp filter (**+SC**) and reference monitor (**+RM**); (3) Graphene-SGX. Latency is in microseconds, except for Graphene-SGX, which is in seconds. Lower latency is better. Overheads are relative to Linux; negative overheads indicate improvement.

an enclave, compared to the native performance.

7.2.6 Process Creation

Process creation is one of the most expensive operations in Graphene. As the worst example of process creation, `fork()+exit()` is 4.4× slower on the Linux PAL, or 6,357× from an enclave (∼1.2s) Profiling indicates that about one-sixth of this overhead is in process creation, which takes additional work to create a clean picoprocess on Linux; we expect this overhead could be reduced with a kernel-level implementation of the process creation ABI, rather than emulating this behavior on `clone()`. Another half of the overhead comes from the checkpointing code in `libLinux` (commensurate with the data in Table 7.6), 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 tuning on `libLinux` implementation.

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.

The experiments 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`, result in substantial overheads relative to Graphene. An efficient implementation of

Test		Linux		Graphene+SC+RM		
		μS	+/-	μS	+/-	
msgget	in-process	3,320	0.7	2,823	0.3	-15 %
(create)	inter-process	3,336	0.5	2,879	3.6	-14 %
	persistent	N/A		10,015	0.7	202 %
msgget	in-process	3,245	0.5	137	0.0	-96 %
	inter-process	3,272	3.4	8,362	2.4	156 %
	persistent	N/A		9,386	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 7.7: 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.

an environment similar to FreeBSD jails [161] would make all reference monitoring overheads negligible.

7.2.7 Inter-Process Communication

Table 7.7 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 [124]. 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 were particularly helpful when a producer/consumer pattern was detected.

7.3 Application Performance

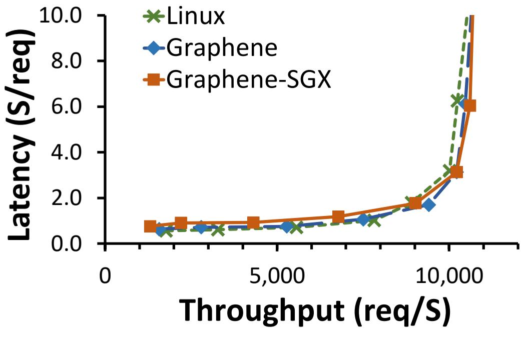
This section evaluates the end-to-end performance of several commercial applications, including server-type applications such as web servers (Apache, Lighttpd, and NGINX), and command-line applications such as language runtime (R and Python), utility programs (CURL), and compilers (GCC).

7.3.1 Server applications

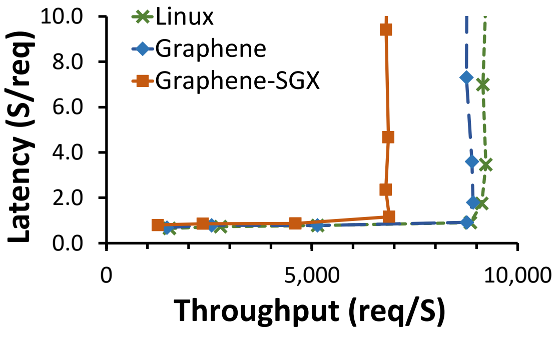
This section measures three widely-used servers, including **Lighttpd** [12] (v1.4.35), **Apache** [1] (v2.4.18), and **NGINX** [16] (v1.10). For each workload, the experiment uses ApacheBench [2] to download static pages on a separate host. The concurrency of ApacheBench is gradually increased during the experiment, to test both the per-request latency and overall throughput of the server. Figure 7.12 shows the throughput versus latency of these server applications in Graphene-SGX, Graphene, and Linux. The paragraphs below discuss each workload individually.

Lighttpd [12] is a web server designed to be light-weight, yet robust enough for commercial uses. Lighttpd is multi-threaded; the experiment tests 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.

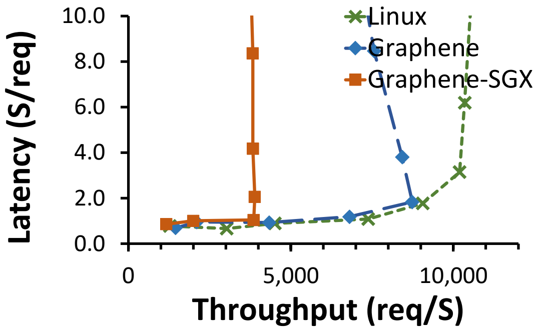
Apache [1] is one of the most popular production servers. The experiment tests Apache using 5 preforked worker processes to service HTTP requests, to evaluate the efficiency of Graphene-SGX across enclaves. This application uses IPC extensively—the preforked processes of a server use a System V semaphore to synchronize on each connection. Regardless of the workload, the response time on Graphene-SGX is 12–35% higher than Linux, due to the overhead of coordination across enclaves over encrypted RPC streams. The peak throughput achieved by Apache running in Graphene-SGX is 26% lower than running in Linux. In this workload, most of the overheads are



(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. The experiment uses an ApacheBench client to gradually increase load, and plot throughput versus latency at each point. Lower and further right is better.

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$\times$ lower than Linux; without SGX, Graphene only reaches 79% of Linux's peak throughput. Using tools like Eleos [130] to reduce enclave exits would help this workload and the integration remains as future work.

7.3.2 Command-line applications

This section evaluates the performance of a few commonly-used command-line applications. Three off-the-shelf applications are tested in the experiment: **R** (v3.2.3) for statistical computing [23]; **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 can be used in an enclave to handle sensitive data—either on a server or a client machine.

The experiment measures the latency or execution time of these applications. In the experiment, 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 [164], 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 [23] 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. The experiment uses an R benchmark suite developed by Urbanek et al. [22], 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

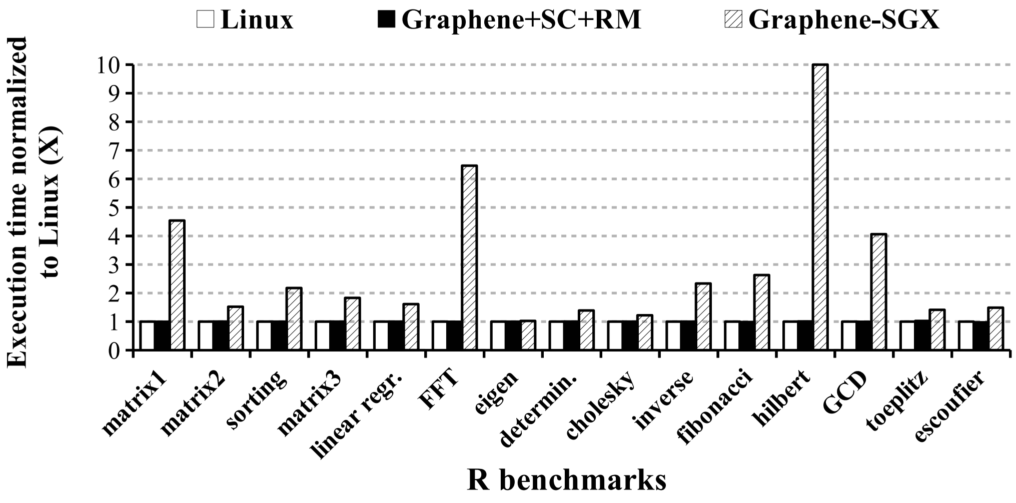
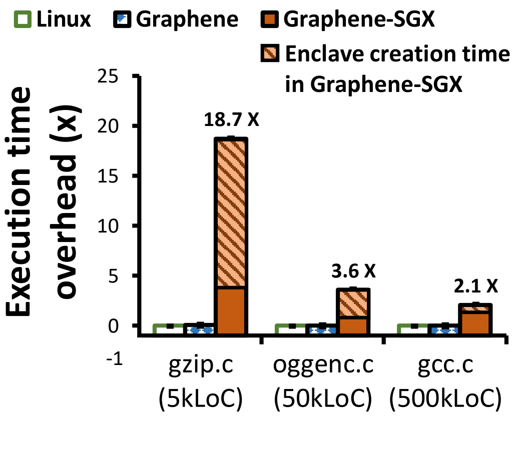
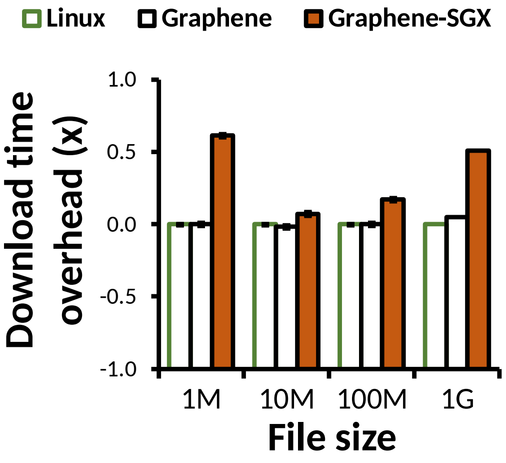


Figure 7.13: Performance overhead on desktop applications, including latency of R, execution time of GCC compilation, download time with CURL. The evaluation compares native Linux, Graphene, and Graphene-SGX.



(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.

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()`. The experiment tests the compilation of the source files with varied sizes, using single C source files collected by MIT [8]. Each GCC execution typically creates five processes, and the experiment runs 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$\times$ 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$\times$ 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), the experiment has to enlarge one of the enclaves (`cc1`) to 2GB, but running on Graphene-SGX (53.1s) is only 2.1$\times$ 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. The experiment evaluates 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

This section evaluates various performance factors of using the Graphene library OS for running unmodified Linux applications. As one of the dominating factors, the PAL ABI performance is primarily determined by the latency of underlying host system calls but burdened with security checks such as system call restriction and reference monitoring. As one of the worst examples, the SGX PAL tends to suffer significant overheads on several PAL calls. The potential factors that are responsible for the overheads include enclave exit and re-entrance, copying PAL call arguments across the enclave boundary, and security checks based on cryptographic techniques.

Despite the potential significant overheads on the PAL ABI, `libLinux` introduces several optimizations based on reducing the average number of PAL calls per emulated system call.

Several system calls, such as `getppid()` and similar system calls, accessing pseudo files (e.g., `/dev/zero`), `brk()`, have shown close-to-native or even better performance which is irrelevant from the PAL ABI implementation. For file system operations, caching and buffering can effectively reduce the impact of PAL call overheads, by deploying spared picoprocess process memory to store temporarily abstraction states. As a result, both server and command-line applications show comparable end-to-end performance on either Graphene or a native Linux kernel. For Graphene-SGX, most of the tested workloads show less than $2\times$ overheads in general.

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 and the impact of changing or deprecating an API. Based on the compatibility metrics, OS developers can strategize 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

The study starts with a research perspective, in search of a better way to evaluate the completeness of system prototypes with a 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. 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 *all* applications and many that are essentially never used. Thus, a system with "partial support" for `ioctl()` is just as likely to support all or none of the Ubuntu applications.

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, the study must take into account 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 [64, 168]. 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`. The final study data is drawn from over 2.9 million installations (2,745,304 Ubuntu and 187,795 Debian).

The thesis introduces two compatibility-related metrics: one for each API, and one for a whole system. For each API, a metric should measure how disruptive its absence would be to applications and end users—a metric we call **API importance**. For a system, a metric should compute a weighted percentage we call **weighted completeness**. For simplicity, this thesis defines 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.

Moreover, this study focuses only 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

This thesis also measure compatibility at the granularity of an OS, using a metric called 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.

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, it is hard to 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, this thesis assumes the installations overlap. Appendix A.2 formally defines weighted completeness.

8.2 Data Collection

This study uses 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

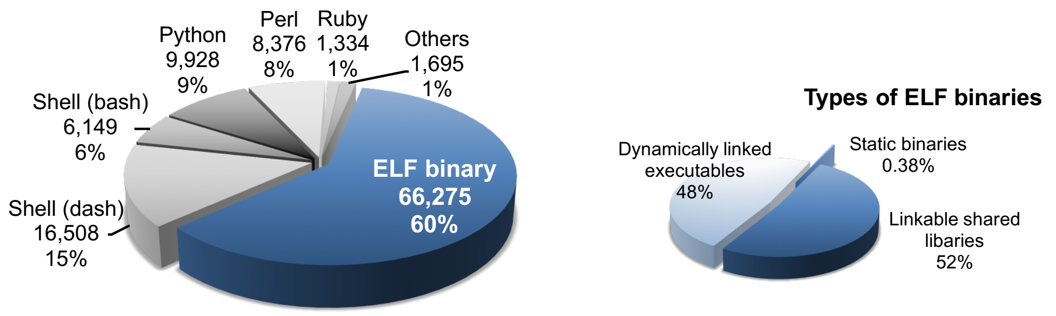


Figure 8.1: Percentage of ELF binaries and applications written in interpreted languages among all executables in the Ubuntu/Debian Linux repository, categorized by interpreters. ELF binaries include static binaries, shared libraries and dynamically-linked executables. Interpreters are detected by *shebangs*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

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

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. the assumption is that binaries can issue system calls directly with inline system call instructions, or can call system calls through a library, such as libc. The static analysis identifies system call instructions and constructs a whole-program call graph.

The study also 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, The assumption is 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 this study.

The study 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. The study uses a simple call-graph analysis to detects system calls reachable from the binary entry point (`e_entry` in ELF headers). The analysis searches all binaries, including libraries, for system call instructions (`int 0x80`, `syscall` or `sysenter`) or calling the `syscall` API of libc. The analysis result shows 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.

The call-graph analysis allows users to select only system calls that are actually used by the application, not all the system calls that appear in libc. The 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, the analysis tracks 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, the analysis assumes that a function pointer assigned to a local variable will be called.

The analysis also hard-codes for a few common and problematic patterns. For instance, the analysis 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. The analysis also spot checks this assumption, but does 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.

The implementation is summarized in Table 8.1, including 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, The 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.

The study assumes 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 the 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. The 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.

The assumption is 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`, the study measures the API importance of these arguments. Verifying bug-for-bug compatibility generally requires

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.)

techniques largely orthogonal to the ones used in this study, and thus this is beyond the scope of this work.

Metrics. The proposed metrics are intended to be simple numbers for easy comparison. But this coarseness loses some nuance. For instance, the 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.

This section evaluates the weighted completeness of four Linux-compatible systems or emulation layers: User-Mode-Linux [67], L4Linux [84], FreeBSD emulation layer [68], and Graphene library OS [164]. For each system, we explore techniques to help identifying the supported system calls, based on how the system is built. For example, User-Mode-Linux and L4Linux are built by modifying the Linux source code, or adding a new architecture to Linux. These systems will define architecture-specific system call tables, and reimplement `sys_*` functions in the Linux source that are originally aliases to `sys_ni_syscall` (a function that returns `-ENOSYS`). Other systems, like FreeBSD and Graphene, are built from scratch, and often maintain their own system call table structures, where unsupported 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`. A previous version of Graphene weighted completeness is only 0.42% complete. The observation shows that the primary culprit is scheduling control—by adding two scheduling system calls, Graphene's weighted completeness is 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 [24], 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 to GNU libc. Both uClibc and musl have a low weighted completeness, because GNU libc's

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()`).

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

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.

9.1 Library OSes and Virtualization

Previous Library OSes. Previous library OSes [18, 45, 69, 119, 138] focus on running single-process applications in a picoprocess or a unikernel for various reasons including compatibility and security isolation. Bascule [45] 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 [18].

A number of recent projects have provided a minimal, isolated environment for web applications to download and execute native code [69, 87, 126, 172, 180]. 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, 60, 95, 109], 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 library OSes is moving functionality from the kernel into applications and reducing the TCB (trusted computing base) size or attack surface. Kaashoek et al. [95] identify multi-processing as a problem for an Exokernel library OS, 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 library OSes, nor will they facilitate dynamically sandboxing two processes.

User Mode Linux [67] (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 [42], 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 [42, 70, 175]. 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 [73, 171], Amoeba [59, 128] and Athena [54] 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 [135], 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 [44] 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 [156] 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 [44] 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 the parent ("chief") before a message could leave the clan [112]. Although this model was abandoned as cumbersome for general-purpose access control [71], the Graphene sandbox design observes that a stricter variation is a natural fit for security isolation among multi-process applications.

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

Migration and Security Isolation. Researchers have added checkpoint and migration support to Linux [107] 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 [154], containers [49], and Solaris Zones [139], 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 [98]. FreeBSD jails apply a similar approach to augment an isolated `chroot` environment with other isolated namespaces, including the network and hostname [161]. Similarly, Zap [131] 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 [149] 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.

9.2 Trusted Execution

Protection Against Untrusted OSes. Protecting applications from untrusted OSes predates hardware support. Virtual Ghost [62] 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 [120], MUSHI [183], SeCage [117], InkTag [86], and Sego [104] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koeberl et al. [102], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [110] built a 2-way sandbox for x86 by separating the Native Client (NaCl) [181] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [121] to protect the piece of application logic from the untrusted OS. Jang et al. [94] build a secure channel to authenticate the application in the Untrusted area isolated by the ARM TrustZone technology. Song et al. [155] 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 [111] 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 [163] on ARM creates an isolated environment for trusted kernel components. Different from SGX, TrustZone separates the hardware between the trusted and untrusted worlds, and builds a trusted path from the trusted kernel to other on-chip peripherals. IBM SecureBlue++ [50] also isolates applications by encrypting the memory inside the CPU; SecureBlue++ is capable of nesting isolated environments, to isolate applications, guest OSes, hypervisors from each other.

AMD is introducing a feature in future chips called SEV (Secure Encrypted Virtualization) [97], 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 [61] 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 [46], SCONE [39], PANOPLY [152], 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 [46] uses the Drawbridge library OS [138] 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 [39] 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 [152] 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 [146] runs MapReduce jobs in SGX enclaves. Similarly, Brenner et al. [51] run cluster services in ZooKeeper in an enclave, and transparently encrypt data in transit between enclaves. Ryoan [89] sandboxes a piece of untrusted code in the enclave to process secret data while preventing the loaded code from leaking secret data. Opaque [185] 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 [150], as well as inter-domain routing in Tor [99].

Several improvements to SGX frameworks have been recently developed, which can be integrated with applications on Graphene-SGX. Eleos [130] reduces the number of enclave exits by asynchronously servicing system calls outside of the enclaves, and enabling user-space memory paging. SGXBOUND [103] is a software technique for bounds-checking with low memory overheads, to fit within limited EPC size. T-SGX [151] 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 [148] 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 [115] uses data-flow analysis at compile-time, to automatically determine the partition boundary in an application.

9.3 System API Studies

API Usage Study. Concurrent with our work, Atlidakis et al. [40] 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 [96] 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 [132]. They find architecture-specific subsystems have highest fault rate, followed by file systems. Harter et al. [83] studied the interaction of a set of Mac OS X applications with the file system APIs—identifying a number of surprising I/O patterns. Our approach is complementary to these studies, with a focus on overall API usage across an entire Linux distribution.

Application Statistics. A number of previous studies have drawn inferences about user and developer behavior using Debian and Ubuntu package metadata and popularity contest statistics. Debian packages have been analyzed to study the evolution of the software itself [77, 129, 144], to measure the popularity of application programming languages [33], to analyze dependencies between the packages [63], to identify trends in package sizes [32], the number of developers involved in developing and maintaining a package [143], 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 [93]. This study is unique in using this information to infer the relative importance 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) [65] 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 [134]. 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 [136]. 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 [153]. 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 [182]. Static binary analysis has been used to automatically generate application-specific sandboxing policies [108]. Dynamic analysis has been used to compare system call sequences of two applications as an indicator of potential intellectual property theft [174], to identify opportunities to batch system calls [140], to model power consumption on mobile devices [133], and to repackage applications to run on different systems [80]. These projects answer very different questions than ours, but could, in principle, benefit from the resulting data set.

Chapter 10

Conclusion

Application developers pay varied efforts to port applications to new OSes or hardware to gain qualitative benefits. Application porting efforts range from recompilation to reimplementation, due to API distinction or host-specific restrictions. Existing library OSes [45, 46, 138] 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 [164], which supports unmodified Linux multi-process applications. In Graphene, the idiosyncratic, Linux multi-process abstractions—including forking, signals, System V IPC, file descriptor sharing, exit notification—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 reuse unmodified 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 an unmodified application, in an environment immune to attacks from host OSes, hypervisors, and hardware devices.

Developers also struggle to prioritize system APIs that are important to compatibility due to skewed observations toward targeted workloads. Alternatively, this thesis suggests a more fractal measurement for estimating how system APIs (e.g., Linux system calls) are used in applications, weighted by 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%.

Appendix A

Formal Definitions

A.1 API Importance

A system installation ($\texttt{inst}$) is a set of packages installed ($\{\texttt{pkg}_1, \texttt{pkg}_2, ..., \texttt{pkg}_k \in \texttt{Pkg}_{\texttt{all}}\}$). For each package ($\texttt{pkg}$) 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:

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

Assuming the packages that use an API are $\texttt{Dependents}_{\texttt{api}} = \{\texttt{pkg} | \texttt{api} \in \texttt{Footprint}_{\texttt{pkg}}\}$. API importance is the probability that at least one package from $\texttt{Dependents}_{\texttt{api}}$ is installed on a random installation, which is calculated as follows:

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

A.2 Weighted Completeness

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

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

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

Because we do not know which packages are installed together, except in the presence of explicit dependencies, we assume package installation events are independent. Thus, the approximated value of weighted completeness is:

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

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] Desktop operating system market share. `https://www.netmarketshare.com/operating-system-market-share.aspx`.

[18] OSv. available at `http://osv.io`.

[19] Perl. `https://www.perl.org/`.

[20] Python. `https://www.python.org/`.

[21] QEMU. `https://www.qemu.org/`.

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

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

[24] *uClibc*. `https://www.uclibc.org`.

[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] 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.

[40] 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.

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

[42] 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.

[43] 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.

[44] 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.

[45] 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.

[46] 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.

[47] 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.

[48] 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.

[49] 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.

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

[51] 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.

[52] 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.

[53] 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 archi-

tecture for cellular computing. In *Proceedings of the IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 2002.

[54] 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.

[55] 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`.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.

[63] 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.

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

[65] 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.

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

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

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

[69] 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.

[70] 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.

[71] 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.

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

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

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

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

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

[77] 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.

[78] 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.

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

[80] 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.

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

[82] 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.

[83] 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.

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

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

[86] 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.

[87] 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.

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

[89] 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.

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

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

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

[93] 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.

[94] 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.

[95] 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.

[96] 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.

[97] 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.

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

[99] 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.

[100] 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.

[101] 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.

[102] 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.

[103] 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.

[104] 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.*

[105] 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.

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

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

[108] LapChung Lam and Tzi-cker Chiueh. Automatic extraction of accurate application-specific sandboxing policy. In Erland Jonsson, Alfonso Valdes, and Magnus Almgren, editors, *Re-*

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

[109] 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.

[110] 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.

[111] 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.

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

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

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

[115] 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.

[116] 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.

[117] 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.

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

[119] 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.

[120] 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.

[121] 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.

[122] 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`.

[123] 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`.

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

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

[126] 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.

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

[128] 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.

[129] 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.

[130] 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.

[131] 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.

[132] 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.

[133] 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.

[134] 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.

[135] 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.

[136] 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.

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

[138] 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.

[139] 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.

[140] 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.

[141] Dennis M. Ritchie. The unix time-sharing system–a retrospective. 1978.

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

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

[144] 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.

[145] 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.

[146] 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.

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

[148] 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.

[149] 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.

[150] 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.

[151] 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.

[152] 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.

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

[154] 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.

[155] 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.

[156] 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.

[157] Hannes Frederic Sowa. zerocopy stream bits on AF_UNIX sockets. `https://www.spinics.net/lists/netdev/msg329905.html`, May 2015.

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

[159] 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.

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

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

[162] 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.

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

[164] 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.

[165] 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.

[166] 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.

[167] Ubuntu Packages. Ubuntu packages. available at `http://pacakges.ubuntu.com`.

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

[169] 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.

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

[171] 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.

[172] 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.

[173] 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.

[174] 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.

[175] 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.

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

[177] 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.

[178] 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.

[179] 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.

[180] 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.

[181] 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.

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

[183] 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.

[184] 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.

[185] 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.

[186] 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.