Securely and Efficiently Reusing Legacy Applications with the Graphene Library OS

A Dissertation Proposal presented

by

Chia-Che Tsai

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

August 2016

Abstract of the Dissertation Proposal

Securely and Efficiently Reusing Legacy Applications with the Graphene Library OS

by

Chia-Che Tsai

For the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

2016

Legacy applications dominate a substantial part of the present software products. In order to reuse the development effort of existing applications, system engineers design emulation layers and translation toolchains for accelerating the process of porting any application from one platform to another. Essentially, the complexity of porting is determined by the distinction of *personalities* among platforms. Reusing the applications from commodity OSes (e.g., Windows, Linux) on a vastly different platform, such as an isolated enclave or a heterogeneous architecture, requires significant rewriting of the applications and systems, to adapt for the new limitations and requirements.

The **library OSes** [Baumann et al., 2013; Madhavapeddy et al., 2013; Porter et al., 2011] are designed to translate the high-level APIs and abstractions relied by the legacy applications, to a narrow, generic host interface. The qualitative benefits of library OSes, such as isolation between mutually untrusting applications, platform independence, and migration of OS state, are comparable to virtualization — at a lower cost [Porter et al., 2011]. However, the reuse of multi-process applications, a vast domain commonly required on UNIX platforms (e.g., Linux, BSD and OSX), is lacked in previous library OSes due to the splitting of OS functionality into application processes, as the so-called **picoprocesses**. Supporting the multi-process abstractions requires multiple library OS instances to collaboratively maintain the system state, as one shared OS for the application.

We design a library OS called **Graphene** [Tsai et al., 2014], which supports legacy Linux, multi-process applications, using a simple, portable host ABI on any platform. In Graphene, the

idiosyncratic, Linux multi-process abstractions — forking, signals, System V IPC, file descritor sharing, exit notification, etc — are coordinated across picoprocesses over host-provided, pipe-like RPC streams. The design facilitates the security isolation of shareable OS state and abstraction via the simple, isolating host ABI. With platform independence, Graphene can extend the support for legacy Linux applications to platforms that are generally hard to port, including the **Intel SGX enclaves**. Graphene has a small memory footprint—3–20 times smaller than KVM —and acceptable performance overhead; for instance, Compiling `bzip2` on four CPU cores has only 8% overhead on Graphene compared to native Linux.

Besides reusing whole applications on diverse platforms, the Graphene library OS also allows securing a sensitive part of an applications written in managed languages (e.g., Java), in an isolated enclave. Using Graphene, we build a prototype framework called **Civet**, which automatically splits a Java application into clean partitions, to be selectively loaded into enclaves. The framework allows various commercial Java applications to utilize the Intel SGX hardware, and apply the Java language protections (e.g., type-checking) to prevent exposure of vulnerabilities to attackers. Using Civet, we isolate the cryptographic components of a SSH session, with only configuration effort for guiding the partitioning.

When building the legacy support in an OS prototype like Graphene, developers often prioritize what they believe to be more *important*, which can be heavily skewed toward their preferred workloads Tsai et al. [2016]. Alternatively, we suggest a more fractal measurement for estimating how system APIs (e.g., Linux system calls) are used in applications, weighted by the application popularity. 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 can be improved from 0.42% to 21.1%.

The trade-off between reusing legacy applications and designing a secure, efficiency system exists in both library OSes and kernels. An performance-centric, heavily-engineered subsystem of Linux, the **file system directory cache**, shows suboptimal lookup latency — causing the execution time of `git-diff` be dominated by `stat` system calls, even when all paths are cached [Tsai et al., 2015]. We build a fast path to improve the hit latency of directory cache, by decoupling path searching from other redundant operations like permission checks and symbolic-link resolution. With the fast path, the hit latency of `stat` system call can be improved for up to 26%, and the execution time of `git-diff` on the whole Linux source is reduced by one-quarter. The design is applied to both Linux 3.19 and Graphene, and generalized to support all OS features, security modules, and most underlying file systems.

In conclusion, this thesis seeks systematic and generalizable solutions, for mitigating the limitations on reusing legacy applications, using the Graphene library OS. The proposed works for fulfilling the thesis include the completion of the Civet framework, implementing missing Linux features and host ABI in Graphene, and porting Graphene to more distinct platforms (e.g., microkernels, Barrelfish Baumann et al. [2009])

Table of Contents

List of Figures / Tables

Figures

Tables

Acknowledgments

Chapter 1

Introduction

As an immense resource, legacy applications and systems represent the accumulated development effort ever made by developers. Up to millions of applications are available in the mainstream App stores [Apple App Store; Google Play; Ubuntu Packages]; In many occasions, developers attempt to port legacy applications from one operating system, platform, or language to another, in order to harvest the past design and implementation. For applications that are not yet ported, users who desire to utilize them are restricted to the operating systems or programming languages that the applications are designed upon. Therefore, users and developers seek systematic approaches such as virtualization, for painless methods to port any legacy applications.

Between different systems, the complexity of porting is primarily caused by the design decisions made by the system developers. The developers' decisions can be based on idiosyncratic abstractions, security mechanisms, resource usage, quality of service, and other countless concerns. For instance, porting an application between Windows and Linux requires translating the system APIs between the OSes. The primary challenge is that system APIs can be vastly different and difficult to translate — Take the memory mapping APIs, `VirtualAlloc` in Windows and `mmap` in Linux, for example. These APIs not only differ in the semantic, but also allocation granularity, security policies and the logic handling edge cases like unaligned addresses. As a result, porting applications often requires reimplementation effort more than simply swapping system APIs and programming language phrases.

As alternatives to virtualization, **library OSes** provides a highly portable, **platform independent** platform, by splitting the OS functionality from a kernel into the application processes [Baumann et al., 2013; Madhavapeddy et al., 2013; Porter et al., 2011]. Compared with monolithic kernels, library OSes minimize the reliance on the correctness or effectiveness of kernels, to enforce isolation on applications that distrust each other and demand a more impenetrable barrier. The isolation of library OSes is based on partitioning the OS instances between the mutually untrusting applications, and leaving the library OSes only a narrow, restrictively defined interface to the host kernel. Within a process with a library OS, or a **picoprocess**, the library OS implements the system APIs (e.g., system calls) and supporting data structures as library functions — mapping the system abstractions and semantics to the narrow host interface. Recent library OSes improve efficiency over full guest OSes by eliminating duplicated features between the guest and host kernel, such as the CPU scheduler, or even compiling out unnecessary guest kernel APIs [Madhavapeddy et al., 2013]. Sometimes, on top of a minimized **microkernel** that often has the size and functionality of a hypervisor, library OSes or applications can bypass the multiplexing of hardware resource to improve the performance [Engler et al., 1995], or separating system components that are intensively shared and bottlenecking the applications, such as bulk allocation of processing buffers [Leslie

et al., 1996]. In total, this can reduce the memory requirements of running a single, isolated application by orders of magnitude, and similarly increase the number of applications which can run on a single system [Madhavapeddy et al., 2013; Porter et al., 2011]. In addition, because of the narrowness of host interfaces, the library OSes can be easily ported to an innovative, vastly distinct platform. A typical example is in **Intel SGX enclaves**, where applications are isolated in a restricted environment that is immune to attacks from untrusted host kernel and hardware peripherals. A library OS can support whole applications in enclaves, without expanded attack surface and porting effort on the applications [Baumann et al., 2014].

Legacy application support is provided in recent library OSes [Baumann et al., 2013, 2014; Porter et al., 2011], by emulating the OS personalities in individual picoprocesses. Among the OS abstractions and features that need to be emulated, the emulation of single-process abstractions, such as accessing unshared files, or creating multiple threads in a process, can be wrapped inside a picoprocess, thus often relatively straightforward. For instance, Drawbridge [Porter et al., 2011] provides Windows abstractions and personalities, with 5.6 million lines of code reused from Windows 7 to emulate the complete OS personality in a single picoprocess, except a few features such as the Windows registry that has to be implemented separately as in-process libraries or services. However, when it comes to emulating multi-process abstractions, which are commonly used by Unix applications, picoprocesses serving multiple processes of an application will need to collaboratively provide shareable OS abstractions, and more importantly, a unified OS view to the application. The multi-process abstractions to be emulated include forking, signals, System V IPC, file descriptor sharing, exit notification, etc. A possible alternative is to rely on the hosts' memory sharing features to share OS states. However, memory sharing is not always an available features in the host: for example, Drawbridge and Bascule [Baumann et al., 2013] cannot emulate process forking because copy-on-write memory sharing is not a platform-independent features. Furthermore, when Haven isolates applications in enclaves, memory sharing is strictly forbidden by the hardware platform, to prevent leaking secret to other enclaves. In a word, emulating OS abstractions and personalities can fundamentally oppose the assumptions or limitations of the system, causing obstacles to supporting legacy applications.

The key to emulating multi-process abstractions in a library OS, especially when memory sharing is not safe or available, is to coordinate shared OS states across picoprocesses using limited host abstractions. The OS states needed to be coordinated are primarily in three types: process states that are inherited at process creation (e.g., `fork`, `execve`); abstraction states that are shared among related processes (e.g., signals, System V IPC semaphores); and namespace states to locate the owners of abstractions (e.g., process IDs, System V IPC keys). A library OS that implements all the idiosyncratic OS states must not overly expand the host interface, and compromise the security isolation or platform independence of the system. As a solution, we present **Graphene**, a Linux-compatible library OS that collaboratively implement various Linux multi-process abstractions, yet appear to the application as a single, shared OS. Graphene instances coordinate all OS states using remote procedure calls (RPCs) over host-provided, pipe-like streams. The use of host RPC streams does not expand the host interface already used by a single picoprocess, and can be naturally isolated by sandboxing the RPC streams from the host.

In particular, the platform independence of library OSes allows porting legacy applications to platforms with more restricted interfaces and functionalities than monolithic kernels, such as Intel SGX enclaves. The essence of isolating applications using library OSes in enclaves is to ensure the security benefits promised by the platform, including application integrity, sealing confidential

information, and attestation. An existing system, Haven [Baumann et al., 2014], packs a Windows-compatible picoprocess into an enclave, using an encrypted virtual disk securely provisioned from a remote, attested server. Haven essentially provides a trusted path to safely load application and library OS binaries, as well as mediating all input and output to the host. The only limitations in the model of Haven, however, are the lack of multi-process support, and being overly reliant on the remote server. Addressing these issues, by porting Graphene to enclaves (**Graphene-SGX**), we reassure multi-process, UNIX applications in a multi-enclave environment. The application integrity in Graphene-SGX is validated on unencrypted, regularly distributed binaries, using checksums signed by the hardware-attested measurement — an off-line model without relying on remote servers. Graphene-SGX ensures that process creation in enclaves can be scalable, and individually configured — a parent enclave can identify and restrict the application binaries to be loaded into any child enclaves it creates.

Improving Efficiency with Legacy Support. One of the other first-order requirements in systems, besides legacy support, is to pursue low latency or high throughput, according to how users define quality of service. Nevertheless, there is a trade-off between the emulation of OS personality and the efficiency of system operations. Graphene, for instance, coordinates OS states over PRC streams, and requires strategies to mitigate the significant RPC coordination overhead. We observe, in a distributed UNIX implementation, placement of shared OS states and messaging complexity are essentially the top performance factors. Improving the coordination overhead will require placing the OS states in the most frequency accessing picoprocesses, and reducing round trips of RPC messages.

Partitioning Legacy Applications. Beside supporting whole applications, we also explore the opportunity of partitioning a legacy application, in a model where it is split into trusted and untrusted components that are isolated by platforms like enclaves. The partitioned model for porting to enclaves is commonly used for quarantining the extremely sensitive execution in an application. The purpose of partitioning is to defend against the whole, untrusted host systems (except the CPUs) as well as other less sensitive components, in order to minimize the trusted computing base. The weakness of the model is, partitioning an application requires additional effort to identify and separate the execution that needs to be protected, and the effort becomes unacceptably expensive when the application is considerably sophisticated. In particular, we are targeting applications implemented in a managed language such as Java, which are more likely to be automatically compartmentalized using programming language techniques.

The motivation for pursuing Java application partitioning is two-folded. On one hand, many enterprise or cross-platform applications are developed in Java, but so far no enclave support for whole or partial Java applications is ever provided. On the other hand, execution isolated in an enclave must be unconditionally trusted; if any vulnerabilities exist in the execution, the attack can infiltrate from the untrusted host to compromise the isolation. However, just as the whole application, the isolated part is also hard to completely eliminate vulnerabilities. By allowing developers to port Java applications into enclaves, protection mechanisms based on the characteristics of Java language, such as type checking, information flow tracking among objects, can further ensure the soundness of isolated execution. The automated partitioning of Java applications into enclaves is an ongoing work.

Measurements for System Completeness. In general, due to the complexity of implementation or preserving the complete specifications, developers must prioritize the development based on what they believe to be more important — what will impact or satisfy more applications or more users. Especially for innovative systems like library OSes, developers struggle to evaluate the progress in a partial prototype. At the center of these problems is the missing of information about how system abstractions or APIs are used in the run-time. In search of better methods for evaluating the legacy support, we conduct a study of Linux system APIs usage among a large sample of Debian/Ubuntu x86_64 applications, weighted by the application popularity collected by the Popularity Contests [Debian Popcons; Ubuntu Popcons]. Based on the study, we create a measurement for estimating the fraction of applications installed by a user that can plausibly use the system. For instance, the current Graphene prototype, in which we hand-pick Linux system calls according to the applications for experiment, can support the complete API footprint in 0.42% of applications installed by a user. The data-driven, principled strategy shows than Graphene can be improved to supporting 21.1% of applications, by simply adding two missing system calls, `sched_setcheduler` and `sched_setparam`.

Optimizing Linux File System Directory Cache. The difficulty of simultaneously achieving compatibility, efficiency, and other qualities can also be observed in commodity operating systems. In the design of a traditional OS such as Windows or Linux, the implementation of OS abstractions are often entangled with security mechanisms and performance optimizations, leading to suboptimal efficiency. The complexity of the OS design makes any structural changes to the OS logic unreasonably hard to adopt. One classic example is the **Linux file system directory cache**, a heavily engineered component designed for avoiding the significant latency of looking up duplicated paths on physical storage, yet also used as data structures for file system features, such as storing permissions, attributes and symbolic links. In our analysis, even when the directory cache is warm (no cache misses), the latency of path-searching system calls still dominates many lookup-bound applications, such as the GIT version control system. The culprit of the suboptimal latency is the requirement of permission checks and processing file system features that interleaves with looking up path prefixes in the directory cache. A more optimal design shall decouple the looking up cached paths from other operations, by fully exploiting locality to cache the results of redundant operations (i.e., permission checks, retrieving attributes, resolving symboling links, etc). The systematic, principled redesign of Linux file system directory cache preserves the compatibility to existing Linux file system features, as well as various security mechanisms (e.g., AppArmor, SELinux) and most underlying storage (e.g., Ext4, Pseudo file systems).

Contributions. This thesis proposes the challenges, solutions and principles, to mitigate limitations from innovative system designs on enabling the execution of legacy applications. The following list describes the contributions of the thesis, from a technical view:

- A Linux-compatible library OS called **Graphene** [Tsai et al., 2014], which runs legacy Linux applications in a platform independent environment. The platforms where Graphene has been ported include Linux, FreeBSD, OSX, Windows and SGX enclaves (also called **Graphene-SGX**). Graphene extends the security isolation on single-process applications, to processes that share UNIX multi-process abstractions, such as forking, signaling, sharing file descriptors, System V IPC, etc. The shared states across multiple processes of an application is coordinated by the distributed implementation of OS states in Graphene, over the host-provided RPC streams. Besides an optional host abstraction — bulk IPC for optimizing

copy-on-write forking — all multi-process abstractions are implemented and coordinated completely over PRC streams, a feature generally provided by most platforms.

- A framework called **Civet**, that combines the isolated execution of SGX and Java language protections, for partitioning legacy Java applications. Civet includes a tool that automates code compartmentalization, generating an enclave with minimal supporting Java classes; a runtime framework that loads signed classes into an enclave, with language protection such as information flow control at the enclave boundary; and Java APIs to seamless access the SGX features such as attestation and secure provisioning. The use cases of Civet includes isolating session keys and supporting cryptography library for a SSH connection, protecting Hadoop workloads that run proprietary algorithms, and a web application deployed over HTTP. **This work is still in progress.**
- A study of **Linux API usage and compatibility** [Tsai et al., 2016], to provide the information missing in the decision-making process of system developers. The study uses a approach to measure platform compatibility as the relative completeness of prototype systems. Rather than considering compatibility a binary property we use a fractional metric which is better suited to measuring the progress of a prototype. Based on the study, we provides a range of insights into current API usage patterns. For instance, we identify an efficient path to implementing a new Linux compatibility layer, maximizing the additional applications per system call. We also identify that usage of many APIs is similarly distributed: some are widely used, and there is a sharp drop with a very long tail of rarely or never-used APIs.
- We identify the suboptimal lookup latency in **Linux file system directory cache** [Tsai et al., 2015]. This heavily engineered OS component optimizes looking up paths in file systems, yet the searching in the directory cache is closely interleaved with permission checks against security models, and file system features such as resolving symbolic links. We optimize the lookup hit latency by decoupling cache searching from other operations. We cache the result of permission checks on path prefixes in a data structure called prefix check cache, which will be invalidated when permission changes. The hit latency of `stat` system calls on a long path can be optimized to up to 27%, improving the execution time for Dovecot IMAP server by up to 12% and GIT version control system by up to 25%.

Organization. The organization of this thesis is presented as follows:

- §2 describes the Graphene approach for supporting multi-process, Linux applications in mutually isolated picoprocesses, or enclaves.
- §3 presents the ongoing work of partitioning Java applications to isolate the most sensitive execution in enclaves.
- §4 discusses new, fractal measurements for evaluating importance of system APIs and completeness of prototype systems, based on a study of Linux API usage and compatibility.
- §5 explains a fast-path design for optimizing the hit latency for looking up in Linux's egacy file system directory cache.
- §6 lists the proposed tasks for the fulfillment of thesis.
- §7 describes the former works and publications related to this thesis.

Chapter 2

The Graphene Library OS

Existing library OSes provide single-process applications with the qualitative benefits of virtualization at a lower cost [Baumann et al., 2013; Madhavapeddy et al., 2013; Porter et al., 2011]. These benefits include security isolation of mutually untrusting applications, migration, and host platform independence. In a library OS, the guest OS is essentially "collapsed" into an application library, which implements the OS system calls and abstractions required by legacy applications and libraries. The library maps high-level APIs onto a few narrowed interfaces to the host kernel. The reduction of host interfaces provides portability to various platforms that can translate the interfaces to host APIs or abstractions, Compared with visualization, library OSes can eliminate duplicated features between the guest to the host kernel such as the CPU scheduler or file system drivers. Therefore, the memory requirements of running a single, isolated application in library OSes is orders-of-magnitude less than running it with visualization [Madhavapeddy et al., 2013; Porter et al., 2011]. Library OSes have also been proven useful for porting legacy applications onto new hardware platforms, such as Intel's SGX enclaves [Baumann et al., 2014].

A key drawback for recent library OSes to run legacy applications, however, is that the support in these library OSes are mostly limited to single-process applications. Many applications, such as network servers and shell scripts, create multiple processes for performance scalability, fault isolation, and programmer convenience. In order for the efficiency benefits of library OSes to be widely applicable, especially for unmodified Unix applications, library OSes must provide commonly-used multi-process abstractions, such as fork, signals, System V IPC message queues and semaphores, sharing file descriptors, and exit notification. Without sharing memory across picoprocesses, library OS must coordinate shared OS states to support multi-process abstractions. For example, Drawbridge [Porter et al., 2011] cannot simulate process forking because copy-on-write memory sharing is not a platform-independent features.

Graphene is a Linux-compatible library OS to run legacy, unmodified Linux applications. In Graphene, multiple library OS instances collaboratively implement POSIX abstractions, yet appear to the application as a single, shared OS. Graphene instances coordinate state using remote procedure calls (RPCs) over host-provided byte streams (similar to pipes). In a distributed POSIX implementation, placement of shared state and messaging complexity are first-order performance concerns. By coordinating shared states across library OS instances, Graphene is able to create an illusion of running in a single OS for multiple processes in an application (as figure 2.1). The Graphene design ensures security isolation of mutually distrusting, multi-process applications on the same host system. Essential to this goal is minimally expanding the host ABI to support multi-processing, as well as leveraging RPCs as a natural point to mediate inter-picoprocess communication. RPC coordination among Graphene instances can be dynamically disconnected, facilitating

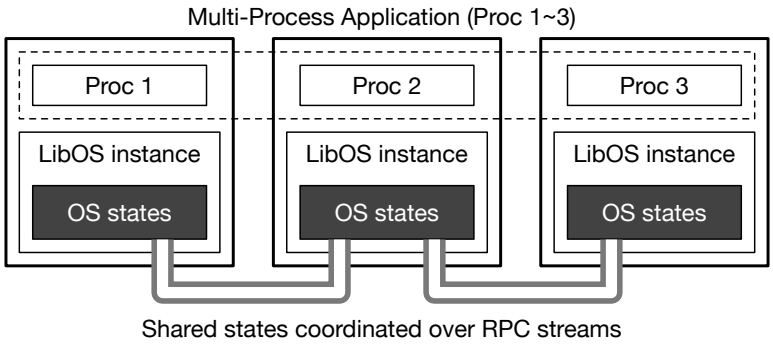


Figure 2.1: Multi-process support model of Graphene library OS. For each process of an application, a library OS instance will serve system calls and keep local OS states. States of multi-process abstractions are shared by coordinating over host-provided RPC streams, creating an illusion of running in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

novel sandboxing techniques. For instance, we develop an Apache web server extension that, upon logging in a given user, places the worker process's library OS in a sandbox with access to only that user's data. We expect more nuanced degrees of trust are possible in future work.

2.1 Implementing Linux Personality

A library OS typically executes in either a paravirtual VM [Madhavapeddy et al., 2013; OSV] or an OS process, called a **picoprocess** [Baumann et al., 2013; Porter et al., 2011], with an interface restricted to a narrowed set of host kernel ABIs. These host ABIs heavily restrict effects outside of the application's address space; as a result, applications in a picoprocess have very little opportunity to interfere with each other, yielding security isolation comparable to a VM. Library OS efficiency comes from deduplicating features, such as hardware management; in a VM these features typically appear in both the guest and host kernels.

Graphene executes within a picoprocess (Figure 2.2), which includes an **unmodified** application binary and supporting libraries, running on a library OS instance. The library OS is implemented over a host kernel ABI designed to expose very generic abstractions that can be easily implemented on any host OS, including virtual memory, threads, synchronization, byte streams (similar to pipes), a file system, and networking. Although the Graphene prototype host kernel is Linux, we adapt a host ABI from Drawbridge/Bascule, which has been previously implemented on Windows, Hyper-V, and Barrelfish [Baumann et al., 2009, 2013; Porter et al., 2011].

Graphene exports 43 host ABIs through a Platform Adaptation Layer (PAL) (Table 2.1). The PAL is a binary injected into the picoprocess by the host, and translates the generic picoprocess ABI into host system APIs. Most of these calls only affect the application-internal state; any calls with external effects are mediated by a trusted **reference monitor** on the host. All Graphene applications are launched by the reference monitor, which installs a system call filter and interposes on permitted kernel calls to ensure isolation (§2.3).

These PAL ABIs should be a sufficient substrate upon which to implement guest-specific semantics, or guest **OS personality**. As an example of this layering, consider the heap memory management abstraction. Linux provides applications with a data segment—a legacy abstraction

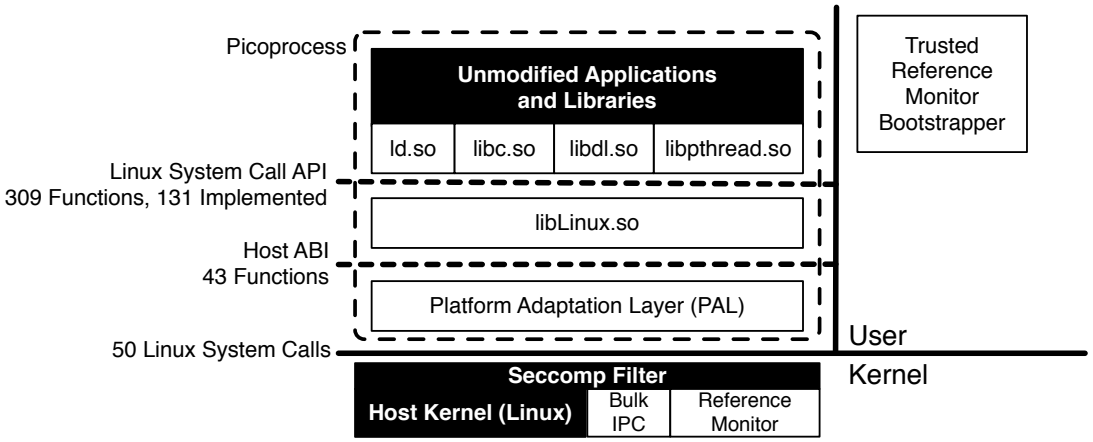


Figure 2.2: Building blocks of Graphene. Black components are unmodified. We modify the four lowest application libraries on Linux: `ld.so` (the ELF linker and loader), `libdl.so` (the dynamic library linker), `libc.so` (standard library C), and `libpthread.so` (standard threading library), that issue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

dating back to original Unix and the era of segmented memory management hardware. 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 `sys_brk`) and the stack grows down until they meet in the middle. In contrast, the PAL ABI provides only three simple functions that allocate, protect, or unmap regions of virtual memory with basic access permissions (read, write, and execute). This clean division of labor encapsulates idiosyncratic abstractions in the library OS, and eliminates the need for redundant hardware management code, such as duplicate low-level page management and swapping heuristics.

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

2.1.1 Multi-Process Support in Library OSes

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

For example, consider the process identifier (PID) namespace. In current, single-process lib-

Adopted from Drawbridge		
Class	**ABIs**	**Description**
Memory	3	Allocate and protect virtual memory.
Scheduling	12	Threads and synchronization.
Files & Streams	12	Files inside a `chroot`-style; jails and byte streams among picoprocesses.
Process	2	Create a child picoprocess, and exit self.
Misc	4	Get random bits, time of day, etc.
Added by Graphene		
Class	**ABIs**	**Description**
Segments	1	Manage x86 segment registers (FS and GS) for TLS.
Exceptions	2	Handle hardware exceptions.
Streams	4	Share stream handles across picoprocesses, rename files, and set stream attributes (file permissions, socket options, etc.)
Bulk IPC	3	Exchange copy-on-write pages.
Sandboxes	1	Move into a new sandbox, with a new manifest applied. Pipes and streams that violates the new policy are closed.

Table 2.1: Classes of host ABI functions adopted from Drawbridge [Porter et al., 2011], followed by ABIs added by Graphene.

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

Design Options. Multi-process support requires extensions to the PAL ABI of recent, single-process library OS designs. Because multi-process abstractions, such as signals or System V IPC, tend to be idiosyncratic, an essential problem is identifying a minimal, host-independent substrate upon which to implement OS-specific abstractions.

We see two primary design options: (1) implement processes and scheduling in the library OS, and (2) treat each library OS instance as a process, and distribute the shared POSIX implementation across a collection of library OSes. We selected the second option, primarily because we expected this would impose fewer requirements on the host, maximize flexibility in mapping processes to physical resources, and facilitate inter-process security policy enforcement.

Implementing processes inside the library OS is also feasible using hardware MMU virtualization, similar to Dune [Belay et al., 2012], but this reintroduces a duplicate scheduler and memory management. Moreover, Intel and AMD have similar, but mutually incompatible MMU virtualization support, which would complicate live migration across platforms. None of these problems are insurmountable, and it would be interesting in future work to compare both options.

Multi-Process Support in Graphene. In Graphene, multiple library OS instances in multiple picoprocesses collaborate to implement shared abstractions, such as copy-on-write fork, signals, exit notification, and System V IPC. For instance, when process A signals process B on Graphene, A's library OS issues a remote procedure call (RPC) to B's library OS over a host-provided byte stream (similar to a Unix pipe), and B's library OS then calls the appropriate signal handler.

Graphene implements all shared abstractions in the library OS, and library OSes cooperatively manage these abstractions over RPC streams. Single-process applications still service system calls from local state, and Graphene includes optimizations to place state where it is most likely to be used, minimizing RPC overheads. The host reference monitor can easily isolate library OSes by

blocking all RPC messages, without the need to understand the library OS details or semantics of these abstractions. In our PID example, only mutually-trusting library OSes can signal each other.

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

Changes to PAL ABI. When implementing Graphene, we found that the Drawbridge ABI lacked 12 PAL calls essential to running a Linux library OS with multi-process support, and Graphene did not require 3 PAL calls to support checkpoint and resume. Of the 12 new calls, 4 are required for single-process Linux and similar ABI have also been added by **Bascule** [Baumann et al., 2013]: rename a file, manage segmentation hardware, and 2 for exception upcalls; 5 are required for stream inheritance, attribute configuration, and sandboxing; and 3 new calls are used to utilize Bulk IPC channel, for optimizing copy-on-write fork (§2.5). We design these new PAL calls in consideration of the platform independence among different hosts; If a PAL call is added for optimizations (e.g., Bulk IPC), it is optional if the implementation is difficult or infeasible on a host. For example, copy-on-write forking can still be supported without Bulk IPC supported on the host. Instead, the inheritance of process state for copy-on-write forking will be completely over regular pipes.

Comparison with microkernels. The building blocks of Graphene are very similar to the system abstractions of a microkernel [Accetta et al., 1986; Baron et al., 1985; Chen and Bershad, 1993; Elphinstone and Heiser, 2013; Klein et al., 2009; Liedtke, 1993, 1995], except a microkernel often has a even narrower, more restricted interface than the PAL ABI. Unlike a multi-server microkernel system, such as GNU Hurd [Free Software Foundation] or Mach-US [Stevenson and Julin, 1995], which implements Unix abstractions across a set of daemons that are shared by all processes in the system, Graphene implements system abstractions as a library in the application's address space, and can coordinate library state among picoprocesses to implement shared abstractions. Graphene guarantees isolation equivalent to running an application on a dedicated VM; this isolation could be implemented on a multi-server microkernel by running a dedicated set of service daemons for each application.

The Graphene host ABI could be described as a hybrid microkernel, which also exposes the file system and network of the host kernel. Similarly, we assume that picoprocesses are provided by a legacy OS kernel, like Linux or Windows, or by a Type 2 hypervisor. We expect that a bare metal hypervisor could export a PAL, but would probably require services from a trusted VM, such as Xen's dom0 [Barham et al., 2003]. Arguably, recent library OS designs might be improved by rethinking the division of labor in the network and file system stacks, but this is beyond the scope of this thesis.

Alternatives. Another approach to support multi-process applications in a library OS would be to use hardware MMU virtualization such as nested paging used by a system like Dune [Belay et al., 2012] in order to implement a second process abstraction, memory manager, and scheduler in the library OS. This approach threatens the efficiency benefits of deduplicating these features. A final option is exposing additional kernel interfaces, such as signals, by adding more system calls

to a picoprocess. This approach undermines host independence, as many of these coordination abstractions tend to be very OS-specific. This approach also harms security isolation, as the library OS now has access to generally porous host kernel interfaces.

Systems must strike a careful balance between the competing goals of security isolation and multi-process coordination. Multi-process applications require OS-managed coordination abstractions such as signals, process exit notification, and System V IPC. These coordination abstractions operate within shared namespaces, such as the process ID namespace and the System V key space. These coordination APIs and namespaces must be consistent among coordinating processes, but can undermine security isolation among unrelated processes on the same host. System designs generally only meet one goal: traditional OSes have a rich but porous coordination interface, while sandboxing systems and virtual machines (VMs) are strictly isolated. This thesis demonstrates that this unfortunate trade-off is not fundamental.

Traditional OS kernels typically provide rich multi-process coordination APIs, but this richness also makes for a very porous attack surface area. For instance, on Windows, a program may inject libraries and create threads in another program [MSDN: SetWindowsHookEx]; similarly, unchecked file descriptor inheritance in Linux can lead to security problems [Software Engineering Instititute, 2015]. Although legacy OSes do enforce some access control rules on these abstractions, kernel developers must audit and add hooks to so much code that users have lost confidence that an OS can comprehensively enforce security isolation on these abstractions.

For achieving strong security isolation on applications, users have turned to virtual machines (VMs). For instance, if two customers host their websites in the same cloud service, the customers will insist on running their web servers in separate VMs for security. VMs take a heavy-handed approach to security isolation — ensuring that every application has a dedicated OS kernel in a hardware-isolated address space. Although virtual machines isolate applications and provide legacy OS abstractions within a VM, coordinating applications must be statically placed in the same VM, and cannot dynamically move to a separate VM. For instance, consider a web service running inside of a VM that wishes to isolate requests for different users in different VMs after authentication. The web server administrator must statically create a VM for each user, introducing substantial overhead; and the developer loses convenient IPC abstractions and must rewrite large swaths of code.

2.2 Coordinating Guest OS States

An application executes on Graphene with the abstraction that all of its processes are running on a single OS. Graphene library OSes service system calls from local library OS state whenever possible, and state is coordinated across picoprocesses via RPC when necessary. Within a sandbox, Graphene picoprocesses coordinate shared state used to implement multi-process abstractions, such as process identifiers, thread groups, and System V IPC including message queues and semaphores, shared file system and file descriptor states (Table 2.2). Similar to previous designs [Baumann et al., 2013; Porter et al., 2011], Graphene uses the host file system; the library OS implements file handles and translates between POSIX and the host ABI. Identifying the best division of labor for a library OS file system is left for future work.

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

Abstraction	Shared State	Strategy
Fork	PID namespace	Batch allocations of PIDs, children generally created using local state at parent.
Signaling	PID to picoprocess map	Local signals call handler; remote signal delivery by RPC. Cache mapping of PID to picoprocess ID.
Exit notification	Remote 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, queue contents	Mappings managed by a leader, contents stored in various picoprocesses. When possible, send messages asynchronously, and migrate queues to the consumer.
Semaphores	Key mapping, count.	Mappings managed by leader, migrate ownership to picoprocess most frequently acquiring the semaphore.
File System	File truncate sizes, FIFO files, domain sockets, symbolic links, file locks	No coordination, create special files in the host to represent symbolic links and file locks.
Shared File Descriptors	Seek pointers	Mappings managed by parent, migrate ownership to picoprocess most frequently accessing the file descriptors.

Table 2.2: Multi-process abstractions implemented in Graphene, coordinated state, and implementation strategies.

conclude with lessons learned from optimizing multi-process performance.

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

2.2.1 Coordination Building Blocks

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

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

Within a sandbox, all IPC helper threads exchange messages using a combination of a **broadcast stream** for global coordination, and **point-to-point** streams for pairwise interactions, minimizing overhead for unrelated operations. The broadcast stream is created for the picoprocess as part of initialization. Unlike other byte-granularity streams, the broadcast stream sends data at the granularity of messages, to simplify the handling of concurrent writes to the stream. Point-to-point streams are simply byte streams between two picoprocesses; two processes may establish a point-to-point stream by passing handles through an intermediate stream or over the broadcast stream. The handle-passing ABI is discussed further in Section 2.5. If a picoprocess leaves a sandbox to create a new one, its broadcast stream is replaced with a new one, connected only to the picoprocess and any children created in the new sandbox.

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

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

2.2.2 Examples and Discussion

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

Figure 2.3 illustrates two sandboxes with picoprocesses collaborating to implement a process ID (PID) namespace. Because PIDs and signals are a library OS abstraction, picoprocesses in each sandbox can have overlapping PIDs, and cannot signal each other. 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 2.3, the `kill` call to `libLinux` will check its cached mapping of PIDs to point-to-point streams. If `libLinux` cannot find a mapping, it may begin by sending a query to the leader to find the owner of PID 2, and then establish a coordination stream to picoprocess 2. Once this stream is established, picoprocess 1 can send a signal RPC to picoprocess 2 (PID 2). When picoprocess 2 receives this RPC, `libLinux` will then query its local `sigaction` structure and mark `SIGUSR1` as pending. The next time picoprocess 2 makes a `libLinux` call, the `SIGUSR1` handler will be called upon return. Also in Figure 2.3, picoprocess 4 (PID 2) waits on picoprocess 3 termination (in the same sandbox with PID 1). When picoprocess 3 terminates, it invokes the library implementation of exit, which issues an `exitnotify` RPC to picoprocess 4.

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

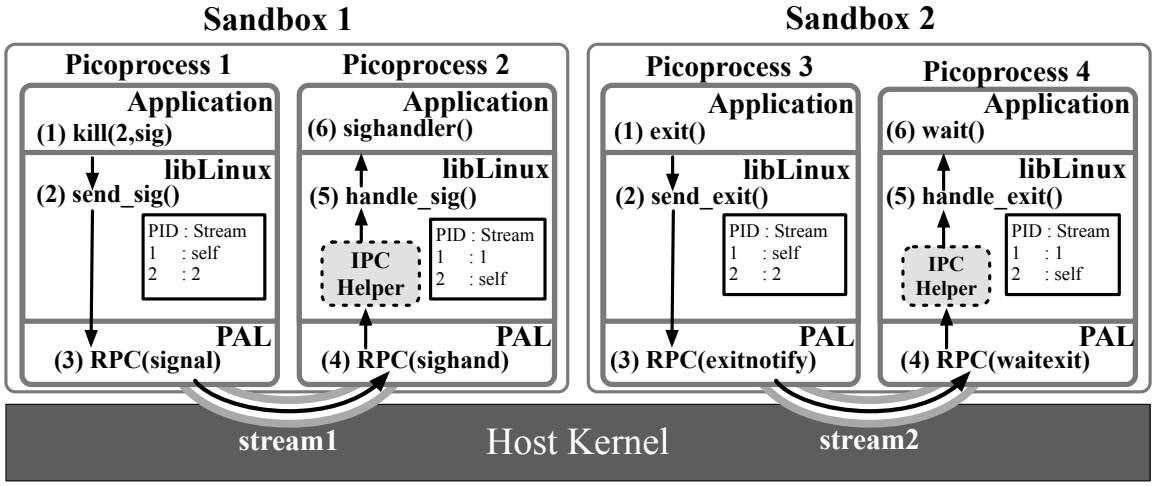


Figure 2.3: Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management. The location of each PID is tracked in `libLinux`; Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation of the `sigaction` interface. Picoprocess 4 waits on an `exitnotify` RPC from picoprocess 3 over stream 2.

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

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

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

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

Semaphores. IPC semaphores follow a similar pattern to message queues, where ownership of a given semaphore is migrated to the picoprocess that most frequently acquires the semaphore. Most

of the overhead in the Apache benchmark (§2.6.3) is attributable to semaphore overheads.

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

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

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

Failure and Disconnection Tolerance. Graphene is designed to tolerate disconnection of collaborating library OS instances, either because of crashes or blocked RPCs. In general, Graphene makes these disconnections isomorphic to a reasonable application behavior, although there may be some edge cases that cannot be made completely transparent to the application.

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

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

2.2.3 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 [Chen and Bershad, 1993; Liedtke, 1993]. We expect that Graphene performance could be improved on a microkernel with a more optimized IPC substrate, such as L4 [Elphinstone and Heiser, 2013; Klein et al., 2009; Liedtke, 1995]; we take a complementary approach of avoiding IPC if possible.

An example of this principle is migrating message queues to the "consumer" when a clear producer/consumer pattern is detected, or migrating semaphores to the most frequent requester. In these situations, synchronous RPC requests can be replaced with local function calls, improving performance substantially. For instance, migrating ownership of message queues reduced overhead for message receive by a factor of $10\times$.

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

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

The coordination within a sandbox is often pairwise. Graphene optimizes the common case of pairwise coordination, by authorizing one side of the coordination to dictate the abstraction state, but also allows more than two processes to share an abstraction. Based on this insight, we observe that *not all shared state need be replicated by all picoprocesses*. Instead, we adopt a design where one picoprocess is authoritative for a given name (e.g., a process ID or a System V queue ID). For instance, all possible thread IDs are divided among the collaborating picoprocesses, and the authoritative picoprocess either responds to RPC requests for this thread ID (e.g., a signal) or indicates that the thread does not exist. This trade does make commands like `ps` slower, but optimizes more common patterns, such as waiting for a child to exit.

Make RPCs asynchronous whenever possible. For operations that must write to state in another picoprocess, the Graphene design strives to cache enough information in the sender to eval-

uate 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.

2.3 Isolating Mutually Untrusting Applications

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

The only kernel-level sharing abstractions the reference monitor must mediate are files, network sockets, and RPC streams — all other allowed kernel ABIs modify only local picoprocess state. In order for the reference monitor to restrict file access, socket and RPC stream creation, each application includes a **manifest file** [Hunt and Larus, 2007], which describes a `chroot`-like, restricted view of the local file system (similar to Plan 9's unioned file system views [Pike et al., 1990]), as well as **iptables**-style [iptables man page] network restrictions.

When a new picoprocess is launched by the reference monitor, it begins execution in a new sandbox. Child picoprocesses may either inherit their parent's sandbox, or can be started in a separate sandbox — specified by a flag to the picoprocess creation ABI. A parent may specify a subset of its own file system view when creating a child, but may not request access to new regions of the host file system. The child may also issue an `ioctl` call to dynamically detach from the parent's sandbox. The reference monitor prevents byte stream creation across sandboxes. When a process detaches from a sandbox — effectively splitting the sandbox — the reference monitor closes any byte streams that could bridge the two sandboxes.

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

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

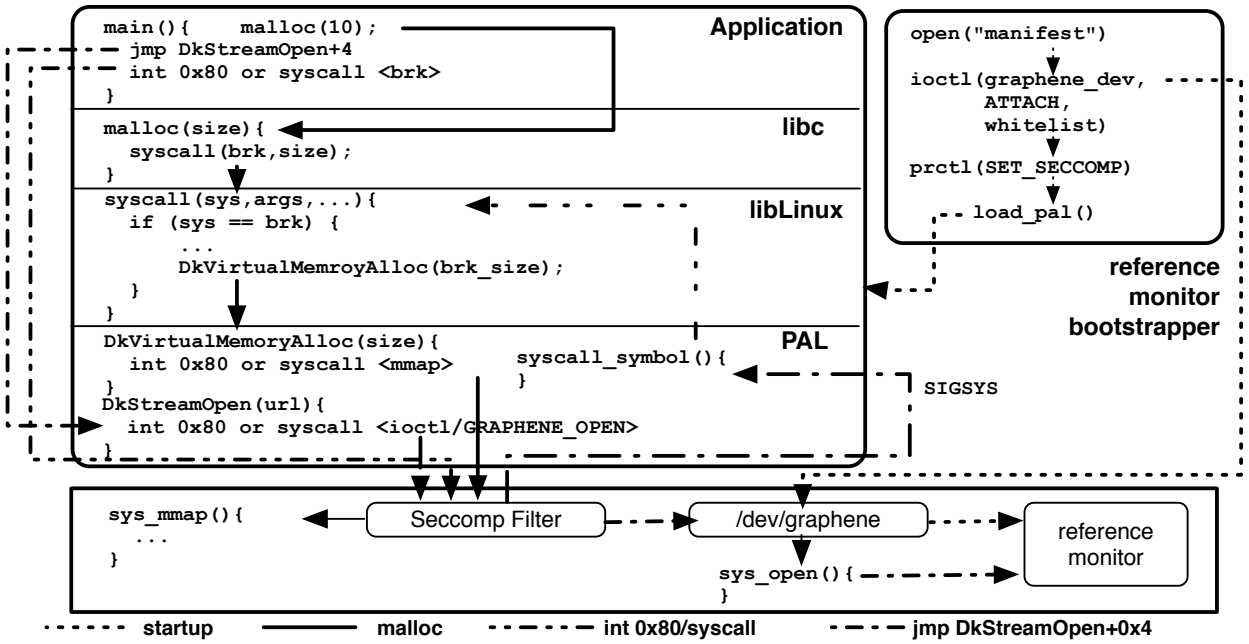


Figure 2.4: System call restriction approach. The reference monitor loads policies into the LSM at startup. A Graphene application requests OS services in three different ways. In the normal case (first line of `main`), `malloc` is invoked causing the invocation of `brk` (`libLinux`) and `mmap` in the PAL. In the second line, the application jumps to an address in PAL, which is permissible. Files are accessed through `ioctl` to `/dev/graphene` and checked by reference monitor. The third line invokes `brk` with an `int` instruction, which is redirected to the `libLinux` function.

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

2.3.1 System Call Restriction

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

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

Unfortunately, the logic to check for allowed paths network addresses cannot be implemented as a seccomp rule, because it involves reading user memory of unknown sizes. In order to avoid

the overhead of trapping to the reference monitor on every use of `open`, `stat`, `bind` or `connect` system calls, we instead force picoprocess to only use `ioctl` system call to Graphene special device (`/dev/graphene`) as alternative interface these system calls. Direct access to these system calls are banned by seccomp filter.

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

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

Example. Figure 2.4 illustrates three possible situations. An unmodified application first invokes the `libc` function `malloc`, which issues a `brk` system call to `libLinux`, which requests memory from the host via a `DkVirtualMemoryAlloc` PAL call, which ultimately issues an `mmap` host system call. The `mmap` host system call is allowed by seccomp because it only affects the picoprocess's address space. The second line of the application jumps to the PAL instruction that issues an `open` system call. From a security perspective, this is permissible, as it is isomorphic to PAL functionality. In practice, this could cause corruption of `libLinux` or application data structures, but the only harm is to the application itself. Because this system call involves the file system, the reference monitor LSM first checks if the file to be opened is included in the sandbox definition (manifest) before allowing the `open` system call in the kernel. Finally, the application uses inline assembly to issue a `brk` system call; because this system call was not issued by the PAL, seccomp will redirect this call back to the PAL, which then calls the `libLinux` implementation.

Process-specific Isolation. Sandbox creation in Graphene can provide more options than virtualization, to reflect the security policy of applications at any timing, in the granularity of picoprocess. A picoprocess can voluntarily detach itself from the current sandbox, dropping its privileges, after finishing security-sensitive operations. If a picoprocess decides one of its children is not trustworthy, it may also start the child under a restricted manifest, or promptly shut down RPC streams to stop sharing OS states. The picoprocess that moves to a separate sandbox will have a restrictive view of the filesystem, and no coordination with the previous sandboxes. In section 2.6, we describe an experiment that improves security isolation of Apache web server without sacrificing functionality.

2.3.2 Reference Monitor

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

newly created picoprocesses in the sandbox.

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

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

Creating New Sandboxes. We add a PAL call which permits a picoprocess to request that it be moved into a new sandbox. This call, as well as file system path checks, are implemented as extensions to the AppArmor LSM [AppArmor]. The new sandbox call closes any open stream handles that cross sandbox boundaries; mediate path lookups; and create a new broadcast stream for multi-process coordination (§2.2.1).

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

2.4 Isolation against Untrusted Hosts

Intel SGX (Software Guard Extension) is a set of new x86/x86_64 instructions introduced in the latest Intel Skylake processors, to create, interface and attest an isolated memory region (**enclave**) inside applications' virtual memory. SGX guarantee any data in an enclave stay encrypted in the DRAMs, using a secret key derived from the application's cryptographic measurements. The secret data store in the enclave memory can only be accessed within the code inside the enclave, and the code must be signed by the developers' private key. The use cases of SGX mostly involve an enclave retrieving a signed attestation from the Intel processor, to exchange provisioning of information from trusted remote servers. The purpose is equivalent to expanding the execution from remote servers to untrusted hosts, to securely harness resources such as CPU cycles and DRAMs.

SGX is an appealing tool for protecting small, highly sensitive execution, against malicious or compromised OSes, hypervisors, or even hardware peripheral. For instance, Hoekstra et al. [2013] show how SGX can be used to build a trusted path from a video chat application to a GPU and network card, which maintains confidentiality and integrity of the video stream, even if the OS is compromised. Similarly, because DRAM contents are encrypted, SGX can resist attacks such as cold-boot attacks [Halderman et al.] or malicious peripheral devices [Hudson and Rudolph, 2015].

A Partitioned Usage Model. To secure an application with SGX, the common usage model requires the developers to partition the application into two parts: the trusted part which runs the sensitive execution in the enclave, and the untrusted part which loads and triggers the execution of the trusted parts. The interaction between the trusted and untrusted parts is through **untrusted**

interfaces. Untrusted interfaces are defined by the developers, and include a few logical entry and exit points of the enclave. In the architectural level, an enclave only has exactly one hardware entry point and one exit point. The architecture forbids the untrusted part to willingly jump into any location of an enclave, to bypass or alter the valid behavior of the trusted part. The control flow of the execution is the enclave transfers to one of the logical entry points, based on the input parameter passed at the hardware entry point.

With the partitioned usage model, the developers can minimize the trusted computing base (TCB) of the enclave, at their best effort, and restraint the behavior of the isolated execution. The partitioning prevents unpredictable behaviors in the enclave, to compromise the confidentiality or integrity. The developers explicitly mark the sites in the application code to determine entry and exit of the enclave.

Using A Library OS. The partitioned usage model restricts the risk of the isolated execution being compromised by the untrusted host, yet requires the developers to cleanly split a part of the application and inject entry and exit points at wherever interaction is needed. Moreover, the developers are responsible for validating and examining the soundness of the partitioned application, to ensure every execution paths in the enclave that are affected by the untrusted entities are carefully checked and sanitized. Porting a legacy application to a partitioned model requires developers to have substantial familiarity to the application code, as well as the principles for developing unexploitable, well-controlled execution in the enclave. When the sensitive execution to be isolated in the enclave becomes considerably complex, the effort for porting and validating the execution quickly becomes unacceptably expensive.

Haven [Baumann et al., 2014] shows a **non-partitioned usage model**, for loading whole, unmodified applications and supporting libraries into enclaves. The non-partitioned model allows application to be run in the enclave as it is, without any porting effort. To support the non-partitioned model, a key requirement is to support the system APIs needed by the application and supporting libraries, right inside the enclave. Essentially, the execution not only is forbidden to access any host system APIs (i.e., using `syscall` or `sysenter` instructions) in the enclave, but also unsafe to rely on the untrusted host to provide system APIs. A library OS, on the other hand, supports system APIs as in-memory function calls, and mediates all input and output during interaction with the untrusted host. For instance, Haven supports unmodified Windows applications in enclaves, using Drawbridge library OS [Porter et al., 2011] to handle all the Windows API calls.

At the low level, the non-partitioned model still requires partitioning, but instead of the applications, it is the library OS itself that needs to be partitioned. More precisely, The PAL that provides host ABI to the library OS is the component that is split across the two sides of the enclave boundary, while the library OS, alone with the isolated application and its supporting libraries, are dynamically loaded. The partitioned PAL interacts with the host through a fixed, narrow untrusted interface, to access host resources if necessary.

2.4.1 Porting Graphene to Intel SGX

The building blocks are shown as figure 2.5. Graphene-SGX is partitioned into two parts: an untrusted PAL to initialize the enclave and provide the untrusted interface; and the trusted library OS including the host ABI, `libLinux`, and core `libc` libraries. Except the host ABI is statically loaded in the enclave since the start up, other components of Graphene-SGX in the enclave are

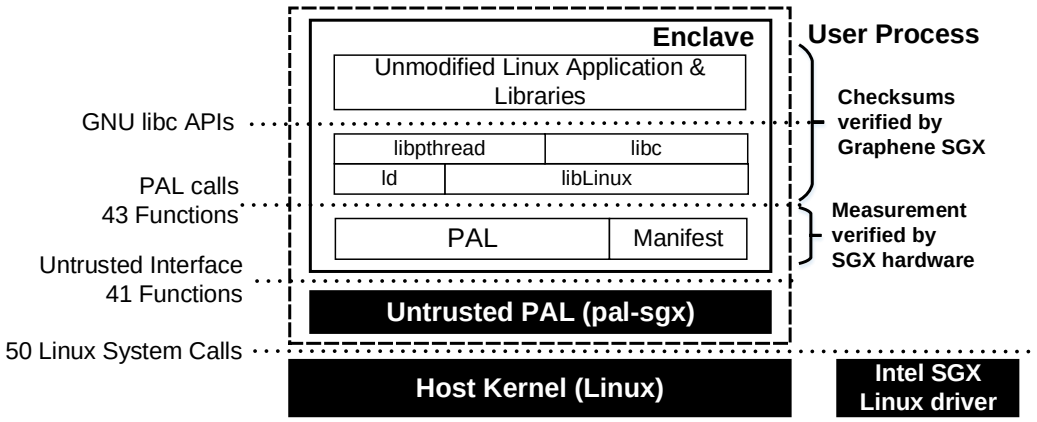


Figure 2.5: The Graphene-SGX building blocks. Green (light) boxes are trusted components trusted, and red (dark) boxes are untrusted. Graphene-SGX is statically linked with `libLinux`, the library OS binary, and basic libraries of GNU library C. To build up the trust, the Graphene-SGX image along with a manifest and application measurements are verified by SGX during bootstrap. Applications and other libraries are verified by Graphene-SGX. The enclave interacts with the host kernel, through untrusted PAL, on a narrowed untrusted interfaces with 41 functions.

dynamically loaded, as needed. After the library OS is laoded in the enclave, the library OS will load the application and its supporting libraries, and call their entry functions. The library OS will handle all the system calls redirected from the applications, whereas it may access host resource through the host ABI calls, which are redirected to the untrusted interface to interact with the host.

Applications Integrity. A key requirement of the non-partitioned usage model is to guarantee the integrity of the secured applications. Because the applications are dynamically loaded by the library OS *after* the enclave starts up, the CPUs cannot verify the integrity of loaded application binaries. An alternative is to place all application binaries in an encrypted virtual disk, which can be decrypted using a securely provisioned key [Baumann et al., 2014].

Graphene-SGX guarantees the integrity of applications by signing each supporting binaries or generating checksums for the binaries. For each secured application, the signing tool will generate a manifest and a correspondent signature. A manifest will contain the attributes of the enclave and library OS, and the checksums of the supporting binaries. The manifest will be loaded in the enclave before the enclave starts, and verified by the hardware as part of the enclave measurement. The integrity of the supporting binaries are validated by PAL before loading from the untrusted host.

By verify the integrity of applications individually, we decouple the deployment and verification of the application binaries; users may simply deploy the manifest and the signature to the untrusted host, and all the binaries, including the library OS, can be loaded from the local file system.

Untrusted Interface. The untrusted interface is defined as Table 2.3. Graphene-SGX draws the untrusted interface right above the calling of the host APIs (Linux system calls) in PAL. Most code inherited from the Graphene Linux PAL stays in the enclave, to keep the states secure. Because the host is not trusted, Graphene-SGX is built upon the assumption that the untrusted interface can be exploited to pass malicious arguments, or return incorrect results. For example, `file_open` may

Classes	#	Untrusted Interface Functions
Entries / Exits	3	`start_enclave exit_enclave start_thread`
Memory	2	`map_untrusted unmap_untrusted`
Scheduling	4	`sleep schedule futex gettime`
Cloning	2	`clone_thread clone_process`
Files	8	`file_open file_mkdir file_rename file_delete file_truncate file_write file_read file_readdir`
Sockets	8	`sock_listen sock_accept sock_connect socketpair sock_send sock_receive sock_shutdown sock_setopt`
File Descriptors	3	`fd_close fd_size fds_poll`
Misc	3	`print_debug load_gdb cpuid`

Table 2.3: The Graphene-SGX untrusted interface, which consists of 41 functions in total. Most of the interface is derived from the host system call footprint of Graphene library OS. Enclave must not trust the hosts to always return right responses or faithfully perform operations.

return a file descriptor that points to a wrong file, or `map_untrusted` may return an address in the enclave. Beside checking for malicious returned results, Graphene-SGX must actively check the behavior of the host.

For instance, if the isolated application opens a stream for IO, Graphene-SGX must guarantee either the stream is protected cryptographically, or assume that nothing is safely read or written. In addition, Graphene-SGX does not include `file_sync` in the untrusted interface, because it does not trust the host to faithfully flush any stream. With a robustly designed untrusted interface, the worst scenario a malicious host can cause is **denial-of-the-service**.

Signing and Launching Applications. A key requirement of supporting the non-partitioned model is to secure the process of dynamic loading. In Graphene-SGX, application binaries are verified by their checksums. The enclave manifest lists all the checksums, and is included in the cryptographic measurement when the enclave starts. The measurement for launching the enclave is distinct for different applications.

When developers port an application to Graphene-SGX, the enclave signature are generated by the signer, on a trusted host. The signing process essentially creates a snapshot of the trusted execution environment, by hashing all the binaries to be loaded. After the signing, developers ship the application, along side with the enclave signature, manifest, Graphene-SGX binaries and all the supporting libraries, to the untrusted hos, on which Graphene-SGX loads the application in the enclave.

Threat model. The host ABI, library OS, `libc` and all supporting libraries and binaries that loaded into the enclave must be considered as part of the TCB. The rest of system, including the untrusted PAL, the host, any hardware peripheral or remote entity that are not attested can be malicious or compromised to launch arbitrary attack on the isolated application. For multi-process applications, each picoprocess created by Graphene-SGX will be isolated in its own enclave. No mutual trust is required between the picoprocesses, unless the **manifest file** allows sharing the multi-process abstractions.

Side-Channel Attacks. We consider side-channel attacks as a potential threat in Graphene-SGX. Attackers may exploit timing channels to expose confidential information in the enclave, and cur-

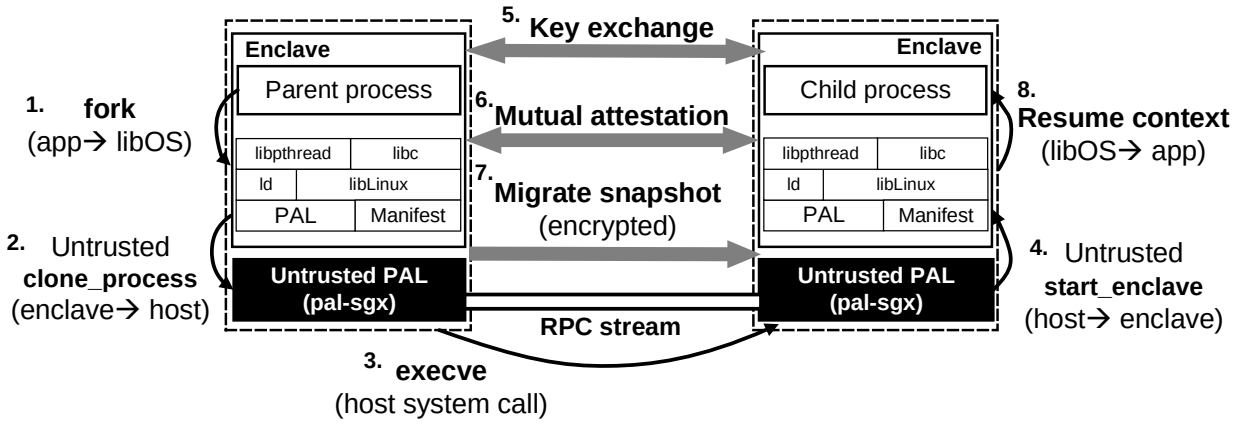


Figure 2.6: Process creation in Graphene. Numbers show the order of operations. When a process forks, Graphene calls `execve` system call on the untrusted host, to create a clean enclave with the same library OS image. Then the two enclaves build up the mutual trust by exchanging a session key, verifying attestation of each other, and migrating the process snapshot from the parent.

rent SGX technology has no defense against these attacks. Xu et al. [2015] show a stronger attack than regular side-channel attacks, as **controlled channel attacks**. This type of attack exploits the fact that an enclave must cooperate with the host OS for page management, the OS can manipulate paging to amplify the side channels for exposing more secrets. Both side-channel attacks and Controlled channel attacks are common problems in all solutions based on SGX, including Haven. In this paper, we consider solving these attacks as out of the scope to our work.

2.4.2 Securing Multiple Processes in Multiple Enclaves

Upon existing platforms using SGX, there is no multi-process abstractions of any kinds that has been supported so far. The main challenge for multi-process support in enclaves is that enclaves cannot share pages, to support either Linux-style copy-on-write forking or sharing abstraction states across processes. Graphene implements multi-process support in zero-sharing fashion. The multi-process abstraction supported by Graphene thus also Graphene includes `fork`, `execve`, signals, System V IPC, etc.

In this section we will describe how Graphene securely creates processes in new enclaves, for supporting `fork` and `execve`, and implements inter-process communication (namespace coordination, signals, System V IPC, etc) with process isolation.

To securely create process across enclaves, Graphene builds trusted paths for enclaves to send process snapshots and coordinate shared states. Since the enclaves do not directly share pages, the coordination must be intermediated by the untrusted host. The untrusted host may launch attacks on the multi-process abstractions of Graphene, by either counterfeiting the behavior of either the forking or forked process, or acting as man-in-the-middle between two processes.

When a isolated process forks, Graphene requests the untrusted host to create a clean enclave of the forking process. The parent and child process in their own enclaves will coordinate over an encrypted RPC streams, with the session key exchange at the beginning of coordination. After securing the RPC streams, the participating processes exchange attestation, ensuring both sides are restricted to running the same application. Once the mutual trust is built between the two processes,

further inter-process coordination is secured. The design of process creation in Graphene is shown as Figure 2.6.

To defend against the man-in-the-middle attack, we take advantage of a user-customized 512-bit field in the attestation structure generated and signed by the SGX hardware. This field is filled with a SHA-256 hash value of the agreed session key, and the current enclave ID, to prove that the attested party is the one who agrees on the key.

Implementing `execve`. Unlike `fork`, `execve` starts a process with a specified binary, often different from the callers, in a clean process state. A key challenge for implementing `execve` is to identify trusted binaries that can be loaded into new enclaves as child processes. Although a child process created by `execve` does not start with a snapshot of its parent, The processes share states through inheriting file handles, or future inter-process coordination. The communication between the processes must be considered internal interaction of the application, thus shall be protected from the host.

As securing `fork`, `execve` is secured by the attestation of the child processes' measurements. Graphene only allows a isolated process to create a child process in an enclave whose measurement is listed in its manifest. In addition, a child process does not verify the measurement of its parent, but instead presents the measurements of all ancestors to remote trusted server when attesting its own integrity. In other word, for the same binary `ls` isolated in the enclave, the measurement reflects the distinction if the binary is called by, say a shell or Apache server.

2.5 Implementation Details in Graphene

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

Alternative PAL Ports. We prove the platform independence of Graphene by porting PAL to **FreeBSD**, **OSX** and **Windows**. With the alternative host PAL, unmodified Linux binaries, along with `glibc` and libLinux, can be transparently run on the host. For FreeBSD, only 1.2 kLoC of the host PAL code need to be rewritten, which are significantly less than FreeBSD Linux compatibility module (10.8 kLoC). PAL components including ELF loader and internal support code can be shared by any PAL ports.

Implementing Linux Personality. The Graphene `libLinux.so` implements a subset of the Linux system call API (currently 131 calls) using only the PAL ABI to interact with the host. We note that Linux exports a very long tail of infrequently used calls. A rough analysis of this tail indicates roughly 100 additional calls that can be implemented with the existing PAL ABI and coordination framework, less than 10 administrative calls that will not make sense to expose to an application, such as loading a kernel module or rebooting the system, and roughly 54 that will require PAL extensions to meaningfully implement, such as controlling scheduling, NUMA placement, I/O privilege, and shared memory. In the last category of system calls, the degree to which

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

Table 2.4: Lines of code written or changed to produce Graphene. Applications and other libraries are unchanged.

actual host details should be exported versus emulated is debatable.

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

In order to use `libLinux.so`, we modified 606 lines of `glibc` to replace system instructions with function calls into `libLinux.so`, and to cooperatively manage thread-local storage with `libLinux.so` (Table 2.4).

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

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

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

Our IPC module is 1,131 lines of code (Table 2.4), runs on multiple versions of Linux (2.6 and 3 series kernels), and does not require Linux kernel changes or recompilation.

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

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

Table 2.5: Startup, checkpoint, and resume times for a native Linux process, a KVM virtual machine, and a Graphene picoprocess, where appropriate. Lower is better. Overheads are relative to Linux.

Synchronization. Perhaps surprisingly, implementing Linux synchronization appears substantially easier than Windows synchronization, as `libLinux` did not require all of the various synchronization ABIs provided by Drawbridge. We believe the reason for this is that Linux has consolidated all user-level synchronization primitives to use futexes [Franke et al., 2002], which are essentially kernel-level wait queues.

2.6 Evaluation of Graphene

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

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

Except for scalability measurements, all measurements were collected on a Dell Optiplex 790 with a 4-core 3.40 GHz Intel Core i7 CPU, 4 GB RAM, and a 250GB, 7200 RPM ATA disk. Our host system runs Ubuntu 12.04 server with host Linux kernel version 3.5, which includes KVM on QEMU version 1.0. Each KVM Guest is deployed with 4 virtual CPU with EPT, 2GB RAM, a 30GB virtual disk image, Virtio enabled for network and disk, bridged connection with TAP, and runs the same Ubuntu and Linux kernel image. We note that recent library OSes are either not openly available or cannot execute unmodified Linux binaries. Unless otherwise noted, Graphene measurements include the reference monitor.

2.6.1 Process Migration and Application Startup

Graphene supports migration of an application from a picoprocess on one machine to a picoprocess on another machine by checkpointing the application, copying the checkpoint over the network, and then resuming the checkpoint. Table 2.5 shows the time to start up a process, VM, or pico-

process; as well as the checkpoint and resume time for KVM and Graphene. Migration across machines is a function of network bandwidth, so we report checkpoint size instead.

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

2.6.2 Memory Footprint

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

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

Unixbench on Graphene uses substantially more memory at a given time than native Linux—more than double. In these samples, however, Graphene also had 3–4× as many processes running; this is because Unixbench simply spawns all of the tasks in the background, rather than executing them sequentially. Because Graphene processes execute more slowly (attributable to a slower `fork`—§2.6.3), a given sample will include more picoprocesses, pushing total memory usage higher. Thus, we expect that further tuning `fork` performance will lower sampled memory usage.

Across all workloads, Graphene's memory footprint is 3–20× smaller than KVM. For all tests, we used a minimal KVM disk image, generated using debootstrap 1.0.39ubuntu0.3 and supplemented only by packages required to obtain, compile, and run the experiments. In order to make memory footprint measurements as fair as possible to KVM, we used both the virtio balloon driver and kernel same page merging (KSM) [Archangeli et al., 2009]. We also reduced the RAM allocated to the VM to the smallest size without harming performance—128MB. We note that memory measured includes memory used by QEMU for VM device emulation, adding a few dozen MB.

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

2.6.3 Application performance

Table 2.6 lists execution time of our *unmodified* application benchmarks (detailed in §2.6.2). All applications create multiple processes, except for lighttpd, which only creates multiple threads. Each data point is the average of at least six runs, and 95% confidence intervals are listed in the table.

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

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

	Execution time (s), +/- Conf. Interval, % Overhead										
gcc/make	**Linux**		**KVM**			**Graphene + RM**			**Graphene-SGX**		
bzip2	2.57	.00	2.70	.00	5 %	2.70	.00	5 %	xx.xx	.xx	xxx %
bzip2 -j4	1.00	.00	1.09	.00	9 %	1.08	.02	8 %	xx.xx	.xx	xxx %
libLinux	7.23	.00	7.55	.00	4 %	8.64	.00	20 %	xx.xx	.xx	xxx %
libLinux -j4	1.95	.00	2.03	.00	4 %	2.54	.00	30 %	xx.xx	.xx	xxx %
gcc	24.74	.02	26.80	.02	8 %	31.84	.00	29 %	xx.xx	.xx	xxx %
Ap. Bnch	Avg Throughput (MB/s), +/- Conf. Interval, % Overhead										
Apache	**Linux**		**KVM**			**Graphene + RM**			**Graphene-SGX**		
25 conc	5.73	.25	4.84	.03	18 %	4.02	.00	43 %	xx.xx	.xx	xx %
50 conc	5.57	.28	4.80	.06	16 %	4.01	.00	39 %	xx.xx	.xx	xx %
100 conc	5.87	.20	4.80	.03	22 %	3.98	.00	47 %	xx.xx	.xx	xx %
lighttpd	**Linux**		**KVM**			**Graphene + RM**			**Graphene-SGX**		
25 conc	6.66	.01	6.46	.03	3 %	5.65	.00	18 %	xx.xx	.xx	xx %
50 conc	6.65	.13	6.41	.02	4 %	4.79	.00	39 %	xx.xx	.xx	xx %
100 conc	6.69	.01	6.39	.03	5 %	4.56	.01	47 %	xx.xx	.xx	xx %
	Execution Time (s), +/- Conf. Interval, % Overhead										
bash	**Linux**		**KVM**			**Graphene + RM**			**Graphene-SGX**		
Unix utils	0.87	.00	1.10	.01	26 %	2.01	.00	134 %	xx.xx	.xx	xxx %
Unixbench	0.55	.00	0.55	.00	0 %	1.49	.00	192 %	xx.xx	.xx	xxx %

Table 2.6: Application benchmark execution times in a native Linux process, a process inside a KVM virtual machine, and a Graphene picoprocess with reference monitoring (+RM).

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

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

2.6.4 Micro-benchmarks

In order to understand the overheads of individual system calls, Table 2.7 lists a representative sample of tests from the LMbench suite, version 2.5 [McVoy and Staelin, 1996]. Each row reports a mean and 95% confidence interval; we use the default number of iterations for each test case. To measure the marginal cost of the reference monitor, we report numbers with and without the reference monitor.

Test	Linux		Graphene			Graphene + RM			Graphene-SGX		
	μS	+/-	μS	+/-	%O	μS	+/-	%O	μS	+/-	%O
syscall	0.04	.0	0.01	.0	-75	0.01	.0	-75	xx.xx	.xx	xxx
read	0.09	.0	0.12	.0	33	0.12	.0	33	xx.xx	.xx	xxx
write	0.11	.0	0.11	.0	0	0.11	.0	0	xx.xx	.xx	xxx
open/close	0.85	.1	3.53	.2	315	5.09	.0	499	xx.xx	.xx	xxx
select tcp	10.87	.0	17.02	.0	56	17.44	.0	60	xx.xx	.xx	xxx
sig install	0.11	.0	0.20	.0	82	0.20	.0	82	xx.xx	.xx	xxx
sigusr1	0.79	.0	0.33	.0	-58	0.33	.0	-58	xx.xx	.xx	xxx
AF_UNIX	4.71	.1	5.71	.0	19	6.37	.1	32	xx.xx	.xx	xxx
fork+exit	67	0	463	4	587	490	3	626	xx.xx	.xx	xxx
fork+exec	231	1	764	5	237	800	6	253	xx.xx	.xx	xxx
fork+sh	576	8	1,720	10	199	1,775	11	208	xx.xx	.xx	xxx

Table 2.7: LMbench comparison among native Linux processes and Graphene picoprocesses, both without and with the reference monitor (+RM). Execution time is in microseconds, and lower is better. Overheads are relative to Linux; negative overheads indicate improved performance.

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

The most expensive system calls occur when `libLinux` inadvertently duplicates work with the host kernel. For instance, many of the file path and handle management calls duplicate some of the effort of the host file system, leading to a 1–3$\times$ slower implementation than native. As the worst example, `fork+exit` is 5.9$\times$ slower than Linux. Profiling indicates that about one sixth of this overhead is in process creation, which takes additional work to create a clean picoprocess on Linux; we expect this overhead could be reduced with a kernel-level implementation of the process creation ABI, rather than emulating this behavior on `clone`. Another half of the overhead comes from the `libLinux` checkpointing code (commensurate with the data in Table 2.7), which includes a substantial amount of serialization effort which might be reduced by checkpointing the data structures in place. A more competitive `fork` will require host support and additional `libLinux` tuning.

We also measure the overhead of isolating a Graphene picoprocess inside the 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 an environment similar to FreeBSD jails [Stokely and Lee, 2003] would make all reference monitoring overheads negligible.

System V IPC. Table 2.8 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 elim-

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

Table 2.8: 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.

inate locking for the readers [McKenney, 2004]. 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 §2.2.3 reduced overheads compared to a naive implementation by a factor of ten. The optimizations of asynchronous sending and migrating ownership of queues when a producer/consumer pattern were detected were particularly helpful.

2.6.5 Security Study

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

Isolation Experiments. This subsection tests whether Graphene meets its goal of providing equivalent isolation to a VM. We conducted an evaluation of the security of the isolation mechanisms in Graphene and analyzed whether a malicious Graphene picoprocess could (i) fork a non-Graphene process, (ii) kill processes from another sandbox or a non Graphene process, (iii) access files not prescribed in its Manifest, and (iv) discover secrets from picoprocesses in other sandboxes or from non-Graphene process through side-channels via the `/proc` file system. The first three attacks use inline assembly to directly issue a system call, and are blocked by the reference monitor; the fourth creates an attack similar to Memento [Jana and Shmatikov, 2012] and is frustrated by the fact that `/proc` is implemented within `libLinux` and the system `/proc` is inaccessible from Graphene.

Analysis of Linux Vulnerabilities. Graphene restricts picoprocesses to 15% of the system call table. To evaluate the impact on system security, we *manually* analyzed all reported Linux vulner-

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

Table 2.9: Manual analysis of Linux vulnerabilities from 2011-2013 and Graphene's prevention.

abilities from 2011–2013 (a total of 291 vulnerabilities) [lin, 2013]. We categorized these exploits by kernel component, listed in Table 2.9. Roughly half of these vulnerabilities required a system call which Graphene blocks in order to exploit the system. Graphene would only allow 5 of the relevant vulnerabilities through its system call filtering and reference monitor. The remaining half of the vulnerabilities were entirely within the kernel or modules, such as bugs in the virtual memory subsystem.

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

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

2.7 Summary

Enabling legacy applications in a restricted environment, such as picoprocesses or enclaves, requires extra effort for mitigating the limitations of platforms, in order to support typical OS personalities. **Graphene**, as described in this chapter, extends the existing library OS designs from isolating single-process or unshared abstractions to include multi-process APIs required by many UNIX applications, such as servers or shell scripts. The challenge that Graphene primarily overcomes is the requirement for coordinating shared states across multiple picoprocesses, to provides a collaborative, unified OS view. Essentially, Graphene implements all shared, multi-process abstractions and OS states based on coordination over host-provided, pipe-like RPC streams. The RPC-based, distributed OS implementation enables multi-process support in Graphene, with minimal extension to the host interface, and a sweet-spot for enforcing inter-application security isolation, by simply sandboxing the RPC streams. Such a model largely reduces the complexity of enforcing security isolation on idiosyncratic multi-process abstractions and shared states. Because the corporative nature of picoprocesses in Graphene, an application can even dynamically impose sandboxing on one of its processes, to reflect per-process, variable security policies.

In principle, we attempt to use Graphene to justify the platform independence of the library OS design, without sacrificing its qualitative benefits, such as isolating mutually-untrusting applications and a narrow attack surface to kernels. Graphene implements a considerable number of common Linux system calls, to support popular, modern applications such as Apache web server, GNU Make, OpenJDK Java VM and the Python runtime. Graphene translates the high-level system APIs used by applications to a host ABI inherited and extended from a previous Windows-compatible library OS [Porter et al., 2011]. In addition, we port the PAL (Platform Adaption Layer) of Graphene to various platforms, including FreeBSD, OSX, Windows, and even a more restricted environment, the Intel SGX enclaves. In particular, Graphene being ported to Intel SGX (**Graphene-SGX**) can isolate applications — either single-process or multi-process — on a host where neither the operating system nor the hardware (except the CPU package) is trusted by the applications. Overall, Graphene shows that, by simply porting the reasonably sized host ABI to a new platform, a whole large spectrum of legacy applications tested on the previous platforms can be activated all together.

Chapter 3

Partitioning Legacy Java Applications (Work-in-Progress)

An increasing fraction of applications handle sensitive data (e.g., medical applications) or implement proprietary algorithms (e.g., algorithmic stock trading). These applications are composed of libraries and other code from third parties, and may run on hardware and hypervisors provided by an untrusted cloud provider. **Application partitioning** is a technique to bound the static or dynamic analysis effort to protect sensitive data and algorithms, while still taking advantage of inexpensive code and cloud hardware. A partitioned application isolates sensitive data and code, typically using language or hardware techniques.

At the hardware level, new CPUs are offering features to protect application-level code and data from a compromised or malicious system software stack, including the OS or hypervisor, or simply isolating portions of the application address space from the rest of the application. Examples include Intel SGX [McKeen et al., 2013], IBM SecureBlue++ [Boivie and Williams, 2013], and ARM TrustZone [TrustZone]. SGX offers entry points to an encrypted region of memory, called an **enclave**. Encryption and remote attestation prevent the hypervisor or OS from reading or modifying the enclave's contents, and validate that the enclave was correctly initialized. This in turn can exclude a cloud provider's hypervisor or OS from the application's trusted computing base (TCB), requiring only trust in the hardware manufacturer (i.e., Intel). SGX also offers protection against malicious devices off the CPU package, such as a compromised storage device.

Although SGX can restrict enclave entry to a few programmer-defined entry points, the programmer must still reason about what those entry points should be, protect against malicious inputs, and verify that code paths within the enclave that cannot inadvertently leak sensitive data. Iago attacks [Checkoway and Shacham, 2013], where an OS offers malicious system call return values, are notoriously difficult to defend against; although limited countermeasures exist for the OS API (typically with the cooperation of a trusted hypervisor) [Kwon et al., 2016],there are no off-the-shelf defenses against Iago-style attacks from libraries or compromised application code. Further, SGX is designed to run a small, native code library inside of a larger application; support for running managed languages in an enclave is currently limited. Writing a security-sensitive application in an unsafe language like C dramatically increases the risk of exploitable pointer or control flow errors [Bletsch et al., 2011; Checkoway et al., 2010; Nergal, 2001], which ultimately disclose sensitive data or undermine the integrity of the enclave.

Language-level techniques can offer more sophisticated insights into how to partition the application, as well as stronger protections [Bittau et al., 2008; Brumley and Song, 2004; Khatiwala et al., 2006]. Similarly, managed languages such as Java, with type-checking, can prevent mem-

ory corruption attacks, while applications developed in C or C++ are often prone to memory-bug exploitation. Because Java is memory safe, it is immune to known control flow attacks, such as return-oriented programming, where control flow is manipulated by unsafe writes to return pointers on the stack or function pointers in objects.

Partitioned application developers need both hardware and language-level protections, but these are difficult to combine in practice. This paper focuses on SGX and the Java language as a running example: SGX enclaves are designed to run native binaries, primarily developed in C or C++, so developers must employ considerable effort to port a Java application or a Java VM to SGX. To our knowledge, no previous effort has even run a JVM in an enclave, much less partitioned part of a Java application into an enclave. For Java language, there is no API support for SGX enclaves, making it challenging to leverage isolation features of SGX in Java applications. The current state-of-the-art is to develop an ad hoc solution for a specific use case. For example, VC3 uses SGX to protect mappers and reducers in Hadoop framework [Schuster et al., 2015]. Although Hadoop is written in Java, the protected code in VC3 cannot be written in Java.

We present **Civet**, a system that combines the benefits of SGX enclave isolation with Java language security features to protect security-sensitive applications. Civet avoids the downsides of taking either approach in isolation. Civet includes three primary parts: a tool that guides the programmer in partitioning an application, ultimately generating an enclave image that includes a minimal Java code base; a runtime framework that loads and verifies classes, and enforces information flow control across the enclave boundary; and a Java API to seamless access to SGX features such as attestation and secure provisioning. Civet is designed to protect the confidentiality and integrity of real-world application code. Civet not only requires very little developer effort to adopt, but also provides the developer with essential insights into how data can flow through the application, leading to better reasoning about the isolation properties of the partition.

Civet addresses design challenges at both the hardware and language level. To extend hardware protection, Civet contributes techniques for dynamically loading Java classes in enclaves with the same code integrity verification as native, static binaries. Civet uses the Graphene library OS [Tsai et al., 2014] to facilitate running a restricted OpenJDK 7 runtime in an enclave. Civet enforces information flow policies at the border of the enclave by incorporating Phosphor [Bell and Kaiser, 2014] instrumentation on all classes in the enclave, tracing both explicit and implicit flows from any confidential data, and preventing tainted information from leaving the enclave by any interface, except via explicit declassification by the developer.

This work presents three use cases for Civet. First we partition and isolate the cryptography library and keys from the rest of an SSH client and server. Second, we present a Hadoop workload with a sensitive algorithm; the Mapper, Combiner and Reducer classes are provisioned from a trusted server, protecting confidentiality of the sensitive code. Third, we present a data manager web application—a building block for medical and other applications that handle sensitive data; we protect the secret client data in an enclave on the client's machine. For each of the use cases, we show that developers or users can utilize Civet to partition the applications with minimal effort and only an understanding of the isolation properties of SGX, not the cumbersome, hardware-level details.

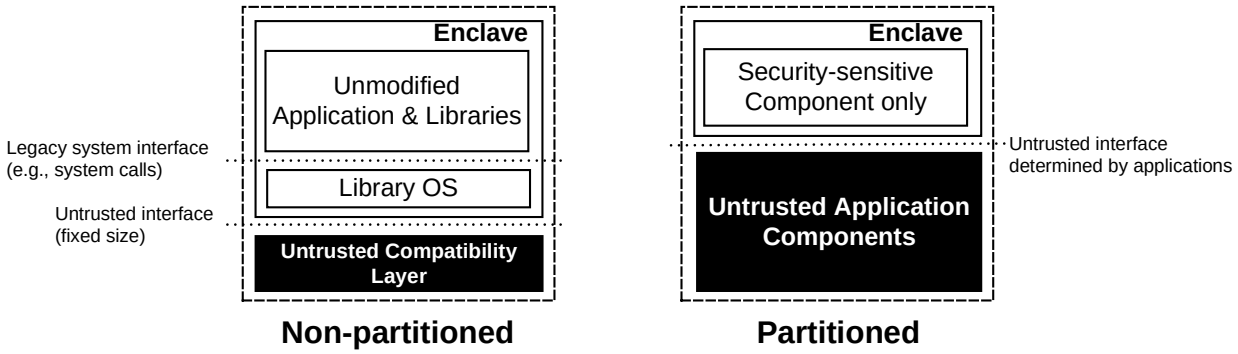


Figure 3.1: Comparison between the Non-partitioned model (e.g., Haven) and partitioned model for protecting applications in enclaves. Green (light) boxes are trusted components and red (dark) boxes are untrusted. The non-partitioned model yields a larger TCB in the enclave, while the partitioned model requires developers to determine the untrusted interface at the enclave boundary.

3.1 Partitioning Java Applications

One model for using SGX is to run an entire application in the enclave. This is exemplified by Haven [Baumann et al., 2014], which runs a Windows application and all supporting libraries on top of a library OS inside of the enclave. This approach is illustrated on the left side of Figure 3.1. The non-partitioned model does not require any application changes, but, in the case of Haven, bloats the TCB by 5.5 billion lines of code. By pulling millions of lines of extraneous code into an enclave, there is a significantly increased risk of vulnerabilities that disclose sensitive data, such as Heartbleed [Heartbleed, 2013]. The non-partitioned model does offer simple deployment and can provide practical benefits, such as protecting an application from an untrusted cloud hypervisor.

We focus on a different usage model for enclaves, where an application is partitioned into the untrusted and trusted sides (right side of Figure 3.1). Only sensitive data and computation are placed inside the enclaves. This model requires the developers to identify what in the application should be protected; harden an interface between trusted and untrusted components; and reason about potential information flows at the enclave boundary [Kilpatrick, 2003]. This effort can be non-trivial and subtle, but for application developers motivated by interests such as regulatory compliance or competitive advantage in business, the additional effort can yield a much smaller trusted computing base (TCB), and thus a reduced attack surface. A key goal of Civet is to minimize the developer's effort to partition an application—both in lines of code changed, and in leveraging language analysis to reason about narrow points in the application's data and control flow.

3.1.1 Challenges in Partitioning Java Applications on SGX

Using a higher-level language can be useful to reduce the risk of semantic errors in an application, yet there are several fundamental, technical challenges to using a managed language, like Java, in an SGX enclave. In part, this is simply an artifact of the SGX design, which is designed for native libraries. To our knowledge, no previous work has successfully executed a JVM inside an enclave.

Challenge 1: Cleanly partitioning classes and objects (§3.3.1). Java encapsulates the placement of object data and code within virtual memory, which facilitates features like inheritance

and garbage collection, but complicates integration with SGX. To program for an SGX enclave, the developer must understand which regions of virtual memory are in the enclave, and which are outside of the enclave. Note that SGX allows code in the enclave to access memory outside of the enclave. Thus, it is easy for a developer to inadvertently write code that discloses a secret, say by using a library that memoizes intermediate results to the untrusted heap. Further, when a Java class inside of an enclave inherits methods or fields from a parent class that is placed outside of the enclave, it is easy to inadvertently pass sensitive inputs to a function outside of the enclave, or update a class field outside of the enclave.

In order for programmers to be able to sensibly program for SGX, they need a model of how objects are placed inside and outside of the enclave at runtime, and as well as a model for if and when updates to objects are propagated across the enclave boundary.

Challenge 2: Dynamically loading byte-code with integrity (§3.3.2). SGX establishes trust by verifying the integrity of the code in the enclave. Therefore, enclaves must be bootstrapped with static, native binaries. However, Java classes are shipped as byte-code, and dynamically loaded by Java VMs. The subtle security challenge for a Java developer is ensuring that, when a class is dynamically loaded into an enclave, by passing a string name of the class, the application needs to be able to verify that this is a trusted class file and that the initial state of the class is correct. Thus, we introduce techniques to manage trust and verify the integrity of dynamically loaded code, including encrypted class files.

Challenge 3: Seamless access of in-enclave objects (§3.3.3). Java code outside of the enclave must be able to call enclave entry points and have some notion of objects that are inside the enclave. Unlike C, where most function addresses are determined once and statically at the dynamic linking phase, Java identifies most functions through object references (e.g., `Foo.toString()`). Thus, the untrusted code must be able to reference objects inside of the enclave in a way that makes sense when the untrusted code behaves correctly, but that does not create unexpected information flows out of the enclave. Similarly, the developer needs a constructor interface to declare which dynamically-created objects should be placed in the enclave and which should be outside of the enclave.

Discussion. Partitioning an application requires some input from the developer in order to identify sensitive data and code. Each of the challenges above highlight cases where Java either hides important information from the developer, or otherwise useful runtime techniques can thwart the isolation benefits of using a hardware mechanism like SGX. A higher-level goal of Civet is to provide constructs for the developer to specify her goals, such as which objects should be isolated in an enclave, and to have the language runtime use these developer-provided hints to make judicious choices on issues such as data placement. A related goal is not eroding the benefits of developing in a high-level, managed language runtime.

3.2 Overview of the Civet Framework

Civet consists of both design-time and runtime components.

Partitioning Java applications into enclaves (Shredder). To cleanly partition Java applications into trusted and untrusted components, Civet provides a design-time tool called Shredder to auto-

mate partitioning. The developer manually identifies trusted classes that should be placed in the enclave, but will still interact with untrusted components. These classes are called entry classes for the enclave. Based on the list of entry classes, Shredder selects all supporting classes required by the entry classes, and creates a static image of java classes, packaged as a signed JAR file. The developer can request for other non-sensitive classes or packages to also be included in the enclave JAR file.

For most cases, Civet only requires the developers to identify the entry classes, and, as desired, to annotate declassifiers for information flow tracking (§3.3.3). This approach minimizes the developer effort required to partition the application.

Civet Runtime Framework. The Civet runtime framework abstracts the low-level semantics of the SGX hardware from the applications. The Civet framework creates two Java execution environments: one in the enclave and one outside the enclave, as illustrated in Figure 3.2. Both environments have an individual Java VM. The Java VM outside the enclave is the default Java VM; the Java VM inside the enclave is a lightweight Java VM, with just enough features to support the trusted components but a smaller TCB. The lightweight Java VM runs on a library OS.

Civet framework creates an enclave for the trusted classes when one of the entry classes is instantiated, or untrusted code calls a static, public method of a trusted class. The Civet framework front-end uses the signed JAR file that contains all the trusted supporting classes as the image to verify and load into the enclave. Figure 3.2 illustrates this process. When the class `Untrusted` instantiates the trusted class `Trusted`, Civet framework creates the enclave, and instantiates `Trusted` inside the enclave so the execution will be isolated.

After the trusted classes are instantiated, the untrusted classes can call public methods on the Trusted objects. Calling a trusted object function from outside the enclave transfers control to the Civet framework back-end, which then calls the appropriate method on the object in the enclave—conceptually similar to a remote procedure call, but on the same CPU core. In the example in Figure 3.2, a call to method `Trusted.process()` from an the `Untrusted` class, causes entry to the enclave to run the method.

We chose to use two JVMs to minimize the risk of the trusted JVM's integrity being compromised. The other sensible option might be to run a single JVM in the enclave that also services the untrusted code. The risk of running only one JVM in the enclave is that the attack surface for the enclave is considerably wider, and there is more risk of attacks on the integrity of the trusted JVM by untrusted code. Of course, one can also place the JVM outside of the enclave, but using an untrusted language runtime seems dauntingly difficult and is beyond the scope of this paper.

Helper APIs for enclave features. Civet provides a Java class (`Enclave`) for application developers to use enclave features, such as attestation and secure provisioning. For attestation, Civet generates a proof of the enclave integrity signed by the CPU, with the hardware measurement of the Civet runtime; the Civet runtime combines this with a measurement of the loaded classes. For secure provisioning, Civet can secure a connection with a remote host, by encrypting the connection and authenticate both sides using attestation. This class is useful for features such as loading a sensitive class file or transferring a secret from a trusted, remote host.

Implementation. Civet is built upon OpenJDK 7, using Java and JNI. Civet requires no changes to the default Java VM in the host, but does modify the lightweight Java VM inside the enclave.

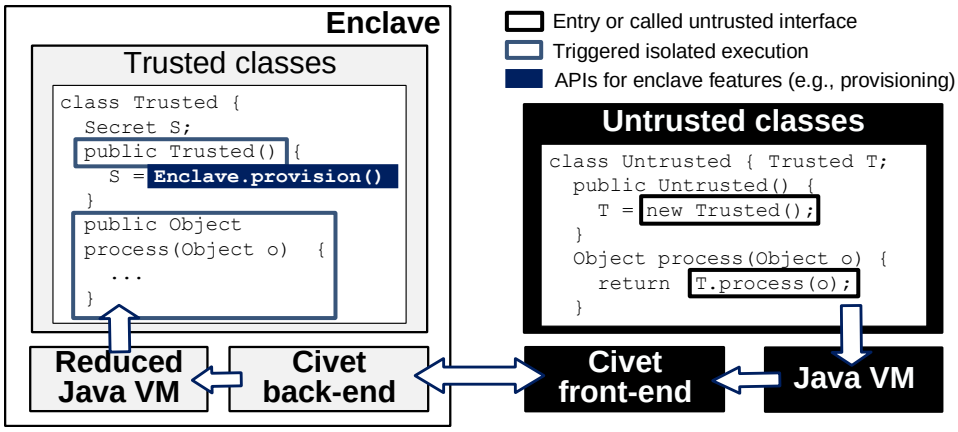


Figure 3.2: How Civet abstracts the SGX hardware protection for Java applications. When an untrusted class (`Untrusted`) calls the constructor of a trusted class (`Trusted`), Civet creates the enclave and instantiates the `Trusted` class inside the enclave. The public methods of the `Trusted` class (`process`) are exported as the untrusted interface of the enclave, and the invocation of these methods will be re-routed into the enclave. Civet also provides APIs for accessing enclave features such as secure provisioning.

3.2.1 Security Properties

Civet provides comparable security properties to an enclave running a static, native binary. First, Civet maintains code integrity by verifying the signed JAR file that contains all the supporting classes, potentially including classes from the Java Standard Library. All methods and objects of the trusted classes are completely isolated inside the SGX enclave. The objects returned from isolated methods of trusted classes are only released from the enclave if the developers explicitly use the Civet's declassifier API to mark the objects as safe.

We explain in more detail below how Civet reduces the enclave's TCB, and provides building blocks for tracking information flows within the enclave, code confidentiality, and remote attestation.

Minimizing the enclave TCB. A Java application often yields a huge TCB, including the Java VM, JNI and loaded classes. For example, the OpenJDK 7 binaries are 40MB in total. The standard runtime library, rt.jar, contains more than 18,604 classes, of which only around 1,000 classes are typically loaded.

The Shredder significantly reduces the TCB of code in the enclave by eliminating any supporting classes that are not required to run the code in the enclave. We further reduce the TCB by removing unused Java VM features such as multi-threaded garbage collection and JIT compilers, unused classes from the Java standard runtime library, and unused APIs from the library OS and the C standard library.

Information flow control at enclave border. An essential concern for application partitioning is ensuring that a bug or vulnerability in the trusted partition does not disclose confidential data that the partitioning was intended to protect. Thus, Civet uses information flow control to prevent implicit or explicit leaks of sensitive secure provisioned data from the enclave. For classes in the enclave, any confidential data, such as a private encryption key, is securely provisioned and protected by the Civet. The Civet runtime framework builds on Phosphor [Bell and Kaiser, 2014]

to instrument classes in the enclave to track information flow through the enclave. At the boundary of the enclave, any variable tainted with a confidential input cannot leave the enclave unless it is either passed through a declassifier.

Civet does allow references to a confidential object to be passed out of and into the enclave, using an opaque **proxy object**. The proxy object can include a serialized and encrypted representation of the data, for literals, or a reference to an object inside the enclave that can be passed as an argument to a subsequent function. For JNI functions that make system calls in the enclave, Civet encrypts all data leaving enclave by encrypting at the library OS level. Civet enforces only confidentiality of the provisioned data, but the infrastructure can be easily extended to ensure data integrity too by propagating taint on enclave inputs.

We note that these opaque proxy objects strike a reasonable balance between ease of use and preventing unexpected information flows out of the enclave. The proxy objects do not contain any indicators about enclave-internal state. If the same object is returned from multiple functions, each opaque reference is unique, and they cannot be compared for equality. Similarly, before a literal return value is encrypted, we add a nonce to the plaintext to avoid comparison of the ciphertext. In the worst case, the untrusted code can leak references via proxy objects, which amounts to a denial-of-service for DRAM—an attack unavoidable within the SGX threat model.

Code Confidentiality. Code confidentiality is a desirable property for algorithms or code that a user wishes to protect, such as a trade secret. The hardware-level SGX code integrity mechanism is based on a cryptographic signature of a static binary in plaintext. Civet can execute confidential code with a dynamic loader that can load encrypted classes from remote, trusted hosts. The remote host uses SGX's remote attestation features to validate the integrity of the Civet enclave.

3.2.2 Threat Model

We assume that any part of the system stack, including the OS, device drivers, and hypervisor can behave adversarially. Similar to other SGX systems, we also assume hardware not in the CPU package, such as the DRAM or GPU can also attempt to attack the enclaves. We assume the attackers have complete information about the SGX hardware implementation, application source (except for confidential code modules), and Civet source code.

An adversary can attempt to exploit a vulnerability in the partitioned applications, by manipulating inputs to the application via the interface between the front-end and back-end.

We assume denial-of-service and side-channel attacks are possible; addressing these attack vectors is out of scope for this paper.

Trusted Components. All code in the enclave is part of the trusted computing base (TCB), including the Civet infrastructure (§3.5); all supporting classes and their JNI; and other resources or classes provisioned from remote hosts. The implementation of SGX hardware is also trusted, and uses adversary-resistant key generators that cannot be compromised by online or offline techniques. We also assume Intel CPUs are resistant to direct, physical attack to the CPU packages, either to modify or peek into the chips.

We also assume that the JVM and JNI code are free from memory corruption and control flow attacks. Proving a JVM implementation correct is beyond the scope, although similar efforts have been made previously to prove a language runtime correct [Yang and Hawblitzel, 2010]. We discourage developers from using JNI code in enclaves if possible. We do not support running

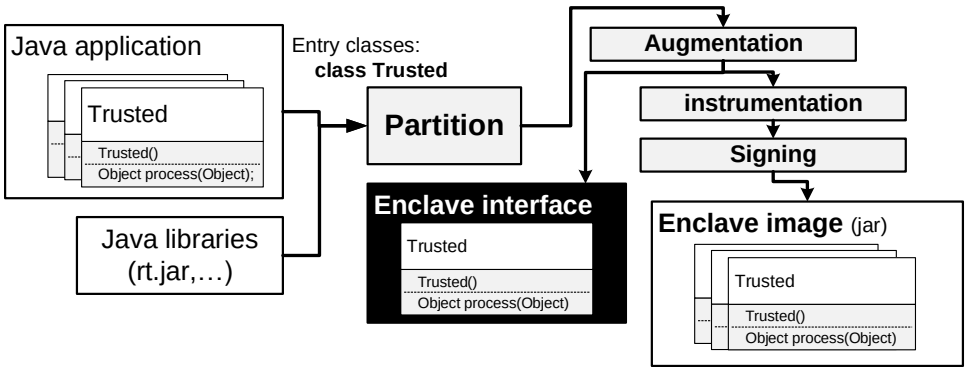


Figure 3.3: The Civet design-time tool, for partitioning, packaging, augmentation, instrumentation, and signing. Civet partitions the Java application based on the entry classes specified by the developers (classes `Trusted`). The Shredder tool recursively pulls all required supporting classes into the package to run in the enclave. The Shredder tool creates an enclave JAR file after augmentation, instrumentation, and signing.

Java application with JIT optimization inside the enclave. Even if running Java application with JIT optimization can improve the performance of execution, we avoid adding the huge JIT compiler to the TCB of the enclaves.

3.3 Java Support for Partitioning into SGX

This section explains the support Civet adds to Java for partitioning applications into SGX enclaves.

3.3.1 Cleanly partitioning classes and objects

Civet includes Shredder to automatically partition Java applications on SGX, requiring minimal developer effort. Figure 3.3 shows the workflow of a developer using the Civet Shredder tool. The developer selects the application's main class, either as a JAR or class files and the classpath of the library. The developer also specifies the list of entry classes that should be in the enclave and export an interface to the code outside of the enclave. The Shredder then identifies all classes that the entry classes depend upon, until the transitive closure of these dependencies converges.

The Shredder creates a single package with all dependencies of the entry classes. This is a reasonable simplification, although it would be relatively easy in future work to add more signed JAR-style packages, if needed. This approach also allows us to reduce the attack surface and overheads by minimizing the enclave entry and exit points, albeit at the expense of duplicating some supporting classes in and out of the enclave.

Unless the application explicitly asks the class loader to dynamically load a class, every piece of code needed during the execution of trusted classes is included in the enclave image. The image includes classes that are used in dynamic casting and parent classes that contain code inherited by the trusted classes. Shredder also includes any required supporting JNI libraries in the image. Shredder does not attempt to partition the JNI libraries to a smaller binary, which we leave for future work.

Based on our case studies (§3.6), we observe that specifying the entry classes and identifying

any dynamically loaded classes, requires minimal developer effort. In all of our use cases, the applications are partitioned with only one entry class, and very few dynamically loaded classes.

3.3.2 Dynamically loading byte-code with integrity

After the Shredder partitions the applications and creates an enclave image as a JAR file, developers can ship the enclave image with the rest of the application. The application is then executed on an untrusted host with the Civet runtime framework installed. Upon the first use, either by creating a trusted object or calling a static method of a trusted class, the Civet runtime framework creates an enclave containing the trusted classes.

Figure 3.4 shows the structure of the Civet runtime framework(a more detailed view of Figure 3.2). The Civet runtime framework is split into the front-end (untrusted) and the back-end (trusted). When the front-end calls into a trusted class, it finds enclave image that contains the class, checks if an enclave is created for the same image in the current Java VM, and, if not, creates an enclave. The trusted classes from the same image share an enclave.

Civet runs a separate, lighter-weight Java VM in the enclave. Running a Java VM in the enclave is a subtle challenge because a Java VM often yields a large system API footprint, and, by default, uses a large heap. To provide the required OS APIs inside the enclave, we use the Graphene library OS [Tsai et al., 2014] [1]. In order to remove unneeded features and balance resource utilization with performance in current SGX enclaves, such as a 128 MB limit on the size of the enclave page cache, we adjust the build-time configuration of the JVM to change multithreaded garbage collector to single threaded, remove multiple JIT engines and stop non-essential threads in the JVM.

When Civet creates an enclave, the SGX hardware measures integrity of the initially-loaded library OS. The library OS then loads OpenJDK 7 and all of the supporting libraries, such as `libc`, the JLI (Java legacy interface) library, and the minimal Java classes needed to bootstrap the class loader. Finally, Java VM loads the enclave image JAR file.

Graphene itself is responsible to maintain the code integrity of the JVM. When Graphene loads a binary or class files, it verifies the integrity of the files by checking their measurements. The measurements of binary or class files are also hashed into the enclave measurement, so no attacks can bypass the integrity check or manipulate Graphene to load a malicious Java VM or bogus enclave image.

If an application requests dynamic loading by a class name, the developers must specify the classes to the Shredder tool. The case that dynamic loading is needed is most commonly seen in the crypto APIs: for example, to instantiate a `Cipher` object, the applications provide a string that describes the transformation of the cipher, such as `AES/CBC/PKCS5PADDING`. Based on the string, the method `Cipher.getInstance()` loads classes `AESCipher`, `PCBC`, and `PKCS5Padding`. The rationale behind this restriction on dynamic loading is that, no matter which class name the application requests, the class loader only searches among the trusted classes, so no malicious classes will be loaded.

[1] Downloaded from `github.com/oscarlab/graphene`

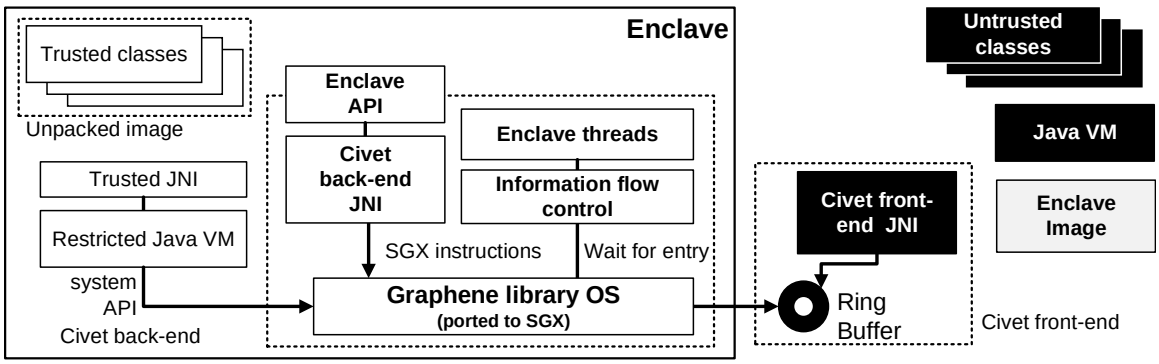


Figure 3.4: Overview of the Civet runtime framework. The Civet runtime framework creates two worlds for an partitioned Java application, each with an individual JVM. The JVM in the enclave is ported using Graphene library OS. Untrusted classes can invoke methods of trusted classes through proxy objects, which can transparently access the enclave interface, through serialization and deserialization over an ring buffer accessed by both untrusted JVM and trusted JVM.

3.3.3 Seamless access to in-enclave objects

For programmer convenience, untrusted code can seamlessly call in-enclave objects in Civet. This is particularly useful when application components are closed-source. All public methods of trusted, programmer-identified entry classes are entry points for the enclave. The Civet runtime framework is responsible for generating glue code for entering and exiting the enclave appropriately, tracking references to objects in the enclave, as well as marshalling arguments and return values for in-enclave functions.

In order to reference objects inside the enclave from outside the enclave, Civet framework uses a byte code generation library — *CGLib* [cglib] to create untrusted proxies for the in-enclave instances. CGLib instruments the class that is being proxied, and redirects the control to a handler assigned by Civet upon any method invocation on the proxy. The proxy then triggers enclave entry to run the trusted method.

In general, supporting classes can be duplicated inside and outside of the enclave. Calls to a supporting class, such as `String`, from inside of an enclave go to the in-enclave version, and calls from outside the enclave go to the untrusted version.

An exception is made for entry classes, which are not allowed to be replicated. Rather, any call to an entry class function is placed inside the enclave. Thus, constructors and static methods of entry classes also cause enclave entries. We note that this design point was taken to minimize programmer effort in porting to SGX; alternatively, we could allow an entry class to be replicated by requiring the programmer to explicitly annotate calls to object functions.

The Civet design-time tool creates untrusted proxy classes for all entry classes, in which all constructors and static methods are redirected to the Civet front-end, which then enters the enclave. We chose this approach because CGLib disallows redirecting constructors and static methods, as this can introduce ambiguity in the invocation target when all classes are in the same JVM.

Passing arguments into the enclave. When a method triggers enclave entry, the arguments of the method have to be passed into the enclave for the invocation. Civet always copy the arguments into the enclave, by serializing the arguments into byte streams, copying the byte streams into the

enclave memory, and then de-serializing into objects. By coping arguments into the enclave, Civet ensures execution of trusted code does not inadvertently leave the enclave. If the code invokes a method on one of the arguments, the in-enclave copy of the class is used on an in-enclave instantiation of the object. Upon de-serialization, the arguments are also automatically type-checked, thus avoiding the risk of memory corruption.

Returning objects. Once the triggered method finishes execution in the enclave, it may return an object or literal back to the untrusted calling function. In general, objects are returned similarly to passing input arguments—by serializing the object to a byte stream and returning the bytes. x In order to ensure confidentiality of sensitive data, Civet takes additional care to check whether a returned object creates an unexpected control flow. At enclave exit, Civet only allows an object to be returned if it is not tainted with any secret data, in which case the object is serialized and passed back to the caller. Section 3.4 details our information flow tracking mechanism.

In cases where the object is tainted and an instance of a trusted class, Civet instead creates a reference in the enclave (to prevent garbage collection of the object internally), and returns an opaque reference type, that causes the untrusted Civet runtime to create a proxy out of the enclave. This policy applies to all constructors. If a proxy object is garbage collected, the destructor calls into the enclave to release the reference on the corresponding object in the enclave. The Civet untrusted runtime is responsible for translating any proxy objects passed as arguments to the enclave into opaque pointers, which the in-enclave runtime then translates to local object references.

In the case of a tainted literal, we encrypt the plaintext return value concatenated with a nonce, using a temporary key, and return the ciphertext. This encrypted literal can then be passed to subsequent enclave calls, where the value is decrypted as part of deserialization.

3.3.4 Remote Attestation and Provisioning

Generating attestation reports. A feature of SGX hardware is the ability to generate an attestation report for a remote entity, demonstrating the integrity of the enclave code at launch time. Civet provides helper API for developers to access these features, with convenience and extended trust. For attestation, Civet generates a report that contains a list of classes loaded inside the enclave, with their measurements. The report is attached with the attestation generated and signed by SGX, but processed by Graphene. The SGX-generated attestation contains both the enclave measurement (proving integrity of Graphene) and the measurement verified by Graphene (proving integrity of other binaries and files).

Note that Civet also includes the dynamic loading state in the attestation report. The attestation generated by SGX only contains the initial state of the enclave, and does not record changes within the executable code after the enclave starts. In both cases, the remote entity is trusting the initially loaded binary to not dynamically load code that could compromise the enclave; however, Civet can offer a more precise accounting of the state of the enclave at the time a report is generated.

Secure provisioning. Civet provides an API that transparently validates a connection to a remote host to load sensitive classes or secret data. To use this API, both sides of the connection must be running in enclaves created by Civet. The API performs key exchange algorithm (e.g., Diffie-Hellman) on the connection, secure the connection with encryption, and authenticate the connection by exchanging the attestation reports. Civet provides convenient helper functions for developers to create a trusted path for provisioning sensitive data to a remote enclave.

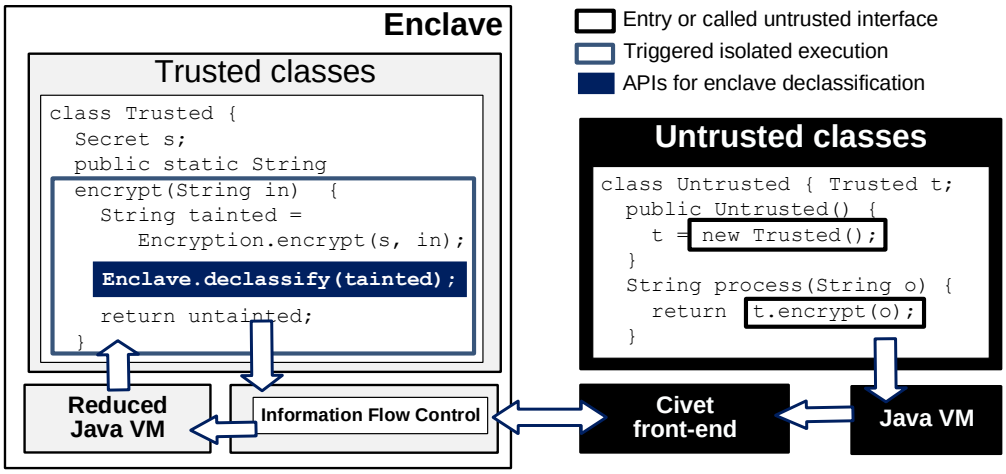


Figure 3.5: How Civet provides declassifier APIs to declassify sensitive data. When the untrusted class (`Untrusted`) from Figure 3.2 now calls the `encrypt` method of a trusted class (`Trusted`), Civet automatically calls the `encrypt` method inside enclave, and pass the `String` parameter. Before returning the result, the trusted class has to use the `declassify` to remove the taint of `tainted` variable that is tainted by the `encrypt` method of class `Encryptor` because of the tainted secret variable `s`.

3.4 Hardening Enclaves using Language Protections

Civet allows Java developers to directly utilize the security features of SGX, such as isolation from an untrusted hypervisor, in combination with language-level features that make the code in the enclave more robust.

As an example of higher-level, language-based analysis, we implemented information flow tracking in Civet. A common usage of enclaves is to protect sensitive data, such as an encryption key; thus, a common concern is that this sensitive data not be inadvertently returned because of an error or exploit within the enclave code. In general, we chose a design point that minimizes programmer effort; to adopt information flow tracking, we do require the programmer to specify secret data classes and declassify objects to be released as is from the enclave.

In the enclave, we implement source-to-sink taint tracking, using the open-source Phosphor library [phosphor]. The programmer manually selects the classes containing secret data that take secret input from a remotely-provisioned source. This taint is propagated to any new variables that result from explicit or implicit flows from a secret object. The only way to remove taint from an object is to pass the object through a Civet declassifier API, which returns an untainted copy of the object.

Civet enforces the policy that only untained data may be returned from an enclave. If tainted data is being returned, the system transparently encrypts the data and removes its taint before letting the data leave the enclave. In the cases where a developer wants to return references to sensitive data, Civet instead returns an opaque reference or, in the case of a literal, encrypts the return value.

The Civet `Enclave` class provides a `declassify(Object o)` API that creates an untainted copy of the object. In practice, we expect this function to be used in conjunction with tests on the returned data, or cryptographic functions to protect the data in transit across an untrusted channel.

Continuing our example from Figure 3.2, in Figure 3.5, if the `Untrusted` class wants to call

the `encrypt` method on the trusted object `t`, Civet front-end transparently passes the argument string `o` to the enclave, and the corresponding `encrypt` method is called in the enclave. The secret `s` is tainted because it was provisioned from remote trusted server as shown in Figure 3.2. As a result, the call to method `encrypt` of class `Encryptor` taints the encrypted output string `tainted`. If the developer had returned this `tainted` variable, the Civet information flow tracking would re-encrypt the ciphertext, and thus make the return value useless for the `Untrusted` class. However, as the developer wants to return the ciphertext as is, she can declassify the `tainted` string by passing it through the declassifier API to get an untainted version of the same object. Such untainted objects can be released from the enclave without further encryption.

In order to protect the confidentiality of sensitive code, Civet also allows classes themselves to be tainted. Civet enforces a policy that any data returned from sensitive code is tainted, and the developer needs to explicitly declassify tainted output data to mitigate concerns around reverse-engineering the code based on brute-force probing of its outputs. Of course, the binary code itself is also not allowed to be copied out of the enclave.

3.5 Implementation Details

In this section, we discuss in detail about how the framework of Civet is implemented.

Graphene library OS for SGX. Similar to Drawbridge in Haven, Graphene library OS can map a larger number of system interfaces to a much narrower host interface. The untrusted interface of Graphene libOS after porting to SGX is mostly identical to the original host interface, therefore Graphene libOS has a finite bound on interface for the enclave.

Graphene libOS uses checksums of files to verify the integrity of supporting binaries and configuration. The checksums of files are collected by a compile-time Signer tool of Graphene libOS. The Signer tool ensures the integrity of the file checksums by including them as part of the enclave's measurement. Even if the same Graphene libOS is used to run the applications, different binaries in the enclaves yield different measurements. Such a design decouples the problem of distributing the library OS and guaranteeing code integrity for each application, as the enclave integrity is based on the integrity of the application — not just the library OS.

Civet makes minor modifications to Graphene library OS to allow the enclave to run in the same process as the the untrusted Java VM. The trusted and untrusted VM share a ring buffer (as shown in 3.4 for communication during control transfer, but the ring buffer itself is not trusted. The Civet framework encrypts tainted security-sensitive data before passing the data on the ring buffer.

Control transfer at enclave entry. Civet transparently transfers the control from the untrusted classes to trusted classes — triggering enclave entry in the process — by intercepting the trusted classes's method or constructor invocation by untrusted classes. Civet intercepts in 2 cases: (a) For an entry class that has finite entry points, Civet creates a bogus class that redirects all the constructors and static methods to the enclave. (b) For other trusted classes, the Civet adds the redirection calls only when a reference is returned by the enclave. On future access of the object in the enclave, a proxy object is created to trigger the interception, using CGLib.

When the control transfers between the trusted and untrusted classes, the arguments and return values of the methods are stored in the ring buffer. Civet relies on serialization and deserialization

to safely transfer object in and out of the enclave. When Civet deserializes an object, the Java VM perform type-checking on the object to sanity check the members.

After Civet transfers the arguments, the Java VM thread that invoke the intercepted method does not directly enter the enclave. On the other hand, several enclave threads that are pre-created during the enclave creation that are block-waiting on the ring buffer. One of the enclave threads consumes the queued job, invokes the method, and returns to block-waiting for more new requests. No variable on the stack has to be persistent for the enclave threads, and only instances of the trusted classes are persisted across enclave entry and exit.

Moreover, upon enclave entry and exit, the objects that can be safely transferred in or out of the enclave must be serialized / deserialized. However, not all classes implement the interface `Serializable`, especially when the classes contain internal states that cannot be simply interpreted. But, Civet assumes that the types of all arguments and return objects must be `Serializable`.

Framework limitations. As we leverage dependency tracking for automated partition, there are few corner cases that the Shredder cannot gracefully handle. Because CGLib creates subclasses of objects when intercepting them, it requires the intercepted object to be never finalized. If a trusted class is also a final class, developers have to manually modify the class definition. Even though final attribute prevents further extension of the class, we argue that the application developer who is building the enclave can safely remove the final attribute as the Civet signs the enclave jar.

We also do not let trusted classes make method or constructor invocation of the untrusted classes, as the untrusted classes may be able to influence and interfere with the execution of trusted classes. Moreover, we only consider the provisioned code and data as security-sensitive. Even though this limits the usage scenarios, we argue that for any right usage of SGX hardware, these limitations are not disruptive.

3.6 Case Studies

In order to evaluate the utility of Civet, we partitioned several example Java applications, which we use in our evaluation.

Session Encryption in SSH Client/Server. In order to show the ability of Civet to protect a confidential data and avoid leaking that data, we partitioned a Java-based SSH client and server [apache-sshd]. In this case, the protected secret is the session key. For both the SSH client and server, we create a control class in the enclave that includes the key generator, encryption engine objects, secret key and the *BouncyCastle* [bouncycastle] cryptography library. The rest of the application is outside of the enclave.

We use information flow tracking to ensure that the only data leaving the enclave is ciphertext output from the encryption algorithm, or plaintext returned from the decryption algorithm. This involves adding lines to the end of both algorithms, and does assume that the encryption and decryption functions are implemented correctly. Attempts to copy the session key directly into an output buffer at any other point in the code will result in encryption of the buffer before leaving the enclave.

`org.apache.sshd.common.FactoryManager` is identified as the entry class for Shredder, because this class returns all the objects required for the SSH Protocol. Shredder generates the

enclave image that includes the secret key classes, random generator, key generator, engine for diffie-hellman key-exchange, and encryption-decryption engine. Both the client as well as the server are partitioned to be run using Civet. The client and server first mutually attests each other, exchange the key, and then setup the secure session. We use these SSH client and server for transferring files using SFTP. For simplicity, we run both client and the server on the same host, but in different enclaves.

Secret Hadoop Algorithm. Civet not only protects confidential data, but also protects confidential code. For instance, if a company has developed an analysis tool that gives them a essential competitive advantage, they must either run their own data center or trust a cloud provider not to leak their tool to any competitors.

To demonstrate this use case, we modify a Hadoop sort algorithm [Kumar, 2014], so that the implementation of its Partitioner, Mapper, Combiner, Sorter and Reducer are isolated from the rest of the Hadoop infrastructure. The algorithm sorts values from a key-value store, in which the input keys and values are encrypted. The output of the sorting algorithm is a sorted, encrypted key value store.

The Hadoop framework schedules proxy Partitioners, Mappers, Combiners, Sorters, and Reducers, and then creates enclaves for these classes. Once a baseline Civet enclave is created, encrypted class files are downloaded from a trusted server using our remote attestation tools, and then decrypted, loaded, and measured for integrity.

The information flow control at the enclave border protects against accidental output of a plaintext key-value pair from the encrypted store, as well as protects the class code file itself. Similarly, intermediate state from the code cannot be inadvertently returned by a function to the untrusted Hadoop framework, although we do allow encrypted outputs to be passed from a Mapper to a Combiner or Reducer. The contents of any code cannot be leaked outside enclave by copying the code into an output object, as the tainted object is automatically encrypted before leaving enclave. Also, in conjunction with the trusted remote server, we rate-limit instantiations of the code to mitigate the risk of brute-force mapping of its outputs or reverse-engineering the code.

Secure Data Manager Web-app. Civet can also secure *Java Network Launch Protocol* (JNLP) applets and web-start apps, effectively extending trust from a remote trusted server to a client enclave using any web-app plugged into a supported browser. This allows the developers to offload some of the computation on secret data to the client machine from the trusted server.

For instance, in a large medical hospital with a centralized repository of patient data, the doctors may want to access any patient data from any terminal using a web browser. A web developer can design an applet such as Secure Data Manager(SDM) [sdm] to store the secret patient data on a client machine, and display this information securely using Intel Protected Audio and Video Path (PAVP) technology [intel-pavp]. However, to protect the sensitive patient information from untrusted system stack, the developer can use Civet to isolate the classes that represent the secure data in an enclave. The developer just has to identify such classes representing the secret data and display methods, and the Civet seamlessly creates enclave for managing secret data. The trusted data manager class authenticates the doctor and loads the provisioned data from the remote trusted server. The Intel PAVP enabled displays can then take the input from the display methods of the trusted data manager class.

For simplicity, we only partition and run the SDM applet from a browser without the support for Intel PAVP display devices. We identify four classes — `SafetyBox`, `AuthenticationInfo`,

`LoginEntry`, and `Type` — as entry class to the Shredder and generate a web-start app image containing the enclave image. The web-app loads normally, and when it tries to access any of the above four classes, Civet seamlessly creates an enclave for those objects.

3.7 Summary

Partitioning an application into trusted and untrusted components provides more plausibility for the soundness of the sensitive execution in enclaves, than isolating the whole application. Similar to the use cases demonstrated in this chapter, in many applications, developers can identify a large fraction of execution to be irrelevant to the target of isolation. **Civet** allows legacy applications written in high-level, managed languages such as Java, to isolate a piece of the execution in the SGX enclave. Unlike the non-partition model, in which the system is only responsible for translating system APIs to untrusted interface, partitioning a Java applications requires generate a clean separation of trusted and untrusted classes and object, and a framework to dynamically load and interface objects in enclaves. Civet essentially enables developers to painlessly partition legacy Java applications into enclave, allowing more flexible use cases of the hardware protection.

At a high level, Language-based and hardware protection are both valuable building blocks for secure applications. Civet shows a constructive synthesis of both technologies — combining their strengths and mitigating downsides of either when used in isolation. Although this work focuses on Java and Intel SGX as a specific case study, we believe the technique is more generic and can apply to other managed languages and hardware isolation platforms. **This work is still in progress, and will be completed for the fulfillment of the thesis.**

Chapter 4

Measuring the Legacy Support

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

Many experimental operating systems add a rough Unix or Linux compatibility layer to increase the number of supported applications [Appavoo et al., 2003; Aviram et al., 2010; Douceur et al., 2008; Zeldovich et al., 2006]. Such systems generally support a fraction of Linux system calls, often just enough to run a few target workloads. One metric for compatibility or completeness of a new feature is the count of supported system APIs [Baumann et al., 2013; Bergan et al., 2010; Porter et al., 2009; Tsai et al., 2014]. System call counts do not accurately estimate the fraction of applications or users that could plausibly use the system. OS researchers would benefit from the ability to translate a set of supported system calls to the fraction of applications that can be directly supported without recompilation. Similarly, it is useful to know which additional APIs would enable the largest range of additional applications to run on the system. In order to indicate general usefulness, a good compatibility metric should factor in the fraction of users whose choice of applications can be completely supported on a system.

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

4.1 Some APIs Are More Equal Than Others

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

These metrics are easy to quantify, but problematic. Simply put, not all APIs are equally important: some are indispensable (e.g., `read` and `write`), whereas others are very rarely used (e.g., `preadv` and `delete_module`). A simple count of system calls is easily skewed by system calls that are variations on a theme (e.g., `setuid`, `seteuid`, and `setresuid`). Moreover, some system calls, such as `ioctl`, export widely varying operations—some used by by *all* applications and many that are essentially never used (§4.2.3). Thus, a system with "partial support" for `ioctl` is just as likely to support all or none of the Linux applications distributed with Ubuntu.

One of the ways to understand the importance of a given interface is to measure its impact on end-users. In other words, if a given interface were not supported, how many users would notice its absence? Or, if a prototype added a given interface, how many more users would be able to use the system? To answer these questions, we must consider both the difference in API usage among applications, and the popularity of applications among end-users. We measure the former by analyzing application binaries, and determine the latter from installation statistics collected by Debian and Ubuntu [Debian Popcons; Ubuntu Popcons]. An **installation** is a single system installation, and can be a physical machine, a virtual machine, a partition in a multi-boot system, or a chroot environment created by `debootstrap`. Our data is drawn from over 2.9 million installations (2,745,304 Ubuntu and 187,795 Debian).

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

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

For each binary in a package—either as a standalone executable or shared library—we use static analysis to identify all possible APIs the binary could call, or the **API footprint**. The APIs can be called from the binaries directly, or indirectly through calling functions exported by other shared libraries. A package's API footprint is the union of the API footprints of each of its standalone executables. We weight the API footprint of each package by its installation frequency to approximate the overall importance of each API. Although our initial focus was on evaluating research, our resulting metric and data analysis provide insights for the larger community, such as trends in API usage.

4.1.1 API Importance: A Metric for Individual APIs

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.

Definition: **API Importance.**
For a given API, the probability that an installation includes at least one application requiring the given 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.

4.1.2 Weighted Completeness: A System-Wide Metric

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

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

Definition: **Weighted Completeness.**
For a target system, the fraction of applications supported, weighted by the popularity of these applications.

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

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

We note that this model of a typical installation is useful in reducing the metric to a single number, but also does not capture the distribution of installations. This limitation is the result of the available package installation statistics, which do not include correlations among installed packages. This limitation requires us to assume that package installations are independent, except when `APT` identifies a dependency. For example, if packages *foo* and *bar* are both reported as being installed once, we cannot tell if they were on the same installation, or if two different installations.

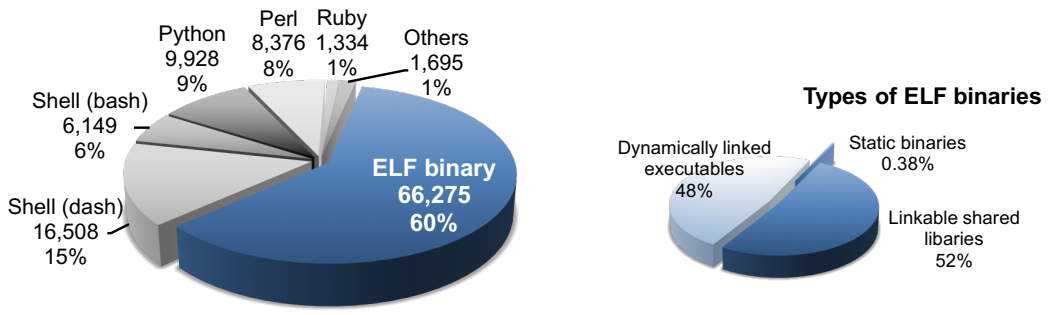


Figure 4.1: Percentage of ELF binaries and applications written in interpreted languages among all executables in the Ubuntu/Debian Linux repository, categorized by interpreters. ELF binaries include static binaries, shared libraries and dynamically-linked executables. Interpreters are detected by *shebangs* of the files. Higher is more important.

If foo and bar both use an obscure system API, we assume that two installations would be affected if the obscure API were removed. If foo depends on bar, we assume the installations overlap. Appendix A.2 formally defines weighted completeness.

4.1.3 Data Collection via Static Analysis

We use static binary analysis to identify the system call footprint of a binary. This approach has the advantages of not requiring source code or test cases. Dynamic system call logging using a tool like `strace` is simpler, but can miss input-dependent behavior. A limitation of our static analysis is that we must assume the disassembled binary matches the expected instruction stream at runtime. In other words, we assume that the binary isn't deliberately obfuscating itself, such as by jumping into the middle of an instruction (from the perspective of the disassembler). To mitigate this, we spot check that static analysis returns as superset of `strace` results.

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

Our study focuses primarily on ELF binaries, which account for the largest fraction of Linux applications (Figure 4.1). For interpreted languages, such as Python or shell scripts, we assume that the system call footprint of the interpreter and major supporting libraries over-approximates the expected system call footprint of the applications. Libraries that are dynamically loaded, such as application modules or language native interface (e.g.,JNI, Perl XS) are not considered in our study.

4.1.4 Limitations

Popularity Contest Dataset. The analysis in this paper is limited by the Ubuntu/Debian Linux's package installer, `APT`, and their package installation statistics. Because most packages in Ubuntu/Debian Linux are open-source, our observations on Linux API usage may have a bias toward open-source development patterns. Commercial applications that are purchased and distributed

through other means are not included in this survey data, although data from other sources could, in principle, be incorporated into the analysis if additional data were available.

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

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

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

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

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

4.2 A Study of Modern Linux API Usage

4.2.1 Spot the Most Valuable System Calls

We begin by looking at the API importance of each system call, in order to answer the following questions:

- Which system calls are the most important to support when implementing a new system, or have high costs to replace, if desired?
- Which system calls are very rarely used and candidates for deprecation?
- Which system calls are not supported by the OS, but still attempted by applications?

There are 320 system calls defined in x86-64 Linux 3.19 (as listed in `unistd.h`). Figure 4.2 shows the distribution of system calls by importance. The Figure is ordered by most important (at 100%) to least important (around 0%)—similar to inverted CDF. The figure highlights several

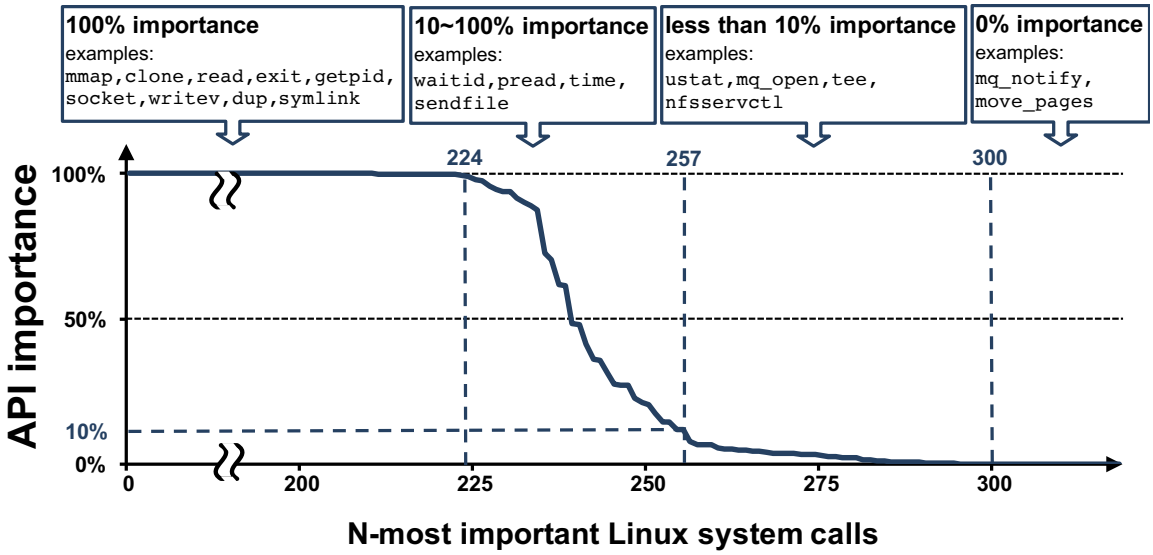


Figure 4.2: The trend of API importance as N-most important system calls among total 320 system calls of Ubuntu Linux 15.04 with Linux kernel 3.19. . Higher is more important; 100% indicates all installations include software that make the system call.

points of interest on this line.

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

Our study also shows the contributors to an API's importance. For instance, Table 4.1 lists system calls that are only called by one or two particular libraries (e.g., libc). These system calls are wrapped by library APIs, so applications depend on them only because the libraries do. To eliminate the usage of these system calls, developers only have to pay minimum efforts to re-implement the wrappers in libraries.

Among the 44 system calls with a API importance above zero but less than ten percent, some are cases where a more popular alternative is available. For instance, Linux supports both POSIX and System V message queues. The five APIs for POSIX message queues have a lower API importance than System V message queues. We believe this is attributable to System V message queues being more portable to other UNIX systems. Similarly, we observed that `epoll_wait` (100%) has a higher API importance than `epoll_pwait` (3%), even though `epoll_pwait` is commonly considered more robust for the same purpose—waiting on file descriptor events. Table 4.2 lists system calls used by only one or two packages—generally special-purpose utilities, such as `kexec_load`, which is used by `kexec-tools`).

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

We also found five system calls `uselib`, `nfsservctl`, `afs_syscall`, `vserver` and `security` system calls that are officially retired, but still have a low, but non-zero, API importance. For

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

Table 4.1: System calls which are only directly used by particular libraries, and their API importance. Only system calls with API importance larger than ten percent are shown. These system calls are wrapped by library APIs, thus they are easy to deprecate by modifying the libraries.

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

Table 4.2: System calls with usage dominated by particular package(s), and their API importance. This table excludes system calls that are officially retired.

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

Table 4.3: Unused system calls and explanation for disuse.

instance nfsservctl is removed from Linux kernel 3.1 but still has API importance of seven percent, because it is tried by NFS utilities such as exportfs. These utilities still attempt the old calls for backward-compatibility with older kernels.

In total, 18 of 320 system calls in Linux 3.19 are not used by any application in the Ubuntu/Debian Linux repository. We list these system calls in Table 4.3. In addition to the issues discussed above, Ten of these system calls do not have an entry point, but are still defined in the Linux headers. Five of the unused system calls such as rt_tgsigqueueinfo, get_robust_list, remap_file_pages, mq_notify, lookup_dcookie provide an interface that is not used by the ap-

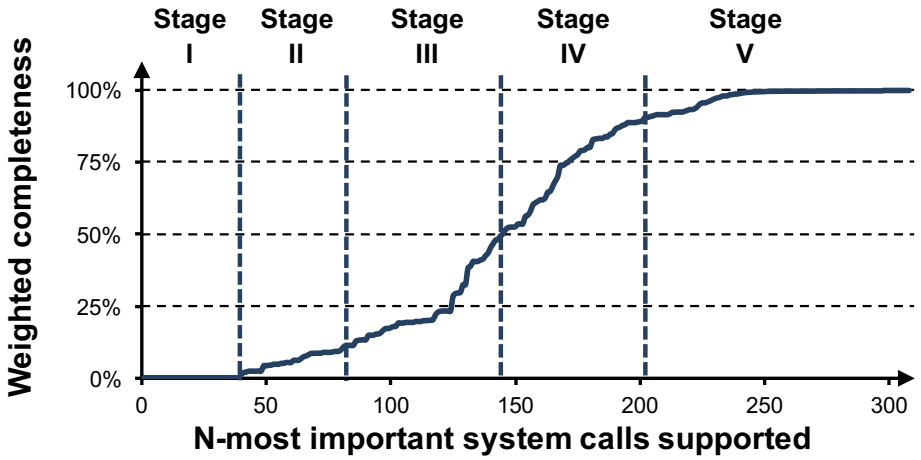


Figure 4.3: Accumulated weighted completeness when N top-ranked system calls are implemented in the OS. Higher is more compliant.

plications. These system calls can be potential candidates for deprecation. However, even though `restart_syscall` is not used by any application, it is internally used by the kernel.

4.2.2 From "Hello World" to Qemu

Figure 4.3 shows the optimal path of adding system calls to a prototype system, using a simple, greedy strategy of implementing the N-most important APIs, which in turns maximizes weighted completeness. In other words, the leftmost points on the graph are the most important APIs, but the y coordinate only increases once enough system calls are supported that a simple program, such as "hello world" can execute. Similar to a CDF, this line continues up to 100% of Ubuntu applications. The graph highlights several points of interest in this curve.

Essentially, one cannot run even the most simple programs without at least 40 system calls. After this, the number of additional applications one can support by adding another system call increases steadily until an inflection point at 125 system calls, or supporting extended attributes on files, where weighted completeness jumps to 25%. To support roughly half of Ubuntu/Debian Linux applications, one must have 145 system calls, and the curve plateaus around 202 system calls. On the most extreme end, qemu's MIPS emulator (on an x86-64 host) requires 270 system calls [Bellard, 2005]. A weighted completeness of 100% implies that all Linux applications ever used are supported by the prototype.

Table 4.4 breaks down the recommended development phases by rough categories of required system calls. We do not provide a complete ordered list here in the interest of brevity, but this list is available as part of our dataset, at `http://oscar.cs.stonybrook.edu/api-compat-study`.

A goal of weighted completeness is to help guide the process of developing new system prototypes. Section 4.2.1 showed that 224 out of 320 system calls on Ubuntu/Debian Linux have 100% API importance. In other words, if one of these 224 calls is missing, at least one application on a typical system will not work. Weighted completeness, however, is more forgiving, as it tries to capture the fraction of a typical installation that could work. Only 40 system calls are needed to have weighted completeness more than 1%.

For simplicity, Table 4.4 only includes system calls, but one can construct a similar path in-

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

Table 4.4: Five stages of implementing system calls based on the API importance ranking. For each stage, a set of system calls is listed, with the work needed to accomplish (# of system calls) and the weighted completeness that can be reached.

cluding other APIs, such as vectored system calls, pseudo-files and library APIs. For example, developers need not implement every operation of `ioctl`, `fcntl` and `prctl` during the early stage of developing a system prototype.

4.2.3 Vectored System Call Opcodes

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

The main expansion is from `ioctl`. Linux defines 635 operation codes, and Linux kernel modules and drivers can define additional operations. In the case of `ioctl`, we obverse that there are 52 operations with the 100% API importance (Figure 4.4), each of which are as important as the 226 most important system calls. Of these 52 operations, 47 are frequently used operations for TTY console (e.g., `TCGETS`) or generic operations on IO devices (e.g., `FIONREAD`).

On the narrow end, `fcntl` and `prctl` have 18 and 44 operations, respectively, in Linux kernel 3.19. Unlike `ioctl`, `fcntl` and `prctl` are not extensible by modules or drivers, and their operations tend to have higher API importance (Figure 4.4). For `fcntl`, eleven out of eighteen `fcntl` operations in Linux 3.19 have API importance at around 100%. For `prctl`, only nine out of 44 operations have API importance around 100%, and only eighteen has API importance larger than 20%.

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

Compared to system calls, `ioctl` has a much longer tail of infrequently used operations. Out of 635 `ioctl` operation codes defined by modules or drivers hosted in Linux kernel 3.19, only 188 have API importance more than one percent, and for only 280 we can find usage of the operations

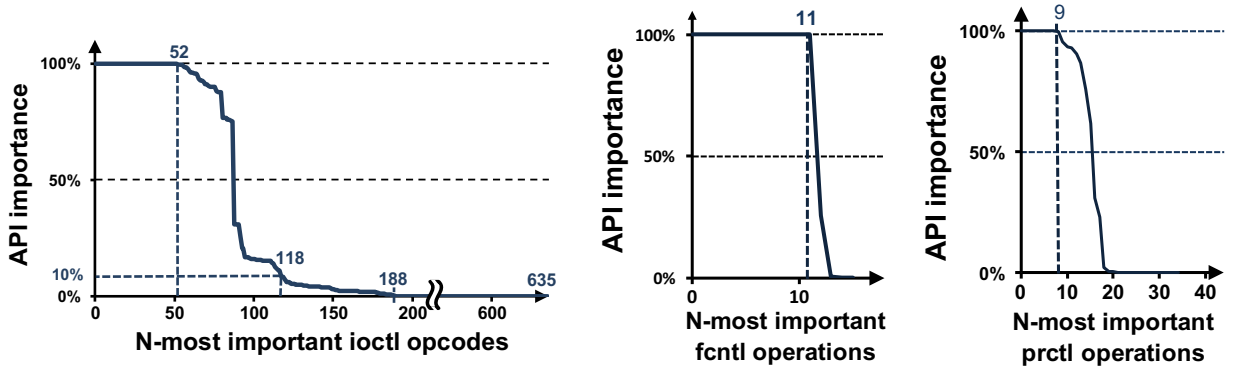


Figure 4.4: Ranking of API importance among ioctl, `fcntl` and `prctl` opcodes. Higher is more important; 100% indicates all installations include software that request the operations.

in at least one application binary. Those unused operations are good targets for deprecation, in the interest of reducing the system attack surface.

4.2.4 Pseudo-Files and Devices

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

We apply static analysis to find cases where the binary is hard-coded to use a pseudo-file. Our analysis cannot capture cases where a path to one of these file systems is passed as an input to the application, such as `dd if=/dev/zero`. However, when a pseudo-file is widely-used as a replacement for a system call, these paths tend to be hard-coded in the binary as a string or string pattern. A common pattern we observed was `sprintf(‘‘/proc/%d/cmdline’’, pid)`; our analysis captured these patterns as well. We also do not differentiate types of access in this study, such as separating read versus write of a pseudo-file; rather we only consider whether the file is accessed or not. Thus, our analysis is limited to strings stored in the binary, but we believe this captures an important usage pattern.

Figure 4.5 shows the API importance of common pseudo-files under `/dev` and `/proc`. These files are ordered from highest API importance; the long tail of files used rarely or directly by administrators is omitted.

Some files are essential, such as `/dev/null` and `/proc/cpuinfo`. These files are widely used in binaries and scripts. Among 12,039 binaries that use a hard-coded path, 3,324 access `/dev/null` and 439 access `/proc/cpuinfo`. However, it is plausible to provide the same functionality in simpler ways. For instance, `/proc/cpuinfo` provides a formatted wrapper for the `cpuinfo` instruction, which one could export directly to userspace using virtualization hardware, similar to Dune [Belay et al., 2012]. Similarly, `/dev/zero` or `/dev/null` are convenient for use on the

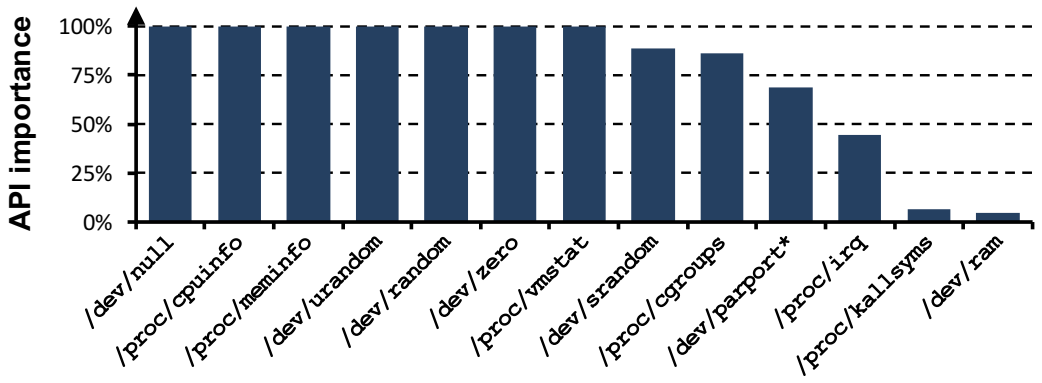


Figure 4.5: API importance distribution over files under `/dev` and `/proc`. Higher is more important; 100% indicates all installations include software that accesses the file.

command line, but it is surprising that a significant number of applications issue `read` or `write` system calls, rather than simply zeroing a buffer or skipping the `write` (e.g., `grub-install`). Thus, in implementing a Linux compatibility layer, a small number of pseudo-files are essential, and perhaps others could be eliminated with modest application changes.

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

Because so many files in `/proc` are accessed from the command line or by only a single application, it is hard to conclude that any should be deprecated. Nonetheless, these files represent large and complex APIs that create an important attack surface to defend. As noted in other studies, the permission on `/proc` tend to be set liberally enough to leak a significant amount of information [Jana and Shmatikov, 2012]. For files used by a single application, an abstraction like a fine-grained capability [Shapiro et al., 1999] might better capture this need. For files used primarily by the administrator, carefully setting directory permissions should be sufficient.

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

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

4.2.5 Reorganizing System Library APIs

In addition to studying kernel interfaces, we also analyze the API importance of APIs defined in core system libraries, such as libc. Most programmers don't directly use the APIs exported by the kernel, but instead program to more user-friendly APIs in libc and other libraries. For instance, GNU libc [gli] exports APIs for using locks and condition variables, which internally use the subtle `futex` system call [Franke et al., 2002].

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

There are several reasons to avoid loading extra code into an application. First, there are code reuse attacks, such as return-oriented programming (ROP) [Shacham, 2007], that rely on the ability to find particular code snippets, called gadgets. Littering a process with extra gadgets offers needless assistance to an attacker. Similarly, when important and unimportant APIs are on the same page, memory is wasted. Finally, the space overhead of large, unused jump tables is significant. In GNU libc 2.21, `libc-2.21.so` essentially has 1274 relocation entries, occupying 30,576 bytes of virtual memory. By sorting the relocation table according to API usage, most libc instances could load only first few pages of relocation tables, and leave the remaining relocation entries for lazy loading.

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

Effects of standard libraries on API importance. Libc and the dynamic linker (`ld.so`) also contribute to the system call footprint of every dynamically-linked executable. This has a marked effect on the API importance of some system calls. The APIs used to initialize a program are listed in Table 4.5. In several cases, such as `set_tid_address`, however, libc or libpthread may be the only binaries using these interfaces directly, indicating that changes to some important system interfaces would only require changes in one or two low-level libraries.

4.3 Linux Systems and Emulation Layers

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.

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

Table 4.5: Ubiquitous system call usage caused by initialization or finalization of libc family.

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 4.6: 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.)

4.3.1 Weighted Completeness of Linux Systems

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

We evaluate the weighted completeness of four Linux-compatible systems or emulation layers: User-Mode-Linux [Dike, 2006], L4Linux [Härtig et al., 1997], FreeBSD emulation layer [Divacky], and Graphene library OS [Tsai et al., 2014]. 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 4.6 shows weighted completeness, considering only system calls. The results also iden-

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 4.7: 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`).

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

4.3.2 Weighted Completeness of Libc

This study also uses weighted completeness to evaluate the compatibility of several libc variants — eglibc [egl], uClibc [ucl], musl [mus] and dietlibc [die] — against GNU libc, listed in Table 4.7. We observe that, if simply matching exported API symbols, only eglibc is directly compatible to GNU libc. Both uClibc and musl have a low weighted completeness, because GNU libc's headers replace a number of APIs with safer variants at compile time, using macros. For example, GNU libc replaces `printf` with `__printf_chk`, which 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).

4.4 Unweighted API Importance

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

Once an API has been identified as having a security risk, and a more secure variant is devel-

oped, one might wish to know how many vulnerable packages are still in the wild, and how many have moved to less exploit-prone APIs. Similarly, one might want to know how many applications have not migrated away from a deprecated API, even if these applications are not widely used.

Definition: **Unweighted API importance.**
For a given API, the probability an application (package) uses that API, irrespective of probability of installation.

One family of APIs prone to security problems are the `set*id` API family. Many of the `set*id` APIs have subtle semantic differences across different Unix variants. Chen et al. [Chen et al., 2002] conclude that `setresuid` has the clearest semantics across all Unix flavors. Table 4.8 shows the unweighted API importance of `set*id` and `get*id` system calls. Most packages have adopted the more clear and secure interface. System calls `setuid`, `setreuid`, and `setresuid` have unweighted API importance of 15.67%, 1.88% and 99.68% respectively. However, for `get*id` system calls, the unweighted API importance suggests that the `getres*id` system calls are only used by roughly 36% of packages.

Directory operations have a long history of exploitable race conditions [Borisov et al., 2005; Cai et al., 2009; Wei and Pu, 2005], or time-of-check-to-time-of-use (TOCTTOU) vulnerabilities. In a privileged application, one system call (e.g., `access`) checks the user's permission, and a second call operates on the file. There are countermeasures that effectively walk the directory hierarchy in user space [Tsafrir et al., 2008]. This approach replaces calls like `access` with `faccessat`, and similar variants. Table 4.8 shows the current unweighted API importance of `*at` system call variants and their older counterparts. We observed that the unweighted API importance of the race-prone `access` is still high (74.24%), whereas `faccessat` is only 0.63%. This suggests about 75% of the packages use the more vulnerable `access` system call instead of the more secure one.

In addition to security-related hints, unweighted API importance indicates whether obsolete APIs have been replaced by newer variants. For instance, `wait4` system call is considered obsolete [wait4 man page], and the alternative `waitid` is preferred, as it more precisely specifies which child state changes to wait for. However, unweighted API importance of `wait4` and `waitid` is 60.56% and 0.24%, respectively. This indicates that 60% of the packages are still using the older `wait4` system call. Table 4.9 shows similar trend for some other system calls. Our dataset provides more opportunity for system developers to actively communicate with application developers, in order to speed up the process of retiring problematic APIs.

Some APIs are specific to a particular OS, such as Linux, and often have more portable variants. Table 4.10 shows the comparison between Linux-specific APIs and their generic variants. The results show most developers prefer portable or generic APIs more than Linux-specific APIs. Except `pipe2`, most API variants that are Linux-specific have unweighted API importance lower than 10 percent.

Finally, we consider system calls with multiple variants where one version has increased functionality. Table 4.11 shows the difference between these system calls. Interestingly, more developers chose the less powerful variants, such as using `select` over `pselect6`, or `dup2` over `dup3`. This indicates that more often than not, developers choose simplicity unless a task demands the functionality of a more powerful API variant.

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

Table 4.8: Unweighted API importance of secure and insecure API variations. Higher is more important.

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

Table 4.9: Unweighted API importance of old (generally deprecated) and new (preferred) API variations. Higher is more important.

4.5 Implications for System Developers

The statistics in Section 4.2 can inform decisions of application developers, library developers, and kernel developers. Similarly, the ability to easily generate a comprehensive data set of API footprints has several practical uses.

One practical benefit of this study is the ability to automatically identify a system call profile of every application distributed with Ubuntu/Debian Linux. In fact, we observed that the total 31,433 applications have 11,680 different system call footprint and 9,133 out of these applications have a unique system call footprint. We note that these numbers may vary with dynamic analyses,

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

Table 4.10: Unweighted API importance of other API variants, and comparison between Linux-specific versions and more portable or generic versions. Higher is more important.

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

Table 4.11: Unweighted API importance among similar API variants. Higher is more important.

but the fact that one third of all Debian/Ubuntu applications have a unique system call footprint is interesting.

System call footprints have been explored previously for identifying malware or software compromises [Lam and Chiueh, 2004]. Linux has recently added seccomp, a Berkeley Packet Filter-based system call filtering framework [Seccomp]; generation of seccomp policies can be easily automated using our framework, reducing the system's attack surface in the event of an application compromise.

These tools can also help OS developers evaluate when it is safe to remove a deprecated interface, or when interfaces appear to be irrelevant to most users (e.g., `remap_file_pages`). In the case of an irrelevant interface, this may either indicate something is a candidate for deprecation (e.g., `lookup_dcookie`), or that a useful or important feature (e.g., `faccessat`) is not getting sufficient traction. Linux developers currently wait as long as six years to retire an interface, allowing ample time for application and library developers to change. Our dataset and methodology can allow more proactive outreach and more rapid system evolution.

The libc function call popularity can similarly help library developers to remove function calls that are not used (222 functions). Moreover, the function call importance distribution can also help reduce the library's memory footprint by organizing the in-memory layout by importance.

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 4.12: Implementation of the API usage analysis framework.

4.6 Implementation Details

This section provides additional implementation details of our analysis framework.

Our analysis is based on disassembling binaries inside each application package, using the standard `objdump` tool. This approach eliminates the need for source or recompilation, and can handle closed-source binaries. We implement a simple call-graph analysis to detect system calls reachable from the binary entry point (`e_entry` in ELF headers). We search all binaries, including libraries, for system call instructions (`int 0x80`, `syscall` or `sysenter`) or calling the `syscall` API of libc. We find that the majority of binaries — either shared libraries or executables — do not directly choose system calls, but rather use the GNU C library APIs. Among 66,275 studied binaries, only 7,259 executables and 2,752 shared libraries issue system calls.

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

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

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

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

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

Our implementation is summarized in Table 4.12. We wrote 3,105 lines of code in Python and

2,423 lines of code in SQL (Postgresql). The database contains 48 tables with over 428 Million entries.

4.7 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 5

Optimizing A Legacy OS Functionality

Legacy operating systems, such as Linux, are full of sophisticated logics, designed for fulfilling requirements and properties that are diverse but comparably important. These requirements and properties often include efficiency of system operations (either latency or throughput), implementation of security mechanisms, and preserving OS specifications. This chapter particularly focuses on an OS feature that commonly exists in many systems, and has cost significant development effort for its improvement — the **file system directory cache**. The file system directory cache is a component of the virtual file system (VFS) layer in many operating system kernels. This caching layer has become a ubiquitous optimization to hide access latency for persistent storage technologies, such as a local disk. The directory cache is not exclusively a performance optimization; it also simplifies the implementation of mounting multiple file systems, consistent file handle behavior, and advanced security models, such as SELinux [Loscocco and Smalley, 2001].

Directory caches are essential for good application performance. Many common system calls must operate on file paths, which require a directory cache lookup. For instance, between 10–20% of all system calls in the iBench system call traces do a path lookup [Harter et al., 2011b]. Figure 5.1 lists the fraction of total execution time several common command-line applications spend executing path-based system calls (more details on these applications and the test machine in §5.5). We note that these system calls include work other than path lookup, and that these numbers include some instrumentation overhead; nonetheless, in all cases except `rm`, the system call times and counts are dominated by `stat` and `open`, for which path lookup is a significant component of execution time. For these applications, path-based system calls account for 6–54% of total execution time. This implies that lowering path lookup latency is one of the biggest opportunities for a kernel to improve these applications' execution time.

Unfortunately, even directory cache hits are costly — 0.3–1.1 μS for a `stat` on our test Linux system, compared to only .04 μS for a `getppid` and 0.3 μS for a 4 KB `pread`. This issue is taken particularly seriously in the Linux kernel community, which has made substantial revisions and increasingly elaborate optimizations to reduce the hit cost of its directory cache, such as removing locks from the read path or replacing lock ordering with deadlock avoidance in a retry loop [Corbet, 2009, 2010]. Figure 5.2 plots directory cache hit latency against lines of directory cache code changed over several versions of Linux, using a path-to-inode lookup micro-benchmark on the test system described in § 5.5. These efforts have improved hit latency by 47% from 2011 to 2013, but have plateaued for the last three years.

The root of the problem is that the POSIX path permission semantics seemingly require work that is linear in the number of path components, and severely limit the kernel developer's implementation options. For instance, in order to open file `/X/Y/Z` one must have search permission to

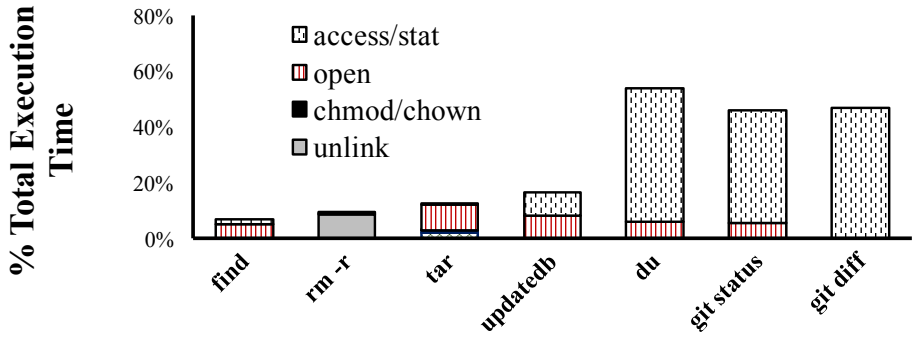


Figure 5.1: Fraction of execution time in several common utilities spent executing path-based system calls with a warm cache, as measured with ftrace.

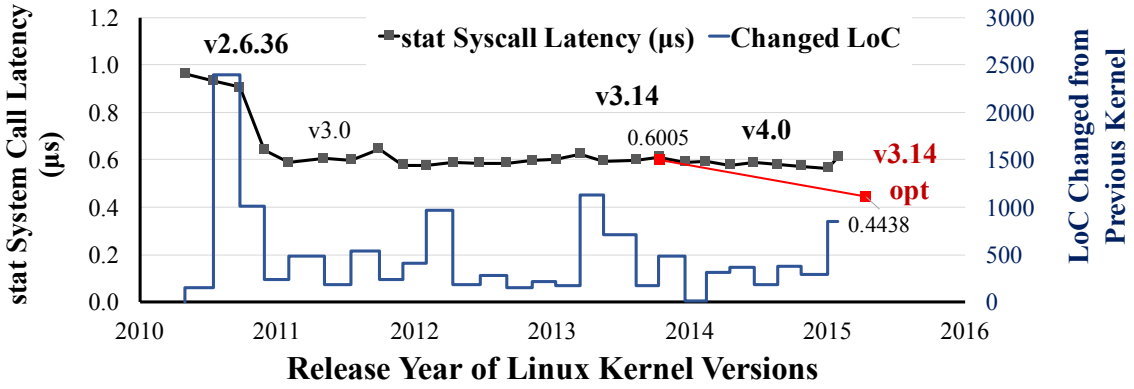


Figure 5.2: Latency of `stat` system call with a long path `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF` on Linux over four years (lower is better), as well as the churn within the directory cache code (all insertions in `dcache.c`, `dcache.h`, `namei.c`, `namei.h` and `namespace.c`). Our optimized 3.14 kernel further reduces `stat` system call latency by 26%.

parent directories `/`, `/X`, and `/X/Y`, as well as permission to access file `Z`. The Linux implementation simply walks the directory tree top-down to check permissions. Unfortunately, when the critical path is dominated by walking a pointer-based data structure, including memory barriers on some architectures for multi-core consistency, modern CPUs end up stalling on hard-to-prefetch loads. Moreover, because so many Linux features are built around this behavior, such as Linux Security Modules (LSMs) [Wright et al., 2002], namespaces, and mount aliases, it is not clear that any data-structural enhancements are possible without breaking backward-compatibility with other Linux kernel features. A priori, it is not obvious that a faster lookup algorithm, such as a single hash table lookup, can meet these API specifications and kernel-internal requirements; to our knowledge, no one has tried previously.

This paper demonstrates that these techniques improve performance for applications that use the directory cache heavily, and the harm is minimal to applications that do not benefit.

5.1 Background of File System Directory Cache

This section first reviews the Unix directory semantics which a directory cache must support; and then explains how directory caches are implemented in modern OSes, including Linux, FreeBSD, Solaris, Mac OS X, and Windows.

5.1.1 Unix Directory Hierarchy Semantics

The most common operation a directory cache performs is a **lookup**, which maps a path string onto an in-memory inode structure. Lookup is called by all path-based system calls, including `open`, `stat`, and `unlink`. Lookup includes a check that the user has appropriate search permission from the process's root or current working directory to the file, which we call a **prefix check**.

For instance, in order for Alice to read `/home/alice/X`, she must have search permission on directories `/`, `/home`, and `/home/alice`, as well as read permission on file `X`. In the interest of frugality, the execute permission bit on directories encodes search permission. Search is distinct from read permission in that search only allows a user to query whether a file exists, but not enumerate the contents (except by brute force) [Ritchie and Thompson, 1974]. SELinux [Loscocco and Smalley, 2001] and other security-hardened Linux variants [AppArmor; Wright et al., 2002], may determine search permission based on a number of factors beyond the execute bit, such as a process's role, or the extended attributes of a directory.

5.1.2 Linux Directory Cache

The Linux directory cache, or **dcache**, caches **dentry** (directory entry) structures, which map a path to an in-memory inode[1] for the file (or directory, device, etc). The inode stores metadata associated with the file, such as size, permissions, and ownership, as well as a pointer to a radix tree that indexes in-memory file contents [Bovet and Cesati, 2005]. Each dentry is tracked by at least four different structures:

- The hierarchical tree structure, where each parent has an unsorted, doubly-linked list of its children.
- A hash table, keyed by parent dentry virtual address and the file name.
- An alias list, tracking the hard links associated with a given inode.
- An LRU list, used to compress the cache as needed.

Linux integrates the prefix check with the lookup itself, searching paths and checking permissions one component at a time. Rather than using the tree structure directly, lookup searches for each component using the hash table. For larger directories, the hash table lookup will be faster than searching an unsorted list of children. The primary use for the hierarchical tree structure is to evict entries bottom-up, in order to uphold the implicit invariant that all parents of any dentry must also be in the cache. Although all dentries are stored in a hash table keyed by path, the permission check implementation looks up each path component in the hash table.

Linux stores **negative dentries**, which cache the fact that a file is known *not* to exist on disk. A common motivating example for negative dentries is searching for a file on multiple paths specified by an environment variable, such as `LD_LIBRARY_PATH`.

[1] Other Unix systems call the VFS-level representation of an inode a vnode.

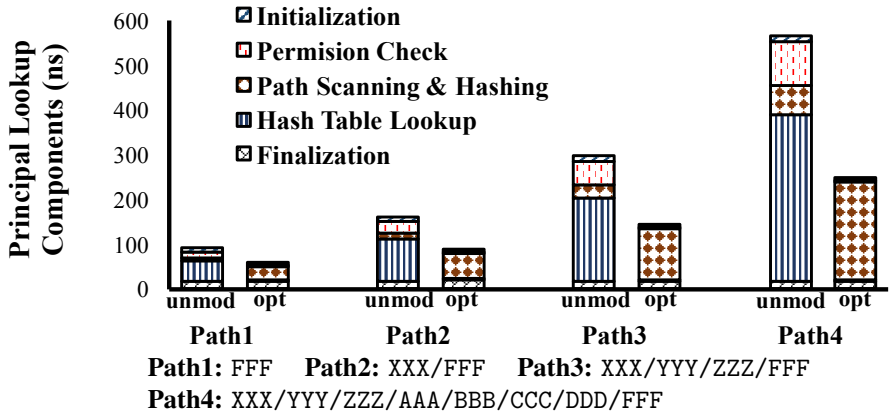


Figure 5.3: Principal sources of path lookup latency in the unmodified and optimzed Linux 3.14 kernel. Lower is better.

Current dcache optimizations. Much of the dcache optimization effort illustrated in Figure 5.2 has improved cache hit latency, primarily by reducing the cost of synchronization in the lookup function with read-copy update (RCU) [McKenney, 2004; McKenney et al.]. RCU eliminates the atomic instructions needed for a read lock and for reference counting individual dentries, pushing some additional work onto infrequent code that modifies the directory structure, such as `rename` and `unlink`.

The most recent Linux kernels also use optimistic synchronization when checking path permissions, using sequence locks (essentially version counters), to detect when the subtree might have changed concurrently with the traversal. If the optimistic fast path fails because of a concurrent modification, the kernel falls back on a slow path that uses hand-over-hand locking of parent and child dentries.

Because the Linux developer community has already invested considerable effort in optimizing its dcache, we use Linux as a case study in this paper. The optimizations in this paper are not Linux-specific, but in some cases build on optimizations that other kernels could adopt.

5.1.3 Opportunities for Improvement

Figure 5.3 shows the time spent in the principal components of a path lookup in Linux, for four paths of increasing lengths. The first-order impact on lookup time is the length of the path itself, which dictates how many times each component will be hashed, looked-up in the hash table, and execute a permission check on each directory's inode. These costs are linear in the number of path components.

The hit latency optimizations described in this paper make most of these operations constant time, except for hashing, which is still a function of the length of the path.

5.2 Minimizing Hit Latency

This section describes algorithmic improvements to the dcache hit path. In the case of a cache hit, one of the most expensive operations is checking whether a process's credentials permit the process to search the path to a dentry top-down (called a **prefix check**). This section shows how

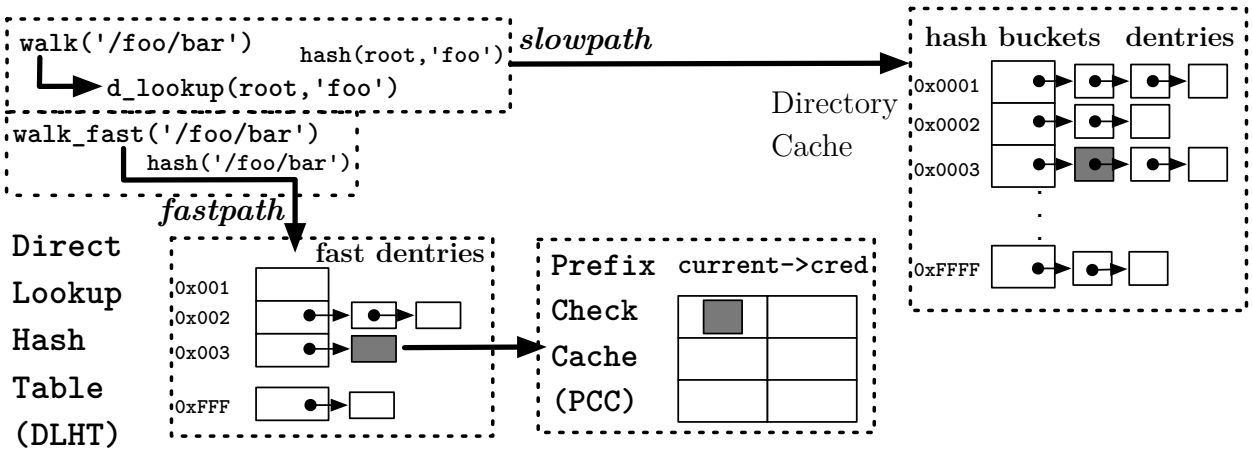


Figure 5.4: Optimized Linux directory cache structure. dentries are chained in hash buckets. To index the hash bucket for a target dentry, the original lookup routine `d_lookup` uses a hashing function with key as a combination of the pointer to parent directory and file name (*slowpath*). Our *fastpath* hashes the full canonical path of target file to look up the dentry in the Direct Lookup Hash Table, and checks the per-credential Prefix Check Cache.

the hit latency can be significantly reduced by caching prefix check results. This section explains the optimization, how it is integrated into the existing Linux directory cache framework, how these cached results are kept coherent with other file system operations, and how we use path signatures to further accelerate lookup.

5.2.1 Caching Prefix Checks

Like many Unix variants, Linux stores cached path-to-inode mappings (dentries) in a hash table (§5.1.2). This hash table is keyed by a combination of the virtual address of the parent dentry and the next path component string, illustrated in Figure 5.4. Virtual addresses of kernel objects do not change over time and are identical across processes.

In practice, prefix checks have a high degree of spatial and temporal locality, and are highly suitable for caching, even if this means pushing some additional work onto infrequent modifications of the directory structure (e.g., `rename` of a directory). RCU already makes this trade-off (§5.1.2).

In order to cache prefix check results, we must first decouple *finding* a dentry from the prefix check. We added a second, system-wide hash table exclusively for finding a dentry, called the **direct lookup hash table (DLHT)**. The DLHT stores recently-accessed dentries hashed by the full, canonicalized path. A dentry always exists in the primary hash table as usual, and may exist in the DLHT. The DLHT is lazily populated, and entries can be removed for coherence with directory tree modifications (§5.2.2).

Each process caches the result of previous prefix checks in a **prefix check cache** (PCC), associated with the process's credentials (discussed further in §5.3.1), which can be shared among processes with identical permissions. The PCC is a hash table that caches dentry virtual addresses and a version number (sequence lock), used to detect stale entries (§5.2.2). When a prefix check passes, indicating that the credentials are allowed to access the dentry, an entry is added to the

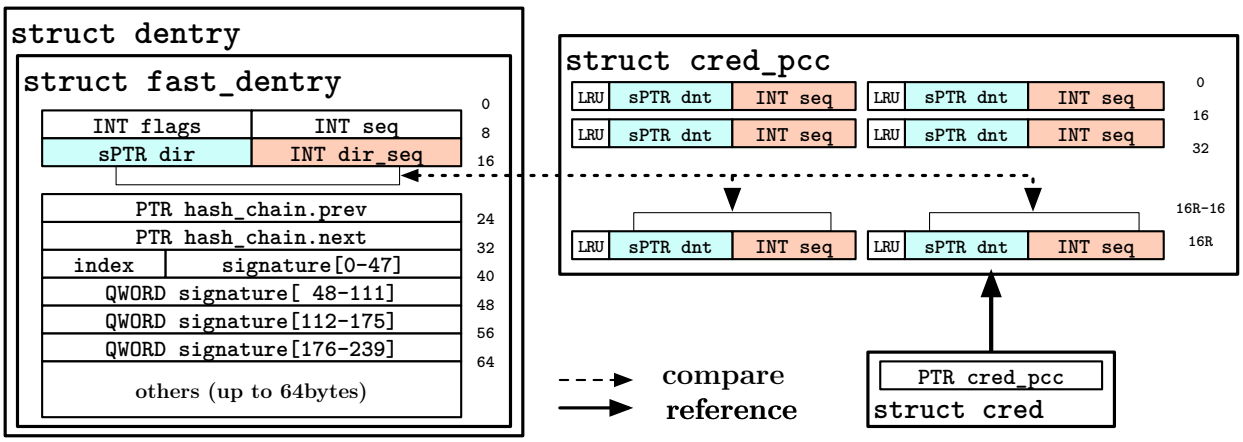


Figure 5.5: Data structures added for fast directory cache lookup. To support *fastpath* lookup, we add a 88-byte fast dentry structure to the original dentry and a variable-sized PCC structure into `cred`.

PCC; entries are replaced according to an LRU policy. A miss in the PCC can indicate a permission denied or that the permission check has not executed recently.

Thus, given any path, the kernel has a *fastpath* that directly looks up the path in the DLHT. If the fastpath hits in the DLHT, the dentry is then looked up in the process's PCC. If a PCC entry is found and the version counter matches the cached counter, the cached prefix check result is used. If the fastpath lookup misses in the DLHT or PCC, or the version counter in the PCC entry is older than the dentry, the code falls back on the original Linux lookup algorithm (the *slowpath*), using the primary hashtable exclusively and traversing one component at a time.

In the case of a relative path, such as `foo/bar` under directory `/home/alice`, we effectively concatenate the relative path and the path of the current working directory. To implement relative paths, Linux already stores a pointer to the `dentry` of the current working directory in each process descriptor (`task_struct`). Rather than `memcpy` the strings, we store the intermediate state of the hash function in each `dentry` so that hashing can resume from any prefix.

The current design includes two very infrequent edge cases. First, a dentry could be freed and reallocated with stale PCC entries. We detect this case by initializing newly allocated dentries with a monotonically increasing version number, allowing PCC entries to detect staleness across reallocation. Freeing a dentry removes it from the DLHT. Second, a version number can wrap around after every 2^{32} initializations of new dentries or renames, chmods, or chowns of non-empty directories; our design currently handles wrap-around by invalidating all active PCCs.

Figure 5.5 illustrates the modifications to the Linux dentry structure. The `fast_dentry` stores the signature, flags, a sequence count, a mount point, lists for managing deep directory entries (§5.4.2), and a list (`hash_chain`) for adding the fast_dentry to a DLHT bucket. The PCC is added to the kernel credential structure (`struct cred`), and stores a tunable number of tuples of dentry pointers and sequence numbers; the system is evaluated with a PCC of 64 KB. Because the highest and lowest bits in each dentry pointer are identical, the PCC only stores the unique pointer bits (8–39 in x86_64 Linux) to save space.

5.2.2 Coherence with Permission and Path Changes

When permissions on a directory or the directory structure are changed, such as with `chmod` or `rename`, any cached prefix checks that include this directory must be invalidated. Our design ensures the safety of concurrent lookups and changes by invalidating relevant PCC and DLHT entries before a change to the hierarchy, preventing stale slowpath lookups from being re-cached, and leveraging VFS-level synchronization to ensure correct slowpath behavior.

First, we ensure that a fastpath lookup cannot complete with stale data after a change to the directory structure. Before a mutation, such as a `rename` or `chmod`, the operation must recursively walk all children in the dcache and increment the `fast_dentry` version counter (`seq`). The `fast_dentry` version counter is used by each process's PCC to detect changes to cached prefix checks on a lookup; incrementing this version counter invalidates all PCC entries for that dentry without directly modifying each PCC. Changes to the directory structure (e.g., `mount` and `rename`) also remove dentries under the old and new path from the direct lookup hash table (DLHT). PCC and DLHT entries are lazily repopulated on the slowpath.

Second, we ensure that the results of a stale slowpath lookup cannot be re-added to the DLHT or PCC by using an atomic, global sequence counter (`invalidation`). The sequence counter is read before and after a slowpath traversal; results are added to the DLHT and PCC only if the counter has not changed, implying no concurrent shootdowns.

Third, we use VFS-level synchronization to ensure that slowpaths synchronize correctly with the mutation. As an example, `rename` acquires both a global `rename_lock` sequence lock, along with per-dentry locks on the old and new parent directory. When the `rename_lock` is held for writing, all lookups on the slowpath (i.e., the current Linux code) must lock each dentry in a hand-over-hand fashion from the root (or current working directory, for relative paths) to the target child. The locks on target dentries obstruct the hand-over-hand traversal until the rename completes. The `invalidation` counter prevents caching the results of slowpath lookups that already passed this point before the dentry locks were acquired. Our implementation follows the VFS's existing locking discipline to avoid deadlocks; it adds version counters that detect inconsistencies and falls back on the slowpath. Thus, relevant PCC and DLHT entries are invalidated before the rename begins, blocking the fastpath; slowpath traversals will block until the rename is complete and the per-dentry locks are released; and a sequence counter ensures that only slowpath traversals that observe the new paths can repopulate the DLHT and PCC.

These recursive traversals shift directory permission and structure changes from constant time to linear in the size of the sub-tree. As one example, to `rename` or `chmod` a directory that has 10,000 descendants with at most depth of 4 takes roughly 330 microseconds to complete. In the original Linux kernel, `rename` and `chmod` are nearly constant-time operations, and only take 4.5 and 1.1 microseconds. A few applications, such as `aptitude` or `rsync`, rely on `rename` to atomically replace a directory, but this is a small fraction of their total work and orders of magnitude less frequent than lookups, making this a good trade-off overall.

Directory References. Unix semantics allow one to `cd` into a directory, and continue working in that directory after a subsequent permission change would otherwise prohibit further accesses. For instance, suppose a process is in working directory `/foo/bar` and `foo`'s permissions change such that the process would not be able to enter `bar` in the future. The process should be able continue to open files under `bar` as long as the process does not leave the directory or exit. Similar semantics

apply to open directory handles. In our design, such a permission change would ultimately result in a blocked PCC entry, and a fastpath lookup would violate the expected behavior. Our design maintains compatibility by checking if the open reference is still permitted in the PCC. If the PCC has a more recent entry that would prevent re-opening this handle, the lookup is forced to take the the slowpath, and this stale result is not added to the PCC.

5.2.3 Accelerating Lookups with Signatures

Our optimized lookup uses 240-bit signatures to minimize the cost of key comparison. Linux looks up dentries in a hash table with chaining. When the hash table key is a relatively short path component, the cost of simply comparing the keys is acceptable. However, a full path on Linux can be up to 4,096 characters, and comparing even modest-length strings can erode the algorithmic benefits of direct lookup. We avoid this cost by creating a signature of the path, which minimizes the cost of key comparisons.

Using signatures introduces a risk of collisions, which could cause the system to map a path onto the wrong dentry. We first explain how signature collisions could cause problems in our design, followed by the required collision resistance properties, and, finally, how we selected the signature size to make this risk vanishingly small.

Signature Collisions. When a user looks up a path, our design first calculates a signature of the canonicalized path, looks up the hash in the global DLHT, and, if there is a hit in the DLHT, looks up the dentry and sequence number in the per-credential PCC.

A user can open the wrong file if the dentry for another file with the same signature is already in the DLHT, and that dentry is in the PCC. For example, if Alice has opened file `/home/alice/foo` with signature X, and then opens file `/home/alice/bar` that also has signature X, her second open will actually create a handle to file `foo`. This creates the concern that a user might corrupt her own files through no fault of her own. This risk can be configured to be vanishingly small based on the signature size (discussed below).

Any incorrect lookup result must be a file that the process (or another process with the same credentials) has permission to access. For a fastpath lookup to return anything, a matching dentry pointer must be in the task's PCC, which is private to tasks with the same credentials. Thus, a collision will not cause Alice to accidentally open completely irrelevant files that belong to Bob, which she could not otherwise access.

Our design correctly handles the case where two users access different files with the same signature, because misses in the PCC will cause both users to fall back on the slowpath. Suppose Bob has opened `foo`, which collides with Alice's `bar`. When Alice opens `bar`, its signature will match in the DLHT, but will miss in the PCC. This causes Alice's lookup to take the slowpath to re-execute the prefix check, ultimately opening the correct file and adding this dentry to her PCC. Thus, if Bob were adversarial, he cannot cause Alice to open the wrong file by changing dcache-internal state.

We choose a random key at boot time for our signature hash function, mitigating the risk of deterministic errors or offline collision generation, as one might use to attack an application that opens a file based on user input, such as web server. Thus, the same path will not generate the same signature across reboots or instances of the same kernel.

Despite all of these measures, this risk may still be unacceptable for applications running as

root, which can open any file, especially those that accept input from an untrusted user. For example, suppose a malicious user has identified a path with the same signature as the password database. This user might pass this path to a setuid-root utility and trick the setuid utility into overwriting the password database. This risk could be eliminated by disallowing signature-based lookup acceleration for privileged binaries or security-sensitive path names, although this is not implemented in our prototype.

Collision Resistance Requirements. The security of our design hinges on an adversary only being able to find collisions through brute force. Our design can use either a 2-universal hash function or a pseudorandom function family (PRF) to generate path signatures. In terms of collision resistance, the difference between a 2-universal hash and a PRF is that the adversary can potentially learn the secret key by observing the outputs of the 2-universal function, but cannot learn the key from the outputs of a PRF. Because our dcache design does not reveal the signatures to the user, only whether two paths have a signature collision, a hash function from either family is sufficient.

One caveat is that, with a 2-universal hash function, one must be careful that timing and other side channels do not leak the signature. For example, one cannot use bits from the signature to also index the hash table, as one might learn bits of the signature from measuring time to walk the chain on a given hash bucket. In the case of our selected function, one can safely use the lower bits from the 256-bit hash output, as lower bits are not influenced by the values in higher bits in our particular algorithm; we thus use a 16 bit hash table index and a 240-bit signature. In contrast, when the signature is generated with a PRF, concerns about learning the signature from side channels are obviated.

Our design uses the 2-universal multilinear hash function [Lemire and Kaser, 2013]. We did several experiments using PRFs based on the AES-NI hardware, and could not find a function that was fast enough to improve over baseline Linux. Using current 128-bit AES hardware, we could improve performance at 4 or more path components, but creating a 256-bit PRF required a more elaborate construction that is too expensive. A more cautious implementation might favor a PRF to avoid any risk of overlooked side channels, especially if a fast, 256-bit PRF becomes available in future generations of hardware.

Probability of Signature Collision. We selected a 240-bit signature, which is comparable to signature sizes used in data deduplication systems, ranging from 128–256 bits. Deduplication designs commonly select a signature size that introduces a risk of collisions substantially less than the risk of undetected ECC RAM errors [Debnath et al., 2010; Quinlan and Dorward, 2002; Srinivasan et al., 2012; Zhu et al., 2008].

We assume an adversary that is searching for collisions by brute force. This adversary must lookup paths on the system, such as by opening local files or querying paths on a web server. Because our hash function is keyed with a random value and the output is hidden from the user, the adversary cannot search for collisions except on the target system. Thus, the adversary is limited by the rate of lookups on the system, as well as the capacity of the target system to hold multiple signatures in cache for comparison.

We calculate the expected time at which the risk of a collision becomes non-negligible (i.e., higher than 2^{-128}) and model the risk of collision as follows. First, $|H(X)| = 2^{240}$ is the number of possible signatures. We limit the cache to $n = 2^{35}$ entries (i.e., assuming 10TB of dcache space in RAM and 320 bytes per entry), with an LRU replacement policy. We calculate the number of

queries (q) after which the risk of a collision is higher than $P = 2^{-128}$ as follows:

$$q \simeq ln(1-p) * \frac{|H(x)|}{-n} \simeq ln(1-2^{-128}) * \frac{2^{240}}{-2^{35}} \simeq 2^{77}$$

At a very generous lookup rate of 100 billion per second (current cores can do roughly 3 million per second), the expected time at which the probability of a brute-force collision goes above 2^{-128} is 48 thousand years.

5.3 Generalizing the Fast Path for Lookup

Thus far, we have explained our fast path optimization using the relatively straightforward case of canonical path names. This section explains how these optimizations integrate with Linux's advanced security modules, as well as how we address a number of edge cases in Unix path semantics, such as mount options, mount aliases, and symbolic links.

5.3.1 Generalizing Credentials

Linux includes an extensible security module framework (LSMs [Wright et al., 2002]), upon which SELinux [Loscocco and Smalley, 2001], AppArmor [AppArmor], and others are built. An LSM can override the implementation of search permission checks, checking customized attributes of the directory hierarchy or process. Thus, our dcache optimizations must still work correctly even when an LSM overrides the default access control rules.

Our approach leverages the `cred` structure in Linux, which is designed to store the credentials of a process (`task_struct`), and has several useful properties. First, a `cred` struct is comprehensive, including all variables that influence default permissions, and including an opaque `security` pointer for an LSM to store metadata. Second, a `cred` is copy-on-write (COW), so when a process changes its credentials, such as by executing a `setuid` binary or changing roles in SELinux, the `cred` is copied. We manually checked that AppArmor and SELinux respect the COW conventions for changes to private metadata. Moreover, a `cred` can be shared by processes in common cases, such as a shell script forking children with the same credentials. Thus, the `cred` structure meets most of our needs, with a few changes, which we explain below.

We store cached prefix checks (§5.2.1) in each `cred` structure, coupling prefix check results with immutable credentials. New `cred` structures are initialized with an empty PCC. When more processes share the PCC, they can further reduce the number of slowpath lookups.

One challenge is that Linux often allocates new `cred` structures *even when credentials do not change*. The underlying issue is that COW behavior is not implemented in the page tables, but rather by convention in code that *might* modify the `cred`. In many cases, such as in `exec`, it is simpler to just allocate another `cred` in advance, rather than determine whether the credentials will be changed. This liberal allocation of new `cred`s creates a problem for reusing prefix cache entries across child processes with the same credentials. To mitigate this problem, we wait until a new `cred` is applied to a process (`commit_creds()`). If the contents of the `cred` did not change, the old `cred` and PCC is reused and shared.

Our `cred` approach memoizes complex and potentially arbitrary permission evaluation functions of different LSMs.

5.3.2 Non-Canonical Paths and Symbolic Links

Our optimized hash table is keyed by full path. However, a user may specify variations of a path, such as `/X/Y/./Z` for `/X/Y/Z`. Simple variations are easily canonicalized in the course of hashing.

A more complex case is where, if `/X/L` is a symbolic link, the path `/X/L/../Y` could map to a path other than `/X/Y`. Similarly, if the user doesn't have permission to search `/X/Z`, a lookup of `/X/Z/../Y` should fail even if user has permission to search `/X/Y`. In order to maintain bug-for-bug compatibility with Linux, our prototype issues an additional fastpath lookup at each dot-dot to check permissions. Maintaining Unix semantics introduces overhead for non-canonical paths.

We see significantly higher performance by using Plan 9's *lexical* path semantics [Pike, 2000]. Plan 9 minimized network file system lookups by pre-processing paths such as `/X/L/../Y` to `/X/Y`. We note that Plan 9 does component-at-a-time lookup, but does not have a directory cache.

Symbolic Links. We resolve symbolic links on our lookup fastpath by creating dentry aliases for symbolic links. For instance, if the path `/X/L` is an alias to `/X/Y`, our kernel will create dentries that redirect `/X/L/Z` to `/X/Y/Z`. In other words, symbolic links are treated as a special directory type, and can create children, caching the translation.

Symbolic link dentries store the 240-bit signatures that represent the target path. The PCC is separately checked for the target dentry. If a symbolic link changes, we must invalidate all descendant aliases, similar the invalidation for a directory `rename`. This redirection seamlessly handles the cases where permission changes happen on the translated path, or the referenced dentries are removed to reclaim space.

5.3.3 Mount Points

Our fastpath handles several subtle edge cases introduced by mount points.

Mount options. Mount options, such as read-only or nosuid, can influence file access permission checks. The Linux dcache generally notices mount points as part of the hierarchical file system walk, and checks for permission-relevant mount flags inline. Once this top-down walk is eliminated, we need to be able to identify the current mount point for any given dentry. We currently add a pointer to each dentry, although more space efficient options are possible.

Mount Aliases. Some pseudo file systems, such as `proc`, `dev`, and `sysfs`, can have the same *instance* mounted at multiple places. This feature is used by `chroot` environments and to move these file systems during boot. A `bind` mount can also create a mount alias.

In our system, a dentry only stores one signature and can only be in the direct lookup hash table by one path at a time. Our current design simply picks the most recent to optimize—favoring locality. If a slowpath walk notices that the matching dentry (by path) has a different signature, is under an aliased mount, and is already in the DLHT, the slowpath will replace the signature, increment the dentry version count, and update the pointer to the dentry's mount point. The version count increment is needed in case the aliased paths have different prefix check results. This approach ensures correctness in all cases, and good performance on the most recently used path for any mount-aliased dentry.

Mount Namespaces. Mount namespaces in Linux allow processes to create private mount points, including `chroot` environments, that are only visible to the process and its descendants. When a

process creates a new mount namespace, it also allocates a new, namespace-private direct lookup hash table. The slowpath always incorporates any mount redirection, and any new signature-to-dentry mappings will be correct within the namespace. Thus, the same path (and signature) inside a namespace will map to a different dentry than outside of the namespace. Similarly, the prefix check cache (PCC) will always be private within the namespace.

As with mount aliases, we only allow a dentry to exist on one direct lookup hash table at a time. This favors locality, and makes the invalidation task tractable when a renamed directory is shared across many namespaces. The invalidation code used for directory tree modifications simply evicts each child dentry from whatever DLHT it is currently stored in.

Network File Systems. Our prototype does not support direct lookup on network file systems, such as NFS versions 2 and 3 [Sandberg et al., 1985]. In order to implement close-to-open consistency on a stateless protocol, the client must revalidate all path components at the server—effectively forcing a cache miss and nullifying any benefit to the hit path. We expect these optimizations could benefit a stateful protocol with callbacks on directory modification, such as AFS [Howard et al., 1988] or NFS 4.1 [Shepler et al., 2010].

5.4 Improving the Hit Rate

The previous sections explain how changes to the structure of the dcache can lower the average hit latency, through algorithmic improvements. This section identifies several simple changes that can improve the hit rate. In the case of a dcache miss, the low-level file system is called to service the system call. At best, the on-disk metadata format is still in the page cache, but must be translated to a generic format; at worst, the request blocks on disk I/O. Although not every application heavily exercises these cases with unusually low hit rates, the evaluation shows several widely-used applications that substantially benefit from these optimizations.

5.4.1 Caching Directory Completeness

Although the Linux dcache tracks the hierarchical structure of directories, it has no notion of whether a directory's contents are completely or partially in the cache. Suppose Alice creates a new directory `X` on a local file system; if her next system call attempts to create file `X/Y`, the dcache will miss on this lookup and ask the low-level file system if `X/Y` exists. This overhead can be avoided if the VFS tracks that all directory contents are in the cache.

A second example is `readdir`, which lists the files in a directory, along with their inode number and their types, such as a regular file, character device, directory, or symbolic link. In the current VFS `readdir` operation, the low-level file system is always called, *even if the entire directory is in cache*. For directories too large to list in the user-supplied buffer, `readdir` may be called multiple times, storing an offset into the directory. To construct this listing, the low-level file system must reparse and translate the on-disk format, and may need to read the metadata block from disk into the buffer cache. As a result, `readdir` is generally an expensive file system operation, especially for large directories.

We observe that repeatedly listing a directory is a common behavior in file systems. For example, a user or a shell script may repeatedly run the `ls` command in a directory. Some applications coordinate state through directory contents, requiring frequent and repeated directory listings. For

example, `maildir` is a popular email back-end storage format [Bernstein], yielding better performance scalability than the older `mbox` format. Maildir stores each inbox or subfolder as a directory, and each individual message is a file within the directory. File names encode attributes including flags and read/unread status. If a message changes state, such as through deletion or being marked as read, the IMAP server server will rename or unlink the file, and reread the directory to sync up the mail list. Similarly, a mail delivery agent (MDA), running as a separate process, may concurrently write new messages into the directory, requiring the IMAP server to monitor the directory for changes and periodically re-read the directory's contents.

Our Linux variant caches `readdir` results returned by the low-level file system in the directory cache. If all of a directory's children are in the cache, the dentry is marked with a new `DIR_COMPLETE` flag. This flag is set upon creation of a new directory (`mkdir`), or when a series of `readdir` system calls completes without an `lseek()` on the directory handle or a concurrent eviction of any children to reclaim space. We note that concurrent file creations or deletions interleaved with a series of `readdir`s will still be in the cache and yield correct listing results. After setting the `DIR_COMPLETE` flag, subsequent `readdir` requests will be serviced directly from the dentry's child list. Once a directory enters the complete state, it leaves this state only if a child dentry is removed from the cache to reclaim space.

One caveat to this approach is that readdir returns part of the information that would normally appear in an inode, but not enough to create a complete inode. For these files or subdirectories, we add dentries without an inode as children of the directory. These dentries must be separated from negative dentries when they are looked up, and be linked with a proper inode. This approach allows `readdir` results to be used for subsequent `lookup`s, cleanly integrates with existing dcache mechanisms, and gets the most possible use from every disk I/O without inducing I/O that was not required.

We note that Solaris includes a similar complete directory caching mode [McDougall and Mauro, 2008], but it is not integrated with readdir or calls other than lookup, is a separate cache (so the same dentries can be stored twice, and both hash tables must be checked before missing), and the comments indicate that it only has performance value for large directories. Our results demonstrate that, when properly integrated into the directory cache, tracking complete directories has more value than previously thought.

File Creation. Directory completeness caching can also avoid compulsory misses on new file creation. Although negative dentry caching works well for repeated queries for specific files that do not exist, negative dentries are less effective when an application requests *different* files that do not exist. A common example of unpredictable lookups comes from secure temporary file creation utilities [CERT Secure Coding]. In our prototype, a miss under a directory with the `DIR_COMPLETED` flag is treated as if a negative dentry were found, eliding this compulsory miss. In our current implementation, this flag will only be set in a directory that has been read or newly created, but other heuristics to detect frequent misses for negative dentries and to load the directory may also be useful.

5.4.2 Aggressive Negative Caching

Negative dentries cache the fact that a path does not exist on disk. This subsection identifies several opportunities for more aggressive use of negative dentries, some of which work in tandem

with direct lookup.

Renaming and Deletion. When a file is renamed or unlinked, the old path can be converted to a negative dentry. Although Linux does convert a cached, but unused dentry to a negative dentry on unlink, this is not the case for `rename` and `unlink` of a file that is still in use. We extend these routines to keep negative dentries after a file is removed, in the case that the path is reused later, as happens with creation of lock files or Emacs's backup ("tilde") files.

Pseudo File Systems. Pseudo file systems, like `proc`, `sys`, and `dev`, do not create negative dentries for searched, nonexistent paths. This is a simplification based on the observation that disk I/O will never be involved in a miss. Because our fastpath is still considerably faster than a miss, negative dentries can be beneficial even for in-memory file systems, accelerating lookup of frequently-searched files that do not exist.

Deep Negative Dentries. Finally, we extended the direct lookup fastpath (§5.2) with the ability to create "deep" negative dentries. Consider the case where a user tries to open `/X/Y/Z/A`, and `/X/Y/Z` does not exist. In the slowpath, the lookup will fail when it hits the first missing component, and it is sufficient to only cache a negative dentry for `Z`. Repeated lookups for this path will never hit on the fastpath, however, because there is no entry for the full path.

In order for this case to use the fastpath, we allow negative dentries to create negative children, as well as deep children. In other words, we allow negative dentry `/X/Y/Z` to create children `A` and `A/B`, which can service repeated requests for a non-existent path. If a file is created for a path that is cached as negative, and the file is not a directory, any negative children are evicted from the cache.

We also create deep negative dentries under regular files to capture lookup failures that return `ENOTDIR` instead of `ENOENT`. This type of lookup failure happens when a filename is use as if it were a directory, and a path underneath is searched. For example, if `/X/Y/Z` is a regular file, and a user searches for `/X/Y/Z/A`, the Linux kernel will return `ENOTDIR` and never create a negative dentry. We optimize this case with a deep, `ENOTDIR` dentry.

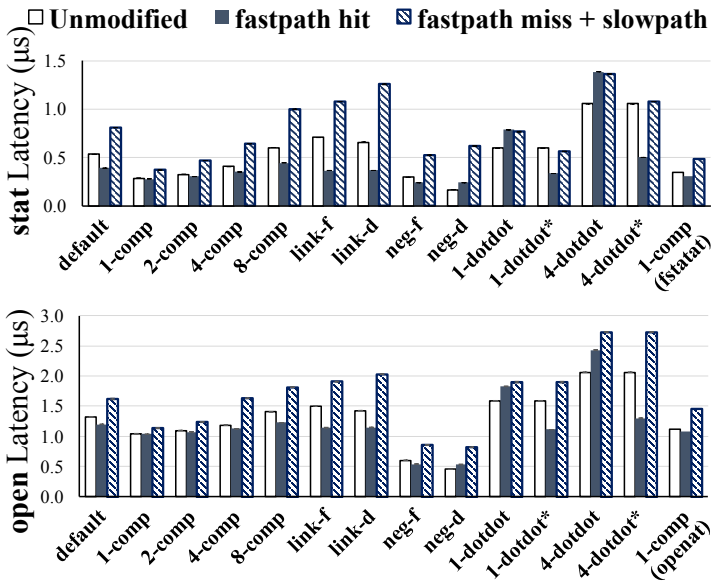
5.5 Evaluation of the Optimized Directory Cache

This section evaluates our directory cache optimizations, and seeks to answer the following questions:

1. How much does each optimization — the lookup fastpath, whole directory caching, and more aggressive negative dentries — improve application performance?
2. How difficult are the changes to adopt, especially for individual file systems?

The evaluation includes both micro-benchmarks to measure the latency of file system related system calls in best-case and worst-case scenarios, and a selection of real-world applications to show potential performance boost by our solution in practice.

All experiment results are collected on a Supermicro Super Server with a 12-core 2.40 GHz Intel Core Xeon CPU, 64GB RAM, and a 2 TB, 7200 RPM ATA disk, formatted as a journaled `ext4` file system, configured with a 4096-byte block size. The OS is Ubuntu 14.04 server, Linux kernel 3.14. All measurements are a mean of at least 6 runs (for the longer-running experiments); most measurements are hundreds or thousands of runs, as needed to ensure a consistent average. Tables and graphs indicate 95% confidence intervals with "+/-" columns or error bars.



(*** represents Plan 9's lexical parent semantics**)
default: `/usr/include/ ... /sys/types.h`
1-comp: `FFF` **2-comp:** `XXX/FFF`
4-comp: `XXX/YYY/ZZZ/FFF`
8-comp: `XXX/YYY/ZZZ/AAA/BBB/CCC/DDD/FFF`
link-f: `XXX/YYY/ZZZ/LLL -> FFF`
link-d: `LLL/YYY/ZZZ/FFF -> XXX/YYY/ZZZ/FFF`
neg-f: `XXX/YYY/ZZZ/NNN (NNN not found)`
neg-d: `NNN/XXX/YYY/FFF (NNN not found)`
1-dotdot: `XXX/../FFF`
4-dotdot: `XXX/YYY/../../AAA/BBB/../../FFF`

Figure 5.6: System call `stat` and `open` latency for micro-benchmark (`lat_syscall` in `LMBench`), based on different path patterns. We include a synthetic evaluation of always missing on the fastpath and falling back to the slowpath, and a comparison with Plan 9's lexical parent semantics, where appropriate. Lower is better.

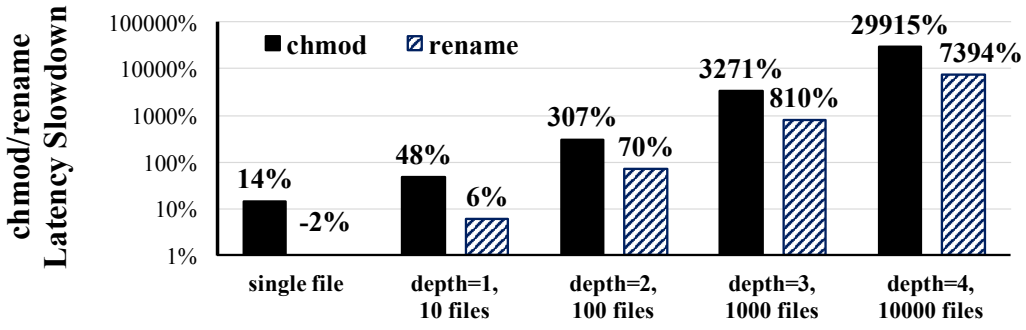
5.5.1 File Lookup Optimizations

Micro-benchmarks. We use an extended LMBench 2.5 UNIX microbenchmark suite [McVoy and Staelin, 1996] to evaluate latency of path lookup at the system call level. Figure 5.6 shows the latency to `stat` and `open` sample paths with various characteristics, including varying lengths, symbolic links, parent (dot dot) directories, and files that are not found.

The primary trend we observe is that, as paths have more components, the relative gain for our optimization increases. For a single component file, `stat` gains 3% and open is equivalent to baseline Linux. For longer paths, the gain increases up to 26% and 12%, respectively.

To evaluate the worst case, we include a set of bars, labeled "fastpath miss + slowpath", which exercise the fast path code, but the kernel is configured to always miss in the PCC. This simulates the full cost of executing the optimized fastpath unsuccessfully, and then walking the O(n) slowpath in the cache. This case does not miss all the way to the low-level file system. The overhead typically ranges from 12–93%, except for path neg-d. In the case of neg-d, the first component is missing, and a component-at-a-time walk would stop sooner than a direct lookup. In general, the neg-d case would be mitigated by deep negative dentries. In practice, these overheads would only be observed for compulsory misses in the dcache, or by an application that exhibits an extreme lack of locality.

We next compare the costs of default Linux parent ("dot dot") semantics to Plan 9's lexical semantics. Enforcing Linux semantics for a path with parent references causes our optimizations to perform roughly 31% worse than unmodified Linux, as this requires an extra lookup per parent. Lexical path semantics, on the other hand, allow our optimization to continue using a single lookup, improving performance by 43–52%. Lexical path semantics have an independent benefit, and



	μS	+/-	μS	+/-	μS	+/-	μS	+/-	μS	+/-
`chmod`	1.86	.00	1.60	.00	4.37	.00	36.38	.01	323	.02
`rename`	3.73	.00	4.68	.00	7.51	.00	40.14	.01	330	.05

Figure 5.7: `chmod` and `rename` latency in directories of various depths and sizes. Lower is better.

could reduce the number of components to walk in a lookup in unmodifed Linux. Although this difference is large, our test applications do not heavily use parent directory pointers, and are not sensitive to this difference.

Caching the resolution of a symbolic link improves performance for paths link-f and link-d by 44% and 48%, respectively. This improvement is insensitive to where in the path the link occurs, as both link-f and link-d walk the same number of components (link-d maps "LLL" onto "XXX").

For files that do not exist (negative dentries), we see comparable improvements to paths that are present. The one exception is long paths that don't exist under a directory early in the path. We believe this case is rare, as applications generally walk a directory tree top-down, rather than jumping several levels into a non-existent directory. In this situation (path neg-d), baseline Linux would stop processing the path faster than our optimization can hash the entire path, even with caching deep negative dentries. Nonetheless, deep negative dentries are an important optimization: without them, stat of path neg-d would be 113% worse and open would be 43% worse than unmodified Linux, versus 38% and 16% slower with deep negative dentries.

Linux also includes `*at()` system call variants, which operate under a working directory—typically using only a single component. Commensurate with the results above, `fstatat()` benefits from our optimizations by 12% for a single path component, and `openat()` is 4% faster than unmodified Linux. Some applications use multiple-component names in conjunction with an `*at` call; in these cases, the benefit of our optimization is proportional to the path length.

To evaluate the overhead of updating directory permissions and changing the directory structure, we measure `chmod` and `rename` latency. In our solution, the main factor influencing these overheads are the number of children in the cache (directory children out-of-cache do not affect performance). Figure 5.7 presents performance of `chmod` and `rename` on directories with different depths and directory sizes. In general, the cost of a rename or chmod increases dramatically with the number of children, whereas baseline Linux and ext4 make these constant-time operations. Even with 10,000 children all in cache, the worst-case latency is around 330 μS. As a point of reference, the Linux 3.19 source tree includes 51,562 files and directories. Initial feedback from several Linux file system maintainers indicate that this trade would be acceptable to improve lookup performance [Farrow, 2015].

Applications	Path Stats		Unmodified kernel				Optimized kernel		
	l	#	s	+/-	hit%	neg%	s	+/-	Gain
`find -name`	39	1	.055	.000	100.0	.18	.044	.000	19.2 %
`tar xzf linux.tar.gz`	22	3	4.039	.024	84.2	.06	4.038	.010	.05 %
`rm -r linux src`	24	3	.607	.008	100.0	.01	.621	.020	-2.32 %
`make linux src`	29	4	868.079	.647	91.2	17.84	868.726	.892	-.07 %
`make -j12 linux src`	29	4	102.958	.597	92.9	20.03	103.308	.288	-.34 %
`du -s linux src`	10	1	.070	.000	100.0	.01	.061	.012	12.65 %
`updatedb -U usr`	3	1	.011	.000	99.9	.00	.008	.000	29.12 %
`git status linux src`	16	4	.176	.000	100.0	.05	.168	.000	4.26 %
`git diff linux src`	16	4	.066	.000	100.0	1.49	.060	.000	9.89 %

Table 5.1: Execution time and path statistics of real-world applications bounded by directory cache lookup latency. Warm cache case. Hit rate and negative dentry rate are also included. The average path length in bytes (l) and components (#) are presented in the first two columns. Lower is better.

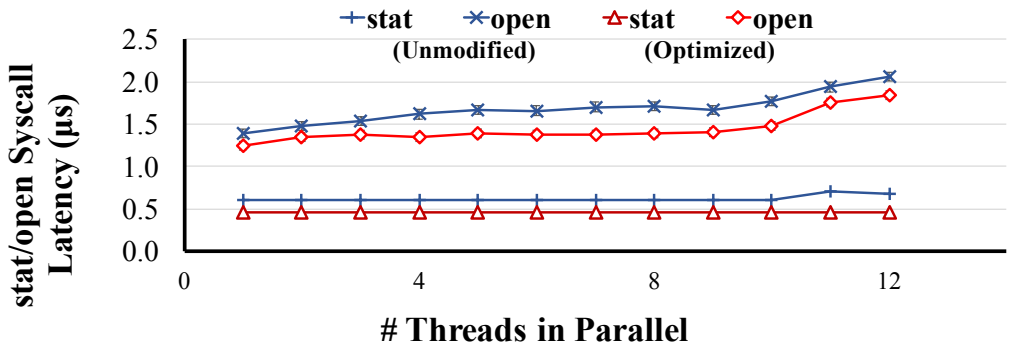


Figure 5.8: Latency of `stat` and `open` (of the same path) as more threads execute in parallel. Lower is better.

Space Overhead. Our prototype increases the size of a dentry from 192 bytes to 280 bytes. Our design also introduces a per-credential PCC of size 64 KB, and a second, global hash table (the DLHT), which includes 2^{16} buckets. Because Linux does not place any hard limits on dcache size, except extreme under memory pressure, it is hard to normalize execution time to account for the space cost. On a typical system, the dcache is tens to hundreds of MB; increasing this by 50% is likely within an acceptable fraction of total system memory. Alternatively, if one were to bound the total dcache size, this induces a trade-off between faster hits and fewer hits. We leave exploration of these trade-offs for future work.

Scalability. Figure 5.8 shows the latency of a `stat`/`open` on the same path as more threads execute on the system. The read side of a lookup is already linearly scalable on Linux, and our optimizations do not disrupt this trend—only improve the latency. The `rename` system call introduces significant contention, and is less scalable in baseline Linux. For instance, a single-file, single-core rename takes 13 μS on our test system running unmodified Linux; at 12 cores and different paths, the average latency jumps to 131 μS for our optimized kernel, these numbers are 18 and 118 μS, respectively, indicating that our optimizations do not make this situation worse for renaming a file. As measured in Figure 5.7, our optimizations do add overhead to renaming a large directory, which would likely exacerbate this situation.

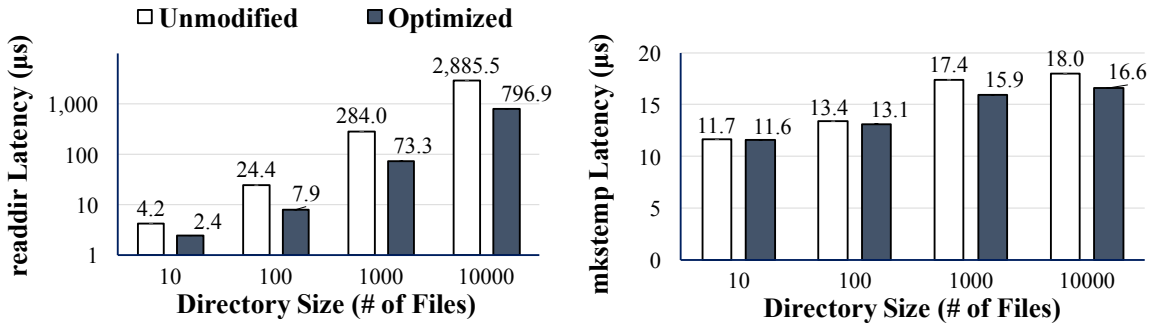


Figure 5.9: Latency in logscale for `readdir` function calls, and latency in microsecond for `mkstemp` function calls, on directories with different sizes. Lower is better.

5.5.2 Caching Directory Completeness

Figure 5.9 shows the latency of a `readdir` micro-benchmark with varying directory sizes. The ability to cache `readdir` results improves performance by 46–74%. Caching helps more as directories get larger. OpenSolaris comments indicate that this idea was only beneficial in UFS for directories with at least 1,024 entries [2]. Our result indicates that there is benefit even for directories with as few as 10 children.

Figure 5.9 also shows the latency of creating a secure, randomly-named file in directories of varying size. We measure from 1–8% improvement for the `mkstemp` library. Although most applications' execution times are not dominated by secure file creation, it is a common task for many applications, and of low marginal cost.

5.5.3 Applications

Command-Line Applications. The improvement applications see from faster lookup is, of course, proportional to the fraction of runtime spent issuing path-based system calls as well as the amount of time listing directories. We measure the performance of a range of commonly-used applications. In most cases, these applications benefit substantially from these optimizations; in the worst case, the performance harm is minimal. The applications we use for benchmarking include:

- `find`: search for a file name in the Linux source directory.
- `tar xzf`: decompress and unpack the Linux source.
- `rm -r`: remove the Linux source tree.
- `make` and `make -j12`: compile the Linux kernel.
- `du -s`: Recursively list directory size in Linux source.
- `updatedb`: rebuild database of canonical paths for commonly searched file names in /usr from a clean `debootstrap`.
- `git status` and `git diff`: display status and unstaged changes in a cloned Linux kernel git repository.

For each application we test, we evaluate the performance in both cases of a warm cache (Table 5.1) and a a cold cache (Table 5.2). To warm the cache, we run the experiment once and drop the first run. For the warm cache tests, we also provide statistics on the path characteristics of each

[2] see line 119 of `fs/ufs/ufs_dir.c` in OpenSolaris, latest version of frozen branch onnv-gate.

App	Unmodified kernel				Optimized kernel		
	s	+/-	hit%	neg%	s	+/-	Gain
`find`	1.39	.01	38	1	1.35	.02	3.1 %
`tar`	4.00	.10	85	0	3.98	.04	.5 %
`rm -r`	1.81	.05	83	1	1.84	.06	-1.4 %
`make`	885.33	.31	100	45	883.88	2.03	.2 %
`make -j12`	114.51	.60	100	47	114.54	.89	.2 %
`du -s`	1.49	.01	6	0	1.46	.02	2.3 %
`updatedb`	.73	.01	34	0	.74	.01	-2.1 %
`git status`	4.13	.03	62	2	4.11	.03	.7 %
`git diff`	.84	.01	61	0	.86	.01	-14.0 %

Table 5.2: Execution time, hit rate, and negative dentry rate for real-world applications bounded by directory cache lookup latency with a cold cache. Lower is better.

application.

Perhaps unsurprisingly, metadata-intensive workloads benefit the most from our optimizations, such as `find` and `updatedb`, as high as 29% faster. Note that `find`, `updatedb`, and `du` use the `*at()` APIs exclusively, and all paths are single-component; these gains are attributable to both improvements to lookup and directory completeness caching.

We note that the performance of directory-search workloads is sensitive to the size of PCC; when we run `updatedb` on a directory tree that is twice as large as the PCC, the gain drops from 29% to 16.5%. This is because an increased fraction of the first lookup in a newly-visited directory will have to take the slowpath. Our prototype has a statically-set PCC size and we evaluate with a PCC sufficiently large to cache most of the relevant directories in the warm cache experiments. We expect that a production system would dynamically resize the PCC up to a maximum working set; we leave investigating an appropriate policy to decide when to grow the PCC versus evict entries for future work.

The application that is improved primarily by our hit optimization is `git`, which shows a gain of 4–9.9%. Cases that are dominated by other computations, such as a Linux compile, show minimal ($\leq 2.3\%$ slowdown). In the cold cache cases, all gains or losses are roughly within experimental noise, indicating that these optimizations are unlikely to do harm to applications running on a cold system. In general, these results affirm that common Linux applications will not be harmed by the trade-offs underlying our optimizations, and can benefit substantially.

Table 5.1 also indicates statistics about these workloads on unmodified Linux. In general, each path component tends to be roughly 8 characters, and *at-based application generally lookup single-component paths, whereas other applications typically walk 3–4 components. Our statistics also indicate that, with a warm cache, these applications should see 84–100% hit rate in the cache, so optimizing the hit path is essential to performance. Finally, make is the only application with a significant fraction of negative dentries (roughly 20%), which is to be expected, since it is creating new binary files.

Server Applications. An example of software that frequently uses `readdir` is an IMAP mail server using the MailDir storage format. We exercise the Dovecot IMAP server by creating 10 mailboxes for a client. We use a client script to randomly select messages in different mailboxes and mark them as read, flagged, or unflagged. Internally, marking a mail causes a file to be re-named, and the directory to be re-read. To eliminate network latency, we run network tests on the

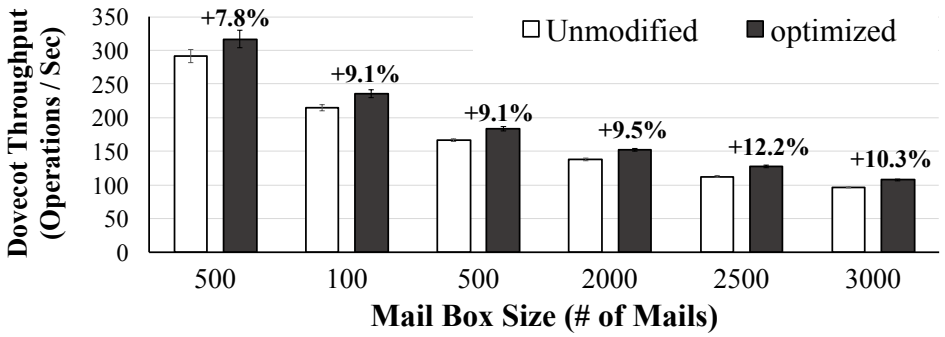


Figure 5.10: Throughput for marking and unmarking mail on the Dovecot IMAP server. Higher is better.

# of files	Unmodified kernel		Optimized kernel		
	Req/s	+/-	Req/s	+/-	Gain
10	27,638.23	43.31	31,491.98	55.24	12.24 %
10^2	7,423.86	11.81	7,934.07	12.76	6.43 %
10^3	1,017.02	0.58	1,081.02	0.38	5.92 %
10^4	99.03	0.11	110.14	0.10	10.09 %

Table 5.3: Throughput of downloading generated directory listing pages from an `Apache` server. Higher is better.

localhost; in a practical deployment, network latency may mask these improvements to the client, but the load of the server will still be reduced.

Figure 5.10 shows the throughput for the Dovecot mail server on both kernels; improvements range from 7.8–12.2%. Commensurate with the `readdir` microbenchmark, larger directories generally see larger improvement, plateauing at a 10% gain. We similarly exercise the Apache web server's ability to generate a file listing using the Apache benchmark (Table 5.3). These pages are not cached by Apache, but generated dynamically for each request. This workload also demonstrates improvements in throughput from 6–12%. Overall, these results show that the readdir caching strategy can reduce server load or improve server throughput for directory-bound workloads.

5.5.4 Code Changes

In order estimate the difficulty of adoption, Table 5.4 lists the lines of code changed in our Linux prototype. The vast majority of the changes required (about 1,000 LoC) are hooks localized to the dcache itself (`dcache.c` and `namei.c`); most of these optimizations are in a separate set of files totaling about 2,400 LoC. Also, the low-level file systems we tested did not require *any* changes to use our modified directory caches. The main impact on other subsystems was actually to the LSMs, which required some changes to manage PCCs correctly. Thus, the burden of adoption for other kernel subsystems is very minor.

Source files	Original (LoC)	Patched/Added (LoC)	
New source files & headers		2,358	
`fs/namei.c`	3,048	425	13.9 %
`fs/dcache.c`	1,997	142	7.1 %
Other VFS sources	3,859	98	2.5 %
Other VFS headers	2,431	218	9.0 %
Security Modules (SELinux, etc)	20,479	76	0.0 %

Table 5.4: *Lines-of-code* (LoC) changed in Linux, as measured by `sloccount` [Wheeler, 2009].

5.6 Summary

The **file system directory cache** is a sophisticated subsystem of the Linux virtual file system (VFS), which is designed for caching the file paths and metadata retrieved from the storage. We observe missed opportunities in the design of the directory cache, primarily in removing the redundant operations that contribute to the lookup latency at cache hits. These operations include checking prefix permissions against security models, detecting mount points, and resolving symbolic links, etc. A fast path for looking up already cached information will fully exploit the locality of path access in the file system, to cache the results of permission checks and other redundant operations. The optimized design not only minimizes the hit latency in favor of frequently accessed paths, but also improves the hit rate, by capturing other missed information (e.g., nonexistent paths) that shall be cached. Essentially, we optimize the hit latency by penalizing the infrequent operations of updating path attributes and permissions, as an acceptable trade-off in most scenarios.

The optimization on the directory cache shows an example for introducing performance improvement into a legacy system, while preserving the existing system abstractions and security mechanisms. We start by investigating the opportunity of rewriting the component-by-component lookup algorithm in the directory cache, and identify the culprit of the suboptimality as the interleaving of looking up data structures and processing file attributes. The principle behind this optimization is to decouple the implementation of an efficient, frequently occuring operation from addressing security concerns and fulfilling system features — a principle we also use in implementing an efficient library OS.

Chapter 6

Proposed Works

The thesis describes our previous contributions to reusing legacy applications in the Graphene library OS and Linux kernels, in consideration of security isolation and efficiency. **The completed works of the thesis are listed as follows:**

- A Linux-compatible library OS called **Graphene** [Tsai et al., 2014], which runs legacy Linux, multi-process applications on a platform-independent host ABI. The system delivers a distributed POSIX implementation over simple RPC streams, and a security isolation model to sandbox applications, on both trusted and untrusted hosts (using SGX enclaves). We optimize the inter-process coordination latency in Graphene, using a simple bulk-copy API on the host, and determining the optimal placement of OS state. (§2)
- A prototype framework called **Civet**, with uses guided partitioning to split the sensitive part of an legacy Java application into SGX enclaves. The framework overcomes several limitations of SGX to allow loading Java classes, including using Graphene to bootstrap Java VM in enclaves, interfaces for accessing in-enclave objects, and exporting enclave feature APIs. Civet applies language-specific protections, such as information flow tracking to filter enclave output. **This work is still in progress.** (§3)
- We suggest measurements for quantitatively evaluating the importances of system APIs in legacy applications, and the completeness of legacy support in a system prototype. The measurements are applied in a study of **Linux API usage and compatibility** [Tsai et al., 2016], to draw several insights for system development. (§4)
- We design a fast path in the Linux **file system directory cache** to improve the hit latency of path lookup [Tsai et al., 2015]. The design decouples path searching and other operations, by caching the results of prefix checking and file attributes in kernel data structures. The optimization is generalized to support most Linux file system features (e.g., pseudo-files, renaming, symbolic links, security modules). (§5)

For the fulfillment of this thesis, we proposed several research and engineering targets, and a concrete schedule for the completion of the proposed works. **The proposed works and schedule are listed as follows:**

- **Schedule for Fall 2016 Semester:**
  - **September:**
    Collect and apply the feedbacks from the committee. A updated thesis proposal shall be submitted to the committee at the end of September, including the new issues

and topics suggested by the committee. For each suggested topics from the committee, present related studies, literatures, challenges and preliminary solutions.

  - **October:**
    Complete and improve the result of partitioning Java applications (§6.2). Complete the Civet framework and support the partitioning of all the proposed use cases (SSH clients, Hadoop algorithm, and secure data manager). Design a more efficient class Shredder, by removing unused methods from all supporting classes. Reduce the binary size loaded into the enclave, including the Java VM, the Graphene library OS, libc, and JNI libraries.

  - **November:**
    Implement missing features in the Graphene library OS (§6.1). Implement scheduling system calls, and hardware support features such as hugepages and NUMA control. Introducing file system drivers, for mounting and coordinating Networked File System (NFS) and Ext4 file system. Improve and evaluate the completeness of Graphene using the measurement and approach described in §4.

  - **December:**
    Porting the Graphene host ABI to Windows, OSX, Barrelfish and L4 (§6.1). Complete and evaluate the partial implementation of the Graphene host ABI on Windows and OSX. For the Windows port of Graphene, remove the dependency on toolchains (i.e., `CygWin`). Implementing the host ABI on Barrelfish, with porting the Graphene library OS to x86 (32-bit) and ARM architecture. Implementing the host ABI using the L4 microkernel.

- **Schedule for Spring 2017 Semester:**

  - **January:**
    Explore the solutions for implementing security models in Graphene (§6.3). Present use cases for demonstrating the practicality of emulating security models in library OSes. Based on the proposed designs, analyze the potential threats in the emulation models. Deliver preliminary result of emulating security models in Graphene (in both picoprocesses and enclaves). Demonstrate proof-of-concepts for the implementation and prevention of vulnerabilities. Consider research contributions and determine a publication plan.

  - **February:**
    Complete the thesis draft. The completed draft must include the updated content since the thesis proposal. Describe new finding, achievements and research contributions. The first draft will be iterated with the adviser and polished. The completed draft must be submitted to the Committee by the end of February, for collecting feedbacks.

  - **March: Collect feedbacks from the Committee and prepare for the thesis defense.**
  - **Mid-April: Doctoral dissertation defense and submit the thesis.**

In the following sections, we will discuss the motivations and observations driving the proposed works. Potential, reasonable solutions or strategies to complete the proposed works will be

described in length, with the discussion of the related issues and principles.

6.1 Generalizing Platform Independence

Existing library OSes support legacy applications by implementing the OS abstractions and personalities, using the interfaces provided on the host platforms. The responsibility of library OSes is essentially translating the abstractions or APIs to the host interfaces and conventions. In addition, to provide **platform independence**, library OSes rely on the definition of host ABI to be generic enough for the underlying platforms [Baumann et al., 2013, 2014; Porter et al., 2011; Tsai et al., 2014] — assuming that porting the host ABI to new platforms will be reasonably easy. The abstractions in the host ABI include the commonly shared features of most platforms, such as file systems, networks, memory mappings, etc, with the semantics defined as translatable as possible. The key to the platform independence is the portability of the host ABI, as well as its completeness to support library OSes for implementing all high-level system APIs.

The narrowness of host interfaces causes limitations for library OSes to implementing OS personalities. In existing library OSes, the host ABI is mostly defined for supporting library OSes with single OS personality: For instance, in Drawbridge [Porter et al., 2011], the host ABI is defined to support a library OS with Windows personality. In Graphene, we implement Linux personality over the Drawbridge host ABI, and observe a few necessary, but missing host abstractions, due to the difference between the platforms. A similar process is taken by another library OS, Bascule, that inherits the Drawbridge host ABI. For instance, Linux applications heavily rely on exception handling, but the relative host interface to set up handlers is missing in Drawbridge, and added in both Bascule and Graphene afterward.

In this thesis we focus on implement Linux personality: Graphene supports 131 Linux system calls, selected by what we considered to be the most commonly used. However, according to our study of Linux system API usage in a large, representative application sample, for any Linux installations, there are only 0.4% of the installed applications whose system API footprint are completely supported by Graphene. Moreover, the study suggests that completeness of Graphene can be largely boosted (from 0.4% to 21%) by adding two scheduling system calls, `sched_setscheduler` and `sched_setparam`. In future, we will guide the implementation of system APIs using the measurements derived from our study.

The Linux system calls that are not implemented in Graphene can be primarily categorized into two types. One type is the APIs to specify policies for scheduling host resources; and the other is to provide functionality of host hardware features. Examples for the former type are the scheduling system calls mentioned earlier. The latter type includes various `ioctl` operations, and **NUMA-related APIs** like `migrate_pages`. These abstractions are impossible to implement in library OSes without extension to the host ABI. To complete these system calls, we will design **new host interface** to expose these host abstractions to applications, or rely on **manifest file** to specify the scheduling policy of hardware resource. Table 6.1 lists the first-order host abstractions that need be provided in the Graphene host ABI.

We also have to demonstrate the neutrality and portability of the host ABI, to justify that Graphene is independent to most host platforms. However, the assumptions made in our host ABI definitions can be violated on specific host platforms due to the platform limitations and characteristics. For instance, although not defined explicitly in the host ABI, a Linux picoprocesses

Host ABI Function	Functions in the library OS	Description
Scheduler policies	`sched_setscheduler`, `sched_setparam`, `sched_setaffinity`	Specifying scheduler policies such as priority level or CPU affinity in the manifests.
Huge pages	`MAP_HUGETLB` flags for `mmap`	Specifying virtual memory to be backed by huge pages.
Memory sharing	`shmget`, `shmat`, `shmdt`	Sharing memory using RPC.
Raw sockets	`socket` with `SOCK_RAW`	Sending RAW packets (e.g., DHCP clients)
NUMA	`set_mempolicy`, `migrate_page`	Accessing host NUMA features.

Table 6.1: List of host ABI functions that are missing in Graphene host ABI. We expect supporting these functions in the future, by either extending the host interfaces or configuration in manifest files.

Feature	Platform	Limitations	Proposed strategy
Memory mappings	Windows	No fine-grained deallocation or protection	Redesign ABI functions for platform independence
Thread-local storage	Windows, OSX	FS or GS register is not available	Overwrite TLS allocation in libc, or binary translation
Position-dependent binaries	Windows, SGX	Memory addresses are occupied or restricted	Reserve virtual memory before picoprocesses start
Special instructions	SGX	`CPUID` and `RDTSC` are forbidden	Exception handling or binary translation

Table 6.2: List of limitations on host features that affects porting the host ABI to the target platforms, and the coping strategies.

requires the host platforms to reserve part of the address space (often near `0x4000000`), for loading position-dependent binaries. Such a requirement is a challenge to the platforms in which the virtual memory of a process is partially preoccupied by the host, or restricted to a specific range. We observe the phenomenon in two of the ported platforms: in Windows, user stacks and PCBs (process control blocks) can overlap with the demanded address; in SGX enclaves, the range of mappable memory is restricted to a preset region. Table 6.2 lists the platform limitations that affects the implementation of host ABI, and the proposed strategies to remove the limitations.

Essentially, justifying the platform independence of Graphene will require porting its host ABI to more diverse platforms, especially ones that do not rely on a monolithic kernel with abundant APIs. One opportunity is to port Graphene to a **microkernel** (e.g., L4 [Klein et al., 2009; L4Family]) or a **hypervisor** (e.g., Qemu, Xen). Instead of relying on host features (e.g., Seccomp filter) to restrict the attack surface of picoprocesses, Graphene ported to a microkernel or hypervisor will minimize both the attack surface and the shared TCB of the host. The functionality of Graphene on L4 will be similar as L4Linux [Härtig et al., 1997], but at a lower cost because Graphene does not port the whole Linux kernel to user space. On the other hande, a microkernel or hypervisor may provide a reduced paravirtual APIs, without the full host abstractions required by the host ABI. A few host abstractions that may be missing are file access, network sockets and other single-process, multiplexed resources. Implementing the host ABI on these platforms will

requires extension of the host paravirtual APIs, or reduction of the Graphene host ABI — for example, running an in-process file system, or a NFS (networked file system) client in picoprocesses.

Barrelfish [Baumann et al., 2009; Zellweger et al., 2014] is a microkernel-style research OS designed for multi-core, heterogeneous environment. Barrelfish runs applications with OS implementation tied to each heterogeneous core, to decouple OS coordination from architectural characteristics such as instruction sets, cache coherence, and memory hierarchies. Not unlike Graphene using a distributed, collaborative OS implementation, Barrelfish duplicates the OS functionality ("OS nodes") with independent, unshared state on each core. Inter-core message-passing is used among OS nodes and applications to coordinate OS state and services. Porting Graphene to Barrelfish can enable the legacy support on the platform, and extend the host architectures for Graphene to include heterogeneous cores. The implementation of Graphene host ABI on Barrelfish can rely on its tool chain that partially supports the common POSIX APIs — including simple file IO, network sockets, and user-space RPC. To allow Graphene to run in a environment with heterogeneous ISAs, Graphene will have to be ported to different architectures, such as x86 and ARM.

We observe that many platforms provide **POSIX-compliant APIs**, which is the closest to a platform-independent specification. Besides a few limitations on the Graphene host ABI, such as TLS support and loading position-dependent binaries (both discussed in Table 6.2), most of the PAL calls can be smoothly ported to POSIX. A POSIX port will allow Graphene to be easily migrated to any POSIX-compliant platforms, with little development effort to deal with the rest of host ABI.

6.2 Partitioning Legacy Applications

In our ongoing work, we explore a system design called **Civet** to automatically partition legacy Java applications into SGX enclaves. As a proof-of-concept, we build a framework prototype that demonstrates the partitioning of three representative use cases described in §3.6. For future works, the implementation can be extended to handle more scenarios.

In particular, a primary challenge in Civet is to improve the effectiveness of the partitioning — in other word, pursuing the minimality of the isolated, security-sensitive components. Civet uses a *Shredder* that identifies the minimal supporting classes required for the isolated execution, to generate a clean, self-contained partitions with the smallest TCB, and less resource for loading and compiling classes in enclaves. We evaluate the effectiveness of the current Shredder design: based on the dependency analysis from the entry classes identified by the developers, even for a minimal "Hello World" enclave, the Shredder generate a partition with ˜3,000 classes, ˜23,000 methods, and ˜388,000 lines of code in total.Compared with the overloaded TCB of the unstripped Java runtime classes (from `rt.jar`), the shredded code effectively reduces xx% of the enclave code. The remaining classes generated by the Shredder includes the dependency of OpenJDK Java VM and the Civet framework, as well as the super classes with overridden methods that are never used. In future, we can further reduce the size of supporting classes, by partitioning at the granularity of methods, to shred all unused code in the isolated classes.

Another important aspect of partitioning is the effort for developers to identify the scope of partitioning, based on the understanding of the execution. Civet requires developers to provide two simple hints for guiding the partitioning: (1) the classes used as the enclave interfaces, and (2)

dynamically-loaded classes. From the developers' perspective, however, the loaded classes are not necessarily exposed to the developers. Due to the encapsulation and run-time loading of Java, the actual loaded classes during execution may be obfuscated to the application developers of Java. A common example is the loading of Java cryptography APIs: Java provides a generic interface to loads encryption or other cryptographic engines based on hard-coded or input strings that describe the algorithm options. To precisely analyze the loading classes in a Java application, Civet can rely on a training phase for detecting dynamically-triggered class loading, or conducting sophisticated date-flow analysis to retrieve the class names from the source or the byte-code.

An alternative approach for partitioning applications will be to determine the minimal partition "bottom-up", starting from the execution or data that needs to be protected. Instead of using a Shredder to remove unused classes, the partitioning can instead rely on a **Collector** that starts with minimal isolated classes, and gradually includes more byte-code into the enclave. The Collector will incorporate the classes that access any protected methods or objects, until it ultimately determines a clean, minimal partition as self-contained and with the narrowest interface.

Besides the effectiveness for partitioning applications, we also observe opportunities in reducing the TCB of infrastructure, including the OpenJDK Java VM and its libraries, JNI libraries, non-Java code of the Civet framework, and the underlying libc and Graphene library OS. In current work, we reduce the unused code on a best-efforts basis. In future, we seek more systematic approach to fundamentally shrink the size of enclave code.

6.3 Emulating Legacy Security Models in Applications

In library OSes, security isolation is enforced on mutually untrusting applications, to disallow OS-level interaction, or sharing any OS and application state. Such a model is fail-safe, but inflexible. On the other hand, many legacy applications are already configured with security policies, based on existing security models adopted in commodity operating systems. For instance, in UNIX systems, applications or processes can be assigned with **credential IDs** or **POSIX capabilities**, to specify the privilege of accessing abstractions or resources. In Linux, more sophisticated, fine-grained security models such as **AppArmor** [AppArmor] and **SELinux** [Loscocco and Smalley, 2001] can further specify the objects (i.e., *"what can I access?"*) and the actions (i.e., *"how can I access it?"*) of access control. Specifically, SELinux can enforce a **multi-level security** model, or **DAC (Discretionary Access Control)**, to control the propagation of information contamination. Both AppArmor and SELinux require configuration efforts, and generally, the developers are responsible for providing the profiles that reflect the appropriate policies. In addition, Linux supports isolation of various **namespaces**, such as PID, mount points, and System V IPC keys, to allow applications to isolate a part of the OS views in selected processes. The isolation of namespaces is often used in the concept known as **containers**. Overall, the usage of these models consists in part of the existing development effort, and defines the least privileges each subject requires to fulfill its tasks, more or less.

Security models can be critical for application: a security model can enforce policies that not only restraint unrelated applications, but also ensure the safe behaviors of cooperating components. In library OSes, applications need a model that, when a picoprocess creates or interacts with others, one can choose to put only partial trust in another. For instance, an Apache web server with privilege-escalating PHP scripts (e.g., suPHP) can isolate the scope of the script execution with

Security Model	In-application example	Proposed strategy
UID-based	Process calls `seteuid()` system call for escalating or dropping privilege	Mapping UIDs/GIDs to sandboxes
Namespaces	Clone child process with isolated namespaces	Isolating RPC streams and IPC helper threads for different namespaces
AppArmor	Specify white-list of allowed files and network address for an application	Translating AppArmor profiles to sandbox rules

Table 6.3: List of security models that are missing in Graphene. We expect emulating in-application policies based on different security models in the future, and propose possible strategies for emulation.

restricted policies. When running in Graphene, the picoprocesses will coordinate OS states such as process ID namespaces or signaling, but restrict access for specific OS abstractions.

At the center of these problems is the fact that existing library OSes including Graphene cannot reflect **non-binary**, precisely-defined policies in applications. The design of picoprocesses disallows any management of credentials and access control inside library OSes, because OS states and data structures are always vulnerable to in-process attackers. In Graphene, every process is granted with the local *root* permissions — allowed to access any OS resources and abstractions in its sandbox. Alternatively, we present a new opportunity in Graphene, to allow *expelling* a picoprocess from its current sandbox (§2.3); however, once a picoprocess is detached from a sandbox, it is forbidden by the reference monitor to coordinate with other picoprocesses by all means.

An alternative for picoprocesses to enforce non-binary security is to use the host interface for specifying capabilities and permissions, and rely on centralized entities—often the kernels—-to mediate all access control. The design is similar to the method adopted in the building blocks of **HiStar** [Zeldovich et al., 2006]. In HiStar, processes are assigned with labels, which will taint the components or resources that the processes ever interact with or access, based on the **information flow control** of the kernel. Applying this model to Graphene will require massive changes to the design; all multi-process abstractions coordinated over RPC will have to be mediated by the kernel, with inevitable extension to the host interface. Another limitation is the prerequisite of trusting the host kernel, which may be impractical on some platforms such as SGX enclaves.

The key to enforcing abstraction-specific policies in library OSes is the placement of abstraction states in picoprocesses. In Graphene, each picoprocess can receive a state replica from the owners of abstractions, while the security policies are completely ignored. However, based on the policies, a picoprocess shall be permitted or rejected for replicating the abstraction state according to whether the process is allowed to share the abstraction. If we ensure all abstraction states to be placed in the permitted picoprocesses, library OSes can approve RPC messaging for coordination, by either permission checks or access control by the host.

For instance, when a process clones a child with namespace isolation (e.g., using `CLONE_NEWPID` flag), the library OS can set up iptables-like firewall rules—a host feature already provided by the reference monitor of Graphene (§2.3)—on the coordinating RPC streams to restrict the access of namespace state. The assignment of the rules on the host must happen prior to the process creation. The library OS will have to isolate the RPC streams and the responding IPC helper threads that manage separate namespaces.

We propose strategies for emulating different security models in library OSes (discussed in ta-

ble 6.3). **Credential IDs**, such as user IDs or group IDs, are commonly used to specify the owner of objects in operating systems, primarily for isolating users that share the same host. In other use cases, however, credentials can matter even in a single application — when a process uses system calls like `seteuid()` and `setegid()` to temporarily change its privileges for accessing resources. We can emulate credential IDs by mapping the credentials to host sandboxes. When a process changes its credential, the host can migrate the picoprocess from a sandbox to another, as long as the migration does not compromise either of them. Another common security model used in Linux, **AppArmor**, allows developers to specify a profile for an application, to white-list all the files, network addresses, and other resources permitted in the application. In fact, the policies specified in AppArmor are conceptually similar to sandboxing in Graphene. We can simply emulate AppArmor model in Graphene by translating the AppArmor profiles into sandbox rules.

Speaking at the high level, emulating security models in a platform-independent way is fundamentally a research problem. From the perspective of developers and users, they choose security models—among the available options of their preferred systems—based on what they see suitable for the applications. The secure platforms (e.g., library OSes, microkernel, SGX enclaves), on the other hand, determine the architectures and mechanisms for enforcing certain security principles. As a future direction, decoupling security models (*"what to secure?"*) and secure platforms (*"how to secure?"*) will provide more options for developers and users to define their desired policies.

Chapter 7

Related Works

7.1 Previous Library OSes

This work extends previous library OSes [Baumann et al., 2013; Douceur et al., 2008; Madhavapeddy et al., 2013; OSV; Porter et al., 2011], which focused on single-process applications, to support coordination abstractions required for multi-process applications, such as shell scripts.

Bascule [Baumann et al., 2013] 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 [OSV].

A number of recent projects have provided a minimal, isolated environment for web applications to download and execute native code [Douceur et al., 2008; Howell et al., 2013; Mickens and Dhawan, 2011; Wang et al., 2009a; Yee et al., 2009a]. 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 [Ammons et al., 2007; Anderson, 1992; Cheriton and Duda, 1994; Kaashoek et al., 1997; Leslie et al., 1996], whereas newer library OSes, including Graphene, focus on providing application compatibility across different hosts without dragging along an entire legacy OS. A common thread among all libOSes is moving functionality from the kernel into applications and reducing the TCB size or attack surface. Kaashoek et al. [Kaashoek et al., 1997] identify multi-processing as a problem for an Exokernel libOS, and implemented some shared OS abstractions. The Exokernel's sharing designs rely on shared memory rather than byte streams, and would not work on recent libOSes, nor will they facilitate dynamically sandboxing two processes.

User Mode Linux [Dike, 2006] (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 [Barham et al., 2003], and, unlike a library OS, does not deduplicate functionality.

7.2 Distributed Coordination APIs

Distributed operating systems, such as LOCUS [Fleisch, 1986; Walker et al., 1983], Amoeba [Cheriton and Mann, 1989; Mullender et al., 1990] and Athena [Champine et al., 1990] 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 [Pike et al., 1990], 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 [Baumann et al., 2009] 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 [Song et al., 2011] 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 [Baumann et al., 2009] could leverage our implementation techniques to efficiently construct higher-level OS abstractions, such as System V IPC and signals.

L3 introduced a "clans and chiefs" model of IPC redirection, in which IPC to a non-sibling process was validated by a the parent ("chief") before a message could leave the clan [Liedtke, 1992]. Although this model was abandoned as cumbersome for general-purpose access control [Elphinstone and Heiser, 2013], 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.

7.3 Legacy OS support for migration and isolation

Researchers have added checkpoint and migration support to Linux [Laaden and Hallyn, 2010] 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 [Soltesz et al., 2007], containers [Bhattiprolu

et al., 2008], and Solaris Zones [Price and Tucker, 2004], 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 [Kerrisk, 2012]. FreeBSD jails apply a similar approach to augment an isolated `chroot` environment with other isolated namespaces, including the network and hostname [Stokely and Lee, 2003]. Similarly, Zap [Osman et al., 2002] 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 [Shan et al., 2012] 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.

7.4 Partitioning Applications and Systems

Several recent efforts have been made to leverage SGX technology in cloud platforms to secure applications against a potentially untrusted host OS or hypervisor. For instance, VC3 [Schuster et al., 2015] runs MapReduce jobs in SGX enclaves. An essential caveat for VC3 is that these mappers and reducers must be written in C or C++, breaking compatibility with the vast majority of Hadoop mappers and reducers, which are written in Java and can run on Civet. Similarly, Brenner et al. [Brenner et al., 2014], developed a transparent encryption layer to the Apache ZooKeeper cluster, running cluster services in an enclave and transparently encrypting data in transit between enclaves. SGX has also been applied to securing network functionality [Shih et al., 2016], as well as inter-domain routing in Tor [Kim et al., 2015].

A number of projects have developed tools for partitioning applications, both with SGX enclaves in mind, and more generally. Atamli et al. [Atamli-Reineh and Martin, 2015] explore different schemes for partitioning applications, and identify different trade-offs in performance and security with different strategies Moat [Sinha et al., 2015] uses formal verification to determine whether an enclave image maintains confidentiality of sensitive data, and identifies potential information leaks. Civet, on the other hand, applies a combination of static and dynamic analysis for information flow tracking of the dynamically-loaded Java code in enclaves, and prevent leakage of confidential information at the enclave boundary .

Orthogonal to SGX, a number of systems have been developed to partition Java applications into pieces that run in multiple JVMs. Addistant [Tatsubori et al., 2001] and J-Orchestra [Tilevich and Smaragdakis, 2002] automatically divide Java applications to run across multiple hosts or JVMs. Zdancewic et al. [Zdancewic et al., 2001] use programmer annotations to partition a code to statically check and dynamically enforce information flow policies. Swift [Chong et al., 2007] partitions web applications such that security-critical data remains on the trusted server, and, secondarily, to minimize client-server communication. Finally, a number of tools have been developed for automatically injecting remote method invocations (RMI) for distributing Java ap-

plications [Aridor et al., 1999; Czajkowski et al., 2002; Diaconescu et al., 2005; Philippsen and Zenger, 1997; Spiegel, 1999; Tilevich and Smaragdakis, 2008]. Civet extends these approaches with techniques appropriate for partitioning code into SGX enclaves.

A number of projects have used developer annotations or runtime tools to implement privilege separation [Brumley and Song, 2004] or enforce least privilege [Bittau et al., 2008] on memory objects. Several systems have also separated programs into trusted and untrusted components, and used different OS instances for servicing each piece of the application [Khatiwala et al., 2006; Singaravelu et al., 2006; Ta-Min et al., 2006]. Civet adopts a similar approach, using developer input to identify classes containing sensitive data to drive the partitioning effort, and contributes a synthesis with hardware-level protections against an untrusted system stack.

A number of systems that protect against an untrusted OS predate SGX, or are designed for hardware platforms with different memory protection techniques, such as ARM TrustZone. Virtual Ghost [Criswell et al., 2014] uses compile-time and runtime monitoring to protect an application from a potentially-compromised OS; Virtual Ghost requires recompilation of the guest OS. Flicker [McCune et al., 2008], MUSHI [Zhang et al., 2012a] and InkTag [Hofmann et al., 2013] protect applications from untrusted OS using SMM mode or virtualization to enforce memory isolation between the OS and a trusted application. Koberl et al. [Koeberl et al., 2014], isolate software on low-cost embedded devices using a Memory Protection Unit. Li et al. [Li et al., 2014] built a 2-way sandbox for x86 by separating the Native Client(NaCl) [Yee et al., 2009b] sandbox into modules for sandboxing and service runtime to support application execution and use Trustvisor [McCune et al., 2010] to protect the piece of application logic from the untrusted OS. Jang et al. [Jang et al., 2015] build a secure channel to authenticate the application in the Untrusted area from the ARM TrustZone. Song et al. [Song et al., 2016] extend each memory unit with an additional tag to enforce fine-grained isolation at machine word granularity in the HDFI system. While these solutions focus on protecting applications from untrusted OS, Chen et al. [Chen et al., 2016], protects pieces of application from the rest of it by restricting access to only parts of memory from specific segments of threads.

7.5 Information Flow Control

Much of the foundational work in language-level information flow was done with a combination of static analysis and runtime checking on a high-level language, such as Java [Banerjee and Naumann, 2002; Chandra and Franz, 2007; Franz, 2008; Hammer et al., 2006; Myers, 1999; Smith and Thober, 2007; Yip et al., 2009]. A number of systems have also developed efficient, dynamic taint-tracking systems for languages such as Java [Haldar et al., 2005; Nair et al., 2008]. These are essential building blocks for the information flow tracking used in Civet. For the most part, these systems assume the integrity of the application and its runtime environment; a few studies have also studied issues in integrating language and OS-level information flow tracking [Roy et al., 2009; Sabelfeld and Myers, 2003]. Civet exposes the capabilities of SGX hardware to the developer and language runtime, providing stronger integrity assurances for the application, reducing the need for instrumentation in untrusted code, and streamlining remote provisioning of sensitive code or data.

Andrew Myers [Myers, 1999] extend the Java language with annotations that are statically checked, and adds a new primitive type *label* which can be dynamically checked for cases when

the label-type cannot be inferred statically. Banerjee et al. [Banerjee and Naumann, 2002], enforce non-interference policy for a Java like language by confining pointers to a high security region. Hammer et al., [Hammer et al., 2006] detect information leakage in Java using dependence graphs and constraint solving on the path conditions instead of type-based enforcement. Haldar et al., [Haldar et al., 2005] enforce source to sink dynamic taint tracking from untrusted inputs to sensitive methods that should not use tainted data. Niar et al. [Nair et al., 2008], extend the information flow tracking to implicit flows by tracking the control flow taint dynamically at runtime. Franz et al. [Franz, 2008], too prevent implicit information flow by tracking the program counter to build a multi-level security aware Java VM by adding a security label to every data item and preventing assignments that would leak secrets. Chandra et al., [Chandra and Franz, 2007] decouple the information flow policy specification and enforcement by statically annotating the Java class files with labels, and then tracking the labels to enforce the specified policy at runtime. Smith et al., [Smith and Thober, 2007] posit that statically checking only a small subset of the core JAVA classes is necessary to enforce information flow control across the I/O boundary without the need to track every data structure. Yip et al. [Yip et al., 2009], extend the concepts of information flow to PHP to check information flow assertions before writing data to a file or network.

Sabelfeld et al. [Sabelfeld and Myers, 2003], survey the language-based information flow security solutions and discuss challenges to support system-wide end-to-end information flow security. Vandebogart et al. [Vandebogart et al., 2007], enforce application-defined security policies from the context of the OS to provide a combination of discretionary as well as mandatory access control. Krohn et al. [Krohn et al., 2007], enforce system-wide decentralized information flow control by interposing a reference monitor to track information flow across OS abstractions such as pipes and file descriptors. Zeldovich et al. [Zeldovich et al., 2006], move the information flow of OS abstractions into a user-level library OS to remove the legacy OS from the TCB. Roy et al., [Roy et al., 2009] combine OS and PL techniques to enforce information flow control at the data structure granularity via JVM and a reference monitor enforces the same information flow policy on OS resources such as files and sockets.

7.6 Improving File System Lookup

Most related work on improving directory cache effectiveness targets two orthogonal problems: reducing the miss latency and prefetching entries. Most similar to our optimization to memoize prefix check results, SQL Server caches the result of recent access control checks for objects [Microsoft].

Reducing Miss Latency. One related strategy to reduce miss latency is to pass all components to be looked up at once to the low-level file system, essentially creating a prefetching hint. Several network file systems have observed that component-at-a-time lookup generates one round-trip message per component, and that a more efficient strategy would pass all components under a mount point in one message to the server for lookup [Duchamp, 1994; Welch, 1994]. A similar argument applies to local file systems, where a metadata index may be more efficiently fetched from disk by knowing the complete lookup target [Lensing et al., 2013; Russinovich and Solomon, 2009]. As a result, this division of labor is adopted by Windows NT and Solaris [McDougall and Mauro, 2008; Russinovich and Solomon, 2009]. One caveat is that, when not taken as a prefetching "hint", this can push substantial VFS functionality into each low-level file system, such as

handling redirection at mount points, symbolic links, and permission checking. Chen et al. note that pushing permission checks from the VFS layer down to individual file systems is a substantial source of difficult-to-prevent kernel bugs in Linux [Chen et al., 2011]. In contrast, this project caches the result of previous prefix checks over paths already in memory to reduce hit latency, rather than using the full path as a prefetching hint.

Another ubiquitous latency-reduction strategy is persistently storing metadata in a hash table. In order to reduce network traffic, several distributed file systems [Brandt et al., 2003; Zhang et al., 2012b; Zhu et al.], clustered environments [LaFon et al., 2012; Xing et al., 2009], and cloud-based applications [Wang and Lv, 2011] have used metadata hashing to deterministically map metadata to a node, eliminating the need for a directory service. The Direct Lookup File System (DLFS) [Lensing et al., 2013] essentially organizes the entire disk into a hash table, keyed by path within the file system, in order to look up a file with only one I/O. Organizing a disk as a hash table introduces some challenges, such as converting a directory rename into a deep recursive copy of data and metadata. DLFS solves the prefix check problem by representing parent permissions as a closed form expression; this approach essentially hard-codes traditional Unix discretionary access control, and cannot easily extend to Linux Security Modules [Wright et al., 2002]. An important insight of our work is that full path hashing in memory, but not on disk, can realize similar performance gains, but without these usability problems, such as deep directory copies [Lensing et al., 2013] on a rename or error-prone heuristics to update child directory permissions [Swift et al., 2001].

VFS Cache Prefetching. Several file systems optimize the case where a `readdir` is followed by `stat` to access metadata of subdirectories, such as with the `ls -l` command [Bisson et al.; Lensing et al., 2013; Thain and Moretti, 2007]. When a directory read is requested, these low-level file systems speculatively read the file inodes, which are often in relatively close disk sectors, into a private memory cache, from which subsequent lookups or stat requests are serviced. Similarly, the NFS version 2 protocol includes a `READDIRPLUS` operation, which requests both the directory contents and attributes of all children in one message round trip [Callaghan et al., 1995]. These file systems must implement their own heuristics to manage this cache. Prefetching is orthogonal to our work, which more effectively caches what has already been requested from the low-level file system.

7.7 Measurement and Study of System APIs

Concurrent with our work, Atlidakis et al. [Atlidakis et al., 2016] 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 [Kadav and Swift, 2012] 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 [Palix et al., 2011]. They find architecture-specific subsystems have highest fault rate, followed by file systems. Harter et al. [Harter et al., 2011a] 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.

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 [Gonzalez-Barahona et al., 2009; Nguyen and Holt, 2012; Robles et al., 2006], to measure the popularity of application programming languages [Amor et al., 2005b], to analyze dependencies between the packages [de Sousa et al., 2009], to identify trends in package sizes [Amor et al., 2005a], the number of developers involved in developing and maintaining a package [Robles and González-Barahona, 2003], and estimating the cost of development [Amor et al., 2005c]. Jain et al. used popularity contest survey data to prioritize the implementation effort for new system security policies [Jain et al., 2014]. This study is unique in using this information to infer the relative importance of of system APIs to end users, based on frequency of application installation.

A number of previous projects develop techniques or tools to identify software incompatibilities, with the goal of avoiding subtle errors during integration of software components. The Linux Standard Base (LSB) [Denis, 2008] 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 [Pavel and Denis, 2009]. 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 [Ponomarenko and Rubanov, 2011]. 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 [Singh and Kaur, 2014]. 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.

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 [Zhang and Sekar, 2013]. Static binary analysis has been used to automatically generate application-specific sandboxing policies [Lam and Chiueh, 2004]. Dynamic analysis has been used to compare system call sequences of two applications as an indicator of potential intellectual property theft [Wang et al., 2009b], to identify opportunities to batch system calls [Rajagopalan et al., 2003], to model power consumption on mobile devices [Pathak

et al., 2011], and to repackage applications to run on different systems [Guo and Engler, 2011]. These projects answer very different questions than ours, but could, in principle, benefit from the resulting data set.

Chapter 8

Conclusion

Due to the diversity of system platforms, developers pay varied efforts to port applications from a system to another, in order to gain qualitative benefits. The porting effort for an application can range from recompilation of the binaries to fundamental re-implementation. Essentially, the difficulty of porting is determined by the distinction between the *personalities* of platforms — either they are Windows, UNIX, library OSes, etc. Existing library OSes [Baumann et al., 2013, 2014; Porter et al., 2011] provide the personalities of monolithic kernels, such as Windows or Linux, within a single picoprocess. The single-process abstractions, such as accessing unshared files or creating in-process threads, can be wrapped inside a library OS; however, multi-process abstractions, on the contrast, require multiple picoprocesses to collaboratively provide a unified OS views.

We design a library OS called **Graphene** [Tsai et al., 2014], which supports legacy Linux, multi-process applications. In Graphene, the idiosyncratic, Linux multi-process abstractions — including forking, signals, System V IPC, file descritor sharing, exit notification, etc — are coordinated across picoprocesses over simple, pipe-like RPC streams on the host. The RPC-based, distributed implementation of OS abstractions can be isolated by simply sandboxing the RPC streams. The beneficial features of Graphene, including isolation among mutually untrusting applications, migration of system state, and platform independence, are comparable to virtualization, but at a lower resource cost. Especially, with platform independence, Graphene can extend the legacy support for Linux applications onto other platforms, including isolated execution platforms like **Intel SGX enclaves**. A Graphene picoprocess isolated in an enclave can seal the execution of a legacy application, in an environment immune to attacks from host kernels and hardware peripherals.

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

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

calls to Graphene, the fraction of applications that can plausibly use the system will grow from 0.42% to 21.1%.

At the high level, the principles for developing OS personalities can be vastly distinct among different specifications, and often entangled with the implementation of security mechanisms and performance optimizations. Similar challenges can be observed in legacy, monolithic kernels. We demonstrate a case of an performance-centric, heavily engineered component, the Linux **file system directory cache**. The directory cache is designed as an optimization for path lookup, yet it interleaves searching path components with permission checks (e.g., searching prefixes) and file system features (e.g., resolving symbolic links), causing suboptimal latency when the cache is warm (no cache misses) [Tsai et al., 2015]. A fast path to improve the hit latency will decouple searching in the directory cache from other operations, by caching the results of prefix checking, symbolic links, etc, in the kernel data structures. In conclusion, this thesis seeks systematic and generalizable solutions, for mitigating the limitations on fulfilling legacy application requirements in innovative system designs (e.g., library OSes, enclaves, file system fast paths).

References

diet libc: A libc optimized for small size. https://www.fefe.de/dietlibc.

The embedded GNU Libc. http://www.eglibc.org.

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

musl libc. http://www.musl-libc.org.

uClibc. https://www.uclibc.org.

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

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

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.

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.

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)*, Helsinki, Finland, 07 2005a.

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*, VI(3):13–16, 06 2005b. URL http://www.cepis.org/upgrade/index.jsp?p=0&n=2167&a=3077.

Juan José Amor, Gregorio Robles, and Jesús M. González-Barahona. Measuring Woody: The size of Debian 3.0. *CoRR*, abs/cs/0506067, 2005c. URL http://arxiv.org/abs/cs/0506067.

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

apache-sshd. Apache SSHD. https://mina.apache.org/sshd-project/.

AppArmor. AppArmor. http://wiki.apparmor.net/.

Jonathan Appavoo, Marc A Auslander, Dilma Da Silva, David Edelsohn, Orran Krieger, Michal Ostrowski, Bryan S Rosenburg, Robert W Wisniewski, and Jimi Xenidis. Providing a Linux API on the scalable K42 kernel. In *Proceedings of the USENIX Annual Technical Conference*, pages 323–336, 2003.

Apple App Store. Apple App Store. URL http://www.apple.com/osx/apps/app-store/.

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

Yariv Aridor, Michael Factor, and Avi Teperman. cjvm: a single system image of a jvm on a cluster. In *Proceedings of International Conference on Parallel Processing*, pages 4–11. IEEE, 1999.

Ahmad Atamli-Reineh and Andrew Martin. Securing application with software partitioning: A case study using sgx. In *Security and Privacy in Communication Networks*, pages 605–621. Springer, 2015.

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.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

Anindya Banerjee and David A Naumann. Secure information flow and pointer confinement in a java-like language. In *In IEEE Computer Security Foundations Workshop (CSFW)*, 2002.

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.

Robert Baron, Richard Rashid, Ellen Siegel, Avadis Tevanian, and Michael Young. Mach-1: An operating environment for large-scale multiprocessor applications. 2(4):65–67, July 1985.

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.

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.

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.

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.

Jonathan Bell and Gail Kaiser. Phosphor: Illuminating dynamic data flow in commodity jvms. In *ACM SIGPLAN Notices*, volume 49, pages 83–101, 2014.

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

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steven D. Gribble. Deterministic process groups in dos. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 1–16, 2010.

Daniel J. Bernstein. Using maildir format. `http://cr.yp.to/proto/maildir.html`.

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.

Tim Bisson, Yuvraj Patel, and Shankar Pasupathy. Designing a fast file system crawler with incremental differencing. *ACM Operating Systems Review*.

Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NDSI)*, volume 8, pages 309–322, 2008.

Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 30–40. ACM, 2011.

Rick Boivie and Peter Williams. Secureblue++: Cpu support for secure executables. Technical report, IBM Research, 2013.

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

bouncycastle. The legion of the Bouncy Castle Java cryptography apis. `https://www.bouncycastle.org/java.html`.

D. Bovet and M. Cesati. *Understanding the Linux Kernel*. O'Reilly Media, Inc., 3rd edition, 2005.

Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Lan Xue. Efficient metadata management in large distributed storage systems. In *IEEE Conference on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2003.

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.

David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, pages 57–72, 2004.

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

B. Callaghan, B. Pawlowski, and P. Staubach. NFS version 3 protocol specification. RFC 1813, June 1995. URL `http://tools.ietf.org/html/rfc1813`.

CERT Secure Coding. FIO21-C. Do not create temporary files in shared directories.

cglib. cglib - Byte Code Generation Library. `https://github.com/cglib/cglib`.

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.

Deepak Chandra and Michael Franz. Fine-grained information flow analysis and enforcement in a java virtual machine. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 463–475. IEEE, 2007.

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`.

Stephen Checkoway, Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, Hovav Shacham, and Marcel Winandy. Return-oriented programming without returns. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 559–572. ACM, 2010.

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

Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M. Frans Kaashoek. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Asia-Pacific Workshop on Systems*, pages 5:1–5:5, 2011.

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.

Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, 2016.

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.

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.

Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web applications via automatic partitioning. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, SOSP '07, pages 31–44, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: 10.1145/1294261.1294265. URL `http://doi.acm.org/10.1145/1294261.1294265`.

Jonathan Corbet. JLS: Increasing VFS scalability. *LWN*, November 2009. `http://lwn.net/Articles/360199/`.

Jonathan Corbet. Dcache scalability and rcu-walk. *Linux Weekly News*, 2010.

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.

Grzegorz Czajkowski, Laurent Daynès, and Nathaniel Nystrom. Code sharing among virtual machines. In *ECOOP 2002—Object-Oriented Programming*, pages 155–177. Springer, 2002.

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*, 1(2): 127–133, 03 2009.

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

Biplob Debnath, Sudipta Sengupta, and Jin Li. Chunkstash: Speeding up inline storage deduplication using flash memory. In *Proceedings of the USENIX Annual Technical Conference*, pages 16–16, 2010.

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.

Roxana E Diaconescu, Lei Wang, Zachary Mouri, and Matt Chu. A compiler and runtime infrastructure for automatic program distribution. In *Proceedings of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 52a–52a. IEEE, 2005.

Jeff Dike. *User Mode Linux*. Prentice Hall, 2006.

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

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.

Dan Duchamp. Optimistic lookup of whole NFS paths in a single operation. In *Proceedings of the USENIX Summer Technical Conference*, 1994.

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.

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.

Rik Farrow. Linux FAST'15 summary. *;login:*, 40(3), June 2015.

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

Hubertus Franke, Rusty Russel, and Matthew Kirkwood. Fuss, futexes and furwocks: Fast user-level locking in Linux. In *Ottawa Linux Symposium*, 2002.

Michael Franz. Eliminating trust from application programs by way of software architecture. In *Software Engineering*, pages 112–126. Citeseer, 2008.

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

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*, 14(3):262–285, 2009. doi: 10.1007/s10664-008-9100-x. URL `http://dx.doi.org/10.1007/s10664-008-9100-x`.

Google Play. Google Play Store. URL `https://play.google.com/store`.

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.

Vivek Haldar, Deepak Chandra, and Michael Franz. Dynamic taint propagation for java. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 9–pp. IEEE, 2005.

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.

Christian Hammer, Jens Krinke, and Gregor Snelting. Information flow control for java based on path conditions in dependence graphs. In *IEEE International Symposium on Secure Software Engineering*, pages 87–96, 2006.

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, 2011a.

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, 2011b.

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

Heartbleed, 2013. CVE-2014-0160, a.k.a, openssl heartbleed bug. Available from MITRE, CVE-ID CVE-2014-0160., dec 2013. URL `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`.

Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the Fourth Workshop on Hardware and Architectural Support for Security and Privacy*, 2013. URL `http://doi.acm.org/10.1145/2487726.2488370`.

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.

John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, pages 51–81, 1988.

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.

trammell Hudson and Larry Rudolph. Thunderstrike: Efi firmwire bootkits for apple macbooks. In *Proceedings of the 8th ACM International System and Storage Conference*, SYSTOR '15, 2015.

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

intel-pavp. Protected audio video path. `http://www.intel.com/content/www/us/en/support/graphics-drivers/000006968.html`.

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

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.

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

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.

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.

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.

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

Tejas Khatiwala, Raj Swaminathan, and VN Venkatakrishnan. Data sandboxing: A technique for enforcing confidentiality policies. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 223–234. IEEE Computer Society, 2006.

Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the USENIX Annual Technical Conference*, pages 273–284, 2003.

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.

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.

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.

M. Krohn, A. Yip, M. Brodsky, N. Cliffer, M. F. Kaashoek, E. Kohler, and R. Morris. Information flow control for standard OS abstractions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007.

Santosh Kumar. Hadoop Java map reduce sort by value. `http://santoshsorab.blogspot.com/2014/12/hadoop-java-map-reduce-sort-by-value.html`, 2014.

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.

L4Family. The l4 microkernel family. URL `http://www.l4hq.org/`.

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

Jharrod LaFon, Satyajayant Misra, and Jon Bringhurst. On distributed file tree walk of parallel file systems. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, 2012.

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

Daniel Lemire and Owen Kaser. Strongly universal string hashing is fast. *The Computer Journal*, page bxt070, 2013.

Paul Hermann Lensing, Toni Cortes, and André Brinkmann. Direct lookup and hash-based metadata placement for local file systems. In *ACM International Systems and Storage Conference (SYSTOR)*, 2013.

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.

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.

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

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

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

P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the USENIX Annual Technical Conference*, 2001.

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.

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.

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.

Richard McDougall and Jim Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Sun Microsystems Press, 2008.

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`.

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

Paul E. McKenney, Dipankar Sarma, and Maneesh Soni. Scaling dcache with RCU. *Linux Journal*. URL `http://dl.acm.org/citation.cfm?id=959336.959339`.

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

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.

Microsoft. Description of the äccess check cache bucket countänd äccess check cache quotaöptions that are available in the sp_configure stored procedure. `https://support.microsoft.com/en-us/kb/955644`.

MSDN: SetWindowsHookEx. MSDN: SetWindowsHookEx function. `http://msdn.microsoft.com/en-us/library/windows/desktop/ms644990%28v=vs.85%29.aspx`.

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.

Andrew C Myers. Jflow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 228–241, 1999.

Srijith K Nair, Patrick ND Simpson, Bruno Crispo, and Andrew S Tanenbaum. A virtual machine based information flow control system for policy enforcement. *Electronic Notes in Theoretical Computer Science*, 197:3–16, 2008.

Nergal. The advanced return-into-lib(c) exploits. `http://phrack.org/issues/58/4.html`, 2001.

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.

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.

OSV. OSv—designed for the cloud. `osv.io`.

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.

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.

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.

Michael Philippsen and Matthias Zenger. Javaparty - transparent remote objects in java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

phosphor. Phosphor: Dynamic Taint Tracking for the JVM. `https://github.com/Programming-Systems-Lab/phosphor`.

Rob Pike. Lexical file names in plan 9 or getting dot-dot right. In *Proceedings of the USENIX Annual Technical Conference*, pages 7–7, 2000.

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.

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.

Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 161–176, 2009.

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.

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.

Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 89–101, 2002.

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.

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

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

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.

Indrajit Roy, Donald E Porter, Michael D Bond, Kathryn S McKinley, and Emmett Witchel. *Laminar: practical fine-grained decentralized information flow control*, volume 44. 2009.

M. Russinovich and D. Solomon. *Windows Internals*. Microsoft Press, 2009.

Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications*, 21(1):5–19, 2003.

Russel Sandberg, David Goldberg, Steve Kleiman, Dan Walsh, and Bob Lyon. Design and implementation or the sun network filesystem. In *Proceedings of the USENIX Annual Technical Conference*, 1985.

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.

sdm. Secure Data Manager. `http://sdm.sourceforge.net/`.

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

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

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.

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

S. Shepler, Storspeed Inc., M. Eisler, D. Noveck, and NetApp. Network file system (NFS) version 4 minor version 1 protocol. RFC 5661, Jan 2010. URL `http://tools.ietf.org/html/rfc5661`.

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.

Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing tcb complexity for security-sensitive applications: Three case studies. In *ACM SIGOPS Operating Systems Review*, volume 40, pages 161–174. ACM, 2006.

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

Rohit Sinha, Sriram Rajamani, Sanjit Seshia, and Kapil Vaswani. Moat: Verifying confidentiality of enclave programs. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, CCS '15, pages 1169–1184, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3832-5. doi: 10.1145/2810103.2813608. URL `http://doi.acm.org/10.1145/2810103.2813608`.

Scott F Smith and Mark Thober. Improving usability of information flow security in java. In *Proceedings of the 2007 workshop on Programming languages and analysis for security*, pages 11–20, 2007.

Software Engineering Instititute. FIO42-C. Ensure files are properly closed when they are no longer needed. `https://www.securecoding.cert.org/confluence/display/seccode/FIO42-C.+Ensure+files+are+properly+closed+when+they+are+no+longer+needed`, August 2015.

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.

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.

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.

Andre Spiegel. Pangaea: An automatic distribution front-end for java. In *Parallel and Distributed Processing*, pages 93–99. Springer, 1999.

Kiran Srinivasan, Tim Bisson, Garth Goodson, and Kaladhar Voruganti. idedup: Latency-aware, inline data deduplication for primary storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 24–24, 2012.

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

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

Michael M. Swift, Peter Brundrett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in Windows NT. In *ACM Symposium on Access Control Models and Technologies (SACMAT)*, 2001.

Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 279–292. USENIX Association, 2006.

Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" java software. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 236–255, London, UK, UK, 2001. Springer-Verlag. ISBN 978-1-124-75297-6. URL `http://dl.acm.org/citation.cfm?id=646158.680014`.

Douglas Thain and Christopher Moretti. Efficient access to many samall files in a filesystem for grid computing. In *Proceedings of the 8th IEEE/ACM International Conference on Grid Computing*, Washington, DC, USA, 2007. IEEE Computer Society.

Eli Tilevich and Yannis Smaragdakis. J-orchestra: Automatic java application partitioning. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 178–204, London, UK, UK, 2002. Springer-Verlag. ISBN 3-540-43759-2. URL `http://dl.acm.org/citation.cfm?id=646159.680022`.

Eli Tilevich and Yannis Smaragdakis. Nrmi: Natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):174–187, 2008.

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

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

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.

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.

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.

Ubuntu Packages. Ubuntu packages. URL `http://pacakges.ubuntu.com`.

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

S. Vandebogart, P. Efstathopoulos, E. Kohler, M. Krohn, C. Frey, D. Ziegler, F. Kaashoek, R. Morris, and D. Mazières. Labels and event processes in the Asbestos operating system. *ACM Trans. Comput. Syst.*, 25(4):11, 2007. ISSN 0734-2071.

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

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.

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, 2009a.

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, 2009b.

Yixue Wang and Haitao Lv. Efficient metadata management in cloud computing. In *ICCSN*, pages 514–519, 2011.

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

Brent Welch. A comparison of three distributed file system architectures: Vnode, sprite, and plan 9. *Computer System*, March 1994.

David Wheeler. Sloccount, 2009. URL `http://www.dwheeler.com/sloccount/`.

C. Wright, C. Cowan, S. Smalley, J. Morris, and G. K. Hartman. Linux security modules: General security support for the Linux kernel. In *USENIX Security Symposium*, 2002.

Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, 2009.

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.

Jean Yang and Chris Hawblitzel. Safe to the last instruction: automated verification of a type-safe operating system. *PLDI*, 2010.

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)*, 2009a.

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, 2009b.

Alexander Yip, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. Improving application security with data flow assertions. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, pages 291–304. ACM, 2009.

Steve Zdancewic, Lantian Zheng, Nathaniel Nystrom, and Andrew C. Myers. Untrusted hosts and confidentiality: Secure program partitioning. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, SOSP '01, pages 1–14, New York, NY, USA, 2001. ACM. ISBN 1-58113-389-8. doi: 10.1145/502034.502036. URL `http://doi.acm.org/10.1145/502034.502036`.

Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in HiStar. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 19–19, 2006.

Gerd Zellweger, Simon Gerber, Kornilios Kourtis, and Timothy Roscoe. Decoupling cores, kernels, and operating systems. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 17–31, 2014. URL `http://dl.acm.org/citation.cfm?id=2685048.2685051`.

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

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, 2012a.

Quan Zhang, Dan Feng, and Fang Wang. Metadata performance optimization in distributed file system. In *ICIS*, Washington, DC, USA, 2012b.

Benjamin Zhu, Kai Li, and Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, pages 18:1–18:14, 2008.

Yifeng Zhu, Hong Jiang, Jun Wang, and Feng Xian. HBA: Distributed metadata management for large cluster-based storage systems. *IEEE Trans. Parallel Distrib. Syst.*, pages 750–763.

Appendix A

Formal Definitions of System API Metrics

A.1 API Importance — A Metric for APIs

Definition: **API Importance.**
For a given API, the probability that an installation includes at least one application requiring the given API.

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

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

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

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

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

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

A.2 Weighted Completeness — A System-Wide Metric

Definition: **Weighted Completeness.**
For a target system, the fraction of applications supported, weighted by the popularity of these applications.

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:

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