Isolating Legacy Applications with Lind

by

Christopher James Matthews
B.Sc., University of Victoria, 2004
M.Sc., University of Victoria, 2007

A Dissertation Submitted in Partial Fulfillment of the
Requirements for the Degree of

DOCTOR OF PHILOSOPHY

in the Department of Computer Science

Isolating Legacy Applications with Lind

by

Christopher James Matthews

B.Sc., University of Victoria, 2004

M.Sc., University of Victoria, 2007

Supervisory Committee

ABSTRACT

Legacy applications, often written in C, can be riddled with bugs. Sarcastically referred to as "veritable bug ranches", pre-existing legacy applications of substantial size and complexity are still commonplace. In this dissertation, I motivate, build and evaluate Lind, a sandbox for legacy applications. Lind decreases the impact of buggy programs on the system that runs them. It does this without changing their code or destroying the non-functional characteristics of the programs—such as performance, portability, light-weightedness and ease of deployment—which are the primary motivators for legacy software written in C. Lind borrows many principles of secure system design to help it isolate legacy applications so they cannot impact the rest of the system. To assess Lind, I evaluate how well legacy applications perform in Lind, how strong the isolation Lind provides is, and how easy it is to port applications to Lind—all to conclude that Lind is a viable proof-of-concept platform for legacy applications.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Software bugs—unintentional flaws in software—are thought to cost the United States many billions of dollars per year [4], while global costs are estimated in the trillions [5]. A portion of these bugs are seemingly benign, but are actually security bugs. Security bugs are hard to identify and can have catastrophic impacts because they allow someone with malicious intent to take over a system. This situation is made worse by intentional government-to-government (or commercial) attacks known as cyber warfare.

This problem is compounded because a single bug in a system is often all that is needed to take over the entire system, or at least cause the system to crash or act slowly or incorrectly. Whether the manifestation of a bug is the result of an intentional attack or a non-malicious bug the results are the same, the system running the application can perform unexpectedly, lose data, or even crash—all to the detriment of other applications running on the system.

It is important to qualify ways applications can interact in the system. Applications can explicitly communicate. This work focuses on subtle interactions that occur when applications share resources. *Resources* are the parts of the system which runs the program. Common resources are the CPU, memory, disk and file system, network, peripheral devices and less tangible things like caches and buffers and even a systems supply of good random numbers.

Modern OSes are designed to provide policies and mechanisms for resource sharing. Though they are not evident in modern OS design, there are some security best practices which I draw on for inspiration. First, systems should be compartmentalized. That means they should be broken up into *fault domains*, so when one part of the system fails, the rest is not effected. There is a fundamental tradeoff OSes must make between applications sharing the system and isolating applications from each other's bad behaviour. Though this idea seems obvious, an application which is fully compartmentalized in terms of all the resources in the system cannot impact another application. In practice there has to be connections between parts of the system, so full compartmentalization is not simple. Modern OSes compartmentalize some resources naturally, sometimes even taking advantage of hardware based assistance, processes having virtual memory being a good example. Other things such as CPU and file system are not well compartmentalized. A delinquent process can easy impact other processes by using too much CPU or filling the file system.

The second best practice is that of the *Principle of Least Authority*[1] (POLA). The POLA was pioneered in the Multics operating system [6]. The principle says:

> Every program and every privileged user of the system should operate using the least amount of privilege necessary to complete the job. [7]

The modern interpretation of the POLA is to give each part of the system as little privilege as possible while still allowing it to operate. This philosophy can reduce the chances of a bug impacting the system as a whole because the part of the system

[1]Also known as principle of least privilege, and the principle of minimal privilege.

where the bug resides may not have the privilege to access many of the resources of the system.

This dissertation deals with *legacy applications*. Legacy applications as pre-existing programs, which still have some importance to their users, but are sometimes considered out of date in terms of their design. The kind of legacy applications Lind deals with are large programs written in C. This dissertation does not deal with legacy systems for which the source code has been lost. In concept this work can be applied to any programming language, though not directly. C is notorious for its weak type system and its ability to alter the programs memory in almost any way. Except in performance critical situations, higher level languages have replaced C as the choice for implementing new systems because of their memory management features, and genrally more structured approach to running a program. Some of the features C programs lack and their related bugs are:

- Garbage collection, or some means to free the programmer from explicit memory deallocation. Lack of garbage collection causes memory leaks which can result in denial-of-service attacks.

- A security model which has finer grained control mechanisms than the entire process and a security model which is not default allow. This violates the POLA. Default allow refers to the fact that by default most OS functionality is enabled, then if a program wants to, each type of functionality can be individually disabled.

- Bounds checking on arrays and buffers so that programs cannot write past the end of the expected memory. Array bounds checking and having an explicit string type solve these problems; however, unchecked this leads to buffer overflow attacks and data and stack corruption.

- A means to detect, and possibly recover from, unexpected behaviour, for example with exceptions. This encourages the system to be put in an unexpected state when an error happens but it is not explicitly checked for.
- System APIs constructed with expectation of invalid input and non-concurrent use. Poorly designed APIs encourage race conditions and bugs.
- Flexible variable scoping options. Scoping options means programmers are not forced to use only globals or locals.
- Silent integer overflow. When an integer silently overflows this can move the system to an unexpected state which can cause malfunctions or open new vectors of attack.
- Typing in variadic functions. Without first class support for variadic functions, format strings have to be used. This enables format string injection attacks and bugs from format strings which do not correctly describe their targets.

These problems, and others are discussed in more detail in Chapter 5. Suffice it to say that the environment in which a C programmer must work is inhospitable and without a large amount of rigor and knowledge about C, it is easy to unintentionally write buggy code. This means there are programs containing a large number of bugs, and OSes which do not fully limit the impact of those bugs. The critical issue is that legacy applications are still used!

1.1 Problem: Legacy Is Here To Stay

The problem is that these legacy systems are still valuable. They do what they are intended to, they cost a lot to rebuild; however, because of C's relance on programmer's skill legacy systems are more susceptible to bugs and attacks. One way to mitigate

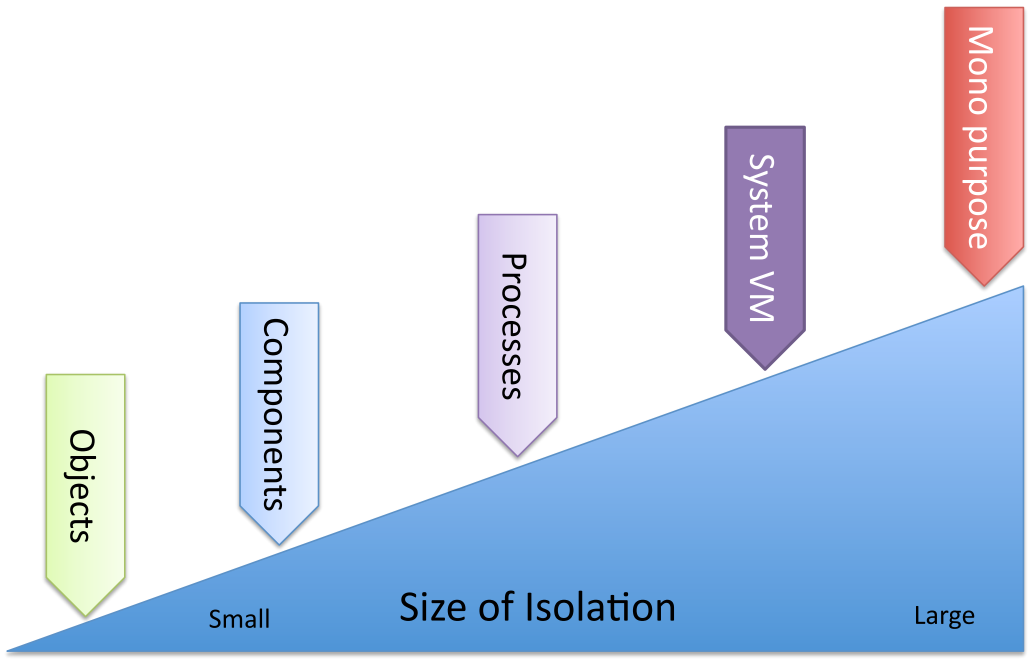


Figure 1.1: Some common forms of isolation and their granularity. On one end of the spectrum there are systems which provide fine grained isolation, for example at the object level, on the other end of the spectrum there are purpose built systems which are intended to only run one thing.

the impact of an attack on a system is compartmentalization—as long as the isolation between programs is sufficient. If you can keep the bug contained within the part of the system it came from, the rest of the system is not at risk.

The key to containment is resource isolation, if legacy applications can be isolated from the rest of the applications in the system, then the bugs damage can be limited. However, the default-allow policy which the OS provides to processes is ripe for impacting other programs on the system.

There are many systems which isolate programs, and there are many resources which need to be isolated each in different ways. One way to organize isolation systems is by the granularity of isolation. *Granularity* refers to the relative size of the thing being isolated. Figure 1.1 illustrates an isolation granularity continuum and notes some popular technologies which fall in certain parts of the continuum.

It is generally the case that technologies trade-off the isolation of granularity for no functional properties like performance, memory use, and ease of use. It is generally the case that as you move to the right of this continuum, the isolation is stronger, meaning more resources are isolated more strongly, however the cost of isolation also goes up. For example object-based isolation in a programming language might not isolate CPU or file system between each object, so they can interact at that level. A purpose built machine on the other end of the spectrum is expensive but offers perfect isolation—no resources interact. To shed some light on the costs of isolation, I will now quantify some of the costs of low-granularity isolation, in this case virtualization. The overheads and isolation properties of virtualization are well studied [8], so what is described here is how they compare to process level isolation.

1.2 What are the Costs of Isolation?

This simple experiment focuses on virtual machine (VM) communication. Virtualization is the gold standard in on machine isolation, because the interface a virtual machine supports is so simple it is easy to multiplex safely. However, that simple interface also mean that each VM must run a full OS inside and communicate with other VMs by means of a virtual network. This setup significantly complicates the software stack. A full OS and file system are run for each VM. Memory and CPU are statically partitioned, so that internal fragmentation can be an issue. Even hardware interrupts are delivered in a different way. This all means that virtualization's isolation comes at a cost.

To assess the impact of virtualization's isolation on simple timed communication experiments, I will first established the anticipated overheads associated with communication costs between virtual machines running on the same host. I will then contrast that with simple processes performing the same communication in a standard OS[2]. The benchmark runs as a simple server in each virtual machine. The server is responsible for waiting for connections and, once received, reading a simple message—which in our configuration of the benchmark, is just an integer value. This message is then forwarded to another server. This experiment can be thought of a high precision inter-process ping.

To measure the latency introduced over N hops, I setup N VMs with a server in each. Each VM forwards its' message to the next VM in a ring formation, so that the message eventually returns to the server that originally sent it. The benchmark measures the time from when the first connection was initiated, to when the message has travelled through all of the servers and arrived back at the originator.

The benchmark can be run in several different configurations, one configuration in which all the servers reside on the same VM, and another in which each server resides in its own VM. Figure 1.2 illustrates these two experimental configurations for $N = 3$ hops: in Figure 1.2 (a) all of the servers reside in a single VM, in Figure 1.2 (b) servers are distributed across domains. These two configurations can start to show us the cost of VM isolation versus process based isolation.

The server program is written in C. Each server process runs in one of two modes:

- Forwarding mode: receives request on a port, reads the message, and then sends that message to another IP and port.
- Master mode: sends a message once every t seconds to a specified IP and port, then waits for the message to come back. The master keeps the time difference from the beginning of the send to the end of the receive. This time difference is the round trip time (RTT) in the system, and what is measured by the benchmark.

[2]The content of this section is published in [9].

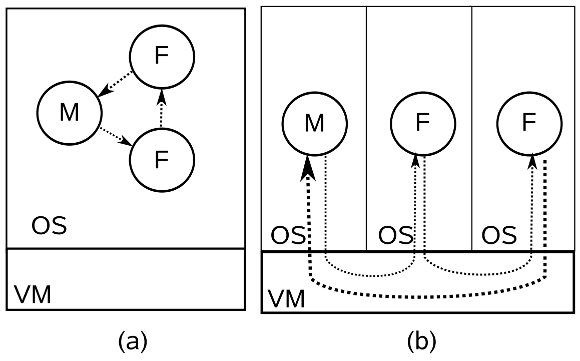


Figure 1.2: A master, M, and forwarders, F, in (a) a single domain configuration, and (b) cross domain configuration. Both configurations are for 3 hops.

A set of programs written in Python takes the role of a supervisory system. One program reads a configuration from a file or remote location and then sets up the server processes that should be running on the VM in which that particular instance of the supervisor program is running. The supervisor makes sure that processes run without aborting from unnatural causes, and aggregates their output and data. Supervisors run on each VM in the experiment, and coordinate with a central server.

The benchmark was run on HP ProLiant DL320 server with a dual core Intel Xeon 3000 series processor and 4 GB of memory. Three platforms were used: two different virtualization platforms and Linux. The Linux distribution used was CentOS 5.2 [10] on all the real or virtual machines. To comply with EULAs and because these results do not serve as a definitive comparison of virtualization platforms, we have anonymized their representation. To control for variability TCP network settings like TCP Nagle were disabled, the experiments were run on dedicated empty networking hardware and the machines had all but essential services and programs disabled.

The communication micro-benchmark was run 50 times, with 10,000 iterations in each run. The benchmark was set to make one hop, so it would just comunicate with itself. The resulting data set is rendered as a mesh diagram in Figure 1.3. The x-axis

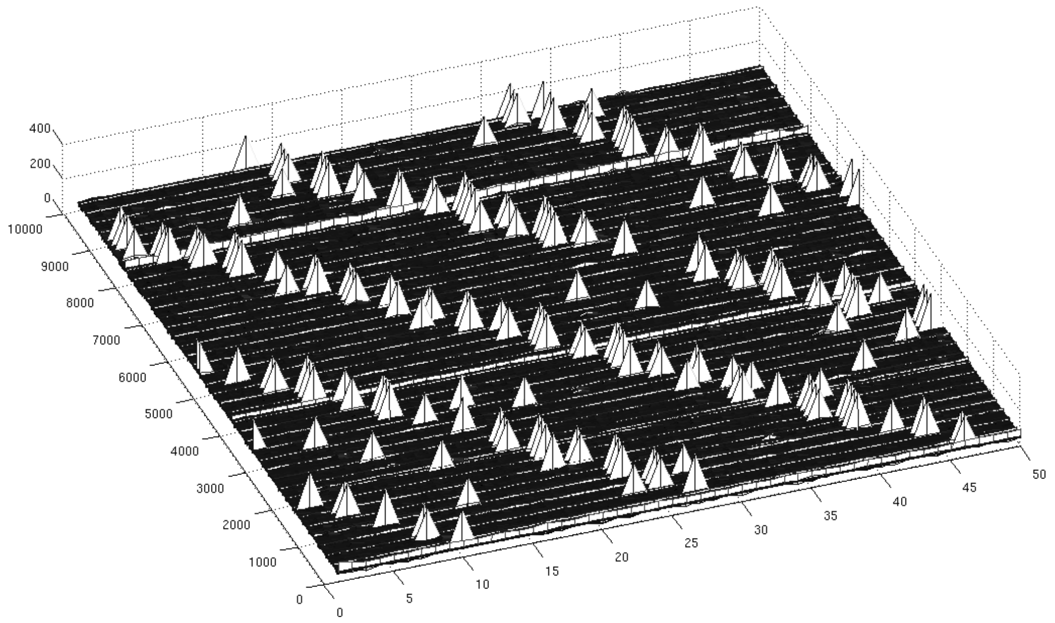


Figure 1.3: A mesh diagram of the round trip packet times. Iterations on the y-axis, experimental runs on the x-axis, and response time on the z-axis. Note the memory between runs at certain iterations.

of this diagram represents the run number, the y-axis represents the iterations of each run, and the z-axis represents the time a single iteration took.

This data was rendered as a mesh diagram to show some of its interesting properties that are not present in the non-virtualized runs. The first artifacts of interest in Figure 1.3 are the horizontal lines that appear at regular intervals. Those lines are specific iterations of the experiment which took longer. They tended to be about twice as long, moving from approximately 50 μsecs to 100 μsecs. This shows that after a certain number of operations, the system takes longer to perform the timed operation for an iteration.

The second artifact of interest is the larger spikes that occur diagonally across runs. These spikes tend to be about 4 times slower than the regular runs. An interesting feature of these artifacts is that they occur between runs, and of further consequence for testing scenarios, the pattern of these spikes reveals that the system has a memory between runs.

To further diagnose origins and possible ramifications of these patterns, two more micro-benchmarks were added to the experiment. These are specifically designed to establish characteristics of networking code and system calls in general. In these benchmarks all the networking code from our timed section was removed. In the first benchmark I left no system calls in the timed region, in the second benchmark I placed a single system call in the code where the networking was occurring. The system call used was a `puts` call, writing a simple string to the standard output stream.

Benchmarks such as these, with no networking, provide insight into the artifacts described above. When all the system calls were removed the results had no artifacts on any platform. When the non-networking system call is added in, the horizontal lines reappear. Upon closer inspection, the lines are there on every platform, but on the virtualized platforms they are much larger. I hypothesize that these lines are buffer flushes from the process's standard output buffer. That would account for their pattern and regularity. A buffer flush is also likely to be an operation that would take longer in a virtualized environment.

Finally, in the benchmark where the networking call was added back into the code, the diagonal patterns reappear in the virtualized environments. This indicates the networking code is the cause of the diagonal lines. This also can likely be explained by buffers in the networking system calls.

Table 1.1 shows some summary statistics from the runs of the benchmark. All the runs were similar in distribution, though VM2 was more variable. Although VM2 was on average slower than VM1 and Linux, it actually finished its runs before VM1. The slowdowns can be attributed to areas of code in the experiment that were not timed, such as the sleep and reset which happens between runs. In this experiment,

Platform	**mean**	**std. dev.**	**min**	**max**
Linux	54.40 μsec	5.05 μsec	52 μsec	421 μsec
VM1	50.36 μsec	5.53 μsec	32 μsec	269 μsec
VM2	81.85 μsec	30.82 μsec	74 μsec	2036 μsec

Table 1.1: The microbenchmark running on three different platforms: Linux and two modern virtualization platforms, VM1 and VM2 respectively.

in terms of timing VM1 behaved very closely to Linux where as VM2 was slower and more variable.

In the case of this workload and these test platforms, the choice of virtualization platform changed the magnitude of the results and their variability. In the next section I will show an experiment where virtualization has a completely different effect.

1.2.1 Quantify Costs with a Web Server

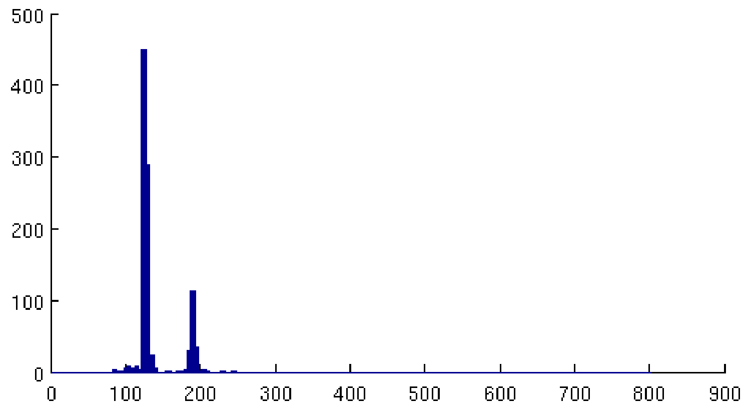


Figure 1.4: Histogram of response times of lighttpd running on single Linux machine. The x-axis is the response time, and the y-axis is the frequency of response at that time.

This third set of experiments is intended to help relate the above results to a more realistic scenario. These experiments record the response times for several configurations of lighttpd [11], a popular lightweight web server.

Using the same measurement framework from before, `HTTP GET` requests were sent to lighttpd, then recorded the time for lighttpd to respond with a 44 B *index.html* file. The experiment was run on the same systems already mentioned, and three configurations were tested:

Single physical machine: client and server were run on a single Linux machine.

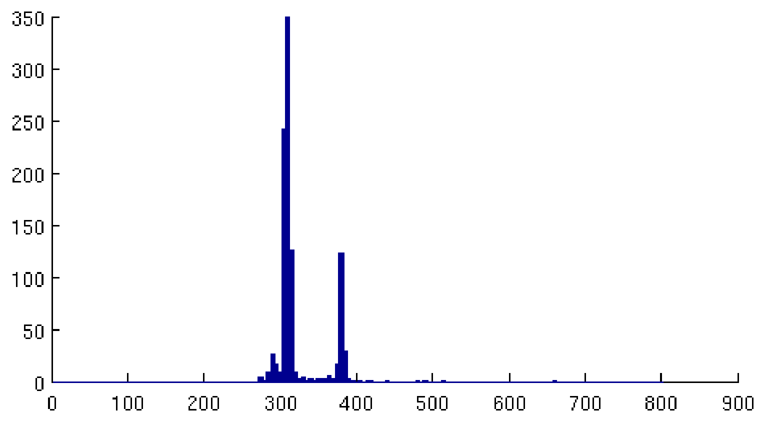


Figure 1.5: Histogram of response times of lighttpd running on two Linux machines. The x-axis is the response time, and the y-axis is the frequency of response at that time.

Two physical machines: client on one machine, server on the other. The machines were connected by a gigabit network on a switch with no other equipment.

Virtualized: client and server in different virtual machines on the same physical machine.

In these experiments the test setup was run 1000 times. Figures 1.4, 1.5 and 1.6 show histograms of the result of these three configurations.

The histograms highlight an interesting artifact of the virtualization. Figure 1.4 shows the plain Linux instance of the experiment on one host, which produces a bimodal distribution. Similarly, Figure 1.5 shows the same test spread over two physical hosts, also producing a similar bimodal distribution shifted to the left. As shown in Figure 1.6 in this experiment the virtualization actually significantly changes the distribution of the results. Furthermore, this change is not just a linear shift of the results, but an actual change in the shape of the distribution when the web server is virtualized in two domains on the same physical host.

Virtualization changed both the magnitude of the data, and the distribution. This quantification allows us to more carefully consider the ways in which tests such as

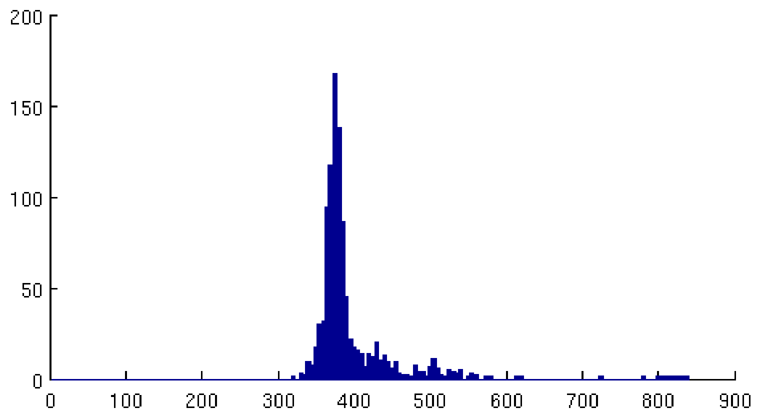


Figure 1.6: Histogram of response times of lighttpd running on two virtual machines on the same physical machine. The x-axis is the response time, and the y-axis is the frequency of response at that time.

these will map into real systems at scale. These experiments do not indicate that virtualization always makes systems run slower on all operations, but that certain operations do take much longer. The degree to which a virtualized deployment differs from a physical system and processes is dependant on workload characteristics, and does not scale linearly from the physical case. Not only does virtualization's isolation make the system slower, but it also makes the system's execution more irregular. This form of coarse grained virtualization is here to stay, as the benefits of isolation in cloud computing scenarios far outweigh the costs of these overheads and irregularities. Given this new norm, we are now faced with the question: what are the costs and benefits at other points along the isolation spectrum?

Virtualization is just one part of the isolation spectrum. I will discuss other froms of isolation in Chapter 5. All make tradeoffs in the isolation granularity space. Large granularity isolation like virtualization has its place. In this dissertation I explore a novel approach to a new form of fine grained isolation, which works at the process level.

1.3 Response: Sandboxing Legacy Applications

In this dissertation I introduce a prototype system called Lind. In Old Norse, Old High German and Old English a “lind” is a shield constructed with two layers of linden wood. Linden wood shields are light-weight, and do not split easily, an appropriate metaphor for a sandboxing system which employs two technologies. The goal of Lind is to run untrusted binary applications in a safe and light-weight manner, so that they are isolated from the rest of the programs on a system.

Lind is a application *sandbox*, a runtime which controls how an application runs and accesses the system that it runs on. Sandboxes control applications by restricting them in some way, then applying a policy to their interactions with the systems. Most sandboxes are for specific languages and operating systems (OSes), where as Lind is designed to run compiled binary applications on many platforms, all while keeping C’s non-functional characteristics like memory footprint and performance intact. This places Lind in a unique location in the isolation granularity spectrum. Lind makes containment and the POLA first class citizens in its design. Lind’s resource access is controlled through flexible polices, allowing a per-application policy approach to building systems. These all combine to make an environment in which it is much harder to unintentionally make a bug that negatively impacts the underlying system.

1.4 Overview

In the rest of this dissertation I will discuss show Lind was built, and evaluate its effectiveness as a proof-of-concept implementation, reflect on other similar systems, and discuss how Lind could be made better. Chapter 2 details Lind’s dual sandbox design and implementation, and why it is essential for the development of a safe, fast and lightweight sandbox. Then in Chapter 4 I will discuss a criteria for evaluation and evaluate Lind's proof-of-concept implementation. Next, Chapter 5 discribes other technologies which isolate parts of systems and reflects on how Lind relates to those technologies. Finally, Chapter 6 discusses some ways in which my proof-of-concept implementation could be improved, and then concludes.

Chapter 2

Design and Architecture of Lind

The goal of Lind is to run applications in a isolated and lightweight manner. This section describes the design goals and requirements of Lind, and then examines the design choices for our proof-of-concept implementation in the context of some best practices when building secure systems.

2.1 Design and Requirements

The design goal for Lind is to provide a sandbox where a legacy C program can be better isolated from the rest of the system, while having minimal impact on workflow practices and performance characteristics. Specifically, we want to make it easy to port legacy C-based applications to Lind, without imposing penalties in terms of execution throughput, and resource utilization, such as memory and disk.

The basic assumption that users make when running a program in a sandbox is that it will not harm their system. That is the basis of the isolation requirement and also a key factor in the effectiveness of Lind. Users will not (at least should not) run code if it cannot be guaranteed to not harm their systems. Unfortunately, there is no way to guarantee programs do not have flaws or backdoors in them, so we have to assume the programs could harm the system. These programs are called *untrusted*, because we cannot guarantee they will not harm the system. Programs could harm the system directly by reading or modifying state, for instance password files or by modifying the kernel, or harm it indirectly by slowing the system so other users can not use the system effectively or at all. Isolating memory and file systems is referred to as *spatial isolation*. Isolating programs so they cannot impact the performance of one another is called *temporal isolation*, or more commonly *performance isolation*. One application being able to slow down or stop other applications is an undesirable trait for the system because it leaves the system open to denial of service attacks and resource exhaustion attacks, hence, performance isolation is an important requirement.

The speed at which a program is executes useful work, called *throughput*, is important to every computer system. Adding the isolation properties discussed above can be done when a system is sufficiently simple. The Repy sandbox (discussed next), is a good example of that simplicity; however, applications that push the bounds of hardware cannot accept a major performance impact to be made "safe". For instance, cryptographic or scientific calculations can operate on long time scales, and imposing the 10 times overhead simply makes a system infeasible. This is one requirement that distinguishes Lind from other systems, our final requirement is to be lightweight. Specifically, to impose a low overhead both in terms of memory and execution throughput.

As I will discuss in more detail in Chapter 5, the heart of any solution for safe computation is a sandbox. A sandbox is an environment that can run programs while restricting their actions. Lind is a prototype architecture for a sandbox to satisfy the isolation and performance requirements described above. Like most sandboxes, Lind performs two key functions: computation and operating system access. For computation, Lind leverages Google's Native Client (NaCl) execution environment [12]. NaCl allows the efficient execution of legacy code in the form of x86 and ARM binaries that are built with a lightly modified compiler toolchain. For operating system access, Lind provides a subset of the POSIX API which is sufficient for many programs. This API is constructed using the Repy sandbox [13]. A detailed discussion of other sandboxes and isolation technologies can be found in Section 5.1.1.

2.2 Native Client and Seattle Repy

Efficient execution and providing a broad array of system services are the two goals of the Lind sandbox. To achieve these goals, the prototype implementation of Lind uses two sandboxes running in parallel: Google's Native Client (NaCl) [12] for running arbitrary native applications efficiently, and the Seattle Project's Repy Python sandbox [13] for running a library OS. The term *library OS* was coined by Porter et. al [14] and describes a system where the bulk of an operating system is implemented as a user-level library instead of in kernel space. To help shed some light on these sandbox choices, and why each makes a unique contribution to Lind, I will describe NaCl and Repy in turn. Both are covered in more detail in Sections 5.4.1 and 5.1.1.

A *native* application is one that is compiled for a specific hardware platform. For example, a C program compiled by the GCC compiler for an x86-64 machine is a native application; it is in a format that is native to the processor. Native applications are fast, and they can even be customized further by having embedded assembly instructions in them.

Software Fault Isolation (SFI), pioneered by Wahbe [15] et. al, is an alternative to hardware memory protection for running two untrusting programs in one address space. SFI restricts the execution of native applications so they cannot execute (write or jump) outside a "fault domain"; however, unlike virtual memory address spaces, transfer of execution between fault domains is fast because there is no hardware context switch involved. SFI is able make execution safe by restricting which instructions can be executed, so control flow is restricted to a region of memory. The original motivation for SFI was to load two programs into one address space, but not allow them to interfere with each other except via a specific interface. SFI is described again in more detail in Section 5.4.1.

In modern OSes, system calls are the means by which all resources external to a program's memory and computation are accessed. NaCl is a modern implementation of SFI [12]. NaCl's intent is to let web browsers safely run untrusted—computationally intense—native code. Unlike Wahbe's SFI implementation, NaCl is not focused on running two programs in one address space, but rather executing one untrusted program while disallowing it access to the operating system in any unforeseen way. Put differently, NaCl is designed to run one untrusted program and one trusted OS gateway in the same process. Games, 3D and sound- and image-processing, are all categories of web applications that benefit from deployment in NaCl.

Instructions in x86 have variable width. That makes it very hard to assess, for any given address, if a verifier is looking at the start or middle of an instruction. One of NaCl's techniques for building verifiable x86 code is to impose an instruction alignment pattern. NaCl makes use of a modified version of the GNU GCC compiler to produce verifiable native code for x86, x86-64 and ARM. Because of the restricted subset of native instructions NaCl uses, it can formally verify that the program can never violate the sandbox. Verification of code happens as it is loaded into memory, then the code is marked *readonly* so it cannot be changed by the program. The sandbox guarantees the program can never write to memory outside the sandbox or allow the control flow to move outside the sandbox without first passing through a trampoline which the system controls. A *trampoline* is an area of memory where once the control flow passes though it the control is guaranteed to be held by NaCl and not the executing program. NaCl can run most programs; however it presents a different system call interface from a POSIX system. The system call interface is that of JavaScript run in the web browser. The program has access to the web browser's DOM and functionality, but the actual system API NaCl presents is very limited, with only simple file operations, and no networking except those provided by the JavaScript interface. These abilities are only satisfactory for simple applications, and quite often significant effort is required to port an application to NaCl as is evident by the existence of the NaCl-ports project [16], a collection of pre-ported software. The way Lind safely expands the NaCl system interface is by using another sandbox which can safely access system resources.

Restricted Python (Repy) is the sandbox Lind uses to safely access system resources; effectively, Repy acts as a system library for Lind. Repy uses a restricted subset of the Python language and provides its own system API. Unlike NaCl, Repy is a language-based sandbox. Repy has a simple system API to limit its attack surface, and allows: private per-application file access, TCP- and UDP-sockets, threading and locking. These services are provided in Repy version 2 which has a simple API of only 34 calls [17]. 34 calls is particularly small when compared with the 311 system calls in Linux 2.6 x86-64 [18]. Repy uses a policy file to describe rate limiting and API restrictions for each program.

Repy was developed as part of the Seattle project [13], and is used to allow developers to run programs on machines which people from all around the world volunteer. Since the machines are the volunteer's, safety is of the utmost priority. Volunteers will not donate resources if there is a chance their machines will be compromised. As of 2012, it is estimated that Seattle has a deployment of 10,000 machines (J.

Cappos, personal communication, October 15, 2012). Seattle's primary use is for education [19–21], security [13, 22] and distributed systems [23] research.

2.2.1 Isolation Model

It is important to describe the assumptions of Lind precisely, to best define and evaluate the isolation Lind provides. The Lind isolation model is similar to most sandboxing technologies. It is partially based on the Seattle threat model detailed in [13]. Put simply, a program running inside of Lind should not be able to negatively impact the system on which it is run.

An application has to ask the system to perform privileged operations on its behalf. The goal of Lind is to restrict a program to some subset of those privileges. The bug or attacker might try to have the system execute privileged operations that were not intentionally exposed by Lind. When a bug causes these operations it may gain access to parts of the system that has information it should not have access to, or to cause a denial-of-service for other applications in the system. Lind does not attempt to stop the bug from performing malicious acts on other systems, though it can help prevent some kinds of malicious acts with a well crafted policy. For example a policy forbidding network connections could prevent spam delivery.

Lind runs arbitrary binary code inside the sandbox, which may accept any input data. The program does not have access to the underlying system except through the sandbox.

2.3 The Lind Dual-Sandbox

To provide native computation and safe access to the system, Lind combines NaCl and Repy. As depicted in Figure 2.1, untrusted programs are run in NaCl, as before, but access to all system-resources is diverted to a Repy program. This program is responsible for accessing the system on behalf of the program, it is called the Lind Library OS.

As depicted in Figure 2.1, the two sandboxes have a communication channel. To service a system call in NaCl, a stub routine marshals its arguments into a text string, and sends the call and the arguments through the channel to Repy. The Library OS then executes the appropriate system call, marshals the result and returns it through the channel; the result is eventually returned as the appropriate native type to the calling program.

Lind is designed to minimize its footprint within the trusted code base (TCB) of these two sandboxes. To achieve that, most of the Lind code is run from within the two sandboxes, the modifications to the sandboxes themselves (and therefore the TCB) was extremely small, and is discussed in Section 3.4.

The dual-sandbox mechanism completes the achievement of the isolation design goals through two features. First, the dual-sandbox ensures that all code can modify only device state, interact with devices, or interact with the outside world through the new trusted operating system interface. Second, the customizability of the interface ensures that the system can only: modify state, interact with devices, or interact with the world at a rate and in a manner specified for the application. For example, any attempt to send spam or execute a denial of service attack would trigger limits on resource consumption and/or allowable addressing, and would be prevented.

The dual sandbox also makes the construction of Lind simpler. The complex part of Lind is the Library OS which runs in Repy; however, Python is a very powerful language, so it significantly simplified the construction of Lind. Even though Python is considered “slow” by some, the internals of an application in Lind are run in NaCl, a very high performance environment. This balances the performance of the system, with the ease of implementation and maintenance of the Library OS component of Lind.

Most importantly, this particular design and architecture for sandboxing ensures the programs are portable. Programs running inside Lind are written to work against a standard POSIX `glibc` interface. The Lind runtime is strictly user-level and designed to work on many different platforms including Linux, MacOSX and Windows. There is more discussion about sandboxing and portability in Section 5.4.1.

Sandboxing also ensures performance isolation. It is used to limit resource consumption, both of computational resources (CPU, memory) and external resources (disk I/O and space, network bandwidth). The interposed system calls rate limit access and total consumption of each class of device on a configurable basis. CPU and memory limits are enforced on a per-process basis.

Finally, this kind of sandboxing ensures that the lightweight goal is met. Overhead for the Lind system is low because the sandbox only incurs overhead when there is a system call; Lind uses a native interface for execution, allowing CPU- and memory-intensive applications to run at speeds that are equivalent to NaCl and near native speed.

2.3.1 Addressing the Portability Goal

For Lind, portability is ultimately determined by the portability of the two sandboxing technologies used. Repy is very portable. Python itself runs on many platforms and architectures, and Repy has been shown to run under Windows (XP or newer), Mac OS X, Linux, BSD variants, and on many portable devices (Nokia devices, Android phones and tablets, iPhones/iPads). A large proportion of Repy code is dedicated to presenting a uniform interface to the sandbox across all these platforms.

NaCl runs on Windows, Mac OS X and Linux, on x86, x86-64 and ARM platforms.

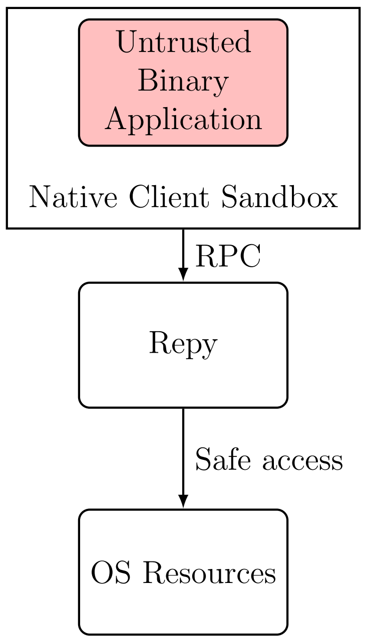


Figure 2.1: The architecture of Lind. System calls are intercepted in NaCl, and sent via a RPC to Repy where they are serviced.

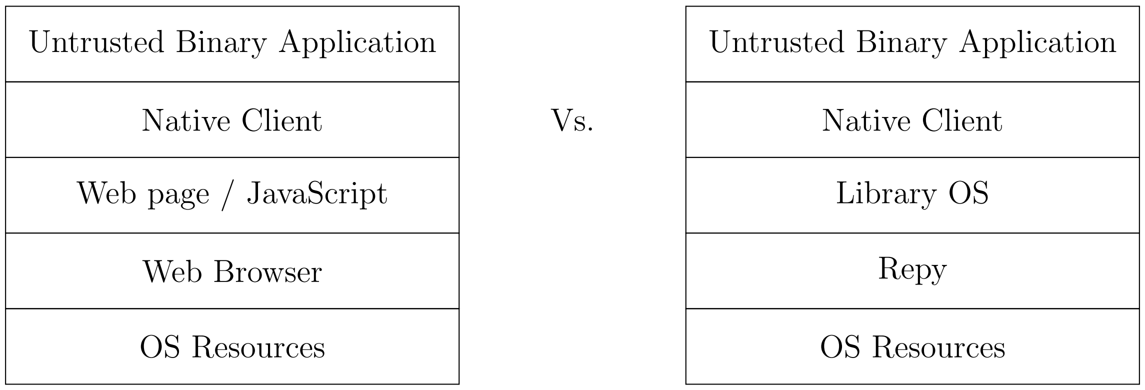


Figure 2.2: The software stacks of NaCl in the Browser and Lind.

Programs compiled for NaCl can use NaCl's PNaCl (Portable NaCl) format, which allows a single executable to be used for all three platforms.

2.4 Lind Compared to NaCl

NaCl was originally designed to be used with a browser, to run computationally-intensive programs under user control and at user initiation in a secure environment. It has been adapted it suit the Lind requirements. This adaption has resulted in no changes to NaCl itself (as detailed in Section 3.1), and a significant re-engineering of the environment in which NaCl programs and NaCl itself are run.

The differences within the systems are shown in Figure 2.2; the stacks are shown side-by-side to highlight their identical functions. The browser plays two fundamental roles in the standard NaCl system: as an application controller (more precisely, as the actuator for user application control) and as an OS sandbox. Lind lacks both a user and a browser; hence these roles are played by other actors. The role of the user in Lind is played by the remote server and it actuates through the library OS. The role of OS sandbox is played in Lind by the Repy sandbox. One way, therefore, to view the Lind system is then as a generalization of the NaCl environment. It would be possible to build and run a web browser in Lind.

2.5 Customizing the Repy Sandbox

A persistent weakness of sandboxing technologies is that they have fixed policies, independent of the applications which run in them. No access to the local file system is a common policy. However, in practice, many applications need restricted, customized, application-specific access to various local resources. As a trivial example, a solitaire program needs to update a score file, but should touch no other system resources. For Lind, and in particular to support the customized networking described in Section 3.1, sophisticated, highly-customized access to the network is required on an application-specific basis. Further, the computational resources of a given Lind application must be tightly-controlled on an application-specific basis using a mechanism called security layers [13]. A security layer is a reference monitor that programmatically blocks access to the system resources consumed by the application. When Lind is setup, it instructs the Repy sandbox to insert the appropriate security layers which in turn grant access if requests fall within the local policy permissions. Concretely, a Repy security layer can block access to a set of IPs / ports or prevent file system access selectively or altogether [13].

Applications in Lind are subject to network bandwidth restrictions set in the policy file and enforced by Repy. These restrictions allow control over networking resources, such as incoming and outgoing bandwidth available to the application. A security layer can be used to customize the resource controls for an application, like modifying the upload and download speeds independently. In addition to throttling network speeds, the networking API is also controlled by Lind, which disables networking calls not required by the application. Unintended network calls can therefore be caught and disallowed by the Lind runtime. Furthermore, restrictions are also placed on the ports available to the application. Ports must be explicitly listed in the applications restrictions file to be available ensuring that network traffic may only be processed on ports explicitly stated to be usable. In addition to restricting the ports, Lind restricts the number of connections allowed by an application. This stops the application from being allowed to flood the network with traffic and in some cases fill a NAT-enabled router's NAT translation table.

All settings are set to their most restrictive by default, then the policy file lists explicit exceptions; for example, file-system open and read are allowed for the OS server. The local policy file makes the same restrictions for the entire Lind runtime, and the application-specific policy is accepted only if it will not violate the global restrictions for the Lind runtime.

The dual-sandbox enforces the granted policy through a combination of techniques. The appropriate system call checks for network and file system permissions. Lind enforces the memory limits by terminating the application if it goes above the allocated memory. Lind enforces the CPU limits by suspending the process (with SIGSTOP on Mac and Linux) when the application approaches the allotted amount of CPU time and then resuming it later. Lind enforces bandwidth constraints by delaying traffic.

Below is an example of a policy file for Repy (and Lind):

```
resource cpu 1.0
resource memory 2500000000
resource diskused 1000000
resource events 1000000000
resource filewrite 100000
resource fileread 1000000
resource filesopened 5000
resource insockets 1000
resource outsockets 1000
resource netsend 100000
resource netrecv 100000
resource loopsend 10000000
resource looprecv 10000000
resource lograte 100000000
resource random 100000000

resource messport 9995
resource messport 9996

resource connport 9995
resource connport 9996

call gethostbyname_ex allow
call sendmess allow
call stopcomm allow
call recvmess allow
call openconn allow
call waitforconn allow
call socket.close allow
call socket.send allow
call socket.recv allow

call open arg 0 is junk_test.out allow
call open arg 1 is rb allow
call open noargs is 1 allow
call file.__init__ arg 0 is junk_test.out allow
call file.__init__ arg 1 is rb allow
call file.__init__ noargs is 1 allow
call file.close allow
call file.flush allow
call file.next allow
call file.read allow
call file.readline allow
call file.readlines allow
call file.seek allow
call file.write allow
call file.writelines allow
call sleep allow
call settimer allow
call canceltimer allow
call exitall allow

call log.write allow
call log.writelines allow
call getmyip allow
call listdir allow
call removefile allow
call randomfloat allow
call getruntime allow
call getlock allow
```

The policy file specifies the rates and limits the application will have access too, as well as which of the system calls are allow, and if needed what parameters can be used in those system calls. The Repy policy langauge is powerful, but it is not explored here because this work does not deal with determining policies, only enforcing them as they are stated.

This chapter identified the design goals and architectural decisions used in the initial Lind prototype. The dual sandbox described provides the isolation necessary to contain native applications, while still providing a high-level environment to build OS-like functionality. Policies are enforced according to application specific needs. The next chapter provides implementation details for the proof-of-concept dual sandbox.

Chapter 3

Implementation of Lind

A dual sandbox can be implemented in one process, or in two separate processes, either as both sandboxes in one process, or one in each process. Two processes gives extra protection from the architecture's hardware virtual memory protection, and any facilites the OS might provide — for example the chroot jails described in Section 5.1.1. The tradeoff made in a two process model is that there is a higher latency and overhead when communicating between the two processes. Using a multi-process model is strategy used in some user level systems to enhance security, for example the Chrome [24] web browser.

Lind runs its two processes in tandem: a NaCl instance that runs as a child of a Repy Python interpreter. Inside the Repy Sandbox, a program runs as the Lind operating system persona. This program services requests for operating system resources, and launches and monitors a NaCl instance running the untrusted application. Inside of NaCl, Lind applications use a modified version of glibc to redirect the standard system call interface through an Unix Domain Socket to the OS server. The OS server then services the system calls using Repy's facilities, then passes the results back to the modified glibc satisfying the original request. The only change needed to the user's program is that is has to be recompiled using the NaCl compiler (which is a slightly modified version of GCC), then be dynamically linked with the Lind glibc library. NaCl also provides a glibc interface; however, Lind's modified glibc provides a much richer system interface than NaCl's glibc because it can exploit the properties of the Repy sandbox.

Repy does not provide an interface to launch system subprocesses, so launching NaCl from an unmodified Repy would not be possible. To add this functionality a new Repy system call was added, `safe_execute`, to allow Repy programs to launch NaCl instances. `safe_execute` uses a process similar to the NaCl `sel_launcher` used in browsers to fork a new process with the program and arguments. When NaCl starts, it establishes an Inter-Module Communications (IMC) connection to communicate, these channels are opened and then handed over to the Repy program. IMC connections are reliable datagram connections, which present a socket interface to the programmer. The Repy program effectively controls the NaCl instance: the Repy program is able to query if the NaCl instance is still running, get its IMC channels, read and write to those channels, and kill the NaCl instance. All other operations are blocked from the Repy program. Using the `safe_execute` mechanism, a Repy program can act as an OS. Reproducing OS functionality at user-level is sometimes called a "Library OS", the next section discusses library OSes in more detail. The library OS Repy program runs an RPC server that allows it to service system calls which come from the NaCl instance.

Lind can be thought of as being structured like the classic Bridge design pattern [25] as Lind provides an abstract OS model to NaCl, which is derived from the Repy API calls.

3.1 The Life of a System Call in Lind

Conceptually, every system call in Lind is executed the same way. This is illustrated in Figure 3.1. The call is marshalled by a custom build of glibc, then forwarded to the Repy OS server, where it is dispatched to a handler. Forwarding to the Repy OS server is done by an RPC call over Unix Domain Sockets. Details of this path are discussed in the following sections.

3.1.1 Implementing Each System Call

Linux has a few hundred system calls; however, many are obscure. In our implementation, the system calls made by glibc broadly fall into three categories. First, system calls that are left alone and allow to pass through to NaCl. These are calls like `brk` for allocating memory. Second, calls implemented through the Library OS mechanism described above, such as `open` or `getdirents`. Third and finally, calls that must be faked because they have no direct analog in Repy—55 system calls for the Lind prototype. For instance, file system permission related calls were faked because Repy programs do not have access to the global file system or any sort of file protection mechanism. Similarly, Repy has no `size` function (in the form of `stat`) for a file, nor does it support directories.

In Lind, to provide these calls it emulates what an OS might do. For example, to emulate directories, the on-disk file name of a file is actually a unique ID, and then a mapping table is used to map paths and filenames to the on disk unique file. This mapping table is persisted on each state changing operation. Lind comes with a `fsck` like tool to reconcile irregularities in the metadata file, and a tools for migrating data in and out of Lind file systems. Other functions simply return dummy values or errors. For example, `getpid` returns a hardcoded PID (since `getpid` is not allowed to fail and

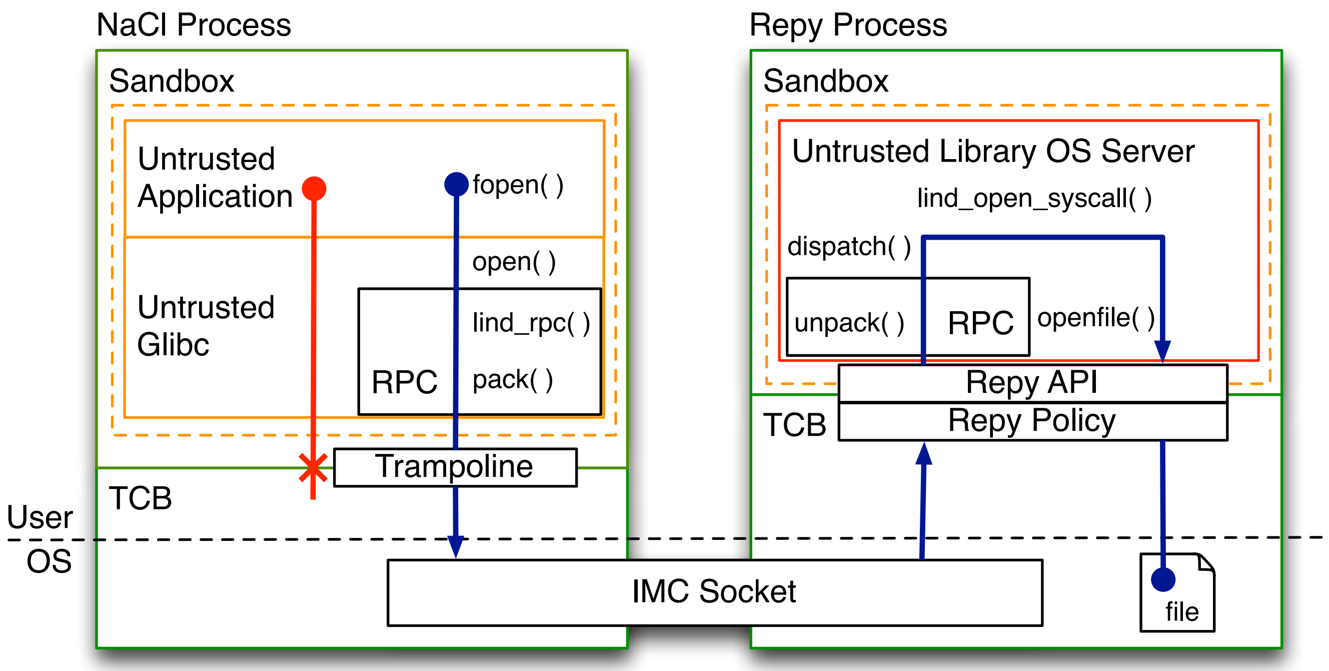


Figure 3.1: The path of a system call during execution. In red, a program tries to make an disallowed system call and is blocked by NaCl. In blue, a successful `open` system call is shown. `Fopen` is called in the program, which leads glibc to make an `open` system call. The `open` system call is replaced with a call to `lind_open_rpc`, that makes a Lind RPC call. The call's arguments are marshaled, then sent via an allowed write call on a pre-established IMC socket. In Repy, the OS server is waiting on the other end of the IMC socket. When the message arrives it is unpacked and the system call is dispatched by the Lind dispatcher to an implementation of the `open` system call. The dispatched calls passes though a module that adheres to the Adapter design pattern [25], and converts the arguments from the native C format to exactly what the Library OS expects. That implementation consults the file table for a free spot, then makes a call to the Repy API's `openfile` function to get a file object. The open file object is stored in a table, while the file handle number or error number is passed via a return RPC back to glibc finally completing the original `fopen` call.

programs don't need to know their PID), and some of the fields of `fstat` and `stat` have reasonable dummy values in them. Other functions that are not implemented just return `ENOSYS` and expect the programmer to deal with the error.

Repy API Name	Difference between POSIX system or Python analog
File system	
openfile	Opens files, not directories, name must be in [A-Za-z0-9]
file.close	
file.read	Rate limited, no cursor
file.write	Rate limited, no cursor
listfiles	Get files in local directory, no subdirectories
removefile	Does not remove if open
Network	
gethostbyname	less configureable DNS
getmyip	get the single external IP of this machine
sendmessage	Send UDP datagram, rate limited
openconnection	Destination and port can be limited
socket.close	
socket.recv	rate limited TCP receive
socket.send	rate limited TCP send
listenforconnection	Port is limited
tcpserversocket.getconnection	rate limited
tcpserversocket.close	
listenformessage	rate limited
udpserversocket.getmessage	
udpserversocket.close	
Threading	
createlock	
lock.acquire	
lock.release	
createthread	rate limited
sleep	does not wakeup early
getthreadname	
Miscellaneous	
log	replaces print statements
getruntime	no clock time, just seconds since program start
randombytes	rate limited hardware generated random string
exitall	Stops all threads and exits program
createvirtualnamespace	loads new code safely
virtualnamespace.evaluate(context)	runs loaded code
getresources	query limits and resources at runtime

Table 3.1: The Repy version 2 system call table.

The isolation of the Repy sandbox is partly due to its simple interface to the system. Table 3.1 lists all of the system calls a Repy program can make and some notable restrictions and differences between the standard Python and POSIX calls. To provide the full functionality of the OS from Repy Lind emulates an OS's behaviour in Repy. For some operations like file `open`, `read` and `write`, emulation is simple because Repy already supports them, though the exact flags and semantics of the calls are different. For instance, Repy does not provide the abstraction of a file cursor, so to make consecutive read calls in the emulated OS Lind has to keep a logical cursor for each file handle. However, in Repy other system call analogs simply do not exist. Repy has no `stat` call for a file, nor does it support permissions or directories.

Num.	Name	Implementation
1	noop	implemented in Lind
2	access	implemented in Lind
3	debug_trace	implemented in Lind
4	unlink	implemented in Lind
5	link	implemented in Lind
6	chdir	implemented in Lind
7	mkdir	implemented in Lind
8	rmdir	implemented in Lind
9	xstat	implemented in Lind
10	open	implemented in Lind and Repy
11	close	passed to Repy
12	read	passed to Repy
13	write	passed to Repy
14	lseek	implemented in Lind
15	ioctl	Case dependent
17	fxstat	implemented in Lind
19	fstatfs	implemented in Lind
23	getdents	implemented in Lind using files
24	dup	implemented in Lind
25	dup2	implemented in Lind
26	statfs	Faked
28	fcntl	implemented in Lind
31	getpid	Hard coded
32	socket	implemented in Lind, passed to Repy
33	bind	implemented in Lind, passed to Repy
34	send	passed to Repy
35	sendto	passed to Repy
36	recv	passed to Repy
37	recvfrom	passed to Repy
38	connect	implemented in Lind, passed to Repy
39	listen	implemented in Lind, passed to Repy
40	accept	implemented in Lind, passed to Repy
41	getpeername	implemented in Lind, passed to Repy
42	getsockname	implemented in Lind
43	getsockopt	implemented in Lind
44	setsockopt	implemented in Lind
45	shutdown	implemented in Lind
46	select	implemented in Lind
47	getifaddrs	implemented in Lind
48	poll	implemented in Lind
49	socketpair	implemented in Lind
50	getuid	Hard coded
51	geteuid	Hard coded
52	getgid	Hard coded
53	getegid	Hard coded
54	flock	implemented in Lind
55	rename	implemented in Lind (Lind specific call)
105	rpc_cia	implemented in Lind (Lind specific call)
106	rpc_call	implemented in Lind (Lind specific call)
107	rpc_accept	implemented in Lind (Lind specific call)
108	rpc_recv	implemented in Lind (Lind specific call)

Table 3.2: The numbered and named Lind system calls and the strategy used to implement the call. The name of the system call comes from the POSIX standard.

Table 3.2 lists all the system calls that are passed through to Lind, and describes what is done to service each call. The details of how each call is implemented for Lind vary widely. To illustrate how the system works Section 3.2 describes the file system implementation in more detail.

3.2 Implementing the File System in the Library OS

The core of the Lind file system is the open, close, read, write, getdents, stat, mkdir and rmdir system calls. These give the program the illusion of a normal file system even though Repy does not allow directories or access to file attributes, and provides different semantics for read and write. Below are the C prototypes for these system calls:

```
int open(const char *pathname, int flags, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int getdents(unsigned int fd, struct linux_dirent *dirp,
             unsigned int count);
int stat(const char *path, struct stat *buf);
int mkdir(const char *pathname, mode_t mode);
int rmdir(const char *pathname);
```

When Lind start, the file system does some pre-initialization. Using the Repy API, Lind reads a file named "lind.metadata" from the local directory. This file contains packed metadata. That metadata is from previous runs of Lind, and the data is loaded into the runtime file system data structures. There are three main data structures: a list of open file handles, and a Python `dict` of inodes and file metadata, and a mapping table to go from a file name and path to an inode number. All these data structures are stored in memory, and written to disk when they are changed.

The open system call is the normal starting point for most file system operations. Given a path, it will return a file descriptor to perform other operations like read and write. When Lind receives the open system call, it parses the path, traverses the path in the inode lookup table. When the Library OS finds the file, it use the Repy `openfile` call to get the backing file's object, it then picks a free entry from the file handle table, and stores a link to the inode and the file object. If the create flag is passed, it adds an entry to the inode and inode lookup table, and creates a new backing file. The backing files are not named the same as the actual files, but rather just "linddata.001", "linddata.002" etc. The simple names for the backing files allows us to store the real file name in the metadata, this is necessary because there are strict rules about the content of a filename in Repy. Finally, the call returns the index into the file handle table, or if an error was encountered, an error number to set the Unix `errno` value to.

The close system call removes the file handle's entry from the file descriptor table. The read and write calls use the file object obtained during open. Repy does not have the notion of a cursor, the read and write function both take an offset as well as a size to read or the string to write. A cursor is stored with the other file handle information, and updated on each read and write.

`Getdents` traverses the inode lookup table to the directory it targets, then reads all the entires from there. A Repy `listfiles` call is not even used since the metadata stores all that information. Likewise, `mkdir` and `rmdir` are just metadata operations on the inode lookup table.

Stat must return many values as shown in the struct below:

```
struct stat {
    dev_t   st_dev;  /* ID of device containing file */
    ino_t   st_ino;   /* inode number */
    mode_t  st_mode;    /* protection */
    nlink_t st_nlink;   /* number of hard links */
    uid_t   st_uid;     /* user ID of owner */
    gid_t   st_gid;      /* group ID of owner */
    dev_t   st_rdev;     /* device ID (if special file) */
    off_t   st_size;     /* total size, in bytes */
    blksize_t st_blksize;/*blocksize for file system I/O*/
    blkcnt_t st_blocks;/*number of 512B blocks allocated*/
    time_t  st_atime;    /* time of last access */
    time_t  st_mtime;    /* time of last modification */
    time_t  st_ctime;    /* time of last status change */
};
```

The device code `stat` must return is based on if the file is a normal file, or one of a few special files Lind supports like `/dev/random` and `/dev/null`. The inode number stat returns is the real Lind inode number. The file's mode is stored in the file metadata, as well as the ownership information. The size of the file is discovered by reading the entire file. This could be made faster by storing a size in the metadata, but it is not that frequent an operation. Block sizes are all hard coded. Times are fake in that they are not the correct wall time, but they are monotonically increasing.

3.3 Building a RPC System for C

Because of the number of system calls that Lind needs to support, and the similar steps needed to marshal each of them to the Repy sandbox, a program was built forward system calls. There are a few key concerns for a program like this, that it is correct for all system calls, and that the RPC stubs it produces are fast. Each time a system call is made, it has to be marshalled and sent, as the overhead of this process is added on to every system call it is prudent to minimize the overhead where possible.

The ideal design in this situation is a zero-copy send (one in which the data is not copied into a buffer before it is sent. Initially in the implementation of the Lind RPC, marshalling was a very error prone part of the code. To make development even harder, debuggers do not work in NaCl, and many 'useful' calls (for example printf, and string formatting) do not work at this level because they have dependencies on the system calls themselves. Marshalling requires knowing the type and direction (in or out) of each parameter of each system call. A simple call like `dup2` with the signature `int dup2(int, int);` is not hard. The parameters are two integers, file handles, and then a return code is passed back out. Whereas, a function like `recvfrom` has a signature `ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen);`. In order, the arguments are:

- the returned value is the size received on the socket or 0 for an error,
- the socket file descriptor,
- a pointer to a buffer to write into,
- the size of that buffer,
- any flags that might be needed by the call, as a bit mask,
- a pointer to another buffer that will hold the connection information in one of several different sized structs (normally sockadder_in or sockadder_in6),
- a pointer to the size of the connection buffer, that is over written with the new size after the call.

A systematic approach was needed to deal with the complex and error prone nature of the marshalling task. To approach the problem, a meta program was built, that produced the marshalling code. The program's output is a set of .c and .h source code files, with RPC stubs, that can be called from the appropriate places. The meta program was honed over time to produce fast error free code.

To define an RPC stub in the program, the user simply calls a function: `syscall( in, reference, out)`, the three parameters are lists of parameters, simple in (pass by value) parameters, then reference parameters, and out parameters. The reference and out parameters, have extra information in them to describe the type and size of each parameter, so the program knows how much to copy. This could be a fixed size like `sizeof(int)` or any expression, for example, one of the other parameters like `*addrlen` or any expression like `strlen(filename)+1`.

For example the `dup2` call:

```
syscall("dup2", [("int", "oldfd"), ("int", "newfd")])
```

and the `recvfrom` call:

```
syscall("recvfrom", [("int", "sockfd"),
                     ("size_t", "len"),
                     ("int", "flags"),
                     ("socklen_t", "addrlen")],
                    [],
                    [("socklen_t*", "addrlen_out", "sizeof(socklen_t
                       )"),
                     ("void *", "buf", "len"),
                     ("struct sockaddr *", "src_addr",
                      "sizeof(struct sockaddr)")])
```

The generator uses the `writev` system call to send the data over the Unix Domain Socket. This call takes a vector of (pointer, length) tuples. The `writev` call traverses this vector, and copies length bytes from each pointer. This means that the marshalling only copies the data once (from the vector locations to the other side of the socket).

The RPC stubs are generated at the start of every build of the system. This means as the RPC generator evolves, all the system calls are updated. This has made it much easier to add new features. Once such feature was a strace, which logged all the calls being made. Another, was a debug mode that dynamically checks the size of all the reference and out parameters, to ensure Lind does not write into any memory outside its bounds. Finally, when threading needed to be add to Lind, it was very simple to add locks to the system calls, so that the data structures could not be accessed concurrently.

Sandbox	TCB Lines of Code	Changes in TCB	Percent Increase
Repy	5866 SLOC	137 SLOC	2.3%
NaCl	138298 SLOC	472 SLOC	0.3%

Table 3.3: Lines of change to the TCB—as counted by SLOCCount [26].

3.4 The Lind Trusted Code Base

One of the goals of our implementation was to minimize changes to the Trusted Computing Base (TCB). The TCB is the part of the system that must be correct for the system to operate without being exploited. Table 3.3 shows that the changes to NaCl and Repy were small. As a result of this Lind has a very minimal impact on the system's TCB, and mainly consists of code to allow the launch of a NaCl subprocesses from a Repy program, and setting up the communication channels between the two. In NaCl this code was built following several other examples in the TCB. Lind's TCB is the native client sel_ldr, the NaCl validator and the Repy sandbox. Some minor additions to both sandboxes were needed to facilitate setup and communication between them. NaCl was set to use the NaCl Python IMC bindings. A new Repy system call was added to launch a NaCl instance, setting up communication between the two sandboxes with Unix Domain Sockets, and code to manage the native code and shared library path used within NaCl. The largest part of the Lind code is the system call subsystem, this all runs as untrusted code in the sandbox.

The TCB calculation omits two code bases, the OS's code, the compiler, glibc and the Python interpreter and any libraries that either use. Officially all those would also be in the TCB. In the case of NaCl the OS, compiler and glibc are in its TCB. In the case of Repy, all those plus the Python runtime and some libraries. Further, the NaCl TCB could be made smaller, the numbers presented here are all the lines that are in the official TCB sections of the NaCl code base; however, those sections include some research code and prototypes that could be removed from a production build.

3.5 The Lind Untrusted Code Base

Sandbox	**Lines of Code**
Library OS	3374 SLOC
Library OS Tests	964 SLOC
GLibC	2529 SLOC

Table 3.4: Lines of code inside the sandbox—as counted SLOCCount [26].

Table 3.4 shows the nearly 7000 lines of code added to Repy and Glibc outside of the TCB. In keeping with the design goals of Lind, the bulk of Lind's functionality stays inside the sandboxes, making it far less likely to be able to harm the system.

3.6 Local File System Integration: `lind-fuse`

Early in the development of Lind it was found that there needed to be a way to move files in and out of a Lind file system. At first this was accomplished by a buggy and hard to use Python script. Later a File System in User space (FUSE) [27] module was built, which is called LindFUSE. The module allows a developer to mount a Lind file system in the native machine's file system. The FUSE module mapped the native Linux system calls to the appropriate OS server calls. Since Lind already provided most of the file system interface, and does so in a similar manner to the native one, all that was done to connect FUSE to Lind was a simple pass-through. LindFUSE made it easy to copy in large numbers of files, change permissions and even edit files right in the Lind file system. LindFUSE allows for easier testing of the file system part of the OS server. Further, any application from the local system can be run in the mounted directory, and all its file system calls are sent to Lind. Others have used this arrangement to find bugs in the Lind file system triggered by VIM and MySQL, even though those applications were never ported to Lind.

3.7 Implementation Effort

A lot of the short commings of the Lind prototype are mainly due to a lack of implementation for the prototype. With more time, the performance of Lind could be improved, more system calls could be implemented, useful debugging tools could be introduced, and POSIX standards could be met more exactly. Despite this, the core Lind prototype is enough to explore the isolation trade-off Lind makes. In newer versions of NaCl debugging and profiling support have been added.

3.8 Discussion and Reflection

For Lind to work, each type of system call the application makes must be able to be serviced. Lind currently has many system calls (55+ calls), but still that is only 1/5 of the number of calls in Linux. One observation is that a lot of system calls are very purpose specific, and most applications make very few types of system calls. The limited set of calls that are implemented is still able to service a wide variety of non-graphical programs.

There are some limitations to the current approach that dictate which system calls can be implemented. First, there is no way in the current Lind model to set up shared memory. This is essential for system calls like `mmap` and graphical applications to work. There are always work around, for example in Xax (described in detail in the Related Work chapter) graphical applications are rendered by streaming `png` images down a socket [28]. Lind does support the `mmap` system call, but only because NaCl does, and the `mmap` implementation only works for shared libraries. It has been to the benefit of Lind that most applications do a good job of abstracting the platform the run on, so when calls like `mmap` do not work they transparently switch to reads. I will discuss how to do a real mmap implementation in Future Work Section 6.1.

Another class of system calls that Lind cannot support are `fork` and `clone`. These are process creation calls. Though the Lind Library OS can spawn new NaCl processes, right now there is no way to move data between NaCl processes. The lack of `fork` is currently the biggest problem with the Lind architecture. Many common security minded legacy applications, for example OpenSSH and Apache, use fork extensively. These missing system calls are discussed more in the Future Work Section 6.1.

Implementing each system call has been a development burden. There are a lot of system calls, a lot of flags, and some calls have complex behaviour. Since all the calls have to be reimplemented inside the TCB, this work is inherent in the architecture of Lind. In Section 4.3 I discuss the work needed to port Tor, and how long it took at add some system calls that were not yet implemented in Lind.

3.9 Summary

In this chapter I describe the design and implementation of Lind, a system designed to provide fast lightweight isolation for untrusted legacy applications. The requirements I had while designing Lind were for it to have good execution throughput, and low overhead, as well as strongly isolate applications from the rest of the system. Isolation can be broken into spatial isolation and temporal isolation referring to changing the state of resources or modifying the timing or availability of resources.

The enabling technologies for Lind are the NaCl and Repy sandboxes. NaCl is a SFI sandbox designed to run untrusted web content, Repy is a Python based sandbox designed to run untrusted distributed systems. Using these sandboxes Lind supports a threat model where an arbitrary binary application can be provided and it will be isolated from the rest of the system.

The design of Lind uses a dual sandbox, this is to provide a defence in-depth design, as well as to minimize the TCB of the system, and provide a good environment for building an OS (Python), while still providing the performance os NaCl for the application. The defence in-depth is achieved because the two sandboxes are used in conjunction, so an attacker would have to circumvent both to attack the system. The minimal TCB comes from the fact that NaCl and Repy can be connected and all the functionality to provide a POSIX OS can be done from within the sandboxes. Further, since both Repy and NaCl run on many platforms, so can Lind.

Repy is a very flexible sandbox, and provides the functionality to customize policies for Lind. This can be simple rate limits per application, or network whitelists, or even automatic end-to-end encryption of traffic—though that would only work between two Lind instances.

Concretely, Lind is implemented by connecting an application running in NaCl with a socket to a Library OS running inside of Repy. System calls are taken from the application in NaCl, marshalled, and sent to Repy to be serviced. The results are then sent back to the application. The Lind Library OS is responsible for performing, emulating or faking each system call, this effectively narrows the wide POSIX system call API to the narrow, and simple to understand, Repy API.

To simplify the building of Lind I produced a C to Python RPC library which can take the prototype of a system call, and produce the RPC code necessary to send it into Repy. To do this I built a simple C code generator. I also created a Lind FUSE file system, so that Lind file systems can be mounted into the local Linux file system.

Lind achieves its goal of having a small TCB, only modifying 2.3% of the Repy code base and 0.3% of the NaCl code base. Most of the nearly 7500 lines of code in Lind runs inside the sandboxes, and thus is not trusted.

Chapter 4

Experimental Methodology and Evaluation

Is Lind worth pursuing as a runtime environment for legacy applications? In building a sandboxing runtime environment, there are a large set of trade-offs to choose from. Lind's characteristics of performance, isolation, security and communication all interplay to make it a unique system in this trade-off space. When building new systems like Lind, all the dimensions of the trade-off space are interdependent. Furthermore, some of the traditional software engineering *ilities* are important too. Foremost, the deployability of Lind, both in terms of where Lind can run, and how easy it is to port applications to Lind, is critical to Lind's overall value as a system.

In the isolation model I describe in Section 2.2.1, Lind must run arbitrary binary code, which may harm the underlying system in any way possible from inside the sandbox. Further, from the system requirements described in Section 2.1 Lind has to be fast and light-weight.

In the context of the threat model and requirements presented, this evaluation answers four questions:

- Is Lind practical? Is it a burden for developers to port applications to Lind?
- Is Lind light-weight? That is, how much memory and disk are needed to run applications in Lind?
- Does Lind slow applications? Specifically, what is the execution throughput overhead on legacy applications?
- Is Lind isolated? Does Lind isolate legacy applications from the system? Specifically, given the isolation model, does Lind protect a system from the impact of bugs?

These questions are the most important criteria for the viability of Lind as a runtime for legacy applications. This evaluation sheds some light on each of these questions.

4.1 Methodology

Given the important questions above, the follow sections outline how the rest of this chapter will evaluate Lind.

4.1.1 Fun? Characterizing Ease of Deployment

For any legacy runtime, it is important to discuss how easy it is to get pre-existing applications to run in it. For a developer, if the effort to port a program to Lind is not worth the security and portability benefits, those efforts could be better spent elsewhere. The deployment part of the evaluation will characterize the practicality of Lind as a platform for legacy applications. This part of the evaluation will describe a experience porting a large application to Lind.

4.1.2 Fat? Characterizing Light Weightedness

Runtime throughput is not the only impact on a legacy application; disk and memory footprint are also important. Technologies like virtualization allow a user to run sandboxed legacy applications; however, a VM running an entire operating system rarely fits in less than 1 GB of memory. Memory overheads like this are prohibitive if each program has to be run in separate VM to maintain the isolation properties of a sandbox.

This part of the evaluation will characterize Lind's memory and disk consumption through simple micro benchmarks.

4.1.3 Fast? How to Characterize Throughput

Execution throughput is important for a runtime system. Though processors get faster, and memory gets cheaper, a wasteful runtime can still undo years of progress. The famous Moore's law describes processor performance doubling every 18 months. Given that performance curve, a 10-times slow down can make a modern system run an application at the speed expected nearly five years ago.

To understand the performance of Lind we have to be able to accurately characterize the execution time and memory overheads it imposes. In Lind, program execution costs fall into two categories, first, the code's execution cost over that of native execution, and second, the increased cost of servicing system calls over that of a normal OS's. For Lind, the execution cost is caused by NaCl. The system call cost is imposed by the communication overhead between NaCl and Repy, and the overhead of executing system calls in Repy instead of directly in the OS. This evaluation uses both micro and macro benchmarks to assess the execution throughput of Lind on several common applications.

4.1.4 Flawless? Characterizing and Evaluating Bugs

Claiming Lind reduces the impact of bugs is a claim that cannot easily be proven. Furthermore, it is likely that because Lind is a research prototype it is full of bugs itself! This evaluation describes how the design of Lind is inherently more isolated than running on the native system, and that Lind helps mitigate some classes of common bugs. This discussion will help show Lind's potential as a platform, but not it's current worthiness.

This evaluation will demonstrate that two types of bugs commonly used as attack vectors, privilege escalation attacks and resource exhaustion attacks, are not able to impact systems when programs are run in Lind.

4.2 Evaluation

4.2.1 Experimental Environment

All tests were run in a Virtual Box [29] VM running Ubuntu 10.04 LTS [30]. This VM is for Lind development, it comes pre-tooled with all the programs that developers needed to compile and run Lind and programs for Lind. Ubuntu 10.04 is a old version of Linux; however, it is the standard NaCl development environment so it was adopted it for Lind. Lind does run on newer Linuxes as well. In this evaluation, tests were run on many different machines, though always in the VM. The standard VM configuration is 6 cores, 8 GB of RAM and 20 GB of disk. This was the configuration used in most of the results, though some were run on other configurations of the VM on other machines with fewer resources. Like tests were always done on the same machine, such that results are not compared between machines or configurations.

For metrics involving execution time the system timer (`gettimeofday`) is used in C, and the Repy `getruntime` timers is used in Repy. The timers are of the same accuracy, though the Repy timer gives time as amount of time since program start instead of Unix time. On Linux systems both the timers are the same resolution, and come from the same time source. However, the source of the timer was the from the VM's clock, not directly the hardware clock.

In the application benchmarks timers were placed at the start and end of the program. That was generally at the beginning and end of the main function. If a program exited somewhere else besides the main function, added a timer there as well. This was done to separate the startup time from the actual program's runtime. Lind has not been designed or optimized for startup performance, so it takes a relatively long time to start Lind—around 2 seconds. This time is mostly spent starting the Python VM and NaCl. This cost would only be encountered if you were launching a lot of short running programs with Lind, and could be mitigated by the improvements suggested in Section 6.2.

Tests were run many times to assure that the results were predictable and repeatable. The number of times each test was run is based on the variance experienced, more variable tests were run with more samples. Some of the tests required setups that could not be easily automated, those tests were run fewer times.

4.3 Porting Tor to Lind

In this section I will describe the effort needed to port Tor to Lind. Tor runs in Lind unmodified; however, it took about two weeks of effort to make it run. The effort was primarily from two things, compiling Tor to run in NaCl, and fixing missing or buggy system calls.

Tor is an good candidate application for Lind. In Tor's default traffic forwarding mode it must accept connections from many untrusted and impossible to trace hosts.

Tor is a traffic forwarding system, and is intended to run in the background. Tor should not consume a lot of CPU memory or disk. Tor's bug tracker mentions that there have been remote code execution bugs in Tor. If Tor could guarantee that it is isolated from the rest of the system, perhaps more people would adopt Tor.

Before I describe in detail what was done, the high level process used to port Tor was:

1. Using Tor's autotools script, compile Tor for the local Linux installations.
2. Run Tor in `strace` to find out what system calls it makes.
3. Look at the autotools script to find out what libraries Tor uses.
4. Download the source code for each of those libraries.
5. Compile each of the libraries for the local Linux installation.
6. Modify each librarie's configuration system to use the NaCl toolchain.
7. Recompile and install each library into the NaCl toolchain.
8. Configure Tor to use the NaCl toolchain, and compile it.
9. Run Tor in Lind - it will fail because of bugs or missing system calls.
10. Find and fix bugs, implement missing system calls.

To better describe the general process of porting an application to Lind, Figure 4.1 is a flowchart depicting the process used while porting most applications to Lind. First the application is profiled with `strace`. That profile will tell the developer which system calls the application makes and what flags are used. Calls and flags which are not currently implemented in Lind should be noted, they must either be non-critical to the application, or implemented in Lind. The Lind manual lists the flags and calls which are currently supported in Lind, so those can be checked by the developer. Obviously checking every flag is very tedious, and it would be very simple to write a script that checks the `strace` profile automatically; however, since it has been Lind developers porting programs so far, they have a much better idea of which system calls

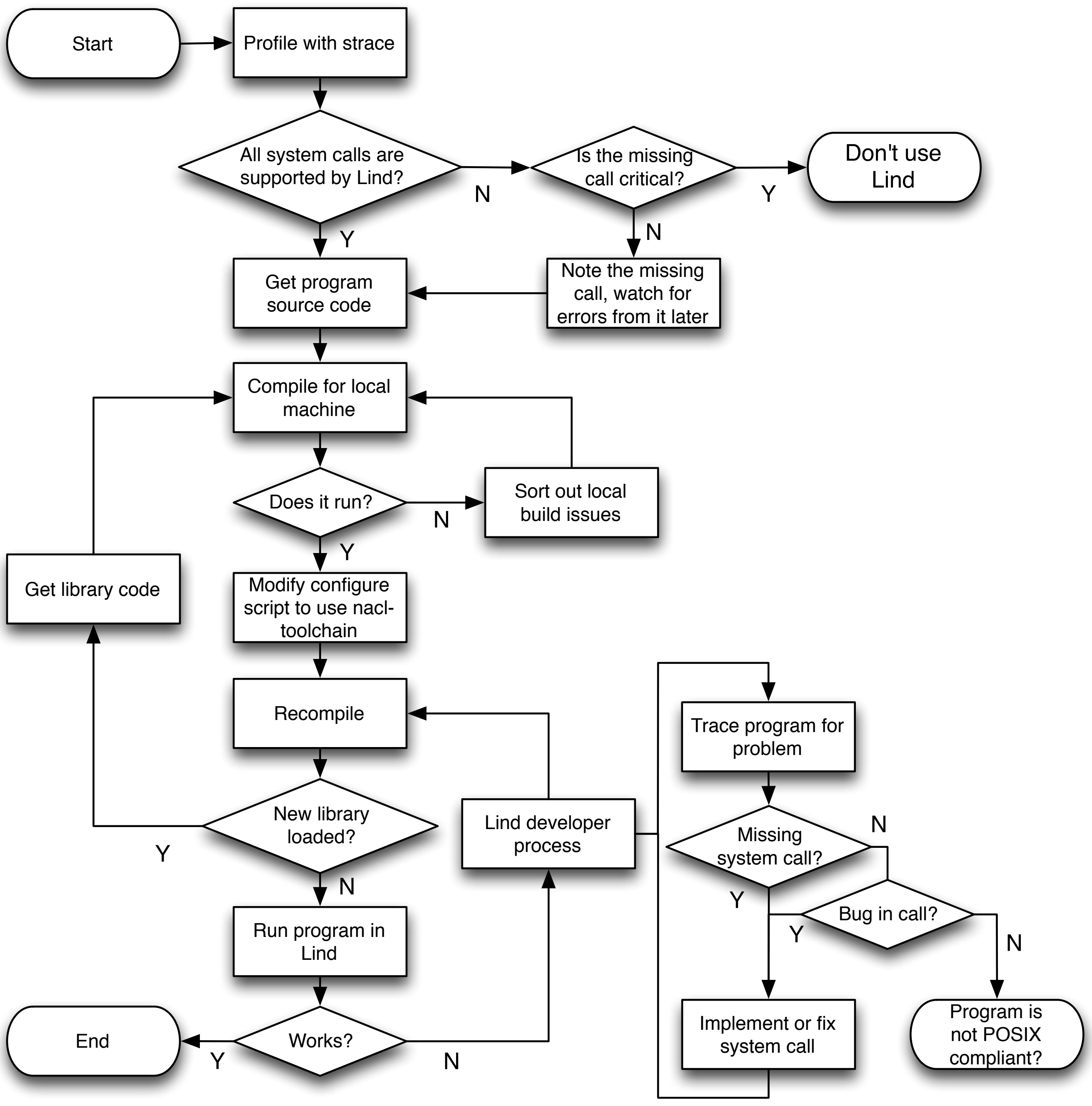

Start
Profile with strace
All system calls are supported by Lind?
N
Is the missing call critical?
Y
Don't use Lind
Y
N
Get program source code
Note the missing call, watch for errors from it later
Compile for local machine
Does it run?
N
Sort out local build issues
Y
Get library code
Modify configure script to use nacl-toolchain
Recompile
Trace program for problem
New library loaded?
Y
N
Lind developer process
Missing system call?
N
Y
Bug in call?
Y
N
Run program in Lind
Implement or fix system call
Program is not POSIX compliant?
End
Y
Works?
N

Figure 4.1: The process to port an application to Lind. The application and libraries must be compiled with the NaCl toolchain. While Lind is still in development, if the program does not work, the system is debugged for missing and buggy functionality.

are likely to cause problems. After the profiling the application has to be compiled using the NaCl toolchain. It is better to first compile the program for your local machine to work out any kinks in the compilation process before switching to the NaCl toolchain. To compile with the NaCl toolchain the applications build system must use the NaCl compiler `nacl64-gcc` (and sometimes programs like `nacl64-ranlib`, `nacl64-nm`, and `nacl64-asm`. Most build systems allow the developer to enter a *build prefix* which has the system transparently switch over to using all tools with that prefix in the name. If the application uses shared libraries, those must be compiled too. Then the application can be run. At this point it should work. If it does not this is a problem with Lind, and a Lind developer should fix it. The Lind problems encountered while working on porting applications were mostly small bugs in the system call implementations or totally missing system calls. For each, the appropriate action can be taken to fix or implement the call. To make this more concrete, I will now describe the process of porting Tor, the details of the errors encountered when porting Tor are discussed in Sections 4.3.1–4.3.2.

4.3.1 Compiling Tor

To get a program to run in Lind, it must be compiled with the NaCl compiler. The NaCl compiler looks like the regular GCC compiler—because it is just an adaption of GCC—however the executables are in different locations and have “nacl64-” prefixed on them. As an example of this prefixing, the command `gcc foo.c -o foo` would be replaced with `nacl64-gcc foo.c -o foo`. All of the commands in the compiler toolchain have this prefix. Custom paths must also be used, so for example, to get the Lind system headers instead of the default system headers `-I /nacl/include -L /nacl/lib` is used where `/nacl/include` is where all the .h files from the Lind glibc were installed and `/nacl/lib` is where all the NaCl compiled shared libraries

.so files are installed. Most build systems are setup so that flags can be passed to control settings like compiler location, and include locations. To modify a standard Makefile to compile for NaCl the following could be used:

```
CC = /lind/sdk/bin/nacl64-gcc
CFLAGS = -c -m64
INCLUDE = -I /lind/sdk/nacl64/include/
LDFLAGS = -melf64_nacl -m64
```

The trick to compiling larger applications is using their build systems to set these settings. When a developer wants to compile a program for a different architecture than what the system is installed as it is called *cross-compiling.* Most modern build systems (especially autotools) are created with this in mind, so adding new architectures is pretty easy.

Tor uses a standard autotools-generated build system. To compile it with the NaCl toolchain, it was simply a matter of adding a new sub-architecture (named "nacl64") to the autotools scripts. In every autotools project there is a file named config.sub that lists architectures and how to find those compilers. Once config.sub is modified, to compile Tor with the NaCl toolchain the developer just has to pass a flag to the configure script that makes Tor build for the "nacl64" architecture. Tor uses three non-standard libraries: `libevent`, `openssl` and `zlib`. These first had to be compiled separately.

Libevent

Libevent 1.4.14 [31] compiled with the same autotools changes mentioned above. Libevent was responsible for making some system calls that were not implemented in the Lind prototype. Those were system calls in the epoll family. Libevent is able to use the *best* available native polling mechanism so they were disabled so the `epoll` system call in `libevent` would fall back to one of the mechanisms that is already implemented, the `select` and `poll` system calls. Ultimately the select system call was forced by setting an environment variable that was accessible in Tor.

OpenSSL

Tor uses the OpenSSL library to encrypt its traffic streams. Unlike libevent's autotools scripts, the OpenSSL 1.0.0e library [32] uses its own hand-coded configuration system written in Perl. OpenSSL's configuration is invoked in the same way as an autotools script; however, it does not have the same options or configuration files. One option passed to OpenSSL told it to build without assembly optimizations, as hand coded assembly might not pass the NaCl validator. This assumption was not tested though. The rest of the options setup a "nacl64" cross compiler environment.

zlib

Tor compresses data streams with zlib [33]. zlib 1.2.7 was compiled from source. Like OpenSSL, zlib has its own configuration system. Configuration options were simple; however, there was a problem that stopped the configuration from working. At one point in the configuration, the script compiles and runs a program to check the machine's integer width. When it runs this program, it checks the output on stdout to see what the bit width of an integer is. In this case it expects a program that prints "4" to stdout. This check fails in Lind for two reasons, first, the stdout output of Lind has some debugging messages in it ("welcome to Lind"), second, Lind programs cannot be run natively, they have to be run by the `lind` command. To fix these issues, a `--silent` flag was added to Lind, so that it can be run with out any debugging output. zlib's configuration system was changed to prepend the command to execute the integer test program with the `lind` command—which other parts of the configuration already did correctly. This allowed zlib's build system to correctly configure and compile itself.

The Lind Library Path

Like the native program loader, to load shared libraries Lind searches a library path. The library path is a directory where shared libraries are loaded from. Since libraries are mapped into memory, validated and then executed, this path is treated specially throughout the system. One difference between Lind and the normal system is that the library path is only one directory instead of a list of directories. On most linux machines, libraries are installed to one of many paths, and all paths are searched. Once the libraries were built, to make them work in Lind the files had to be copied to the correct Lind library. If this copy was not done, Tor would compile (because the compiler checks more directories for libraries), but would fail in the program loader because it would not find the library files the compiler had access to.

4.3.2 Running Tor

At this point, Tor was able to run, but failed with an error. In the version of NaCl that Lind is based on there is no debugger support, so `printf` based code tracing is the best approach to see what is going on. Lind has a verbose option that causes it to print out all the system calls being made, and their arguments.

The first problem Tor had at runtime was missing configuration and support files. The location for these is hardcoded into Tor. A Lind file system was constructed, and the files copied in. The second problem was that libevent was failing on a missing system call. This was the `socketpair` system call. The third problem was that libevent was causing a segmentation fault when it was being initialized. This was traced back to more missing system calls, and then disabled those event handling modes in libevent. Those modes were device polling mode (using `/dev/poll`) and epoll mode (using the `epoll` system call). All the errors encountered are listed in Table 4.2.

Symptom	**Error**	**Fix**
Stop in loader	Lind library path does not have libraries	Copy libraries to library folder
Stop in settings parsing	Configuration and support files missing	Make a Lind file system with needed files
Segmentation fault in libevent init	Missing `socketpair` system call	Implement the socketpair system call
Segmentation fault in libevent init	Missing `/dev/poll` file and `epoll` system call	Disable the device poll and epoll modules in libevent
Assertion in lseek	The argument to lseek can be a long as well as an integer	Updated lseek call
Assertion in access system call	A previously unexpected combination of arguments was passed	Allowed the combination of arguments
`listen` system call fails	Tor requests a listen queue of 128, Linds current max queue was 100	Allowed queue length of 128
`socket` system call fails	Tor passes undocumented flag combination in the socktype parameter	Allow the socktype parameter to contain both a socket type in the lower bits and flags in the upper bits
`getpid` system call fails	The return value of the call was set to the length of the return value instead of 0	Make getpid return 0 on success
Tor can not lock settings file	`flock` system call is unimplemented	Implement `flock` call
`flock` call fails	Tor passes unexpected combination of flags to flock	Correctly parse unexpected flags
OpenSSL fails to load	OpenSSL ca not get random seed from the system because there is no `/dev/urandom` file	Implemented mknod system call with handles for a few custom files like `/dev/urandom` and `/dev/null`
`connect` system call fails	Tor makes more connections than Repy allows	Increase connection limit and free sockets from 10 to 100
Tor can make a certificate	Certificate made, but missing `rename` system call fails to put the file in the correct place	Implement the `rename` system call
Assertion failure in stat	Tor called stat on a device created by mknod, this was an untested code path	The device number in the stat struct had to be correctly serialized
`listen` call fails	Tor binds to port 0, which is a little known feature that tells the system to bind to any open port	On bind to 0, find a free port and use it instead of 0
Assertion in recv	Tor requests data from a socket larger than the RPC buffer	Shrink `recv` calls larger than the buffer size
Tor fails logging	`unlink` system call missing	Added C portion of the unlink call

Table 4.2: Errors encountered while porting Tor.

Task	**Time**
Building and reading strace profile	2 hours
Exploring Tor codebase	1 day
Downloading and compiling zlib, libevent and OpenSSL	4 days
Altering Tor autoconf script	2 hours
Setting up environment variables for Tor	1 day
Setting up Tor file system	1 day
Fixing missing system calls and bugs	5 days

Table 4.3: The developer time it took to port Tor.

Table 4.3The total time spent porting Tor was about two weeks. One week was spent on strace and compiling Tor and the libraries. The other week was spent getting the Tor file system ready, and finding bugs in the Lind implementation.

The nature of these errors highlights a few things about porting an application to Lind. First, as Lind starts to behave more like the Linux/POSIX specifications describe, fewer errors will be encountered. Specifically, the more system calls and options Lind supports, the fewer problems new applications will encounter. There is a finite set of system calls (about 300) and functionality, so as the common set of functionality gets covered, porting will be easier. Second, getting the system call interface "just right" is hard. Many errors were because of my (or other developers) misconceptions about how system calls will be used. There are projects on the formal verification of system calls [34], that would likely be useful to Lind. Third, without knowledge of the application being ported, that was the case for myself and Tor, a lot of time has to be spent during debugging to understand how the application works. In the case of Tor's 102 KSLOC that understanding took me a while, but would not for the application's developer.

One debugging technique which was tried without success was to annotate failing system calls with an assertion. To do this programatically required looking for all uses of the `ENOSYS` return code. If those locations could be found, it would be possible to track all failing system calls by placing failing assertions near those returns.

Unfortunately there were so many uses of `ENOSYS` in glibc, no easy automated way to annotate them was found. This might take a week of dedicated effort.

A lot of time was spent on the build systems of Tor and its libraries. Future developers would not have to worry about this as once an library is compiled for Lind it can just be reused. As a collection of common libraries is built up, it will be less likely that a developer will need to port a library.

4.3.3 Execution Throughput

To evaluate Lind's execution cost, the reader first needs an idea of the expected overhead for NaCl applications. The overhead associated with running NaCl modified native code is documented in [12, 35], and shows execution overhead (and slight speedups) in the 0%–10% range, slowdowns depend on the instruction composition of the program, and the number of system calls it makes.

To build on their NaCl performance work and to ensure its relevance to Lind, a micro benchmark called Primes was created. Primes is a memory and compute bound benchmark which calculates all the prime numbers below 1 billion. The Primes benchmark uses 64-bit numbers to do the computation, and uses a standard sieve algorithm.

To see the impact of Lind on compute intensive applications like the Primes, it was run natively, in NaCl and in Lind. The runtimes of each are compared. The Primes benchmark is simple enough that is can be run in all three environments. The rest of the applications in this evaluation are too complex to be run unmodified in NaCl because of their I/O requirements.

To evaluate the system call overhead, a simple micro benchmark that makes a single system call is used. Timers are placed all over the system call path, this allows us to focus in and see where the overhead of a system call takes place. The single system call overhead test is sampled 500 times, and run with one system call, `getpid`.

Although Lind is still a proof of concept, it can run many popular applications. Some common applications are used as benchmarks to assess how Lind handles real applications. These simple applications incude, grep, wget, and a web server. Two large programs are evaluated, netcat and tor.

4.3.4 Isolation

As mentioned in the Lind isolation model in Section 2.2.1, it is assumed that a program running in Lind has bugs that might make the program act not in the system's best interest. Lind's goal is to prevent the program from causing harm to the rest of the system. The two bugs this evaluation will look at are privilege escalation bugs and resource exhaustion bugs. Section 4.9 discusses why a privilege escalation in Lind is harder.

Resource exhaustion bugs are a simple denial of service problem where a resource that the program needs to do its work is used up so that the program (or another application) can no longer do its work. This could manifest as simply as a logger writing so many messages that the log takes up too much disk space and other programs can no longer write new files, or as a program using a lot of memory and causing other programs to be swapped out.

There are some best practices from the security domain to help prevent with resource exhaustion. For instance, it is recommended that the `/var/` directory in Linux be in a separate file system, so that a full file system can not stop the system from logging, and so that a full log file system can not crash the rest of the system. Another best practice saves a small percentage of the file system for the root user, so even when the file system gets full the root user can run programs. By allowing you to set resource limits for programs, Lind lets you ration resources effectively. Ration programs on a per-application basis and ration every resource, not just disk.

For every run of a Lind program, a restrictions file must be supplied. This file specifies the quantities of CPU, memory, read-, write-, and other-disk operations, and network bandwidth each Lind application can use. This is similar to the authorizations that modern cell phone OSes use. The restrictions file also authorizes some special system resources like access to time, high quality random numbers and specific TCP and UDP ports.

This evaluation shows that the Lind restriction mechanisms work, and can either block or rate limit the resources they guard. These restriction tests take place with the rest of the execution throughput performance tests below, and show that rate limiting will effectively slow or stop programs that are using more resources than allowed.

4.3.5 Compute Workloads

A benchmark called *Primes* was created. It performs a simple prime number exploration. Primes is a classic computationally bound problem with minimal I/O. The Primes benchmark calculates the primes below 10^9, and stores them in an array. It uses the standard sieve technique, that allocates space for the array, then calculates the primes in a tight loop, and then finally prints the number of primes found and exits.

Though the Primes benchmark is not complex, it produces two security concerns, both can result in slowing or terminating other important applications: first, memory consumption can grow too large; second, the program can monopolize CPU cycles. Imagine a server where your program was running along side hundreds of these Primes benchmarks, then they all suddenly allocate so much memory your program is swapped out. One would hope the OS's resource scheduling handles this gracefully, but the reader will later be shown that is harder to do with resources like disk were the OS cannot over commit.

The policy mechanisms described in Section 2.5 allow us to address these issues. The resource file for the Primes application is about 15 lines of simple statements. For example, the policy to describe the CPU and memory has two commands: `resource cpu 1.0` will limit the program to one virtual CPU's worth of computation and `resource memory 25000000` sets the maximum memory to 25 MB. File system, networking, and other unneeded calls are disabled by setting a Repy security layer. Disabling these calls helps Lind adhere to the POLA.

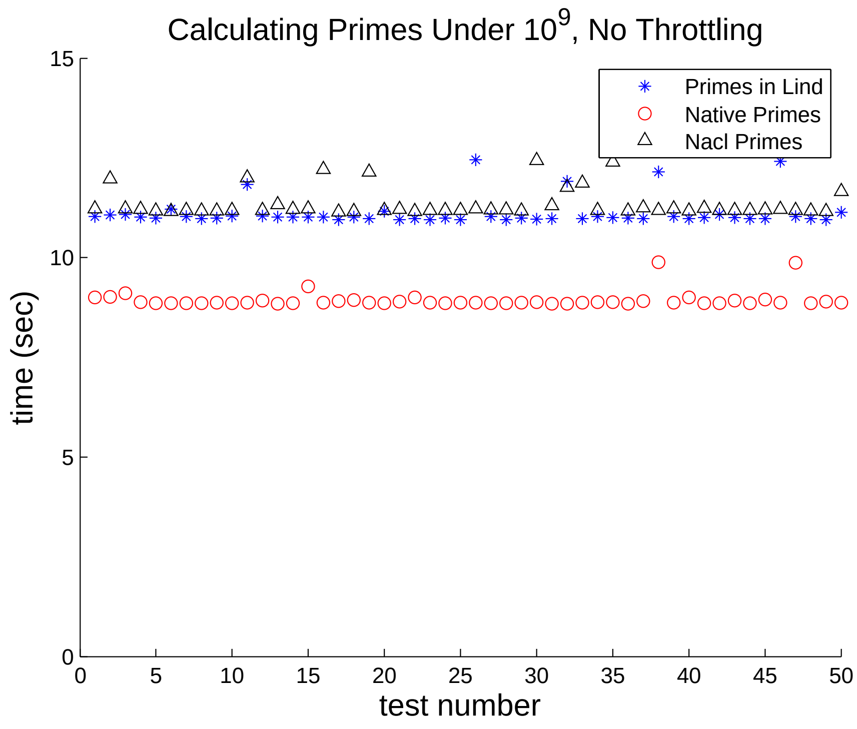


Figure 4.2: The execution time for 50 runs of the Primes benchmark, running natively, in NaCl and in Lind.

The Primes application was run 50 times in each of the native OS, NaCl and Lind. The time from the start of the calculation to after the result count prints is

Environment	Min (sec)	Max (sec)	Mean	Standard Deviation
Lind	10.9463	12.4500	11.1245	0.3586
NaCl	11.1440	12.4330	11.3531	0.3505
Native	8.8403	9.8779	8.9367	0.2081

Table 4.4: The execution time for 50 runs of the Primes benchmark

measured. Figure 4.2 shows the primes application execution times in Lind with 80% the performance of native. Both NaCl and Lind had the same distribution of results which was bimodal, with a median of 11.007 for Lind and 11.196 for NaCl. Both had some outliers, in the 11.8-12.4 range, but these were less than 10% of the runs. Other compute bound programs are expected to behave similarly. Most importantly, Lind performs at the same throughput as NaCl, so the introduction of Lind does not decrease the compute performance of an application's native code execution. Thus, compute based workloads that will perform well in NaCl should also perform well in Lind.

4.4 Overhead of a System Call

This micro benchmark quantifies the overhead of a single system call, and classify the parts of the system call's execution that take the most time. As described in more detail in Section 3.1, the path of a system call starts in the program, passes through glibc, through the Lind RPC module, and then into an IMC socket, where it is transmitted to Repy. In Repy a system call is received on the other end of the IMC socket, dispatched, then serviced by the Library OS. The reverse path is then taken to return the results back to glibc and ultimately the program.

Since the code spans NaCl and Repy, user- and kernel-level, and two programming languages, using a profiling tool was not possible; however, it was possible to insert timers in the code. At specific points in the execution, the system clock is read, then

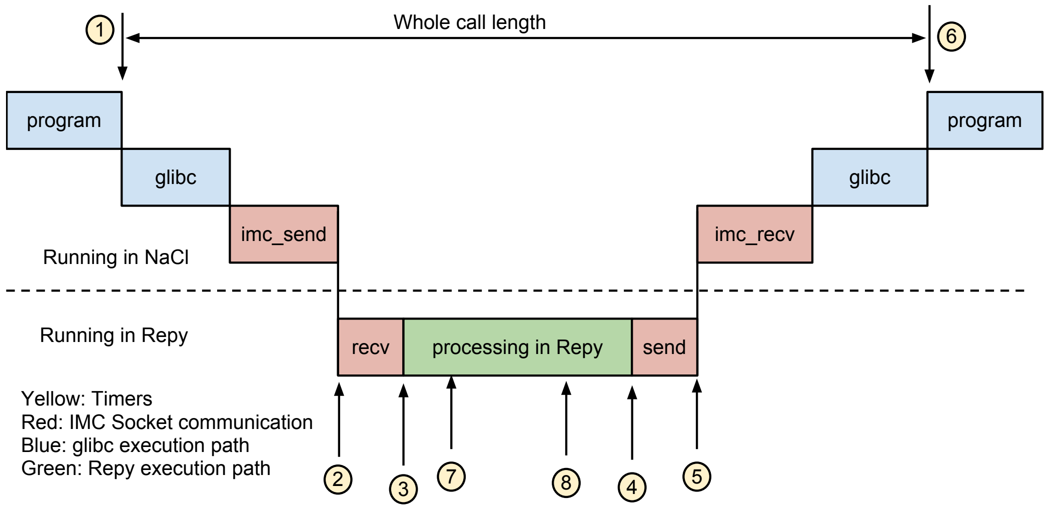


Figure 4.3: Timer locations on the path of a Lind system call.

stored in an array. Those values are subtracted to find out how long each operation took. Figure 4.3 illustrates the timer positions in Lind and the program.

The overhead for specific system calls was checked, measuring the time spent receiving, processing, and sending back the system call. The `read`, `write`, `open`, `close`, and `seek` system calls were all examined. For this evaluation I show the `getpid` system call, because its implementation is simple. The experiment was run with 500 runs. As illustrated in Figure 4.3, times were taken at eight sample points in the path of the system call's execution. Subtracting timer values allows us to break down the overhead times into their different parts. For example, the actual system call is performed in Repy between timer seven and eight, $(t_8 - t_7)$, and $(t_3 - t_2) + (t_5 - t_4)$ is the time spent on socket I/O in Repy.

Figure 4.4 shows 500 runs of the `getpid` system call. The trace shown also includes other system calls, so the initial system calls which are needed to start the system are included. You can see those calls take longer to service than the getpid calls. The experiment takes about 75 runs to reach a stable state, before that there is some variation caused by the system settling down.

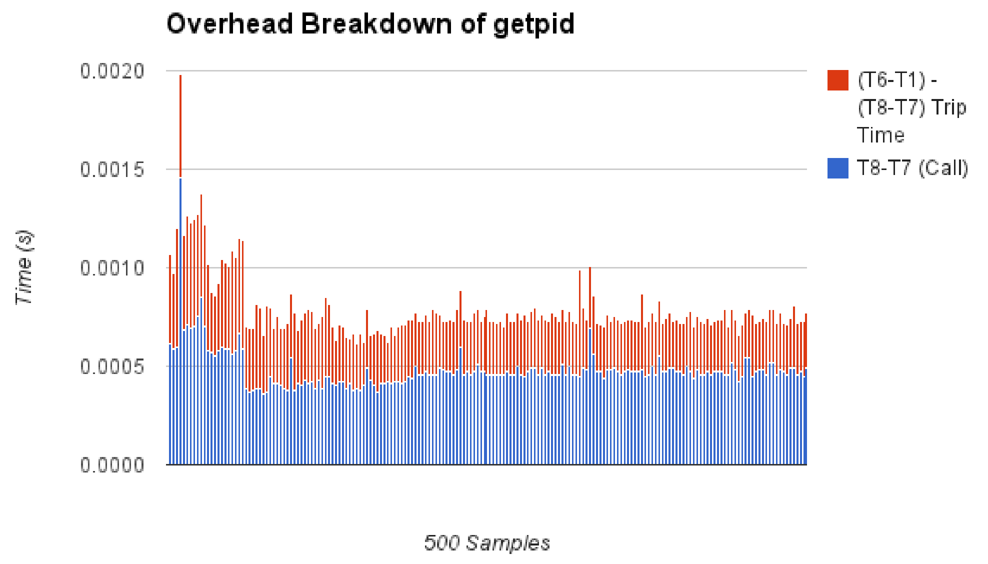


Figure 4.4: Execution time breakdown of the `getpid` system call.

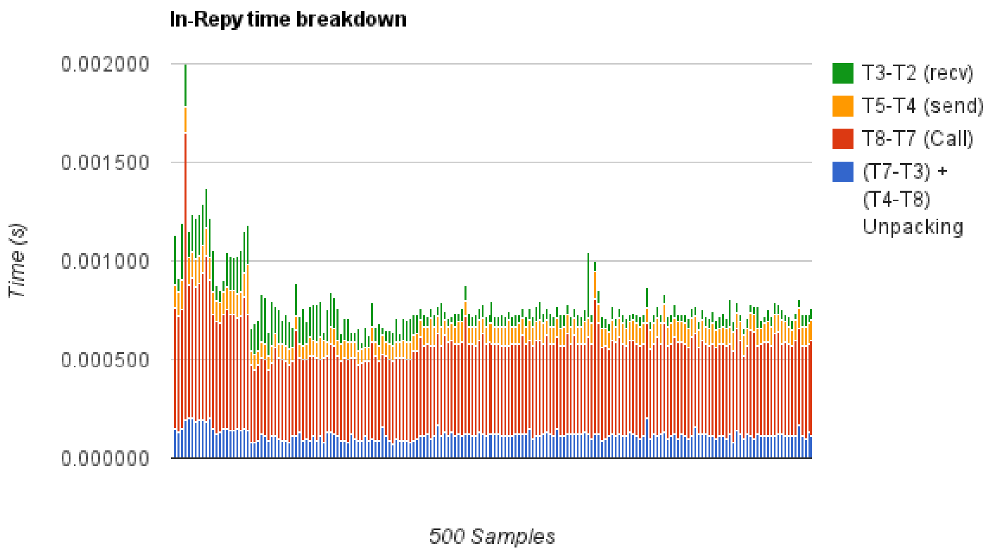


Figure 4.5: Execution time breakdown of the `getpid` system call in Repy.

Component	**Mean time**	**std. dev.**	% of total
Total time	0.0007474 s	0.00006055 s	100%
Call time	0.0004648 s	0.00004930 s	62%
Send time	0.0000920 s	0.00001500 s	12%
Recv. time	0.0000713 s	0.00004416 s	9%
Packing time	0.0001141 s	0.00003933 s	15%

Table 4.5: Time components from the stable part of Figure 4.5.

Figure 4.5 elaborates on the overhead incurred in the Repy part of the system call, showing time spent sending and receiving data, making the system call and marshalling the call's arguments. On average 62% of time of the `getpid` call spent in the Library OS, 20% is lost in IPC overhead, and 15% is lost to the RPC marshalling overhead. Lind's `getpid` system call returns a hard coded value; however, in this experiment Lind was run in "safe mode" when many extra assertions and tests are performed on each call, and log entries are printed to a file and to the screen. This is why the `getpid` call takes longer than you might expect at first glance. Both figures have spikes in the data, they are assumed to come from scheduling context switches. The initial 10 system calls are open and stat system calls, so they were not counted. Figure 4.5 indicates that communication takes some of the time, however, the call and marshalling code take a lot of time as well. Both the call and marshalling systems are candidates for easy optimization.

As an update to this section, later in the Lind development the Python profiler was run on Repy code. With the profiler it was possible to reduce some of the higher overhead operations, though the times presented here still are representative of the breakdown of overheads.

4.4.1 Application Performance

This evaluation collects a variety of programs which are already in common use today. These programs will contrast Lind's performance with applications running natively, and show that Lind's restriction mechanisms perform as expected. These applications are GNU grep [36], GNU wget [37], and a web server called nweb [38].

These well known applications all run unaltered and correctly inside Lind. Each is benchmarked, and at the same time, resource exhaustion is addressed through specific polices for each application.

Programs		
Name	command(s)	Version
GNU Grep	`grep`, `egrep`	version 2.9
GNU Wget	`wget`	version 1.13
IBM Nweb	`nweb`	August 8, 2012
GNU Netcat	`nc`, `netcat`	version 0.7.1
Tor	`tor`	version 0.2.3
Libraries		
Name	Library	Version
Lib Event	`libevent.so`	1.4.14b-stable
OpenSSL	`libcrypt.so`	1.0.1c
zlib	`zlib.so`	1.2.7

Table 4.6: Software which was compiled-for and run-in Lind for this evaluation. For a full list of software run in Lind see Appendix A.

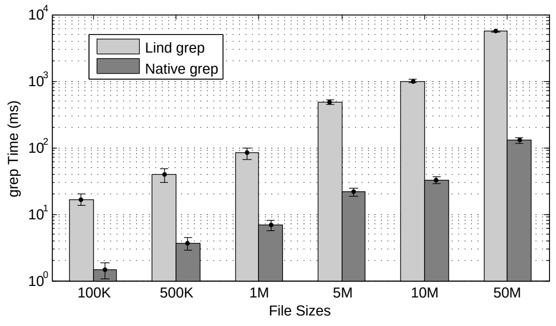


Figure 4.6: Lind and Native performance of `grep`.

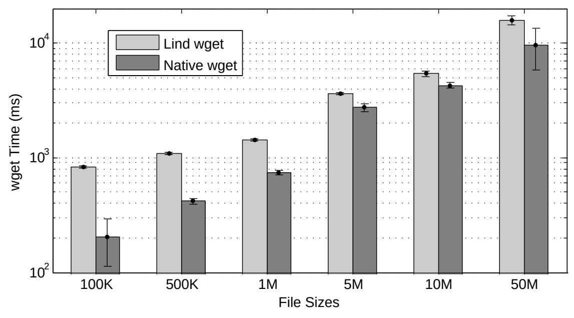


Figure 4.7: Lind and Native performance of `wget`.

Beyond the simple applications listed in Table A.1 two complex applications, Tor [39] and GNU netcat [40], were evaluated.

Finally, Table A.1 also lists some other programs that were able to be run in Lind, but did get used in this evaluation.

GNU Grep

GNU Grep is a popular Unix tool for searching for strings and regular expressions within files. Grep's performance is impacted by access to disk I/O operations. In this benchmark, Lind is first initialized with a blank file system, and then 25 books

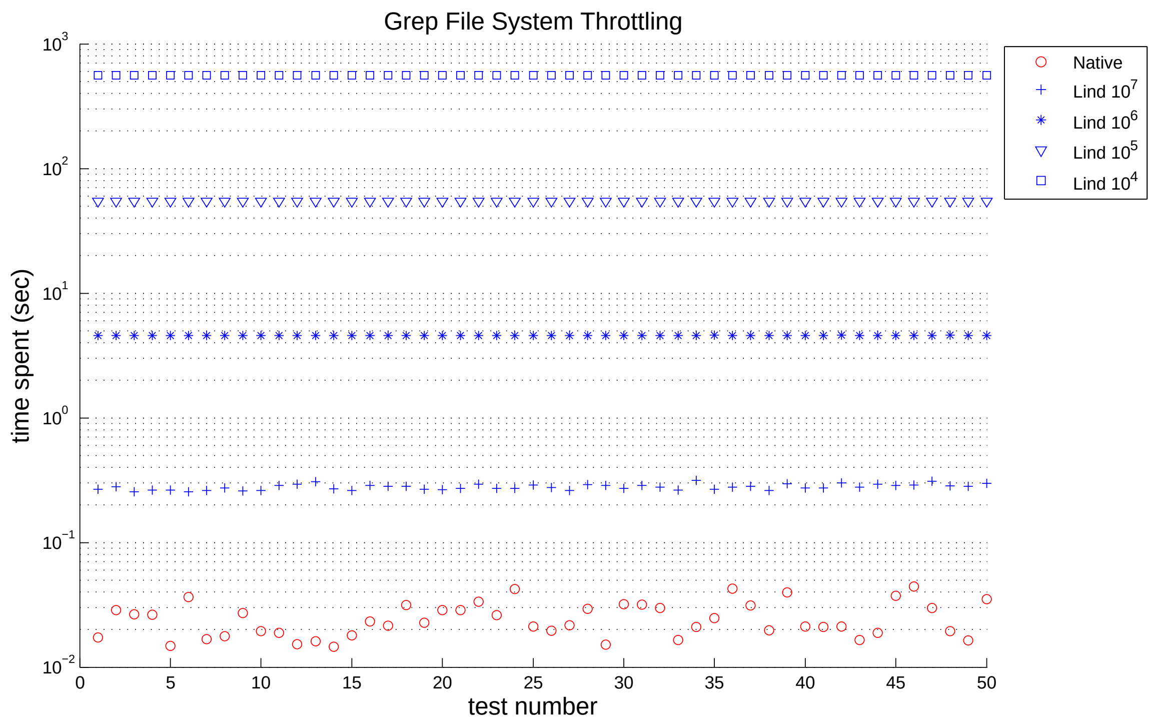


Figure 4.8: Grep execution times running natively versus in Lind. Throttling is applied with 10^7 B/s, 10^6 B/s, 10^5 B/s and 10^4 B/s of read operations. Note the log scale on the time axis.

from Project Gutenberg [41] are copied into that file system. Total sizes of the files range from 100 KB to 50 MB. The task given to grep is to search for some common words in many files, the times it took to complete this task in several configurations are illustrated in Figure 4.6. The command used to run Grep was:

```
lind ./grep 'He' /10609.txt.utf8 /10.txt.utf8 /11.txt.utf8 /1342.txt
    .utf8 /1400.txt.utf8 /1661.txt.utf8 /2591.txt.utf8 /30601.txt.
    utf8 /38840.txt.utf8 /4300.txt.utf8 /5200.txt.utf8 /76.txt.utf8
    /98.txt.utf8
```

In the test results presented in Figure 4.6 and Figure 4.8, Grep ran on the native system, as well as in Lind. Figure 4.6 identifies a 10 to 40 times throughput overhead imposed by Lind over a native application when rate limiting is not triggered. However, the execution time variance in Lind is mush lower than running `grep` natively. The main resource exhaustion concern for Grep is I/O. Restricting the number of reads `grep` is allowed per second addresses these concerns. In an experiment, running

Grep with various read-rate limitations, the reader can see how these limits impact overall execution performance. When Grep was run at a 10^7 B/s read rate, it showed a small performance slowdown compared to native execution. A read rate of 10^7 B/s is about 9.5 MB/s. Because in this experiment Grep did not need to perform more than 10^7 B of read the times show no rate limiting impact. Experiments restricted the reads by 10^6 B/s (950 KB/s), 10^5 B/s (95 KB/s), and 10^4 B/s (9500 B/s), corresponding degrees of slowdown were observed, ranging from 16x, to 2000x. This slowdown occurs because the file used in our experiment is approximately 4.3 MB, thus the number of reads per second that is needed is larger than what is provided, consequently, the impact of the restrictions is observable.

GNU Wget

GNU Wget is a common Unix tool for fetching webpages. Wget and a simple web server called Nweb are run in Lind to further illustrate performance isolation and resource restrictions. Wget uses both file system and network system calls to receive and write the requested webpage; however, performance is primarily dominated by network bandwidth.

To mimic the real world, in this experiment both the client and server side of `wget` are located on a WAN. In this case the client that runs `wget` is in Victoria Canada, and a simple web server is hosted on Emulab in Utah. This is to simulate the latency on a normal connection to a server on the Internet. Files on Emulab, varying in sizes from 100 KB to 50 MB, are requested by the `wget` client in Victoria. Figure 4.7 demonstrates the differences in performance in Lind and native `wget`. Note that because of the bandwidth bottleneck over WAN, the performance of Lind and native `wget` does not differ significantly. The throughput overhead of Lind never exceeded 10x.

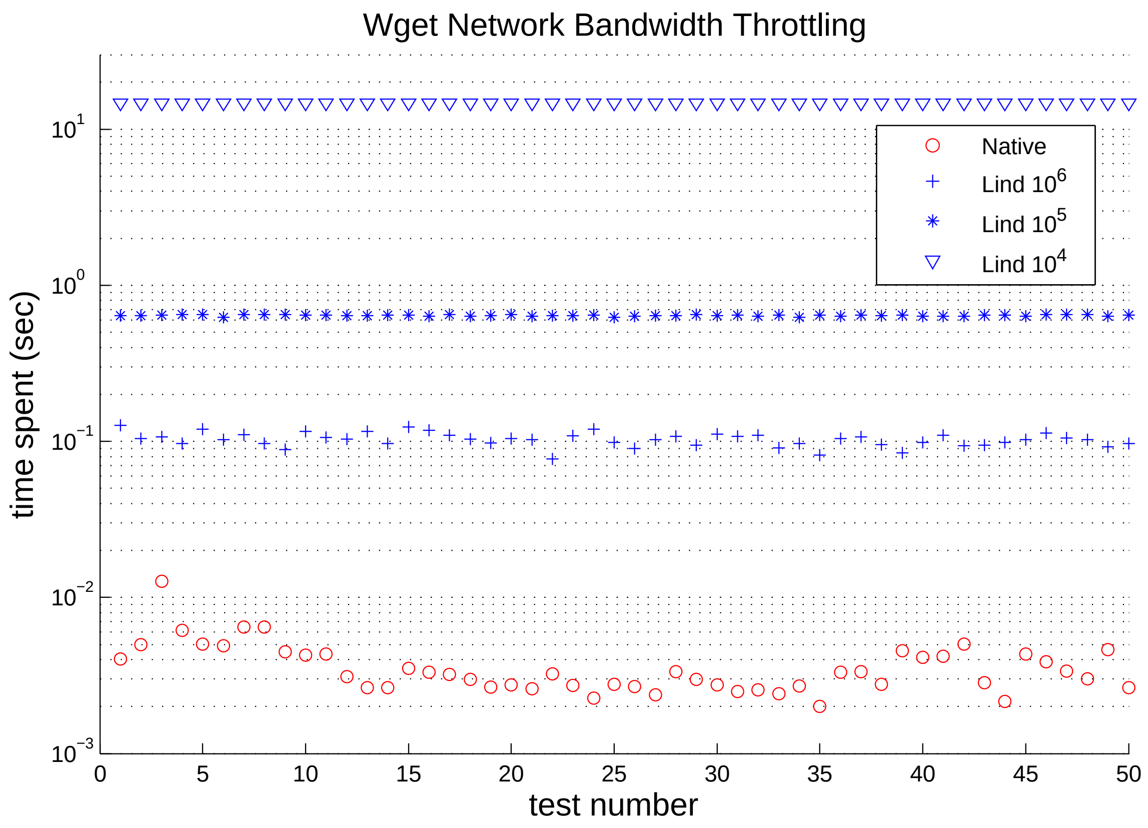


Figure 4.9: Native and Lind performance of GNU Wget on a 151 kB file hosted locally. Throttling is run with 10^6, 10^5 and 10^4 receive bytes per second.

The bandwidth restrictions in Lind shown in Figure 4.9 demonstrate how decreasing the bandwidth available to the application increases execution time. A 151 KB file is hosted locally then requested with wget. Lind normally executes with bandwidth 10^6 KB/s and introduces a 10x overhead penalty relative to native. As the available bandwidth is decreased I observe an increase in execution time, showing how the network throttling can restrict an application's network resources.

While Wget's slowdown is roughly 30 times in some cases, it was intentionally throttled by our default policy.

Simple Web Server

NWeb is a simple web server. It is able to serve static files from the local directory. NWeb was chosen because its code is simple (only 500 SLOC), and because, at the time Lind was still being debugged, thus testing with a simple program was essential. In this benchmark I ran NWeb and restricted three parameters: CPU, outgoing bandwidth, and file system read rate. These are the most common resource exhaustion scenarios. This benchmark uses a more meaningful metric, the transmission rate of the web server.

Each performance isolation criteria was tested with a simple web server running in Lind and an external client, the transfer size in each set of experiments was 11 MB. A somewhat larger than normal file size was chosen to ensure resource restrictions and performance isolation could be clearly observed. Figure 4.10 compares the unlimited web server's performance with throttled performance. Each bar in the graph manipulates one parameter at a time, and had a sample set size of 10. The leftmost bar sets a baseline for unthrottled performance. The CPU bar illustrates the transmission rate impact of restricting the application's CPU parameter to 10%. The bandwidth bar shows the impact of setting a bandwidth restriction of 100 KB/s, while the rightmost

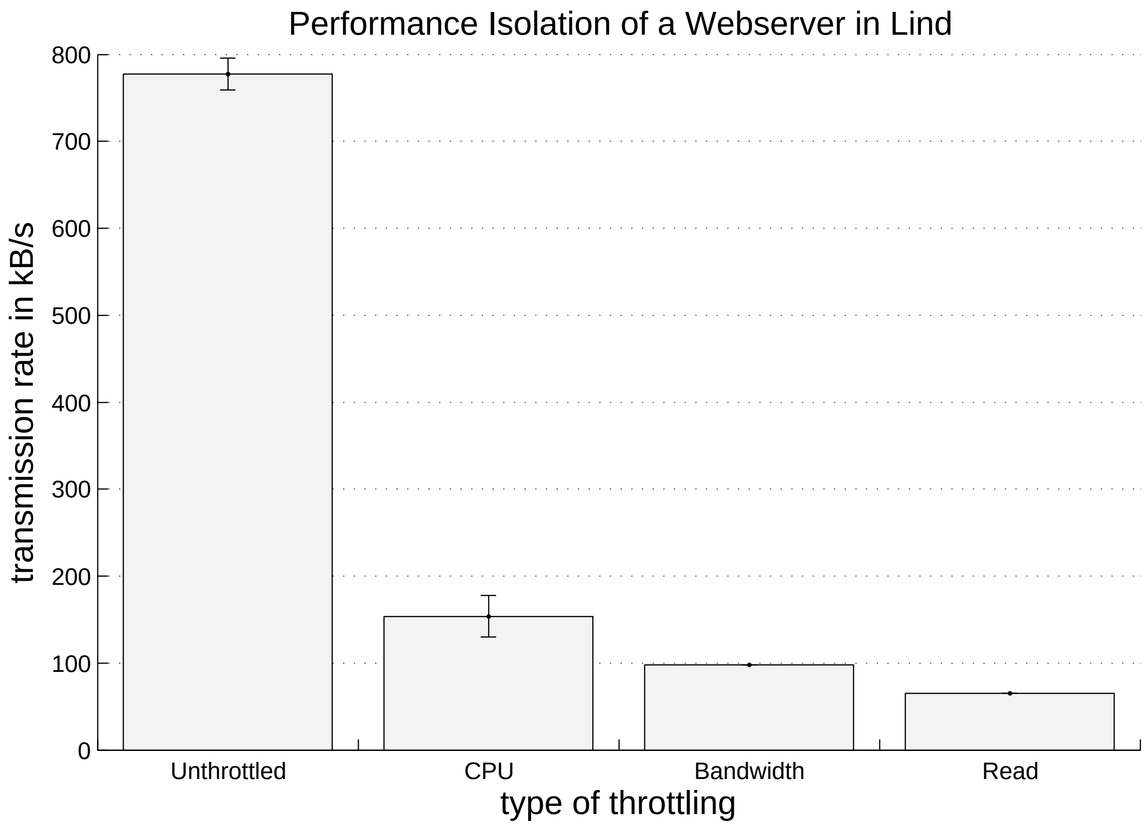


Figure 4.10: The NWeb web server running in Lind, and running with CPU, network, and file throttling enabled. Note the missing error bars on bandwidth and read, that data was lost.

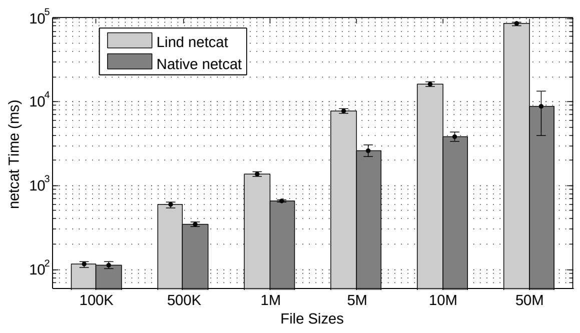


Figure 4.11: Transmission time of binary data files in netcat over the WAN.

bar reflects effective restriction for file reads with a rate of 10^4 B/s. These experiments on the simple web server indicate that applications running within Lind are effectively bounded by performance isolating criteria enforced by Repy. Bandwidth restrictions to 100 KB/s resulted in the expected amount of throttling. Likewise limiting CPU to 10%, as well as, in a separate experiment, limiting reads to 10^4 B/s both showed the expected results.

Netcat

`netcat`, often referred to as a “Swiss-army knife” for TCP/IP [40], is a program for reading-from and writing-to network connections using the TCP and UDP protocols. As a versatile network-investigating and debugging tool, `netcat` has a rich set of built-in capabilities; capabilities such as port scanning, transferring files, and port listening. Netcat could also be used as a back door. In this section, Lind's impact on the throughput of `netcat`'s binary data transfers is evaluated.

In a simple file transferring scenario, acting as a server, one networked system opens a TCP connection by listening for inbound connections on a specific TCP port. The second system connects and transfers data messages to the server by creating an outbound TCP connection to the same port. The server side of `netcat` stays running until the end of message is reached. How netcat is used in practice is a pipe is used to send data to the netcat client (that is then sent) and the server pipes data it receives into a file. This amounts to a simple file transfer—using no application level protocol—just sending raw bytes of a file through the socket.

Similar to `wget`, in this experiment, the client and server side of `netcat` run on a WAN. The client side on Emulab runs a native version of `netcat`, whereas the server side in Victoria runs `netcat` both as a Lind application and a native application, respectively. The transmission times of binary data files in various sizes, and the corresponding memory cost are measured and compared. Figure 4.11 shows the data transmission times. From Figure 4.11, Lind adds an extra overhead to netcat that is under 10x.

Tor

Tor [39] is a popular low-latency anonymous communication service, designed to preserve user privacy in network communications. Tor is designed to protect users against trafc analysis, and to permit communication when services are blocked by local Internet providers. Tor's intended users include journalists, free speech advocates, and privacy advocates. Tor uses a combination of randomized routing and hop-by-hop encryption to ensure privacy and anonymity. In a Tor circuit, each relay knows only its predecessor and successor, so no node knows both sender and destination. Further, each hop is separately encrypted. Tor is an attractive candidate as a Lind application for several reasons. First, the Tor network is resource-constrained, with constant appeals for new relays. Second, both bandwidth and computation requirements are well within the range of any computer. Third, the requirement for hop-by-hop encryption argues for use of native code wherever possible. Fourth, the value of Tor is given by the number of Tor relays, and by the number of clients for which Tor is accessible. Both of these considerations argue for the support of a large number of device types

Benchmark	Native Code	Lind	% Increase
Digest Tests:			
Set	57.31 nsec/element	67.24 nsec/element	17.33%
Get	50.75 nsec/element	50.18 nsec/element	-1.123%
Add	12.52 nsec/element	14.47 nsec/element	15.57%
IsIn	8.93 nsec/element	10.59 nsec/element	17.58%
AES Tests:			
1 Byte	16.02 nsec/B	25.17 nsec/B	57.12%
16 Byte	8.37 nsec/B	14.94 nsec/B	78.49%
1024 Byte	7.11 nsec/B	14.18 nsec/B	100.57%
4096 Byte	7.13 nsec /B	14.25 nsec/B	100.14%
8192 Byte	7.17 nsec/B	14.37 nsec/B	100.42%
Cell sized	7.36 nsec/B	13.85 nsec/B	88.18 %
Cell Processing:			
Inbound	3790.52 nsec/cell	7040.96 nsec/cell	85.75%
(per Byte)	7.45 nsec/B	13.83 nsec/B	-
Outbound	3830.62 nsec/cell	7051.98 nsec/cell	84.10%
(per Byte)	7.53 nsec/B	13.85 nsec/B	-

Table 4.7: Results from Tor's builtin benchmark program.

and operating systems, on a single executable. Lind reduces the support burden for Tor routers. Fifth, Tor's effectiveness is dependent on its security properties, volunteers will not run Tor if they think it makes their computer slower, and there are known attacks on Tor based on resource exhaustion and privilege escalation. Finally, Tor is a good candidate as a legacy system because it is implemented with approximately 102 KSLOC, with dependencies on other C libraries like OpenSSL, libevent and zlib. It would require a large effort to port Tor.

Tor runs unmodified in Lind. Traffic flowing through Tor is sporadic, so measuring live Tor throughput is not a good gauge of performance; however, Tor's source code comes with a benchmark that executes several of Tor's common operations. This benchmarks is used as a gauge of Lind's impact on the performance of Tor.

Table 4.7 summarizes the results of the benchmark. The benchmark focuses on cryptographic operations, these are CPU and memory intensive; however, the benchmark also makes system calls like `getpid`, and reads to `/dev/urandom`. Table 4.7 is summarized from the real benchmark's output. The benchmark does not have documentation, but it is short so the code was read to see what it is doing. The digest operations time the access to a map of message digests. The AES operations time

Size	Median Time
0 B	0.000246 s
500 B	0.000259 s
1000 B	0.000242 s
1500 B	0.000268 s
2000 B	0.000281 s
2500 B	0.000281 s
3000 B	0.000283 s
3500 B	0.000287 s
4000 B	0.000297 s
4500 B	0.000302 s
5000 B	0.000308 s
5500 B	0.000313 s
6000 B	0.000324 s
6500 B	0.000321 s
7000 B	0.000326 s
7500 B	0.000331 s
8000 B	0.000335 s

Table 4.8: Results from different sizes of null system calls.

encryptions of several sizes and message digest creation. Cell processing executes full packet encryption and decryption. Lind slows the operations between 0%–85%. Besides the system call being serviced, these slowdowns are caused by the differences in the code produced by NaCl, and the increased overhead from Lind system calls. As described in Section 4.3.1, when the OpenSSL library was compiled for Tor, the assembly optimized encryption routines were turned off, so the encryption will be less optimal for the processor.

4.5 Size of System Calls

The size of a system call could impact the runtime performance of Lind. This experiment quantifies the impact of the size of a system call on its execution speed. In this experiment, 500 null system calls were made, at a number of different sizes, all the way from 0 B, to 8000 B in increments of 500 B. 8000 B represents the current system call size limit in Lind.

The 500x17 samples represented in Table 4.8 collected were centered around a single point at each size; however, there were some extreme outliers in each data set. To remove the impact of outliers, median is used as the estimation of execution time.

The time does trend upwards as the size of the system call increases, though not very much, especially considering the overall system call time of a normal write call which is 100x larger—around 0.0012 s. The slow upward trend is expected; there is more data to copy each time.

One conclusion to this experiment is that it would be better to use the largest system calls possible. Lind's limit of 8 KB was only selected because of the default limit of the Unix Domain Socket and similar to what the OS uses normally. A limit of 64 KB might make large system call workloads perform much better, as we can trade many small system calls for fewer large ones.

4.6 Memory and Disk Consumption

Execution throughput is not the only metric that matters, memory and disk are finite resources, so Lind's impact on them matters. To quantify the impact Lind has on memory use, memory usage is tracked in Lind uses versus memory the same native program uses.

Memory is measured on a per-process granularity. Since Lind runs as three processes, their totals are summed. Sampling is done by a script polling the `ps` command for the memory resident set size (RSS), at an interval of one second. This gives me the memory usage over the life of the process, the maximum of those values is then taken as the peak memory consumption. The peak is expected to be representative of the overall usage. Results are collected in bytes, then converted into megabytes because this is the most common representation of memory usage. Lind is run in three processes, so the total of the three is used.

Memory usage in Linux is a complex issue, mmaped files and shared libraries are counted in the RSS total, even though they do not occupy actual physical memory

Program	Memory			Disk		
	Lind	Native	Delta	Lind	Native	Delta
netcat	28.2 MB	0.9 MB	27.3 MB	110 KB	78 KB	32 KB
primes	154 MB	122 MB	32 MB	13 KB	8.5 KB	4.5 KB
grep	31 MB	1.9 MB	29.1 MB	457 KB	509 KB	−52 KB
wget	31 MB	1.2 MB	29.8 MB	382 KB	387 KB	−5 KB
nweb	30 MB	0.5 MB	29.5 MB	22 KB	27 KB	−5 KB
tor	53.7 MB	24.6 MB	29.1 MB	5.2 **MB**	1.8 **MB**[1]	3.4 **MB**

Table 4.9: Memory consumption and disk usage of programs running in Lind and natively.

in some cases. Even the two Python VMs used in Lind are forked from each other, and should share most of their memory. The numbers presented here should treated as a worst-case scenario.

Table 4.9 shows the peak RSS memory usage for each of the programs while running the previous benchmarks.

Lind's memory usage is actually very similar to the native programs, except for a constant additional amount of memory (approximately 30 MB). That 30 MB is comprised of the additional overhead of running the NaCl `sel_ldr` process to run the program, as well as two Python processes, one to run Repy, and one that is an external monitor process. Repy's two Python processes are forked from one other, so they share some copy on write memory and libraries. These two processes sharing of memory is not counted in these numbers. I will describe in the Future Work section that it would be possible for many Lind programs to share the same Python process, though this functionality is not implemented in our current version of Lind.

Disk space usage varies only slightly over native applications. The only outlier is Tor, which is much bigger. In the case of Tor, I compiled the Lind version with a static version of OpenSSL, zlib and libevent. When those libraries are added to the native build it becomes 5.3 MB, almost exactly the same size as the Lind version. These results were expected. The only difference between a Lind executable and a

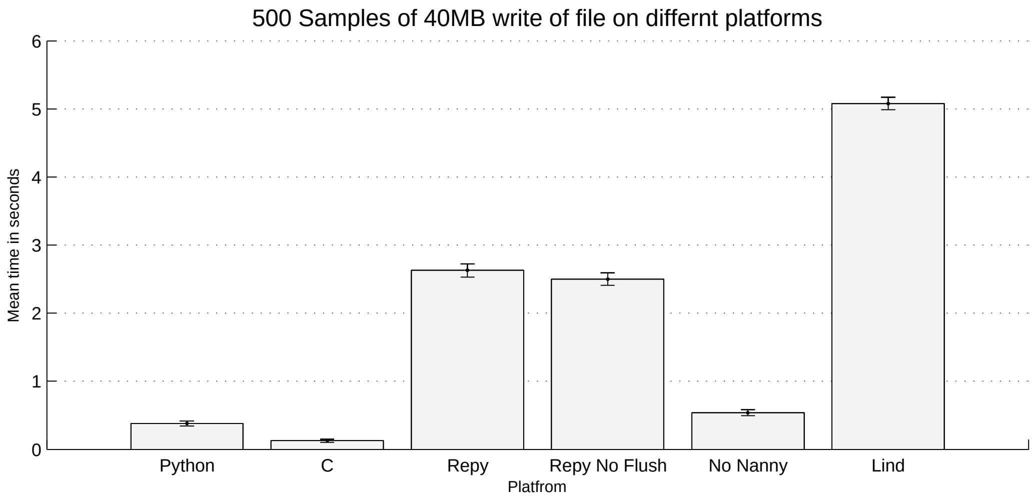


Figure 4.12: Time to write 40 MB to disk on different platforms.

native executable that would effect the size is that the Lind executable must use NaCl's formatting rules, which means instruction blocks must be aligned to 32 byte boundaries. In cases where exact alignement is not possible, NOP padding is used [12]. This should not manifest as a drastic change in code size. Lind's results are smaller in some cases, this is probably because the library code Lind uses is smaller than a native program. The changes are so minor, a detailed study is not warranted.

4.7 Quantifying I/O Slowdown

The observations presented above show that I/O based applications are at least 10 times slower in the Lind prototype. This section further quantifies where that slow down comes from. The benchmark used in this section writes a 40 MB file to the file system. To help isolate where the 10x slow down comes from, this benchmark was run on several different platforms, C, Python, Repy, Lind, and some modified versions of Repy.

Figure 4.12 shows a barplot of the 6 configurations run 500 times each, the mean and standard devotion is shown. Histograms showed that the larger results were normal, the faster results (C and Python) had more noise in them.

The first notable observation from Figure 4.12 is that Repy is much slower than Python, in fact in these runs Repy's I/O performance accounts for slightly more than half of Lind's slowdown and in this case is about 7x slower than Python. To help break down this overhead caused by Repy, Repy was modified in two ways: the Repy writeat system call was changed to not seek or flush, the calls to the Repy nanny were also removed. The nanny is the rate limiting system, and even though it was set to not rate limit at all, you can see it has a significant impact on the performance of writes, removing seek and flush did not have as much of an impact.

These results highlight an important lesson learned, Repy's I/O performance is the dominant factor in the Lind overheads, and unfortunately, Repy was not built with performance in mind. Optimistically though this gives Repy a lot of room to improve its I/O performance. For instance the nanny calls could be made asynchronously, so that they are not blocking the I/O path.

4.8 Quantifying Isolation

Throughout this chapter, there is an assumption that rate and functionality limiting a program helps enhance temporal isolation with other programs. This section presents simple benchmarks which shows that limiting can help a process behave more regularly.

The first simple experiment is to simply check that a program which writes to disk is not able to excede the policy limit. A simple logging program was used, and told to write a large log. The Repy policy was changed to allow 1 MB of disk usage, then the program was run.

```
lind@hypervisor:~/test$ lind ./big_file.nexe
Disk use '3467323' over limit '1000000.0'. Impolitely killing child!
Terminated
```

The console trace below shows Repy "impolitely" stoping the offending program by killing the NaCl and Repy processes. This experiment shows that Lind can effectivly cap the amount of disk space a program is able to use. The example policy file in Section 2.5 shows many other resources which can be limited or capped.

Another example, setting a memory cap:

```
lind@hypervisor:~/test$ lind ./big_file.nexe
Memory use '7122944' over limit '2500000.0'. Impolitely killing child!
Terminated
```

One thing to note is that Repy monitors usages by polling, so there may be a short span (less than a second) where the program can excede the limit before it is killed.

Opening files in the local file system also fails.

```
 int fd = open("/test", O_CREAT|O_WRONLY, S_IRWXU|S_IRWXO|S_IRWXG);
```

The file `/test` is created within the Lind file system, not the native file system.

4.9 Mitigating Privilege Escalation

Privilege escalation is categorized by horizontal privilege escalation and vertical privilege escalation. Horizontal escalation occurs when the program gains access to resources that another equally privileged part of the system has access too. For example, an authenticated user gaining access to a different authenticated user's restricted files, or a program being able to crash a different program are both cases of horizontal escalation. Vertical escalation is when an program is able to gain higher than permitted rights in the system. An example of vertical escalation is when a non-root program is able to operate as root, or when a program is able to run code in the OS kernel. This section will deal with how Lind reduces the risk of both horizontal-escalation and vertical-escalation attacks.

Horizontal escalations are prevented by Lind because of Lind's share nothing model: Nothing is shared between Lind programs. Specifically, the Lind file system is not shared between programs at runtime, so the file system is not a vector of attack, socket connections can be blocked to localhost so communication over sockets cannot be made to other local processes. Lind programs cannot send signals to other programs. Lind programs cannot create or open Unix domain sockets, or any other type of special files. Without any shared medium there is nothing to propagate the programs influence into another part of the system. The only way a program is able to access another program on the system in any way is by circumventing the NaCl or Repy sandboxes, irrespective of Lind, or accessing a side channel which has not anticipated.

One vector for horizontal escalation in Lind that still exists is if a program is able to put something in a Lind file system, then another program is run using that file system. For example, when the benchmark section was setup, one program was used to copy files into the file system, then the actual benchmark programs were run. That copy program can do whatever it likes to the file system, so if it is able to change it in a way to attack the benchmark program that is a possible attack. This implies that Lind programs must consider a file system as input when they are started, and check that input as they should any other untrusted input.

Vertical privilege escalation is mitigated by the sandboxes Lind makes use of. As discussed in Section 2.2 and 5.1.1, both NaCl and Repy present an interface to the system that is intended to be very hard to exploit, and use strong mechanism to stop access to the system in any way besides using the safe interface. As long as the NaCl and Repy sandboxes function correctly, the only way Lind could allow a vertical privilege escalation is by introducing a new attack vector itself. As discussed in Section 3.4, Lind makes only minor changes to those interfaces, specifically, one new system call in Repy and the implementation of that system call. To help ensure that this minor change could not be a source of vulnerability, the code was reviewed by two security researchers before it was committed—a common practice to assure code is secure. Still, the code should only be counted as a proof of concept, not a real battle tested secure system.

4.10 Summary and Discussion

On some workloads Lind is fast. In the Primes benchmark, Lind was able to perform at 80% of native execution speed, because it is a CPU bound computation. In other CPU bound benchmarks like Tor, Lind's proof-of-concept implementation ran at worst at 15% of native speed, though most benchmarks ran closer to 80% of native speed. On I/O bound benchmarks Lind's proof-of-concept implementation was much slower. In the non-rate limited case Grep ran 11x slower than native. The wget example took 200 ms. Though the Lind I/O overhead is large, it is still within the standard operating time scale of the internet. A 200ms page load is not uncommon on the Internet.

Lind's memory use is quite a bit larger than native programs. The increase in memory is caused by several things. Lind uses a three process model: the NaCl process, the Repy Python process, and the Repy Python monitor process. The Repy processes have full Python interpreters running in them. The NaCl process has code setting up the SFI, and then processing all the different types of system calls. This three process model does add a lot of memory overhead for small applications, but should be less noticeable for large applications, as this part of the memory overhead is fixed at about 30 MB.

In I/O heavy programs Lind's performance penalty is due to several factors. First, in some cases, Repy is rate limiting all requests going out of the sandbox. This means disk access, and network I/O are all throttled. Second, NaCl imposes a small runtime overhead, which can be optimized. For the purposes of the Lind prototype, these optimization were not pursued, because Lind was a proof of concept used to demonstrate that Lind did not introduce any additional overheads for CPU and memory intensive applications.

Finally the system call round trip time has been made much longer in Lind. System calls now have to go through my modified glibc, be marshalled, then be passed down a Unix Domain Socket, be unmarshalled, dispatched, processed by Repy, then a reply must go through the reverse path. That is why I/O bound applications are slower in Lind. Unix Domain Sockets were not the best choice for interprocess communication; however, they are supplied by NaCl, so no addition to the TCB was needed to use them. This choice was a tradeoff between security and ease of implementation and performance. As I describe in future work, I have done an initial exploration of an implementation that reduces the Unix Domain Socket overhead by moving Repy and NaCl into the same process.

The simple timing experiment in Section 4.4 shows that a sizeable portion of system call time goes into executing the system call in Repy. During Lind's proof-of-concept implementation, no time was spent profiling the Library OS server. The 11x slowdown in I/O operations could be sizeably reduced by optimizing Repy and the Library OS server. For instance, the *stat* system call actually reads the entire file to find out how big it is. This naive way of finding file size was chosen because

Repy does not have a size call itself. A better way to get file size would be to store each file's size in the file system metadata, and update it on writes. This alone would likely cut the 11x overhead in half. There are many locations in Repy and the Library OS where a dedicated programmer could trim runtime overhead, but as a proof-of-concept implementation those tasks were omitted.

In trade for a slowdown it was shown that Lind helps prevent resource exhaustion bugs, as well as privilege escalation bugs. Overall, a 10 times overhead is a steep price to pay for isolated legacy computation; however, the paradigm still holds promise for some applications, specifically, those that perform relatively few system calls, and those in which isolation is paramount—many of these initial costs in the prototype can be mitigated through engineering efforts, and given the relative immaturity of the Lind implementation there is lots of room for performance gains.

To investigate how easy it is to port an application to Lind, I described the steps necessary to port Tor to Lind. The two weeks to port Tor was primarily spent on adding and fixing system calls and setting up the build systems of Tor and the libraries it uses. These times will be reduced as more libraries are precompiled for Lind, and bugs are shaken out of Lind's implementation.

Chapter 5

Related Work

Program isolation began when computer systems switched from single job OSes to multiprogrammed time sharing OSes. As soon as more than one program was loaded into the system the opportunity to have them interoperate and interfere became possible. Resources had to be partitioned and shared, and the modern OS—as a resource and information control mechanism—was born. Dijkstra's "THE"-multiprogramming system [42] was a single-user system and introduced software enforced memory segmentation where the programmer did not have to directly write code based on physical memory drum locations. Enforcement was done through the ALGOL compiler, but was done in the name of multiprogramming, not security. The first OS to embrace security as a fundamental design property was Multics [43]. Multics used a hardware enforced memory segmentation in a scheme with a permissions table similar to modern X86 virtual memory page tables. The modern notion of virtual memory is a logically isolated address space each process runs in. These systems all took the idea of sharing resources, then provided means of isolating the sharers.

Other abstractions of program execution also exist. Just as OSes are primarily about sharing and multiplexing resources, Virtual Machines [44] are an abstraction

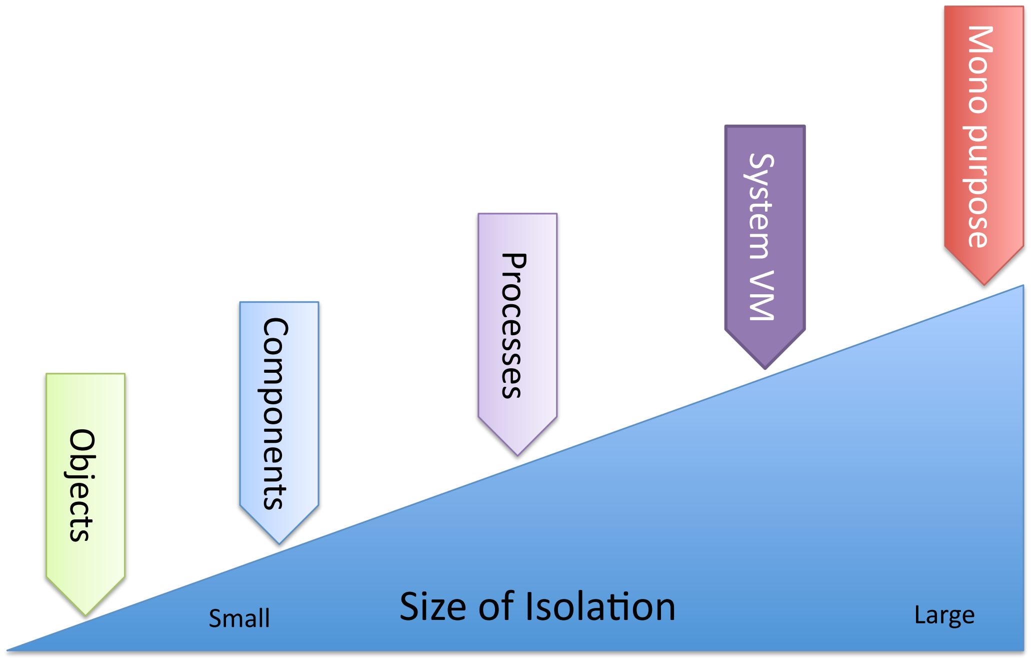


Figure 5.1: Isolation comes in many forms, with many sizes of granularity. Lind, tries to provide isolation stronger than process isolation, but not by using a machine abstraction like VMs do.

that partition one physical machine into many logical machines, effectively creating many smaller isolated machines—isolating OSes on the same machine from each other. This is the opposite approach to isolation, in that it isolates then shares instead of how OSes share then isolate. The modern mechanics to support the VM abstraction span all the way from the hardware to support of modern processors [45], to the binary rewriting [46] used by VMware, to blurring the hardware interface as in the Xen Hypervisor [47]. If the multiplexing of hardware does not have to happen at the lowest levels of the system, the OS can provide logically independent machine interfaces. Good examples of this technique are the Linux kernel's KVM project [48], Linux containers [49] and Vservers [50]. Since virtual machines are logically isolated machines, programs running inside of them are isolated from each other.

Now that we have seen the design, implementation and evaluation of Lind, we can revisit Chapter 1's granularity spectrum. Figure 5.1 shows that spectrum again. Lind provides stronger isolation than processes, but with different runtime characteristics than a VM. This places Lind in an interesting place in this spectrum. There have been and are many other technologies for isolating programs, the rest of this chapter covers those in depth. The two ideas of processes and VMs are similar to Lind's isolation mechanisms, so they are covered in depth. The rest of this chapter is structured as follows. Section 5.1 will discuss the general area of systems security and best practices to inform system design. Section 5.1.1 will discuss modern isolation mechanisms in the context of these best practices and in the context of Lind's design.

5.1 Building Secure Systems

Isolation and security go hand-in-hand. A well isolated system is more secure. The rest of this section discusses some relevant security best practices, and how they impact isolation.

The Open Web Application Security Project (OWASP) defines a model for thinking about security problems [51]. In their model, a *threat agent* makes use of an *attack vector* to exploit a *security weakness* to circumvent *security controls* to cause a *technical impact* which eventually causes a *business impact*. Systems work mainly concerns itself with the attack vector through to the technical impact. Attack vectors are the technique by which the attacker compromises the system, and security weaknesses are the problems which the attack makes use of. Security controls are the mechanisms that are setup to control how parts of the system are allowed to interact. Finally, technical impacts are actions like executing some code or gaining access to an asset—some information or execution the attacker would otherwise not have access to.

There are many possible security weaknesses. Some common weaknesses are listed in Table 5.1 [1, 2]. These all result in varying degrees of compromise, and can be compounded. The worst case scenario is a remote code execution in which the threat agent can run arbitrary code on the machine.

There is much research on fixing these security weaknesses, but Lind works further down the chain, at the security control level. So if a program has had one of these weaknesses exploited, now what can it do? Hopefully, in a sandboxed environment, it can do nothing more than it already could. This is a fundamental principle of sandboxing, resource access policies are decided outside the application, so when the application is misbehaving it cannot do more than it should be able to. For example, if an application reads files only in directory A, there is no time ever when it should be accessing directory B. Instead of allowing the application to decide what it should access, this rule can be enforced by the sandbox. Another way to think of it is: sandboxes are an effective mechanism to enforce the POLA onto an application even when the application is not trusted.

To combat these attack vectors, and as a means to block technical impacts, many security best practices are recommended. Some that are relevant to the design of a sandboxing system [3] are listed in Table 5.2.

Common Name	**Description**
Buffer overruns	Miscalculate buffer size so program writes data off the end of a buffer to overwrite program code or data.
Uncontrolled format string	Format strings used by `printf` and similar functions subverted to leak private data.
Integer overflows	Integer wrap around violates programer's expectations.
SQL injection	Unescaped user input used in SQL statement that allows attacker to change the meaning of the statement.
Command injection	Unescaped user input is placed in a shell command which can then change the meaning of the command.
Failing to handle errors	An unexpected error puts the system into a state the programmer did not expect.
Unprotected network traffic	Sending data or program state without encryption.
Weak passwords	Passwords which can be easily guessed.
Storing data without protection	Writing private data or program state to disk without adequate protection.
Information leakage	Provide an interface to the system which provides more data than expected.
Improper file access (TOUTTOC[1])	Trigger program races and errors by changing the file system in unexpected ways.
Trusting network name resolution	DNS servers the program uses are subverted so network connections are to a different party than expected.
Race conditions	Problems caused by accessing the system in parallel.
Strong random numbers	Encrypting without a source of real randomness.
Poor usability	User interface design which leads the user to perform an unintended action.
Improper pathname	Uncontrolled filename from outside allows more access to the system than expected.
Download without integrity check	Program downloads code or updates, however download is subverted and they run something else.
Untrusted functionality	Using code from untrusted sources.
Use of dangerous function	Using something which can do a lot more than just the intended task.
Unnecessary privileges	Same as the POLA mentioned before.
Incorrect permissions	Permissions not correct for some use case, or too open.

Table 5.1: Common security weaknesses from OWASP [1] and SANS [2].

Practice	**Description**
Secure the weakest link.	If control of the system can be gained from anywhere, attackers will attack the most vulnerable part of the system.
Practice defence in depth.	Have multipul layers of enforcement, so if one level of security control fails, others can still catch the problem.
Fail securely.	Try to envision all failures, even those that are unlikely, then deal with those situations as if they were common. This includes ensuing that error recovery is secure, for instance making sure that no private information is leaked in error messages or written into logs.
Follow the principle of least privilege.	Give each part of the system the least possible privilege, so if it is subverted it cannot do more than what was intended.
Compartmentalize.	Break the system up into well isolated parts, so if one is subverted the others can still function.
Keep it simple.	Complex system are hard to understand, and can be hard to detect if they are acting inappropriately.
Promote privacy.	Privacy is easier to manage if it is a first class concern in the system.
Remember that hiding secrets is hard.	Try not to build a system that can be compromised if the attacker gains access to a "secret", specifically something about the implementation or single key or password.
Be reluctant to trust.	If possible, operate in a way where you trust as few things as possible. The fewer trusted parts of the system, the less
Use community resources.	There are many online forums where you can get great up-to-date information such as CERT and SANS.

Table 5.2: Some security best practices which apply to building sandboxes [3].

Sandboxes provide restricted execution environments—a key design element behind Lind. The following overview surveys a variety of mechanisms for controlling access to system resources and ensuring controlled application behaviour.

5.1.1 Sandboxing in the Wild

Probably the most popular approach to sandboxing is language-based sandboxes. A language based sandbox requires that the untrusted program be written in a particular programming language, that, because of the semantics of the language can be isolated by the language's runtime. Examples of these sandboxes are Java's JVM [52], and Adobe's ActionScript's Flash sandbox [53, 54].

Programming language VMs, such as Java, Silverlight, JavaScript and Flash are commonly used sandboxes that have achieved widespread adoption. These sandboxes combine untrusted application code with an interpreter and standard libraries. Standard libraries consolidate routines to perform I/O, network communication, and other system sensitive functionality. Though many sandboxes implement the bulk of their standard libraries in a memory-safe language like Java or C#, flaws in memory-safe code can still pose a threat [55,56]. Furthermore, memory-safe languages must include their compilers and runtimes in their TCBs.

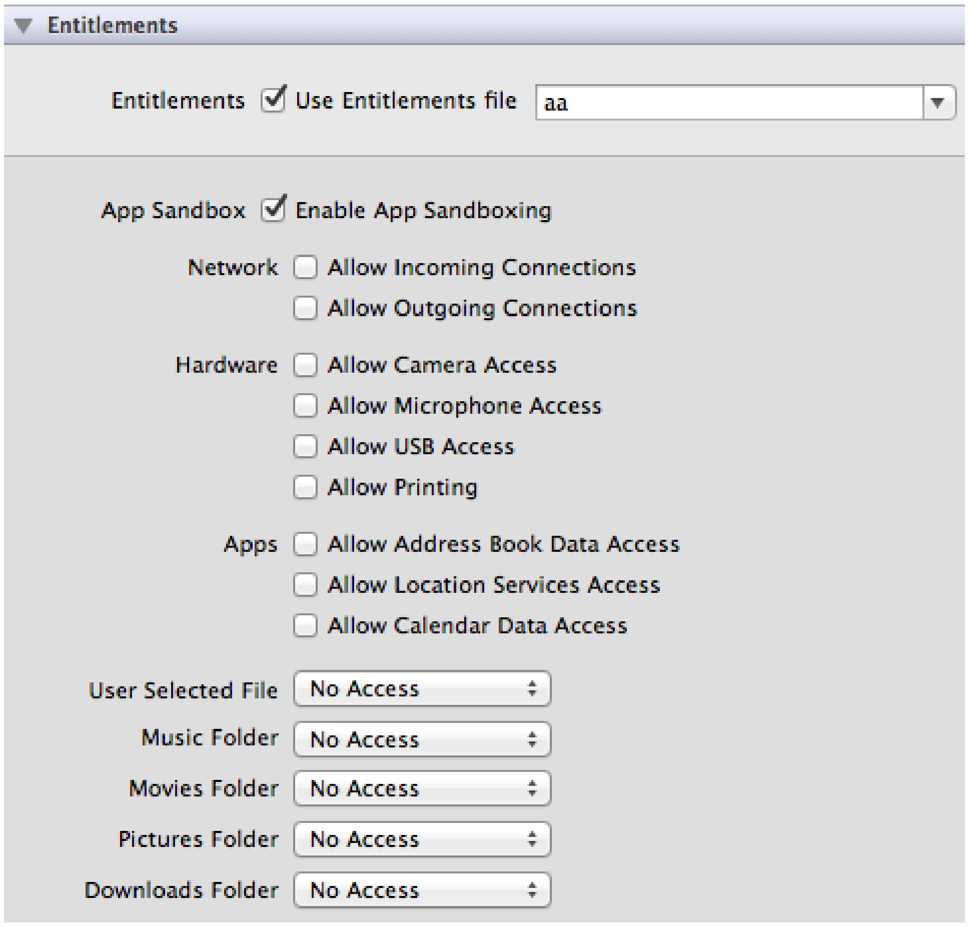
Building a safe language-based sandbox which has no impact on the system it runs in is Repy's goal. As mentioned in Section 2.2, Repy is a subset of the Python language. Repy uses a library called `safe.py` written by Phil Hassey to run a Repy program in a custom subset of Python. The subset chosen intentionally blocks all of the system libraries, then disallows Python `import` statements, which could then reenable them. Specifically, Repy is Python without: the `import` statement, global

[1]TOUTTOC stands for Time Of Use To Time Of Check.

variables, `stdin` input, and a collection of keywords and attributes which may be used to escape the sandbox—for example the `exec` dynamic code loading call [57].

Because Repy hides the Python system API, it provides its own calls to access the system. This API is detailed in Section 3.1.1, and was designed to be minimal, simple to use, and easy to formally verify. On top of this API, some of Python's standard libraries are re-implemented. However, they are not inside the TCB of Repy, but rather in the untrusted space. Repy provides security policies that are factored out of the Repy kernel, so there is a simple dynamic means of controlling policy. The Repy API is provided by a set of *security layers*, each of which has a capability to access one type of resource in the system [13]. Each security layer is also isolated from the untrusted code and the system, so if a security layer is compromised then at worst it can only allow the actions it is responsible for to occur. Much of the logic of system access in Repy is factored into these safe security layers, meaning a lot of Repy is outside of its TCB.

OS-based mechanisms are also a popular means to isolate applications. These sandboxes use OS specific and OS/sandbox-specific APIs to setup customized application-specific environments for processes and subprocesses to run. For example, all applications sold in the Mac App Store must use Apple's OSX sandbox [58]. This setup requires a signed permissions file, (that Apple calls *entitlements*), describing which parts of the system the program can access—files, networking, webcams, and other devices. When executing, the application is restricted to these resources by the OS. By default, a program can access only its own private folder. Figure 5.2 shows the configuration of the sandbox, in which the developer can select the entitlements they want for their program. It is important to note that this is an opt-in system because the developer could just select all the entitlements; however, the POLA tells us that it is best to select as few entitlements as possible.



Entitlements

Entitlements ✓ Use Entitlements file aa

App Sandbox ✓ Enable App Sandboxing

Network ☐ Allow Incoming Connections
☐ Allow Outgoing Connections

Hardware ☐ Allow Camera Access
☐ Allow Microphone Access
☐ Allow USB Access
☐ Allow Printing

Apps ☐ Allow Address Book Data Access
☐ Allow Location Services Access
☐ Allow Calendar Data Access

User Selected File: No Access
Music Folder: No Access
Movies Folder: No Access
Pictures Folder: No Access
Downloads Folder: No Access

Figure 5.2: In Apple's XCode application, selecting the application sandbox's *entitlements*. These are the operations and data that the application will be able to access when it is run.

Since Windows 2000, Microsoft OSes have shipped with a sandboxing mechanism [59]. It is not widely used as a general purpose sandbox, but some developers use it to build more secure applications that are specifically targeted to the sandbox. The sandbox works by setting up tokens that limit a child processes access to the file system, rate limit the child's access to resources (for instance the clipboard), and stop the child from performing operations like launching other programs and doing some graphical operations or shutting down the system. The sandbox interface does not span the entire Windows API though, so it is not considered as much a sandbox as a deterrent.

One notable use of the Windows sandboxing technique is in the Adobe sandbox. Adobe's strategy for sandboxing Flash and Acrobat programs is to contain zero day exploits [53, 54]. On Windows, Adobe's Flash and Acrobat sandboxes use several of Windows safety features including data execution prevention, modified system call tables, and file system permissions. A program launched in this sandbox must first re-enable these system calls, and create another child with more permissions to escape the sandbox. This is no small task, as the system calls to do so are blocked.

Operating system level mechanisms can also be used to achieve a lot of the goals of Lind. These approaches have the most success functionality wise, however, they also are the most invasive and error prone methods because they require (often complex) changes to the OS's kernel or worse building a whole new OS, and they tend to be more of a one-size-fits-all solution.

One simple sandboxing mechanism that is used commonly on Unix based machines is a *chroot jail* [60]. Chroot jails change an applications perceived root directory to somewhere else in the file system, so the application can no longer travel any higher up in the directory tree structure than the jail's root. It is a common best practice to run daemons that do not need access to most of the system in their own dedicated chroot jail. For example, by default the Network Time Protocol (NTP) daemon is run in a chroot jail.

However, there is a ease of use problem with chroot jails—files that do need to be accessed need to be copied or linked into the jail. Since libraries, most importantly `glibc`, are loaded into programs on Linux, it may actually be hard to figure out which files a program will need to access.

As an example of this complexity, the DNS subsystem in glibc will access many files in `/etc/` on its first use. The logging files stored in `/var/log/` are another good example of files that are needed. All these dependancies must be sorted out manually, or worse the programmer will just give blanket access to `/etc/` for the jail—a violation of the POLA since not all the files in `/etc/` were needed. Some of these issues can be side-stepped by invoking the jail after the program is running, since open files are not effected by the invocation of the jail. One known problem with chroot jails is that once in the jail another chroot system call can be used to change the root again, possibly back to the original root. Further, file systems can still be accessed if there is a path to the device file from which the file system is created. Chroot jails cannot do any special enforcement of read-only attributes. So chroot is more of a deterrent than an absolute file system restriction mechanism.

Some other Unix facilites are commonly used in sandboxing. The `rlimit` system call sets a per-process memory limit on the heap or address space of a process. When processes allocate more memory than their `rlimit`, they are terminated by the OS. The `nice` system call can change a processes priority, so that it receives less CPU time. Processes can also be started and stopped with signals, the technique Repy uses to slow applications which use too much CPU time.

In use today there are several virtualization systems which provide sandbox like isolation. Operating system-level virtualization platforms modify the OS to provide logically isolated runtimes for processes. OS-level virtualized programs always share the same kernel, but have restricted views of the system and access to resources is controlled. OS-level virtualization is also referred to as *container-based virtualization*, presumably the name stems from Solaris. Containerized OS's isolate processes at the system call level, imposing a customized environment and restrictions through system calls. Solaris Containers [61] allow a super-user to break process up into separate *zones* each of which has separate: networking, storage, and namespace. Processes within a zone can see and modify only other processes in the same zone. Memory, CPU and physical hardware like the network interface can be shared amongst all the zones or assigned to a single zone, however these resources are presented in an abstract form. To minimize storage use, file systems can be shared through a copy-on-write mechanism so all the zones do not have to keep a separate copy of the root file system. Similar to zones, FreeBSD *jails* [62] extend the chroot jail notion to apply to the whole system instead of just the file system. In FreeBSD jailed processes are bound to specific IP addresses, and have no access to routing- and raw-sockets, and no access to processes running outside the same jail. Jails also have their own copy of the file system, that is not shared.

For Linux, several container-based virtualization platforms are popular, most notably: OpenVZ [63], LXC [49], Virtuozzo [64] and Linux-Vservers [50]. These all operate similarly to zones and jails, each providing limited access to the system for a collection of processes. Containers have appeal because of their low system overhead, and are conjectured to be the best runtime for large collections of diverse programs [65].

One detractor from container-based isolation is that it is built into the OS, this causes problems in two ways: it makes it harder to maintain the container system [66] and because it is an expansion of the OS interface it could lead to new security and stability problems. Furthermore, the isolation of container-based systems is weaker than more heavy weight virtualization platforms. For instance, processes running in LXC still share many common buffers, and process can even share common memory pages (for instance `glibc`) [67]. These optimizations are part of what makes containers so light-weight, by sacrificing some isolation. Unlike more heavy-weight isolation mechanisms, container-based virtualization normally has the capacity to run thousands of sandboxes on one machine, which is important for isolating modern workloads.

Though they are primarily concerned with the prevention of malware and not security in general, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) are techniques many OSes employ to enhance the security of native processes [68]. DEP uses the paging mechanism to mark pages in the stack and heap memory segments as not executable, and pages in the code segment as not writable. The effect of those two changes is that new data introduced to the program after it starts cannot contain instructions that can be executed. ASLR furthers this protection by randomizing the layout of the code in the code segment so that “stack smashing” techniques cannot stitch together code at known offsets to perform malicious work.

AppArmour [69] and SELinux [70] are Linux kernel based sandboxes. Program policies work by restricting the parts of the file system a program can access and the system calls it can make. Profiles exist for many of the most popularly exploited applications, such as web browsers and server programs. AppArmour tries to make it very easy to write policy files, and has a *learning* mode in which policy violations are logged so the policy can be revised. This sort of learning mode would make it easier to make policies in Lind too. These system impose a discretionary access control model on resources and the file system. This allows principles like users and programs to be explicitly identified, making a rule like "the firefox program can write only to the `/tmp/` directory" possible. Rules can be composed, so base rules can be made for classes of applications with the same needs, for example "gnome applications need read access to `/usr/local/gnome/`, firefox is a gnome application".

Function	**OS**	**Resource(s) protected**	**How**
Restricted Tokens	Windows	system calls, rate limit, file system	Restricts system calls and OS enforcement.
chroot	Unixes	file system, file system accessible OS controls (/proc)	Changes a processes perceived root folder.
SIGSTOP & Nice	Unixes	CPU	Suspends a process or decreases the number of cycles it is scheduled for.
rlimit	Linux, Solaris	Memory	OS kills process if it goes over set limit.
ulimit	Linux, Solaris	Open files, number of children, number/size of pipes, stack size, CPU time, number of locks	Kill processes if user goes over set limits
DEP, ASLR	Windows, Linux	Program instructions	Randomizes locations of things mapped into memory so that no absolute offset will ever work in an attack.
Containers	Solaris	File system, process control, network	OS mechanism to force processes to work in same group only.
Jails	FreeBSD	File system, network addresses	OS level mechanism.
LXC	Linux	File system, processes control, networking	Linux kernel extension.
Linux-vservers	Linux	File system, process control, some networking	Linux kernel patch.
AppArmour	Linux	File system	Linux kernel extension.
SELinux	Linux	File system	Linux kernel extension.
Native Client	Windows Linux Mac	Everything except CPU	Block system calls to everything, then pipe resources back from the browser.

Table 5.3: Sandboxing related OS functionality.

5.2 Systems Providing Sandboxing

There are several sandboxing systems with similar intent to Lind, that I will discuss in more detail now. Xax [28] has goals very similar to Lind and NaCl, as it aims to provide a light-weight, secure sandbox for running legacy applications in web browsers.

Xax introduces the idea of a *picoprocess*, a replacement for a hardware VM with an application level API instead of a hardware API. The Xax hypervisor runs in a web browser, and downloads untrusted programs, it then loads those programs in a specially constructed protection domain. This protection domain is constructed using the OS's mechanisms for controlling system calls. In the case of Linux, this is using the `ptrace` system call to perform system call interposition. Inside the picoprocess is a *portability layer* that converts requests for resources into a platform specific format then makes a *xaxcall*, (similar to a hypercall in a hypervisor) or system call, to get the request resources. These request are then satisfied by the web browser.

Xax is a very interesting system in the context of Lind. It is similar in many ways. Lind uses SFI to block system calls, whereas Xax uses OS mechanisms. Xax programs can run at full native speed, Lind programs have the runtime overhead of SFI; however, as is pointed out in their paper it is possible for a untrusted program in the sandbox to escape and run at full privilege if the parent crashes, this is not possible with SFI. Furthermore, SFI can be validated at load time, to ensure an unsafe program is never executed at all.

Both Lind and Xax transfer system calls to another process to be serviced. However, the Xax system call servicing code is part of the TCB, whereas it is not trusted in Lind. Xax claims a small change to the TCB of about 5000 SLoC, and that its TCB is small relative to an OS. It is hard to gauge what this means because the Xax paper does not mention if the implementation is in a type safe language. Xax's TCB is not small because it must include the browser as well.

Xax takes a more cavalier approach to running legacy applications. Lind tries to run legacy application as-is, with no changes to the source code or application semantics. Xax requires modifications to the legacy code, but more worryingly, in some cases when provided services would be hard to implement in the Xax model, they modify the semantics of the system. Examples of that are using a RAM file system instead of a persistant file system, and applying the browser's single origin policy to all socket connections of the program. Modifying the system's semantics—that will

System	**CPU Memory**	**System call**	**Provider**	**File system**	**Network**
Xax	No	Yes	Browser	Ramdisk	Javascript's
NaCl	No	Yes	Browser	Limited	Javascript's
Lind	Yes	Yes	Repy	Full	Full
Windows	Yes	Partial	OS	Restricted	Restricted
Ostia	Some	Yes	Process	Full	Full

Table 5.4: Comparison of some similar systems to Lind, and what they isolate.

surely violate some developer's expectations—then applying Xax to 3.3 million lines of pre-existing code without any discussion of the impacts to the functionality of the code is problematic.

In the context of Xax's goals—to run legacy applications from the web, in a web browser—those trade-offs make sense, but as a general purpose platform for running legacy applications Xax is lacking.

Xax showed best case performance which is better than Lind's and showed worst case performance much worse than Lind's. Xax has slightly better native code performance because it does not encounter the overheads induced by SFI. Xax I/O performance was similar to Lind, they state it is in the range of 7x to 161x slower. Xax also does not deal with resource exhaustion style attacks. Further, the Xax sandbox does not allow for anything but user-threading—effectively limiting the applications scalability on modern hardware.

5.2.1 System Call Interposition

The key to accessing resources in a program is through the use of system calls into the OS, so working at the system call level can be another effective mechanism for sandboxing. Instead of rebuilding an OS to change the way system calls work, modifying calls as they happen though *system call interposition* (SCI) can offer a number of advantages that make it attractive for building sandboxes. This is similar to the approach that Lind takes; however, the key difference is that SCI systems normally just allow or disallow calls, they do not reimplement the servicing of the call as is done in Lind. When interposing on system calls, a system can delegate calls or filter them. Filtering systems approve or block the system call, then make the real OS system call, delegating systems approve or block the call, then send them to another user-level process to make the actual resource request. Approaches for delegation and filtering have been extensively studied, along with their respective trade-offs with security and performance [71]. Janus Version 2 (J2) [72, 73] uses filtering and sandboxing, while Ostia [71] (like Lind) uses delegation. A key observation about SCI is that it can be error prone [74]. Garfinkel [74] identifies several common points of pain for filtering SCI systems, which could equally apply to Lind, so I will discuss each below. Ostia also provides a hybrid interposition architecture, that allows for kernel-level enforcement and user policies. The kernel module enforces policy, denies direct access to restricted resources, and delegates certain calls to the emulation library, that sends transformed system calls to agents. An agent reads the policy file and handles the delegation of calls.

The first point of pain mentioned is incorrectly replicating the OS functionality. SCI systems need to exactly match the underlying OS, because they emulate the OS to make policy decisions. This can manifest as not mirroring OS state correctly, for example not tracking all calls which modify a file handle, or manifest as bugs from not implementing the full OS functionality correctly, for example the canonicalization of path names is complex and error prone. Lind does not suffer from this class of errors, because it does not mirror the OS code, but rather actually implements its own library OS. A bug in Lind state or code only causes the application running in Lind to experience unexpected behaviour, unlike SCI systems where it results in a violation of the security constraints.

Overlooking indirect-paths to resources is the second point of pain. In a read-only file system policy examples of this include: using Unix domain sockets to write to the file system, allowing the OS to write a core dump to the file system, passing already open discriptors to the application so it can write through them, and having another process do the writing such as the local DNS daemon. In Lind, because the primitive operations of the Repy sandbox are so simple, none of the situations listed are possible in Lind. It is also important to note though that Lind (intentionally) does not implement the functionalities listed in the examples above.

Race conditions are the third point of pain. They do not effect Lind for two reasons, first, to simplify the prototype implementation Lind uses a global lock, second, Lind makes all the metadata calculations itself, and does not use the OS metadata (because it is not available), so there is no chance of unexpected state changes.

Incorrectly subsetting a complex interface is the fourth point of pain. This is the situation where the interface selected still allows some forms of attacks. In the case of Lind, this is not possible in the sandbox; however, it may be possible if the Repy system call API has preexisting problems. The hope is that the Repy API is small enough, and well enough thought out that it does not have this kind of problem. Lind also side steps this issue with a global lock, and by not guaranteeing that what Lind provides in a system call is exactly what the real OS would provide. The global lock makes sure that only one system call is executed at a time which will impact the performance of multithreaded code.

Sandboxing is a good first step to building more secure systems. None of the sandboxes discussed here claim to solve system security in general. In fact, as they are used more heavly, it is likely they will introduce new security problems of their own. Because of the nature of the security problems discussed above, sandboxing can only help solve some of those problems. Sandboxing does have some concrete security benifits though, so sandboxing is nessessary but not sufficent for securing legacy applications.

5.3 Alternate System Structures

There has been a lot of work isolating applications in the context of virtual memory. Multics introduced memory protection hardware and CPU based ring protection levels [43,75]. Multic's segmented- and paged-memory protection and ring-based instruction execution now form the basis of how modern x86 processors and OSes isolate processes from each other and the OS—though Multic's protection was only at the segment-level, not a page-level.

Outside mainstream OSes, several research OSes have introduced novel isolation and sandboxing primitives. Palladium is an intra-address space isolation mechanism that is aimed at making dynamic software extensions and components safe using commodity x86 hardware memory protection [76]. For kernel-level extensions Palladium runs each extension in a separate x86 segment, in the same address space, but with different protection levels than the main OS. For user-level extensions Palladium exploits page-level and segment-level protection hardware so as to make the changes to the application minimal. Mondrix [77] is research prototype of Linux that uses Mondriaan Memory Protection (MMP)[1] a hardware and software intra-address space memory protection scheme to enforce fine grained isolation. Much like Nooks [79], Witchel et al. use Mondrix to isolate Linux device drivers to stop them from crashing the system. MMP hardware does not actually exist, however it could provide more fine grained isolation than NaCl, without having to recompile, and with less overhead

[1]In the literature Mondriaan is spelt with both one "a" and two. In their first paper the authors mention that the work was named after the famous painter Pieter Cornelis Mondriaan, who later changed his last name to Mondrian [78].

than SFI [80]. MMP itself does not allow for system call blocking and filtering like NaCl does though.

Lind shares some things in common with Exokernel and Microkernel OSes. Exokrenel type OSes try to change the primitives that the OS exposes to the user-level application so that they can have more control over the hardware [81,82]. Exokernels are motivated by extra performance for applications that have unusual resource access patterns so standard one-size-fits-all OSes perform poorly. This leads to applications implementing the OS primitives that best suit the application at user-level [83]. Microkernel OSes, for example Minix [84] and L4 [85], try to reduce the kernel size by providing as much functionality as possible as user-level services, which other processes can then access. By doing this they make kernel services easier to build and maintain, and also gain many favourable security traits [86–90] even the ability to transparently distribute work and data over a LAN [91]. Pioneered in Nucleus [92], microkernels most commonly keep virtual memory, scheduling and IPC functionality in the kernel, but move all other services into user-level processes. In fact, a common microkernel design principle, Liedtke's minimality principle, says:

> "A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality" [93].

Like Lind, in a microkernel a program must access resources though IPC, and like Lind the user-level isolation of microkernels makes them more secure. Similarly to Lind, both exokernels and microkernels allow customized user-level services to be written to provide applications with resources—most common are network and file system implementations. Micro-kernels are known for their complex performance trade-offs [93]. Kea [94] is an OS that can dynamically include services in the kernel or run them at user-level and even restart and replace services, thus allowing the system to restructure itself as needed.

It has also been proposed that the Microkernel and Exokernel models are perfect for the development of extensible applications [95]. This observation comes about because most applications are structured around the use of shared libraries and services, common pieces of data and implementation shared between many processes in the system. Banerji conjectures that structuring a whole system into modules, then using call gates to control how they communicate could be an effective way to recompose a system [95].

Virtual Machines (VMs) run by Virtual Machine Monitors (VMMs) divide the system up as abstract machines with the same interface as the underlying hardware [44]. Some popular notable VMMs are Xen [47], VMWare [46] and KVM [48]. Others have argued that instead of providing the OS interface to programs like microkernels do, the best way to securely divide work on a system is to use VMMs then replicate the OSes [96] in each VM. VMMs have been shown to have very good temporal- and spatial-isolation [8], except in a few situations [97]. The reason VMMs work well is that the hardware interface is very simple compared to almost any other API. In the context of running applications securely, VMs are very high overhead, each VM must run an entire instance of a OS inside itself. Mirroring the problems with early microkernels, VMMs have the problem of a large TCB, so there has been lots of work to factor parts of VMMs into VM level services [98,99]. Also mirroring microkernels, the Denali Isolation Kernel attempted to make very lightweight VMs, so that each application on a machine could run in a separate VM [100]. This was also the goal of the Windows feather-weight virtual machine (FVM), another OS-level VMM, which attempts to make single program VMs possible by aggressively sharing resources between VMs, but still trying to maintain strong isolation properties.

Drawbridge [14] is a research OS that uses lightweight processes and a library OS to present a Windows persona to a wide variety of Windows applications. This is accomplished by moving a large portion of the OS into the process, and presenting a simplified system virtual machine like interface to each process. This approach brings many of the benefits of VM based temporal, spatial- and fault-isolation properties to a per-process level.

Wedge [101] is a system which provides fine grained isolation to applications so that the POLA can be concretely applied to individual parts of an application. Wedge uses the native Linux protection mechanisms, `fork`, threads, and shared memory, to make it easy for a developer to decompose applications into *compartments*. An interesting design methodology about Wedge is that, unlike the OS, by default Wedge grants no privileges to each compartment. This is an effective way to push the application closer tos conforming to the POLA. Wedge provides a tool called Crowbar, which helps the developer figure out how to decompose their system. Crowbar helps developers deduce which code will need what privileges, thus allowing each compartment to have minimal permissions. From a security engineering standpoint Wedge is a great idea; however, each application has to be refactored by a developer to run in Wedge compartments. Wedge does not introduce an isolation mechanism, but rather shows us how to better make use of existing mechanisms.

Another system which tries to make better use of existing isolation mechanisms is Ribbons [102]. Much like Wedge, Ribbons uses `fork` and shared memory to make isolated containers; however, Ribbons uses them to create a new container permission model, more fine grained than threads. Ribbons does this by introducing RibbonJ—an extension to the Java language. Each *ribbon* is composed of at least a single thread and some Java objects. Ribbons can then grant permissions to other ribbons to interact with them, and create new threads and objects. RibbonJ provides new language constructs to make ribbon interactions easy to define and understand. Ribbon's most interesting evaluation target is the Tomcat server, which hosts multiple untrusting web applications. This is a case of language virtual machine multitasking, which is a hard problem because of the resource sharing which must go on [103].

Building on type safe langages is another way to build sandboxes. Type safe langages can prevent buffer overflows and stack smashing attacks. In the OS context, the SPIN OS was built in a type-safe language [104], and more recently, the Singularity OS is a microkernel which was built in C# [105] and runs applications compiled with a custom compiler. The challenge with both OSes and applications built in type-safe languages is obtaining the best performance. This challenge stems from two factors: the overhead the langage imposes at runtime to check types that could not be statically checked, and the inability to access low level system features like pointers and CPU features.

System	**Intended Target**	**Mechanism(s) of isolation**
Xax&NaCl	Browser	Send all resource requests back to the browser.
Ostia&Janus	All	Stop all system calls and check them for safety.
Palladium	Extensions	x86 memory isolation and gates.
Mondrix	Extensions	Special hardare.
Nooks	Kernel Extensions	SFI and hardware.
Exokernel	Whole system	Direct hardware access to a library OS and RPC communication.
Microkernels	Whole system	Address spaces, RPC, protected procedures.
Ribbons	Java modules	processes, shared memory, Java extension.
Drawbridge	Windows applications	Provide address spaces and block devices to library OS.
SPIN	Spin applications	Type safe language.
Singularity	Singularity applications	Type safe language.
Wedge	Application Extensions	Linux mechanisms, POLA.
Virtualization	OSes	Hardware support, or paravirtualization.

Table 5.5: Research systems providing isolation.

System	**Level**	**Provided from**	**Mechanism(s) of isolation**
Xen	Hardware	Hypervisor	Separate pages tables, virtualization hardware, multiplexed device drivers, paravirtualization.
VMWare	Hardware	Hypervisor	Instruction rewriting, multiplexed device drivers.
KVM	Hardware	Linux kernel	Extends Linux kernel using hardware virtualization support.
Vservers	System call	Linux kernel	extended kernel data structures.
LXC	System call	Linux kernel	Minimally extended kernel data structures.

Table 5.6: Virtualization systems providing isolation.

5.4 Sandboxing in Applications

There are some well known cases studies in best practice application specific sandboxing, I will now discuss the OpenSSH server and Google's Chrome web browser. Both these applications have to be very security conscious because they receive arbitrary input from the internet—placing them both in a high risk situation for remote exploits.

Among the most dangerous OpenSSH bugs are those called "pre-auth exploits", these are bugs which can be exploited before an authenticated encrypted connection can be established. They are particularly worrying because the attacker does not need to be able to log into the system. Further compounding the situation, because the OpenSSH server must operate as many different users and create new pseudo-terminals it must be run as root. Therefore, when the server is compromised the attacker will have root privileges. The OpenSSH server has done several things to try to reduce the chances of these bugs impacting the system before they can be correctly patched. In 2002, Niels Provos of the OpenSSH project started work on something he called "Privilege Separated OpenSSH" [106, 107]. On the server, running each client connection in a separate isolated process was the goal of the project. The project is now part of the mainline OpenSSH server. In the initial implementation a child was created on every network connection, the child was given the open socket, ready to be read from, then the child had to report back to the parent the status of the connection. The idea was that if an attacker was somehow able to exploit the server, the only thing they would have access to is the child's memory, not the entire server. The child was isolated by changing its GID and UID so that it could not access any files or other processes, and isolated by running the child in a chroot jail, as described in Section 5.1.1—this effectively makes a child which has no process privileges or access to the file system. When the child says that the connection is valid through the very narrow API with the parent, it is replaced with a post-authentication process. This process has the privileges of the logged in user, but still needs to contact the parent to perform more privileged operations like creating new terminals. The approach taken is quite specific to the OpenSSH server and its needs, but does provide a good case study of how to build a high security network application.

The Chrome web browser from Google must be able to tolerate all sorts of attack types, spanning malicious content types to buggy JavaScript programs. Sandboxing, applying frequent automatic updates, and blocking known bad sites are 3 techniques Chrome uses to be more secure [24]—sandboxing being of interest to us. Chrome keeps each webpage's rendering engine in a separate process, and using a defence-in-depth strategy each process is wrapped in 3 layers of sandboxes. The inner-most sandbox is a JavaScript language sandbox. The secondary sondbox employs intra-process techniques like ASLR. The outermost sandbox (similarly to the OpenSSH sandbox) is a OS sandbox which limits what the renderer process can do. The rendering processes are controlled by a higher privilege browser kernel, which also uses the second level sandbox from above. Like the OpenSSH sandbox, the Chrome sandbox makes the renderer processes communicate with the kernel to get access to files and user-information such as cookies and histories. Some challenges the Chrome sandbox faces is that some renderers actually need to communicate with each other, for example frames on the same webpage. Also plugins like Java and Adobe Flash cannot be run in the renderer sandboxes because they expect more access to the underlying OS.

5.4.1 Software Fault Isolation

Lind uses NaCl as a binary sandbox, so it is important to expand the discussion in Section 2.2 about how NaCl and SFI work.

The foundational work of instruction re-writing for safety was originally presented by Deutsch and Grant [108]; however, their technique was not for general programs. Software Fault Isolation (SFI) was pioneered by Wahbe [15] et al. as an alternative to hardware memory protection for running two untrusting programs in one address space. The motivation for this is speed, because separating two programs as processes introduces communication overhead which is unacceptable in some cases—mainly program extensions or plugins. Since the two programs do not trust each other, there needs to be a mechanism to isolate them. In SFI's case, it isolates memory writes and code jumps so the programs cannot write into each other's memory or execute each other's code. SFI restricts the execution of native applications so they cannot execute (write or jump) outside a "fault domain". However, unlike virtual memory address spaces, transfer of execution between fault domains is fast because there is no hardware context switch involved. SFI is able to make execution safe by restricting what instructions can be executed, so control flow is restricted to a region of memory. SFI is able to load two programs into one address space, but not allow them to interfere with each other except via a specific interface. This type of isolation is spatial only, the programs can still effect each other temporally. In SFI system calls can be prevented from happening in one of the programs by disallowing the system call instruction, thus allowing one program to act as a trusted resource provider to the other—the approach NaCl has taken.

SFI is a type of proof-carrying code (PCC) [109] because its structure can be formally verified before it is run to prove its runtime constraints. PCC systems have been used to make many other interesting systems; for example, a provably safe ML byte code [109], a safe kernel-level packet filter, and a safe efficient garbage collector [110].

Wahbe's SFI implementation worked on MIPS and Alpha, both RISC architectures. McCamant and Morrisett introduce a SFI implementation called Pittsfield, for x86-32 [111]. Pittsfield pioneers the current state of the art techniques in CISC SFI sandboxing, and describes many optimizations to offset execution throughput disadvantages. Pittsfield is evaluated in a compression program which includes the compression library in the compressed file, introducing a interesting situation where data and code are packaged together, but the code should not be trusted.

NaCl is a modern implementation of SFI for x86 [12] and later x86-64 [35], which is similar to Pittsfield; however, instead of attempting to run two programs in one address space, NaCl runs one untrusted program and a trusted runtime to service the untrusted application. NaCl's intent is to let web browsers safely run untrusted—computationally intense—native code. NaCl is not focused on running two programs in one address space, but rather executing one untrusted program while disallowing it access to the operating system in any unforeseen way, so NaCl is similar to the system call inter-positioning sandboxes mentioned in Section 5.2.1. Games, 3D, sound- and image-processing, are all categories of web applications that could possibly benefit from deployment in NaCl.

NaCl uses static analysis to check code constraints. Instructions in x86 have variable width. That variable width makes it very hard to verify, for any given address, if a verifier is looking at the start or middle of an instruction. One of NaCl's techniques to build verifiable x86 code is imposing an instruction alignment pattern. NaCl makes use of a modified version of the GNU GCC compiler to produce verifiable native code for x86, x86-64 and ARM. Because of the restricted subset of native instructions NaCl uses, it can verify that the program can never violate the sandbox. Verification of code happens as it is loaded into memory, then the code is marked readonly so it cannot be changed by the program. The sandbox guarantees the program can never write to memory outside the sandbox or allow the control flow to move outside the sandbox without first passing through a trampoline which the system controls. More recent work has shown that dynamically loaded libraries[2] and JITed code can be safely loaded into the system with a modified verification process [112], opening the possibility of using more elaborate programs like language virtual machines and programs which dynamically add extensions. Recent work has used formal methods to prove the NaCl verifier correct [113], further helping us trust there is no way to escape the NaCl sandbox.

NaCl can run most programs; however it presents a different system call interface from a POSIX system. The system call interface is that of JavaScript run in the web browser. The program has access to the web browser's DOM and functionality, but the actual system API NaCl presents is very limited, with only simple file operations (open a local file, read, write, close), and no networking except those provided by the JavaScript web-sockets interface. These abilities are only satisfactory for simple applications, and quite often significant effort is required to port an application to NaCl as is evident by the existence of the NaCl-ports project [16], a collection of pre-ported software. The way Lind safely expands the NaCl system interface is by using another sandbox which can safely access system resources.

Vx32 is a runtime with similar goals to Lind and NaCl. Vx32 is a user-level library that allows sandboxing of a program with SFI, and then restriction of which system calls the restricted program can make. Unlike Lind it does not deal with temporal isolation. Unlike other SFI work, Vx32 does restrict both reads and writes between fault domains, though it has a 30% overhead, higher than that of previous work. Like Palladium, Vx32 uses the x86 segmentation hardware to control data reads and writes. To guard control flow, Vx32 uses dynamic instruction translation.

[2]Typical Linux shared libraries (DSOs) were already supported because they are loaded at program start, dynamic loading is done by systems using the `dlopen`, `dlsym` and `dlclose` calls in the `libdl.so` package. Web browsers and apache are good examples of applications which load extensions this way.

In programs with a lot of computation Vx32 will be very fast, but in programs with a lot of control flow, Vx32 can be 2 times slower than native.

Castro at. al introduce a SFI system called Byte Granularity Isolation (BGI) for retrofitting SFI fault domains onto existing device drivers [114]. They observe that finer granularity is needed to isolate kernel modules, and present a SFI system which isolates at a byte-granularity.

XFI is another modern SFI implementation that makes use of SFI's static verification, in-line guard checks and twin execution stacks. Guard checks are responsible for checking properties that cannot be statically verified. Guard checks can be inserted: manually, by the compiler, or through binary rewriting. Since their checks are software based, they claim to be very flexible in the types of checks they can perform, and be very architecture independent.

MisFIT [115] is another x86 SFI implementation which is used in VINO [117], which is discussed below. MisFIT makes the observation that object oriented programming lends itself well to program extension, and allows the developer to structure their extension system around C++ objects which are contained in SFI containers. MisFIT implements its own x86 SFI system, though since its security properties were weak and not verified, their results are not well accepted [111].

Of all the SFI implementations discussed here, the only one deployed and supported in a major way is NaCl, making it the best choice for Lind.

Uses of SFI

SFI has been used in some interesting ways. Nooks [79] isolates Linux kernel modules, so that when they crash the whole kernel does not crash as well. They use two techniques: sandboxing and micro-reboots. Nooks uses both a SFI-based and user-level sandbox. Micro-reboots conceptually refresh small parts of the system. Engler

System	Use	Mechanism
Wahabe's SFI	Isolating extensions	MIPS and Alpha instruction replacement.
Pittsfield	Isolating extensions	x86-32 instruction replacement.
NaCl	Isolate untrusted browser extension	ARM and x86 instruction replacement and instruction clustering, dynamic code insertion, system call restrictions, dual sandbox.
Vx32	Isolate extensions	Read and write restrictions + easy user-level library interfaces.
BGI	Isolate existing kernel extensions	byte-granularity isolation domains.
XFI	Isolate extensions	instruction replacement, dual stacks, guard checks in non-verifiable locations.
MisFIT	The VINO OS	Isolate C++ objects in SFI domains.

Table 5.7: Systems that use SFI.

[116] suggests using SFI as a secure means of extending kernels, giving Exokernel like access to raw resources without the usual resultant system structure. In VINO [117], Small suggests the same thing with the context of making RDBMs faster [117].

Binary rewriting techniques, similar to SFI, can be used to do other interesting things like provide transparent cross-architecture compatibility [118]. In that work, they transparently ported x86 Windows NT applications to the Alpha platform.

5.5 Security and Lind

Table 5.8 revisits the common security weaknesses listed above, and discusses how Lind impacts each of them. Lind helps with many of them. In most cases it does not deal with the problem directly, but rather isolates the program so that after the problem has occurred nothing beyond what the policy specifics can happen.

Beyond these, Lind may have some other impacts on security. Lind's interface with the system is very simple. So simple it might make it possible to apply formal verification techniques on the interface.

Lind may also have some negative security impacts. Lind could open up new side channels which might be exploited. The Unix domain socket might be one such channel. Other good security practices like whole system logging are harder with Lind, because there is no shared file system to log to.

5.6 Summary

There are a broad class of security problems which are caused by the lack of isolation in a system. Some problems like buffer overflows and injection attacks are related to system implementation, while others like insecure data storage and network communication are a result of a poor system design. Decades of OS research has produced many interesting isolation mechanisms with many differing intents. When these mechanisms can be used to stop an attack, or limit the scope of an attack they become interesting from a security point of view. Multics pioneered process level isolation, with new mechanisms like hardware assisted virtual memory. Because the chief task of OSes is to multiplex and share resources, many isolation technologies have sprung up in modern OSes to help isolate programs when necessary, and all modern OSes have some mechanisms to impose isolation. Jails, containers, and subprocesses with restricted rights are just some of the ways modern OSes like Linux, Windows and Solaris isolate workloads from untrusting parties.

Research systems span the gamut of isolation granularities, all the way from the ultra-fine grained isolation of MisFIT, RibbonsJ and BGI, all the way up to per-OS isolation modern VMMs like Xen and Drawbridge provide. These systems all intend to isolate extensions or subprograms of some sort, and, within that space isolate different resources, with varying levels of overhead in terms of memory, execution time and even programmer effort.

Like all the systems discussed here, Lind intends to isolate programs. It does so with a lightweight isolation mechanism SFI, provided by NaCl. SFI systems uses instruction level manipulation to produce code that can be verified to be safe—safe in that it can not leave the sandbox. Lind sandboxes applications on a per-process level, much like Xax and NaCl, but does not use a browser as a trusted resource provider.

Problem	**Effect of Lind**
Buffer overruns	Lind will isolate the in program so any injected code will have the same capabilities as the original program.
Uncontrolled format string	Lind will isolate the in program so any injected code will have the same capabilities as the original program.
Integer overflows	Lind will isolate the in program so any injected code will have the same capabilities as the original program.
SQL injection	Lind does not help.
Command injection	There is no forking in Lind, so no command injection as well.
Failing to handle errors	When program is in unexpected state it still can only do what its policy allows.
Unprotected network traffic	Repy can transparently encrypt TCP connections, this has never been tested in Lind though.
Weak passwords	Lind does not help.
Storing data without protection	Repy can transparently encrypt data before sending to disk.
Information leakage	Lind can block some standard channels for leakage (like network) for applications which don't use them.
Improper file access (TOUTTOC[1])	Lind file systems are not shared so this is not possible.
Trusting network name resolution	Lind does not help.
Race conditions	Lind does not help.
Strong random numbers	Lind replaces all random numbers in the OS with strong random numbers.
Poor usability	Lind does not help.
Improper pathname	Less likely because file systems are not shared.
Download without integrity check	Lind does not help.
Untrusted functionality	Lind does not help.
Use of dangerous function	Dangerous functions can only do as much as the policy dictates.
Unnecessary privileges	Lind reduces the default privileges an application has, and requires a application policy to have more privileges enabled.
Incorrect permissions	File systems are not shared, so this won't matter

Table 5.8: Common security weaknesses and how they play out in Lind.

Chapter 6

Future Work and Conclusions

There are several parts of Lind which, if developed, could significantly improve the viability of Lind as a platform for legacy applications. Missing system calls and performance are two opportunities I will discuss here.

6.1 Missing System Calls

In the development of Lind, as they have been needed, new system calls have been added. However, there are some system calls that do not fit into the Lind model as easily. Two outstanding system calls that are used heavily are `fork` and its relatives and `mmap`. `fork` and `mmap` both pose a particular problem, because their implementations cannot be easily executed in another process, then returned through RPC.

In Unix based systems, process creation happens using the `fork` then `exec` model. One of the modifications made to Repy is a function that launches a NaCl subprocess with a pre-set executable in it. The fork model is hard for Lind because the child should be a duplicate of the parent. There is no easy way to support this in Lind. While not modifying the TCB, the only way to accomplish this would be to copy the entire stack and heap out of the NaCl untrusted process, make a new NaCl process with Repy, then copy all that data back into the new process. This operation would be slow, and hard to build. Fork is thought of as a lightweight process creation mechanism because it can use copy-on-write, and this assumption would be violated by this type of implementation.

A second simpler implementation would be possible allowing the fork system call to work in NaCl. This would allow the native OS's fork to duplicate the process. This would require some adjustment to how Lind currently handles the resource connection sockets with Repy and an update to the accounting code for resources. It also raises the question whether it breaks Lind's isolation to have two processes sharing the same Repy instance. In terms of isolation fork has to be handled very carefully. Of course, unrestricted forking should not be allowed, so it would have to be rate limited. Like Repy's CPU monitoring process, the children of the NaCl process could be checked to make sure they are not spawning out of control, and since Repy creates the original NaCl process, it will have the OS' permission to kill the children. Further work would have to be done to ensure that a forked NaCl process is still as secure as the original and that forking does not violate the sandbox integrity.

Exec also poses an interesting problem. NaCl provides mechanisms for dynamic code loading; however, they were designed to work in the context of JIT compilers [112] and the loading of libraries. It may not be possible to replace the entire process' address space because those tools were not built to work at that scale.

Mmap poses less significant engineering challenges, though it still does not easily fit into the Lind model. Mmap is used for shared memory access and file access, the later is what I will discuss here. With respect to file access, it is used for easy serialization of pointer based structures, as well as for fast random access to large files. Simple mmap could be implemented by copying the file into the memory requested, then checking if the memory has changed. The problem with this approach is that mmap is normally used for performance reasons, not just for file access.

Modifying the implementation of NaCl, a simple demand paging scheme could be implemented that marks all the pages of the mmaped region as not accessible, then catches the signals the OS sends to NaCl when the pages are accessed. This is similar to how some JVMs do garbage collection [119]. This approach presents some interesting security challenges, as it would be possible for malicious signals, or too many signals to be generated. By default NaCl disables most signals so that they are unable to interfere with its operation. It remains to be seen if signals can be delivered while running untrusted code, and still be serviced safely.

6.2 Lind's Performance

Performance is important to Lind. NaCl's execution is quite fast, but the dual-process model Lind uses introduces some overheads. Repy is not optimized for system access, but rather for safety. There are two ways in which Lind's performance can be improved: removing the IPC overhead and optimizing Repy.

In recent builds of Lind, the Python profiler is able to operate in Repy—something that was previously not possible. A significant portion of Lind's overhead while servicing system calls comes from Repy, and those overheads can be broken down to a few things: slow marshalling code and slow Repy API calls.

Early in the development of Lind a struct module had to be written. Its job was to covert a Python string containing a struct into a list of corresponding Python types. For example, a C struct containing an `int` and a `char *` would be converted into a list with a Python integer object and a string object. This conversion is very slow, since it does not use native types, but rather arithmetic. For instance the Python integer would be created by reading 4 bytes from the string, then using bit operations and arithmetic to compose the integer. The justification for this is that it is a totally in-Repy solution, and never leaves the sandbox. Python has a real struct module, that uses C code to convert the types, and it is efficient. If it were ported to Repy that could speed up Lind. Even if the TCB could not be expanded with the struct module, the existing Repy struct module could probably be optimized somewhat, for example, by using an 8 bit indexed look up table instead of the bit shifting arithmetic.

Perhaps the largest hit to Lind's performance is that of Repy's system calls. In recent benchmarking, it was found that Repy's file I/O operations are more than 10 times slower than the corresponding Python operations. If some effort could be applied to speeding up these system calls it could provide a significant performance boost to Lind applications which make a lot of system calls.

Finally, Lind's performance is hampered by the dual sandbox model. The dual sandbox was used to provide more layers of security and simplify the TCB implementation of Lind. A possible alternate design would load Repy and NaCl into the same address space. This was a design called *Fast Lind*. Fast Lind is a tradeoff, since there is no IPC overhead or marshalling overhead, therefore, a lot of Lind's overheads would disappear; however, it would be much harder to guarantee the safety of Fast Lind. NaCl makes many security assumptions. For instance, it disables signals, and runs itself in a chroot jail. To make Repy work inside of NaCl, the entire Python interpreter must be run inside of the NaCl TCB. Python requires access to signal handlers and files which NaCl by default bans. It may be possible to run the two in the same process; however, the security ramifications may be subtle, and the TCB complexity will increase drastically. In the initial prototype of Fast Lind, Repy was able to run in NaCl, but it caused the process to segmentation fault on occasion, testament to the complexity[1] of this kind of implementation. With significant work, Fast Lind would be possible.

Lind's startup performance is also relatively slow. Both NaCl and Repy (including Python) have to be running before a Lind application can start. One way to amortize this startup overhead is to keep a clean copy of Repy and NaCl running, then fork them for each new Lind program, instead of launching them from scratch. This would require some changes to the program handling in several places in Lind, but still may be worth the time savings.

6.3 Lind Cloud

Using Lind as a PaaS cloud might also be an interesting opportunity. Since applications run in complete isolation from each other and are relatively light weight, Lind could make an interesting scalable cloud platform. Some issues that would have to be addressed are how Lind instances are controlled remotely, how their I/O is handled, and how a program and data are packaged, and what sort of remote data access should be provided. If files in the cloud can be accessed, Lind could be used as a way to move computation to data.

6.4 Access to Data

Especially in the context of Lind cloud, one important feature would be to provide access to local data either in the form of local devices such as web cams and other sensors, or provide access to sets of local files. Local file access, beyond simple access though copying files to the LindFS that is already provided by Lind, would give direct

[1]Complexity is always the enemy of security minded systems.

access to large files or changing files on the system. One device Lind already provides access to is the system's random number generator.

Device access would not be technically hard, instead the complexity of the system's policy and security guarantees become harder. For example, if it is possible to modify the Lind processes through the file system (in Linux this would be the `/proc/` files), the direct access might be used to escape the sandbox. Some thought would also have to go into a scheme for authorizing which files should be accessible. One way to do this could be through a manifest or some other extension to the Repy policy file that explicitly lists files which are accessible.

6.5 MacroComponents

One prototype built was for a Lind component model. In this model each software component was placed in a separate NaCl process, and interactions between them were done with message passing. Communication was asynchronous, so that components could not block the progress of other components. Components could not access each others memory, however they did share the same system resources. This kind of component model allows for the construction of very interesting systems. This is in some ways similar to the motivation of Wedge [101]. This gives the developer a way to decompose an application into well structured components, each isolated from the rest of the application. This might be easily applied to shared libraries that are loaded into an application, as they provide a well known and natural interface and fault domain.

6.6 Better Policy Enforcement

One issue with Repy, and Lind in general, is that building a good application policy is hard. The policy should try to be minimal, but still not block the application from normal operation. AppArmor [120] and Wedge [101] both come with tools to help the application developer determine a good policy. However, they are both limited in scope.

One approach that might work for Lind is to determine the program policy by analyzing strace logs of correct (however that is defined) runs of the application. Whether this is possible or not, the effectiveness of Repy is based on good polices, so developing them should be studied in more detail. One area that might provide inspiration is mobile OSes, which also use policy to allow applications access to resources.

6.7 Conclusion

In this dissertation I motivated, introduced and evaluated a sandbox for legacy applications called Lind. Lind was designed to run existing legacy applications written in C while maintaining C's good portability, performance and memory footprint. Lind is a combination of two sandboxing technologies: NaCl and Repy. The combination of these two allows for applications to be run in their binary format, but allows Lind to have a small TCB and simple design. I evaluate Lind on several micro-benchmarks to establish some performance characteristics then run several large legacy applications in Lind. The evaluation shows that Lind holds promise as a sandbox, though an industrial strength implementation would need to focus more on performance. Lind has some limitations because of the two sandbox model, as some system calls are more difficult to support. Despite this, I have shown that Lind provides an interesting point in the trade-off space of isolation and container granularity.

Appendix A

Additional Information

A.1 Software Tested in Lind

Programs		
Name	command(s)	Source
K&R Cat	`cat`	from “the C programming language” [121]
GNU Grep	`grep`, `egrep`	version 2.9
GNU Coreutils	`md5sum`, `cp`, `ls`,`wc`	version 8.9
GNU Wget	`wget`	version 1.13
IBM Nweb	`nweb`	August 8, 2012
GNU Netcat	`nc`, `netcat`	version 0.7.1
Tor	`tor`	version 0.2.3
FFmpeg	`ffmpeg`	
Network Troubleshooter		Written by myself
Name	command(s)	Source

Libraries		
Lib Event	`libevent.so`	1.4.14b-stable
OpenSSL	`libcrypt.so`	1.0.1c
zlib	`zlib.so`	1.2.7

Table A.1: Software which was compiled and run in Lind.

Bibliography

[1] M. Howard, D. LeBlanc, and J. Viega, *24 Deadly Sins of Software Security: Programming Flaws and How to Fix Them*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 2010.

[2] B. Martin, M. Brown, A. Paller, D. Kirby, and S. Christey, "2011 CWE/SANS top 25 most dangerous software errors," Common Weakness Enumeration, 7515 Colshire Drive, McLean, VA, Tech. Rep. 1.0.3, September 2011.

[3] J. Viega and G. Mcgraw, *Building Secure Software: How to Avoid Security Problems the Right Way (Addison-Wesley Professional Computing Series)*. Addison-Wesley Professional, Oct. 2001. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20\&path=ASIN/020172152X

[4] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," National Institute of Standards and Technology, Tech. Rep., 2002.

[5] J. Hunker, *Creeping Failure - How We Broke the Internet and What We Can Do to Fix It.* Mcclelland and Stewart, 2010.

[6] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, Jun. 2005. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1451869

[7] J. H. Saltzer, “Protection and the control of information sharing in Multics,” *Commun. ACM*, vol. 17, no. 7, pp. 388–402, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361067

[8] J. N. Matthews, W. Hu, M. Hapuarachchi, T. Deshane, D. Dimatos, G. Hamilton, M. McCabe, and J. Owens, “Quantifying the performance isolation properties of virtualization systems,” in *Proceedings of the 2007 workshop on Experimental computer science*, ser. ExpCS ’07. New York, NY, USA: ACM, 2007. [Online]. Available: http://doi.acm.org/10.1145/1281700.1281706

[9] C. Matthews, Y. Coady, and S. Neville, “Quantifying artifacts of virtualization: A framework for mirco-benchmarks,” in *The proceedings of the 2009 IEEE International Workshop on Quantitative Evaluation of large-scale Systems and Technologies*, 2009.

[10] “CentOS—Community ENTerprise Operating System is a free rebuild of source packages freely available from a prominent north american enterprise Linux vendor.” accessed January 29, 2013. [Online]. Available: https://www.centos.org/

[11] “Lighttpd: fly light,” 2007, accessed March 1, 2007. [Online]. Available: http://www.lighttpd.net/

[12] B. Yee, D. Sehr, G. Dardyk, J. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: A sandbox for portable, untrusted x86 native code,” in *Security and Privacy, 2009 30th IEEE Symposium on*, may 2009, pp. 79 –93.

[13] J. Cappos, A. Dadgar, J. Rasley, J. Samuel, I. Beschastnikh, C. Barsan, A. Krishnamurthy, and T. Anderson, "Retaining sandbox containment despite bugs in privileged memory-safe code," in *Proceedings of the 17th ACM conference on Computer and communications security*, ser. CCS '10. New York, NY, USA: ACM, 2010, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/1866307.1866332

[14] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt, "Rethinking the library OS from the top down," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS '11. New York, NY, USA: ACM, 2011, pp. 291–304. [Online]. Available: http://doi.acm.org/10.1145/1950365.1950399

[15] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," *SIGOPS Oper. Syst. Rev.*, vol. 27, no. 5, pp. 203–216, Dec. 1993. [Online]. Available: http://doi.acm.org/10.1145/173668.168635

[16] B. Chen, "Naclports: Ports of various open-source projects to Native Client," January 2013, accessed January 10, 2013. [Online]. Available: https://code.google.com/p/naclports/

[17] J. Cappos, "Future Repy library reference - Seattle," accessed December 9, 2012. [Online]. Available: https://seattle.cs.washington.edu/wiki/FutureRepyAPI

[18] "Linux cross reference—linux/arch/ia64/kernel/entry.s," accessed October 9, 2012. [Online]. Available: http://lxr.linux.no/linux+v3.6.1/arch/ia64/kernel/entry.S#L1471

[19] S. A. Wallace, M. Muhammad, J. Mache, and J. Cappos, "Hands-on internet with Seattle and computers from across the globe," *J. Comput. Sci. Coll.*, vol. 27, no. 1, pp. 137–142, Oct. 2011. [Online]. Available: http://dl.acm.org/citation.cfm?id=2037151.2037181

[20] J. Cappos, I. Beschastnikh, A. Krishnamurthy, and T. Anderson, "Seattle: a platform for educational cloud computing," in *Proceedings of the 40th ACM technical symposium on Computer science education*, ser. SIGCSE '09. New York, NY, USA: ACM, 2009, pp. 111–115. [Online]. Available: http://doi.acm.org/10.1145/1508865.1508905

[21] J. Cappos and I. Beschastnikh, "Teaching networking and distributed systems with Seattle: tutorial presentation," *J. Comput. Sci. Coll.*, vol. 25, no. 5, pp. 308–310, May 2010. [Online]. Available: http://dl.acm.org/citation.cfm?id=1747137.1747196

[22] J. Cappos and J. Jacky, "Model-based testing without a model: assessing portability in the Seattle testbed," in *Proceedings of the 5th international conference on Systems software verification*, ser. SSV'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 4–4. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929004.1929008

[23] L. Collares, C. Matthews, J. Cappos, Y. Coady, and R. McGeer, "Et (smart) phone home!" in *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, &#38; VMIL'11*, ser. SPLASH '11 Workshops. New York, NY, USA: ACM, 2011, pp. 283–288. [Online]. Available: http://doi.acm.org/10.1145/2095050.2095098

[24] C. Reis, A. Barth, and C. Pizano, “Browser security: lessons from Google Chrome,” *Commun. ACM*, vol. 52, no. 8, pp. 45–49, Aug. 2009. [Online]. Available: http://doi.acm.org/10.1145/1536616.1536634

[25] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design patterns: elements of reusable object-oriented software.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[26] D. A. Wheeler, “generated using 'SLOCCount',” accessed October 15, 2012. [Online]. Available: http://www.dwheeler.com/sloccount/

[27] “Filesystem in Userspace,” accessed January 26, 2013. [Online]. Available: http://fuse.sourceforge.net/

[28] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch, “Leveraging legacy code to deploy desktop applications on the web,” in *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 339–354. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855765

[29] “Oracle VM VirtualBox,” accessed January 26, 2013. [Online]. Available: https://www.virtualbox.org/

[30] “Ubuntu 10.04.4 LTS (Lucid Lynx),” accessed January 26, 2013. [Online]. Available: http://releases.ubuntu.com/lucid/

[31] N. Mathewson and N. Provos, “libevent—an event notification library,” accessed September 1, 2012. [Online]. Available: http://libevent.org/

[32] M. J. Cox, R. S. Engelschall, S. Henson, and B. Laurie, “OpenSSL: The open source toolkit for SSL/TLS,” September 2012. [Online]. Available: http://www.openssl.org/

[33] G. Roelofs and M. Adler, "A massively spiffy yet delicately unobtrusive compression library (also free, not to mention unencumbered by patents)," September 2012. [Online]. Available: http://www.zlib.net/

[34] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: formal verification of an os kernel," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 207–220. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629596

[35] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen, "Adapting software fault isolation to contemporary cpu architectures," in *Proceedings of the 19th USENIX conference on Security*, ser. USENIX Security'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1929820.1929822

[36] "grep – GNU project – Free Software Foundation (FSF)," accessed December 9, 2012. [Online]. Available: http://www.gnu.org/software/grep/

[37] "GNU Wget," accessed December 9, 2012. [Online]. Available: http://www.gnu.org/software/wget/

[38] N. Griffiths, "nweb: a tiny, safe web server (static pages only)," January 2012, accessed January 30, 2013. [Online]. Available: http://www.ibm.com/developerworks/systems/library/es-nweb/index.html

[39] "Tor project: Anonymity online," January 2012, accessed January 30, 2013. [Online]. Available: https://www.torproject.org/

[40] G. Giacobbi, “The GNU Netcat project,” September 2012. [Online]. Available: http://netcat.sourceforge.net/

[41] M. Hart, “Free ebooks by Project Gutenberg,” November 2012, accessed December 9, 2012. [Online]. Available: http://www.gutenberg.org/

[42] E. W. Dijkstra, “The structure of the THE-multiprogramming system,” in *Proceedings of the first ACM symposium on Operating System Principles*, ser. SOSP ’67. New York, NY, USA: ACM, 1967, pp. 10.1–10.6. [Online]. Available: http://doi.acm.org/10.1145/800001.811672

[43] A. Bensoussan, C. T. Clingen, and R. C. Daley, “The Multics virtual memory: concepts and design,” *Commun. ACM*, vol. 15, no. 5, pp. 308–318, May 1972. [Online]. Available: http://doi.acm.org/10.1145/355602.361306

[44] G. J. Popek and R. P. Goldberg, “Formal requirements for virtualizable third generation architectures,” *Commun. ACM*, vol. 17, no. 7, pp. 412–421, Jul. 1974. [Online]. Available: http://doi.acm.org/10.1145/361011.361073

[45] *Intel 64 and IA-32 Architectures Software Developer’s Manual: System Programming Guide*, August 2012.

[46] “VMWare Workstation,” accessed January 5, 2013. [Online]. Available: http://www.vmware.com/

[47] 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 *SOSP03*. New York, NY, USA: ACM, 2003, pp. 164–177.

[48] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori, “kvm: the Linux Virtual Machine Monitor,” in *OLS ’07: Proceedings of the*

Linux Symposium, vol. 1, Jun. 2007, pp. 225–230. [Online]. Available: http://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf

[49] "LXC homepage," accessed December 10, 2012. [Online]. Available: http://lxc.sourceforge.net

[50] J. Gélinas and H. Pötzl, "Linux VServers project," accessed December 9, 2012. [Online]. Available: http://linux-vserver.org/

[51] J. Williams and D. Wichers, "Open web application security project top 10 2010," Open Web Application Security Project, Tech. Rep., 2010.

[52] "Java virtual machine—Wikipedia," January 2013. [Online]. Available: https://en.wikipedia.org/wiki/Java_virtual_machine

[53] K. Randolph, "Inside adobe reader protected mode – part 1 - design," Adobe Secure Software Engineering Team (ASSET) Blog. [Online]. Available: http://blogs.adobe.com/asset/2010/10/inside-adobe-reader-protected-mode-part-1-design.html

[54] ——, "Inside adobe reader protected mode – part 3 – broker process, policies, and interprocess communication," Adobe Secure Software Engineering Team (ASSET) Blog. [Online]. Available: http://blogs.adobe.com/asset/2010/11/

[55] D. Dean, E. W. Felten, D. S. Wallach, and D. Balfanz, "Java security: Web browsers and beyond," in *Internet besieged*, D. E. Denning and P. J. Denning, Eds. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1998, pp. 241–269. [Online]. Available: http://dl.acm.org/citation.cfm?id=275737.275753

[56] L. Gong, “Java security architecture revisited,” *Commun. ACM*, vol. 54, no. 11, pp. 48–52, Nov. 2011. [Online]. Available: http://doi.acm.org/10.1145/2018396.2018411

[57] J. Cappos, “All of the Python you need to forget to use Repy,” January 2013. [Online]. Available: https://seattle.cs.washington.edu/wiki/PythonVsRepy

[58] “Apple sandbox,” accessed April 28th, 2012. [Online]. Available: http://phys.org/news/2011-11-apple-sandbox-angers.html

[59] D. LeBlanc, “Practical Windows sandboxing,” accessed December 14, 2012. [Online]. Available: http://blogs.msdn.com/b/david_leblanc/archive/2007/07/27/practical-windows-sandboxing-part-1.aspx

[60] R. E. Smith, “Mandatory protection for internet server software,” in *Proceedings of the 12th Annual Computer Security Applications Conference*, ser. ACSAC ’96. Washington, DC, USA: IEEE Computer Society, 1996, pp. 178–. [Online]. Available: http://dl.acm.org/citation.cfm?id=784588.784626

[61] *System Administration Guide: Oracle Solaris Containers-Resource Management and Oracle Solaris Zones.*

[62] P. Kamp and R. N. M. Watson, “Jails: Confining the omnipotent root,” in *In Proc. 2nd Intl. SANE Conference*, 2000.

[63] Swsoft, “OpenVZ Homepage,” http://openvz.org/.

[64] “Parallels virtuozzo containers,” accessed January 10, 2013. [Online]. Available: http://www.parallels.com/products/pvc/

[65] S. Soltesz, H. Pötzl, M. E. Fiuczynski, A. Bavier, and L. Peterson, “Container-based operating system virtualization: a scalable, high-performance alternative

to hypervisors," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 275–287, Mar. 2007. [Online]. Available: http://dx.doi.org/10.1145/1272998.1273025

[66] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker, "patch (1) considered harmful," in *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, ser. HOTOS'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251123.1251139

[67] M. Helsley, "LXC Linux container tools: Tour and set up the new container tools called Linux Containers," February 2009, accessed January 10, 2013. [Online]. Available: http://www.ibm.com/developerworks/linux/library/l-lxc-containers/

[68] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[69] "AppArmour," accessed April 28th, 2012. [Online]. Available: http://www.nds8.co.uk/apparmour.asp

[70] P. A. Loscocco and S. D. Smalley, "Meeting critical security objectives with Security-Enhanced Linux," in *Proceedings of the 2001 Ottawa Linux Symposium*, 2001. [Online]. Available: http://lwn.net/2001/features/OLS/pdf/pdf/selinux.pdf

[71] Mendel, "Ostia: A delegating architecture for secure system call interposition," in *Proceedings of the 11th Annual Symposium on Network and Distributed System Security (NDSS 2004)*, Feb. 2004.

[72] I. Goldberg, D. Wagner, R. Thomas, and E. Brewer, "A secure environment for untrusted helper applications (confining the wily hacker)," in *Proceedings of the Sixth USENIX UNIX Security Symposium*, 1996.

[73] D. A. Wagner, "Janus: An approach for confinement of untrusted applications," University of California, Berkeley, Tech. Rep. CSD-99-1056, 1999.

[74] T. Garfinkel, "Traps and pitfalls: practical problems in system call interposition based security tools," in *In Proc. Network and Distributed Systems Security Symposium*, 2003, pp. 163–176.

[75] P. Green, "Multics virtual memory — tutorial and reflections," accessed January 1, 2013. [Online]. Available: ftp://ftp.stratus.com/vos/multics/pg/mvm.html

[76] T.-c. Chiueh, G. Venkitachalam, and P. Pradhan, "Integrating segmentation and paging protection for safe, efficient and transparent software extensions," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 140–153. [Online]. Available: http://doi.acm.org/10.1145/319151.319161

[77] E. Witchel, J. Rhee, and K. Asanović, "Mondrix: memory isolation for Linux using Mondriaan Memory Protection," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, ser. SOSP '05. New York, NY, USA: ACM, 2005, pp. 31–44. [Online]. Available: http://doi.acm.org/10.1145/1095810.1095814

[78] "Piet Mondrian—Wikipedia," accessed January 1, 2013. [Online]. Available: http://en.wikipedia.org/wiki/Piet_Mondrian

[79] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, ser. SOSP '03. New York, NY, USA: ACM, 2003, pp. 207–222. [Online]. Available: http://doi.acm.org/10.1145/945445.945466

[80] E. Witchel, J. Cates, and K. Asanović, "Mondrian memory protection," in *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, ser. ASPLOS-X. New York, NY, USA: ACM, 2002, pp. 304–316. [Online]. Available: http://doi.acm.org/10.1145/605397.605429

[81] 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 fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 251–266. [Online]. Available: http://doi.acm.org/10.1145/224056.224076

[82] D. R. Engler and M. F. Kaashoek, "Exterminate all operating system abstractions," in *Proceedings of the Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, ser. HOTOS '95. Washington, DC, USA: IEEE Computer Society, 1995, pp. 78–. [Online]. Available: http://dl.acm.org/citation.cfm?id=822074.822387

[83] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie,

"Application performance and flexibility on exokernel systems," in *Proceedings of the sixteenth ACM symposium on Operating systems principles*, ser. SOSP '97. New York, NY, USA: ACM, 1997, pp. 52–65. [Online]. Available: http://doi.acm.org/10.1145/268998.266644

[84] J. Howatt, "Operating systems projects: Minix revisited," *SIGCSE Bull.*, vol. 34, no. 4, pp. 109–111, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/820127.820179

[85] J. Liedtke, "Toward real microkernels," *Commun. ACM*, vol. 39, no. 9, pp. 70–77, Sep. 1996. [Online]. Available: http://doi.acm.org/10.1145/234215.234473

[86] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure?" *Computer*, vol. 39, no. 5, pp. 44–51, May 2006. [Online]. Available: http://dx.doi.org/10.1109/MC.2006.156

[87] N. Hardy, "KeyKOS architecture," *SIGOPS Oper. Syst. Rev.*, vol. 19, no. 4, pp. 8–25, Oct. 1985. [Online]. Available: http://doi.acm.org/10.1145/858336.858337

[88] C. R. Landau, "Security in a secure capability-based system," *SIGOPS Oper. Syst. Rev.*, vol. 23, no. 4, pp. 2–4, Oct. 1989. [Online]. Available: http://doi.acm.org/10.1145/70730.70731

[89] J. S. Shapiro, J. M. Smith, and D. J. Farber, "EROS: a fast capability system," in *Proceedings of the seventeenth ACM symposium on Operating systems principles*, ser. SOSP '99. New York, NY, USA: ACM, 1999, pp. 170–185. [Online]. Available: http://doi.acm.org/10.1145/319151.319163

[90] G. Heiser, K. Elphinstone, I. Kuz, G. Klein, and S. M. Petters, "Towards trustworthy computing systems: taking microkernels to the next level,"

SIGOPS Oper. Syst. Rev., vol. 41, no. 4, pp. 3–11, Jul. 2007. [Online]. Available: http://doi.acm.org/10.1145/1278901.1278904

[91] G. Hamilton and P. Kougiouris, "The Spring Nucleus: a microkernel for objects," in *Proceedings of the USENIX Summer 1993 Technical Conference on Summer technical conference - Volume 1*, ser. Usenix-stc'93. Berkeley, CA, USA: USENIX Association, 1993, pp. 11:1–11:15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1361453.1361464

[92] P. B. Hansen, "The nucleus of a multiprogramming system," *Commun. ACM*, vol. 13, no. 4, pp. 238–241, Apr. 1970. [Online]. Available: http://doi.acm.org/10.1145/362258.362278

[93] J. Liedtke, "On micro-kernel construction," in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 237–250. [Online]. Available: http://doi.acm.org/10.1145/224056.224075

[94] *Fourth International Conference on Configurable Distributed Systems, 1998, Proceedings, Annapolis, MA, USA, 6 May, 1998.* IEEE, 1998.

[95] A. Banerji, J. M. Tracey, and D. L. Cohn, "Protected shared libraries: a new approach to modularity and sharing," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '97. Berkeley, CA, USA: USENIX Association, 1997, pp. 5–5. [Online]. Available: http://dl.acm.org/citation.cfm?id=1268680.1268685

[96] G. Heiser, V. Uhlig, and J. LeVasseur, "Are virtual-machine monitors microkernels done right?" *SIGOPS Oper. Syst. Rev.*, vol. 40, no. 1, pp. 95–99, Jan. 2006. [Online]. Available: http://doi.acm.org/10.1145/1113361.1113363

[97] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 199–212. [Online]. Available: http://doi.acm.org/10.1145/1653662.1653687

[98] D. G. Murray, G. Milos, and S. Hand, "Improving Xen security through disaggregation," in *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, ser. VEE '08. New York, NY, USA: ACM, 2008, pp. 151–160. [Online]. Available: http://doi.acm.org/10.1145/1346256.1346278

[99] P. Colp, M. Nanavati, J. Zhu, W. Aiello, G. Coker, T. Deegan, P. Loscocco, and A. Warfield, "Breaking up is hard to do: security and functionality in a commodity hypervisor," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, ser. SOSP '11. New York, NY, USA: ACM, 2011, pp. 189–202. [Online]. Available: http://doi.acm.org/10.1145/2043556.2043575

[100] A. Whitaker, M. Shaw, and S. D. Gribble, "Scale and performance in the Denali isolation kernel," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. SI, pp. 195–209, Dec. 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844147

[101] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "Wedge: splitting applications into reduced-privilege compartments," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, ser. NSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 309–322. [Online]. Available: http://dl.acm.org/citation.cfm?id=1387589.1387611

[102] K. J. Hoffman, H. Metzger, and P. Eugster, “Ribbons: a partially shared memory programming model,” in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, ser. OOPSLA ’11. New York, NY, USA: ACM, 2011, pp. 289–306. [Online]. Available: http://doi.acm.org/10.1145/2048066.2048091

[103] G. Czajkowski and L. Daynés, “Multitasking without comprimise: a virtual machine evolution,” in *Proceedings of the 16th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, ser. OOPSLA ’01. New York, NY, USA: ACM, 2001, pp. 125–138. [Online]. Available: http://doi.acm.org/10.1145/504282.504292

[104] B. N. Bershad, S. Savage, P. Pardyak, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, “Extensibility safety and performance in the SPIN operating system,” in *Proceedings of the fifteenth ACM symposium on Operating systems principles*, ser. SOSP ’95. New York, NY, USA: ACM, 1995, pp. 267–283. [Online]. Available: http://doi.acm.org/10.1145/224056.224077

[105] G. C. Hunt and J. R. Larus, “Singularity: rethinking the software stack,” *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 2, pp. 37–49, Apr. 2007. [Online]. Available: http://doi.acm.org/10.1145/1243418.1243424

[106] N. Provos, M. Friedl, and P. Honeyman, “Preventing privilege escalation,” in *Proceedings of the 12th conference on USENIX Security Symposium - Volume 12*, ser. SSYM’03. Berkeley, CA, USA: USENIX Association, 2003, pp. 16–16. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251353.1251369

[107] N. Provos, “Privilege separated OpenSSH,” accessed January 1, 2013. [Online]. Available: http://www.citi.umich.edu/u/provos/ssh/privsep.html

[108] P. Deutsch and C. A. Grant, “A flexible measurement tool for software systems,” in *Information Processing 71*, Ljubljana, Yugoslavia, 1971, pp. 320–326.

[109] G. C. Necula, “Proof-carrying code,” in *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL ’97. New York, NY, USA: ACM, 1997, pp. 106–119. [Online]. Available: http://doi.acm.org/10.1145/263699.263712

[110] C.-X. Lin, Y.-Y. Chen, L. Li, and B. Hua, “Garbage collector verification for proof-carrying code,” *J. Comput. Sci. Technol.*, vol. 22, no. 3, pp. 426–437, May 2007. [Online]. Available: http://dx.doi.org/10.1007/s11390-007-9049-z

[111] S. McCamant and G. Morrisett, “Evaluating SFI for a CISC architecture,” in *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, ser. USENIX-SS’06. Berkeley, CA, USA: USENIX Association, 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267336.1267351

[112] J. Ansel, P. Marchenko, U. Erlingsson, E. Taylor, B. Chen, D. L. Schuff, D. Sehr, C. L. Biffle, and B. Yee, “Language-independent sandboxing of just-in-time compilation and self-modifying code,” in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, ser. PLDI ’11. New York, NY, USA: ACM, 2011, pp. 355–366. [Online]. Available: http://doi.acm.org/10.1145/1993498.1993540

[113] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, “RockSalt: better, faster, stronger SFI for the x86,” in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI ’12. New York, NY, USA: ACM, 2012, pp. 395–404. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254111

[114] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, "Fast byte-granularity software fault isolation," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ser. SOSP '09. New York, NY, USA: ACM, 2009, pp. 45–58. [Online]. Available: http://doi.acm.org/10.1145/1629575.1629581

[115] C. Small and M. I. Seltzer, "MiSFIT: Constructing safe extensible systems," *IEEE Concurrency*, vol. 6, no. 3, pp. 34–41, Jul. 1998. [Online]. Available: http://dx.doi.org/10.1109/4434.708254

[116] D. Engler, M. F. Kaashoek, and J. O'Toole, "The operating system kernel as a secure programmable machine," in *Proceedings of the 6th workshop on ACM SIGOPS European workshop: Matching operating systems to application needs*, ser. EW 6. New York, NY, USA: ACM, 1994, pp. 62–67. [Online]. Available: http://doi.acm.org/10.1145/504390.504407

[117] C. Small and M. Seltzer, "VINO: An integrated platform for operating system and database research," 1994.

[118] A. Chernoff and R. Hookway, "DIGITAL FX!32 running 32-bit x86 applications on Alpha NT," in *Proceedings of the USENIX Windows NT Workshop on The USENIX Windows NT Workshop 1997*, ser. NT'97. Berkeley, CA, USA: USENIX Association, 1997, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267658.1267660

[119] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss, "Cramm: virtual memory support for garbage-collected applications," in *Proceedings of the 7th symposium on Operating systems design and implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 103–116. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298466

[120] “The AppArmor security project,” January 2013. [Online]. Available: http://wiki.apparmor.net/

[121] W. B. Kernighan and D. M. Ritchie, *The C Programming Language*, 2nd ed. Prentice Hall Professional Technical Reference, 1988.