Cooperation and Security Isolation of Library OSes for Multi-Process Applications

Chia-Che Tsai Kumar Saurabh Arora Nehal Bandi Bhushan Jain William Jannen
Jitin John Harry A. Kalodner[†] Vrushali Kulkarni Daniela Oliveira[†] Donald E. Porter

Stony Brook University [†]Bowdoin College

{chitsai,karora,nbandi,bpjain,wjannen,jijjohn,vakulkarni,porter}@cs.stonybrook.edu
{hkalodne,doliveir}@bowdoin.edu

Abstract

Library OSes are a promising approach for applications to efficiently obtain the benefits of virtual machines, including security isolation, host platform compatibility, and migration. Library OSes refactor a traditional OS kernel into an application library, avoiding overheads incurred by duplicate functionality. When compared to running a single application on an OS kernel in a VM, recent library OSes reduce the memory footprint by an order-of-magnitude.

Previous library OS (libOS) research has focused on single-process applications, yet many Unix applications, such as network servers and shell scripts, span multiple processes. Key design challenges for a multi-process libOS include management of shared state and minimal expansion of the security isolation boundary.

This paper presents Graphene, a library OS that seamlessly and efficiently executes both single and multi-process applications, generally with low memory and performance overheads. Graphene broadens the libOS paradigm to support secure, multi-process APIs, such as copy-on-write fork, signals, and System V IPC. Multiple libOS instances coordinate over pipe-like byte streams to implement a consistent, distributed POSIX abstraction. These coordination streams provide a simple vantage point to enforce security isolation.

1. Introduction

Library OSes provide single-process applications with the qualitative benefits of virtualization at a lower cost [10, 34, 42]. These benefits include security isolation of mutually untrusting applications, migration, and host platform compatibility. In a library OS, the guest OS is essentially “collapsed” into an application library, which implements the OS system calls and supporting data structures as library functions—mapping high-level APIs onto a few paravirtual interfaces to the host kernel. 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 [34]. 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 [34, 42].

A key drawback of recent library OSes, however, is that they are limited to single-process applications. Yet 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, and exit notification.

This paper describes **Graphene**, a novel, Linux-compatible library OS. In Graphene, multiple libOS 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.

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-libOS communication. RPC coordination among Graphene instances can be dynamically disconnected, facilitating novel sandboxing techniques. For instance, we develop an Apache web server extension that, upon logging in a given user, places the worker process’s libOS in a sandbox with access to only that user’s data. We expect more nuanced degrees of trust are possible in future work. The contributions of this paper are:

- Graphene, a Linux library OS, which supports real-world, multi-process applications including a shell, web server, and compiler, which can be efficiently checkpointed and migrated.
- A framework for implementing multi-process APIs across cooperating library OS instances.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

EuroSys 2014, April 13 - 16 2014, Amsterdam, Netherlands
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2704-6/14/04…$15.00.
http://dx.doi.org/10.1145/2592798.2592812

- A thorough evaluation of the overheads of Graphene. Memory footprints are an order of magnitude smaller than KVM, and several applications perform comparably to a Linux process.
- A thorough analysis of Graphene security isolation.

Graphene's design gives the user and system administrator a high degree of flexibility in isolating arbitrary groups of unmodified application processes, while upholding the efficiency and host compatibility benefits of recent library OSes.

2. Background and Overview

Recent library OSes [10, 34, 40, 42] are designed for security and efficiency, but are limited to single-process applications. A libOS typically executes in either a paravirtual VM [34, 40] or an OS process, called a *picoprocess* [18], 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 1), which includes an *unmodified* application binary and supporting libraries, running on a libOS instance. The libOS 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 [9, 10, 42].

Graphene exports 43 host ABIs through a Platform Adaptation Layer (PAL) (Table 1). The PAL is injected into the picoprocess by the host kernel, and translates the generic picoprocess ABI into host kernel system calls. Most of these kernel calls only affect the application-internal state; any calls with external effects are mediated by a trusted *reference monitor*. All Graphene applications are launched by the reference monitor, which installs a system call filter and interposes on permitted kernel calls to ensure isolation (§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 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,

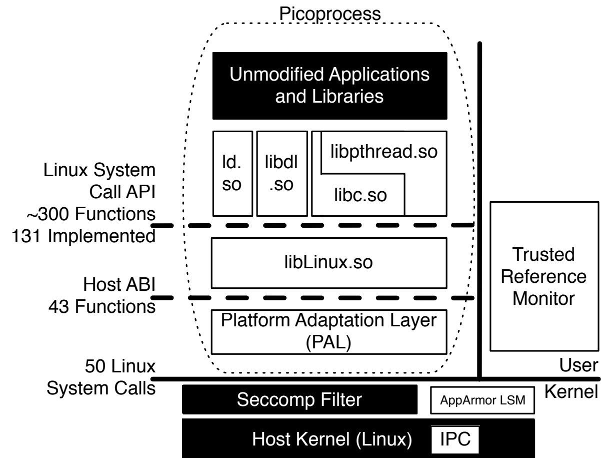

Figure 1. Graphene architecture. Black components are unmodified. We modify the four lowest application libraries: `ld.so` (the ELF linker and loader), `libdl.so` (the dynamic library linker), `libc.so`, and `libpthread.so`, to issue Linux system calls as function calls directly to `libLinux.so`. Our Linux library 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 ensures libOS isolation. We modify the AppArmor LSM and add a small module for fast, bulk IPC.

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 Multi-Process Support in a Library OS

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 libOS designs. The underlying design challenge is minimally expanding a tightly-drawn isolation boundary with-

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 for TLS.
Exceptions	2	Handle hardware exceptions.
Streams	3	Share stream handles and rename files.
Bulk IPC	3	Exchange copy-on-write pages.
Sandboxes	1	Move into a new sandbox, closing handles to other picoprocesses.

Table 1. Classes of host ABI functions adopted from Drawbridge [42], followed by ABIs added by Graphene.

out also exposing idiosyncratic kernel abstractions or reduplicating mechanisms in both the kernel and the libOS.

For example, consider the process identifier (PID) namespace. In current, single-process libOSes, 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 libOS 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 libOS instance as a process, and distribute the shared POSIX implementation across a collection of libOSes. 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 [11], 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.

Graphene Approach. In Graphene, multiple libOS 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 libOS issues a remote procedure call (RPC) to B's libOS over a host-provided byte stream (similar to a Unix pipe), and B's libOS then calls the appropriate signal handler.

Graphene implements all shared abstractions in the libOS, and libOSes 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 libOSes by blocking all RPC messages, without the need to understand the libOS-level details or semantics of these abstractions. In our PID example, only mutually trusting libOSes can signal each other.

PAL ABI Changes. When implementing Graphene, we found that the Drawbridge ABI lacked 11 PAL calls essential to running a multi-process Linux libOS, and Graphene did not require 3 PAL calls to support checkpoint and resume. Of the 11 new calls, 4 are required for single-process Linux and have also been added by Bascule: rename a file, manage segmentation hardware, and 2 for exception upcalls; 4 are required for stream inheritance and sandboxing; and 3 new calls are used to optimize copy-on-write fork (§5).

Comparison with microkernels. The building blocks of Graphene are very similar to the system abstractions of a microkernel [3, 8, 14, 19, 28, 32, 33]. Unlike a multi-server microkernel system, such as GNU Hurd [22] or Mach-US [48], 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 simulated 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 [7]. Arguably, recent libOS 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 paper.

3. Enforcing Security Isolation

Graphene ensures that mutually untrusting applications cannot interfere with each other, providing security isolation comparable to running in separate VMs. Graphene reduces the attack surface exposed to applications by restricting access to the host kernel ABI and prevents access to unauthorized system calls, files, byte streams, and network addresses with a reference monitor.

Graphene contributes a multi-process security model based on the abstraction of a **sandbox**, or a set of mutually trusting picoprocesses. The reference monitor permits picoprocesses within the same sandbox to communicate and exchange RPC messages, but disallows cross-sandbox communication. The current work focuses on all-or-nothing security isolation, although we expect this design could support controlled communication among mutually distrusting libOSes in future work.

The Graphene reference monitor is implemented using an unprivileged daemon as well as extensions to the AppArmor LSM [1], which checks file and socket policies in the kernel. In order for the reference monitor to restrict file access, each application includes a *manifest* file [24], which describes a `chroot`-like, restricted view of the local file system (similar to Plan 9's unioned file system views [41]), as well as iptables-style 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 a kernel 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` creation. 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.

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.

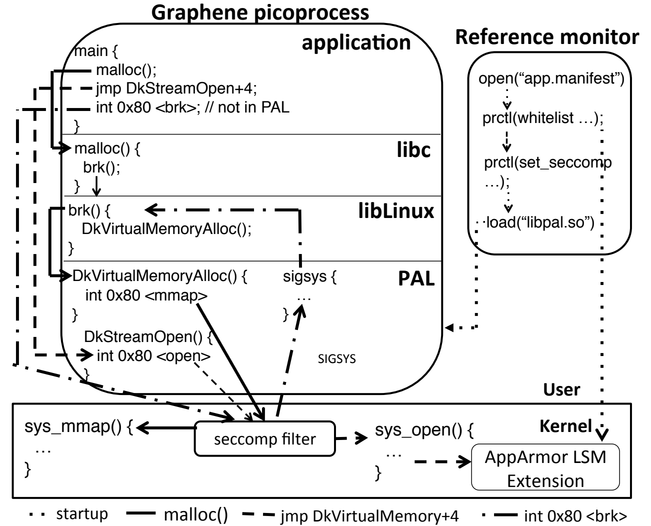

Figure 2. System call restriction approach. The reference monitor loads policies into the LSM at startup. A Graphene application requests OS services in three different ways. In the normal case (first line of `main`), `malloc` is invoked causing the invocation of `brk` (`libLinux`) and `mmap` in the PAL. In the second line, the application jumps to an address in PAL, which is permissible. The LSM checks the `open` system call. The third line invokes `brk` with an `int` instruction, which is redirected to the `libLinux` function.

Graphene restricts the host system call table using seccomp [44], 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 cannot be implemented as a seccomp rule. In order to avoid the overhead of trapping to the reference monitor on every `open` or `stat` system call, we instead extend AppArmor [1] to enforce file system isolation in the kernel.

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 aug-

ment 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 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.

4. Guest Coordination Framework

An application executes on Graphene with the abstraction that all of its processes are running on a single OS. Graphene libOSes service system calls from local libOS 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 and semaphores (Table 2). Similar to previous designs [10, 42], Graphene uses the host file system; the libOS implements file handles and translates between POSIX and the host ABI. Identifying the best division of labor for a libOS file system is left for future work.

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

4.1 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 libOS 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 [22]; 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 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.

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

4.2 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

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.

Table 2. Multi-process abstractions implemented in Graphene, coordinated state, and implementation strategies.

structures to call the appropriate signal handler directly. Graphene implements all three of Linux's signaling namespaces: process, process group, and thread IDs.

Figure 3 illustrates two sandboxes with picoprocesses collaborating to implement a process ID (PID) namespace. Because PIDs and signals are a libOS 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 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 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, 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.

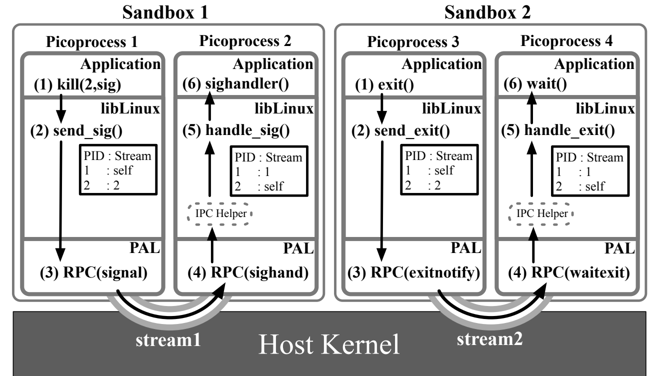

Figure 3. Two pairs of Graphene picoprocesses in different sandboxes coordinate signaling and process ID management. The location of each PID is tracked in `libLinux`; Picoprocess 1 signals picoprocess 2 by sending a signal RPC over stream 1, and the signal is ultimately delivered using a library implementation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

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 (§6.3) is attributable to semaphore overheads, and, in ongoing work, we will likely optimize this by using shared memory to reduce semaphore latency.

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.

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. None of our target applications have required a shared seek cursor, and it is not currently implemented, but would be a straightforward extension to current RPC mechanisms.

Failure and Disconnection Tolerance. Graphene is designed to tolerate disconnection of collaborating libOS 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 libOS 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. The Graphene prototype does not currently implement leader recovery, but the design makes recovery straightforward. If a leader failure is detected, by timeout or an RPC channel disconnection, a simple consensus algorithm over the broadcast channel is sufficient to elect a new leader, such as selecting the picoprocess with the lowest process ID. Assuming that all picoprocesses in the sandbox are trusted, leader state can be reconstructed by querying each picoprocess in the sandbox.

4.3 Lessons Learned

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

Service requests from local state whenever possible. Sending RPC messages over Linux pipes is expensive; this is unsurprising, given the long history of work on reducing IPC overhead in microkernels [14, 32]. We expect that Graphene performance could be improved on a microkernel with a more optimized IPC substrate, such as L4 [19, 28, 33]; 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.

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

Component	Lines	(% Changed)
glibc	606	0.07%
libLinux	31,112	
PAL	11,644	
Linux kernel IPC module	1,131	
AppArmor LSM isolation extensions	888	16.63%
Reference monitor	3,568	

Table 3. Lines of code written or changed to produce Graphene. Applications and other libraries are unchanged.

5. Implementation Details

Linux host 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).

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 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 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 3).

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 libOS.

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 3), 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 *previously established byte streams* within a sandbox. Handle passing facilitates inheritance and general-purpose RPC.

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 [21], which are essentially kernel-level wait queues.

6. Evaluation

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?

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

Table 4. 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.

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.

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 4 shows the time to start up a process, VM, or picoprocess; as well as the checkpoint and resume time for KVM and Graphene. Migration across machines is a function of network bandwidth, so we report checkpoint size instead.

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

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.

Figure 4 lists memory overheads of a diverse set of *unmodified* applications, including a `make -j 4` of Graphene's `libLinux` using the `gcc` compiler (v4.4.5), the lighttpd web server (v1.4.30) with 4 threads, the Apache web server (v2.4.3) with 4 processes, and the Bash shell (v4.1) executing the shell script test (`multi.sh`) from the Unixbench suite (v5.1.3) [2]. We measure memory utilization based on the maximum kernel-reported resident set size of each process or VM. For most applications, memory usage was fairly constant across inputs, so we only display representative examples.

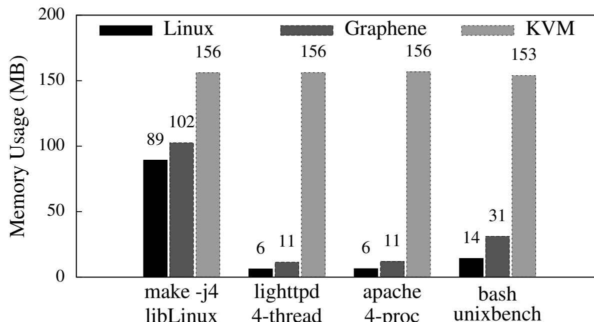

Figure 4. Memory usage of applications on Linux, Graphene, and KVM, in MB. Lower is better.

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`—§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 times 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) [6]. 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.

6.3 Application performance

Table 5 lists execution time of our *unmodified* application benchmarks (detailed in §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`

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

Table 5. Application benchmark execution times in a native Linux process, a process inside a KVM virtual machine, and a Graphene picoprocess with reference monitoring (+RM).

(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 [2], as well as 300 iterations of a simple shell script benchmark that runs 6 common shell script commands (`cp`, `rm`, `ls`, `cat`, `date`, and `echo`).

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

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

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

Table 6. 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.

6.4 Microbenchmarks

In order to understand the overheads of individual system calls, Table 6 lists a representative sample of tests from the LMbench suite, version 2.5 [36]. Each row reports a mean and 95% confidence interval; we use the default number of iterations for each test case. To measure the marginal cost of the reference monitor, we report numbers with and without the reference monitor.

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

The most expensive system calls occur when `libLinux` inadvertently duplicates work with the host kernel. For instance, many of the file path and handle management calls duplicate some of the effort of the host file system, leading to a 1–3× slower implementation than native. As the worst example, `fork+exit` is ˜5.9× slower than Linux. Profiling indicates that about one sixth of this overhead is in process creation, which takes additional work to create a clean picoprocess on Linux; we expect this overhead could be reduced with a kernel-level implementation of the process creation ABI, rather than emulating this behavior on `clone`. Another half of the overhead comes from the `libLinux` checkpointing code (commensurate with the data in Table 4), 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 imple-

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

Table 7. Microbenchmark 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.

mentation of an environment similar to FreeBSD jails [49] would make all reference monitoring overheads negligible.

System V IPC. Table 7 lists microbenchmarks which exercise each System V message queue function, within one picoprocess (in process), across two concurrent picoprocesses (inter process), and across two non-concurrent picoprocesses (persistent). Linux comparisons for persistent are missing, since message queues survive processes in kernel memory.

In-process queue creation and lookup are faster than Linux. In-process send and receive overheads are higher because of locking on the internal data structures; the current implementation acquires and releases four fine-grained locks, two of which could be elided by using RCU to eliminate locking for the readers [35]. Most of the costs of persisting message queue contents are also attributable to locking.

Although inter-process send and receive still induce substantial overhead, the optimizations discussed in §4.3 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.

6.5 Scalability

We compare the scalability of Graphene's RPC substrate with the scalability of Linux pipes, using a microbenchmark that compares the cost of ping-ponging a no-op RPC within a sandbox. For this experiment, we used a 48-core SuperMicro SuperServer, with four 12-core AMD Opteron 6172 chips running at 2.1 GHz and 64 GB of RAM. The performance of Graphene closely matches Linux (Figure 5), indicating that the Graphene RPC mechanism doesn't introduce any scalability bottlenecks above the scalability of IPC on the host OS (Linux). The relative performance differences are more variable above 24 cores, which we believe are the result of host-level scheduling. We hasten to note that these are worst-case stress tests; based on our application behaviors

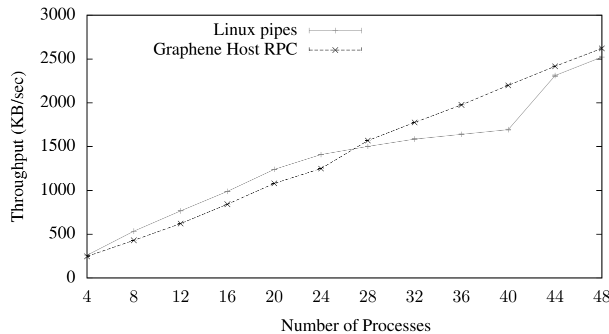

Figure 5. Scalability of Linux pipes and Graphene RPC on a 48 core machine. Pairs of processes concurrently exchange 10,000 1-byte messages.

as well as tuning experience, we expect RPC messages to be infrequent and to scale further in practice.

6.6 Security

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 [25] and is frustrated by the fact that `/proc` is implemented within `libLinux` and the system `/proc` is inaccessible from Graphene.

Analysis of Linux Vulnerabilities. Graphene restricts picoprocesses to 15% of the system call table. To evaluate the impact on system security, we *manually* analyzed all reported Linux vulnerabilities from 2011–2013 (a total of 291 vulnerabilities) [50]. We categorized these exploits by kernel component, listed in Table 8. 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

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 8. Manual analysis of Linux vulnerabilities from 2011-2013 and Graphene's prevention.

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 libOS cannot 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.

7. Related Work

This section surveys related work on library OSes, distributed OSes, and migration and isolation of a process.

7.1 Previous Library OSes

This work extends previous library OSes [10, 18, 34, 40, 42], which focused on single-process applications, to support coordination abstractions required for multi-process applications, such as shell scripts.

Bascule [10] 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 [40].

A number of recent projects have provided a minimal, isolated environment for web applications to download and execute native code [18, 23, 37, 52, 53]. 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 [4, 5, 15, 26, 30], 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. [26] 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 [17] (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 [7], and, unlike a library OS, does not deduplicate functionality.

7.2 Distributed Coordination APIs

Distributed operating systems, such as LOCUS [20, 51], Amoeba [16, 38] and Athena [13] 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 [41], 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 [9] 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 [47] 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 [9] 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 [31]. Although this model was abandoned as cumbersome for general-purpose access control [19], the Graphene sandbox design observes that a stricter variation is a natural fit for security isolation among multi-process applications.

7.3 Legacy OS support for migration and isolation

Researchers have added checkpoint and migration support to Linux [29] 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 [46], containers [12], and Solaris Zones [43], 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 [27]. FreeBSD jails apply a similar approach to augment an isolated `chroot` environment with other isolated namespaces, including the network and hostname [49]. Similarly, Zap [39] 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 [45] 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.

8. Conclusion

Graphene extends library OS designs to include multi-process APIs required by common applications, such as a shell or web server. Graphene demonstrates efficient, selective coordination of shared state across multiple library OS instances—maintaining host independence. Applications on Graphene enjoy both strong security isolation with acceptable performance and low memory overheads.

Acknowledgements

We thank the anonymous reviewers, Vyas Sekar, and our shepherd, Bryan Ford, for insightful comments on earlier drafts of this paper. Anchal Agarwal, Amit Arya, Imran Brown, Gurpreet Chadha, Naveen Kalaskar, Manikantan Subramanian, and Sourabh Yerfule contributed to the Graphene prototype implementation. This research was supported in part by NSF grants CNS-1149730, CNS-1149229, CNS-1161541, CNS-1228839, and the Office of the Vice President for Research at Stony Brook University.

References

[1] AppArmor. `http://wiki.apparmor.net/index.php/Main_Page`.

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

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

[4] G. Ammons, J. Appavoo, M. Butrico, D. Da Silva, D. Grove, K. Kawachiya, O. Krieger, B. Rosenburg, E. Van Hensbergen, and R. W. Wisniewski. Libra: A library operating system for a JVM in a virtualized execution environment. In *VEE*, pages 44–54, 2007.

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

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

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP*, pages 164–177, 2003.

[8] R. Baron, R. Rashid, E. Siegel, A. Tevanian, and M. Young. Mach-1: An operating environment for large-scale multiprocessor applications. 2(4):65–67, July 1985.

[9] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new OS architecture for scalable multicore systems. In *SOSP*, pages 29–44, 2009.

[10] A. Baumann, D. Lee, P. Fonseca, L. Glendenning, J. R. Lorch, B. Bond, R. Olinsky, and G. C. Hunt. Composing OS extensions safely and efficiently with Bascule. In *EuroSys*, 2013.

[11] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis. Dune: safe user-level access to privileged CPU features. In *OSDI*, pages 335–348, 2012.

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

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

[14] J. B. Chen and B. N. Bershad. The impact of operating system structure on memory system performance. In *SOSP*, pages 120–133, 1993.

[15] D. R. Cheriton and K. J. Duda. A caching model of operating system kernel functionality. In *OSDI*, 1994.

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

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

[18] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *OSDI*, 2008.

[19] K. Elphinstone and G. Heiser. From L3 to seL4: What have we learnt in 20 years of L4 microkernels? In *SOSP*, 2013.

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

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

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

[23] J. Howell, B. Parno, and J. R. Douceur. How to run POSIX apps in a minimal picoprocess. In *USENIX ATC*, pages 321–332, 2013.

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

[25] S. Jana and V. Shmatikov. Memento: Learning secrets from process footprints. In *IEEE S&P*, pages 143–157, 2012.

[26] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on exokernel systems. In *SOSP*, pages 52–65, 1997.

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

[28] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. sel4: Formal verification of an OS kernel. In *SOSP*, pages 207–220, 2009.

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

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

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

[32] J. Liedtke. Improving IPC by Kernel Design. In *SOSP*, pages 175–188, 1993.

[33] J. Liedtke. On micro-kernel construction. In *SOSP*, pages 237–250, 1995.

[34] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. Unikernels: Library operating systems for the cloud. In *ASPLOS*, 2013.

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

[36] L. McVoy and C. Staelin. lmbench: Portable tools for performance analysis. In *USENIX ATC*, pages 23–23, 1996.

[37] J. Mickens and M. Dhawan. Atlantis: robust, extensible execution environments for web applications. In *SOSP*, pages 217–231, 2011.

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

[39] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. In *SOSP*, pages 361–376, 2002.

[40] OSV. OSv—designed for the cloud. `osv.io`.

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

[42] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. Hunt. Rethinking the library OS from the top down. In *ASPLOS*, pages 291–304, 2011.

[43] D. Price and A. Tucker. Solaris Zones: Operating system support for consolidating commercial workloads. In *LISA*, pages 241–254, 2004.

[44] SECure COMPuting with Filters (seccomp). `http://kernel.ubuntu.com/git?p=ubuntu/ubuntu-precise.git;a=blob;f=Documentation/prctl/seccomp_filter.txt;hb=HEAD`, 2012.

[45] Z. Shan, X. Wang, T.-c. Chiueh, and X. Meng. Facilitating inter-application interactions for os-level virtualization. In *VEE*, pages 75–86, 2012.

[46] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *EuroSys*, 2007.

[47] X. Song, H. Chen, R. Chen, Y. Wang, and B. Zang. A case for scaling applications to many-core with OS clustering. In *EuroSys*, 2011.

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

[49] M. Stokely and C. Lee. The FreeBSD Handbook, 3rd Edition, Vol 1: Users's Guide, 2003.

[50] Linux Kernel Security Vulnerabilities (http://www.cvedetails.com). `http://www.cvedetails.com/`, 2013.

[51] B. Walker, G. Popek, R. English, C. Kline, and G. Thiel. The LOCUS distributed operating system. In *SOSP*, pages 49–70, 1983.

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

[53] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *IEEE S&P*, 2009.