Where the Bugs Are: Better Virtualization Security by Treading Commonly Used Kernel Paths

Yiwen Li
New York University
liyiwen@nyu.edu

Ali Gholami
KTH Royal Institute of Technology
gholami@kth.se

Chris Matthews
Apple Inc.
chris.matthews@apple.com

Yanyan Zhuang
New York University
yyzh@nyu.edu

Justin Cappos
New York University
jcappos@nyu.edu

Abstract

Building secure systems that allow untrusted programs to run without triggering inherent vulnerabilities is very challenging. Through techniques such as system call interposition, operating system virtualization, and library OSes, security researchers and system developers have attempted to minimize the risk of such flaws. However, these technologies can add new code to the system's TCB that often has new exploitable vulnerabilities. In addition, the complex functions that these systems use can enable access to portions of the underlying kernel that could trigger these flaws.

This paper introduces a new metric to determine where kernel flaws are likely to be located, based on the hypothesis that commonly-used kernel paths, executed by popular applications on a daily basis, contain fewer vulnerabilities than less-used paths. With this insight, we devise the safely-reimplement design, a new technique that limits kernel access to only commonly used paths, and reimplements complex OS functions within a sandbox. Finally, we use this design to prototype and test a secure virtualization system called Lind. The results show that for the 35 bugs we examined in Linux kernel version 3.14.1, Lind can reduce the threat of attacks on the OS kernel to less than 3%. This result is about an order of magnitude better than existing systems like VirtualBox (40%), VMWare Workstation (31%), Docker (23%), LXC (34%), QEMU (14%), KVM (14%), and Graphene (23%).

1 Introduction

Despite the efforts of programmers to develop systems free of vulnerabilities, researchers frequently uncover new flaws in operating system kernels that can be triggered by untrusted programs. To reduce security flaws a variety of techniques, including system call interposition, operating system virtualization and library OSes, have been proposed. However, evaluating the level of security in such virtualization technologies through a quantitative approach is a challenging issue. As security comparisons between diverse systems are often qualitative or predictive in nature, it is hard to discern which practices actually better protect the kernel.

In this work, we devise a metric that provides a strong indication of where bugs will arise in the Linux kernel. Namely, we demonstrate that executing lines of kernel code that are not used by popular programs represent a substantial security risk. We support this hypothesis with data from a quantitative analysis of resilience to flaws in two versions of the Linux kernel, and using this data to predict where within the kernel bugs will later be discovered. We found that only 2.5% - 4.0% of flaws occur in commonly used code, despite accounting for only 32.2% - 34.5% of the reachable kernel code.

Guided by this metric, we propose the safely-reimplement design, a new technique for building virtualization systems. In this design, the operating system interface (i.e., POSIX) is reimplemented in a restricted environment (i.e., a sandbox) that limits kernel access to only commonly-used paths. Creating such a safe space requires building complex operating system functionality (i.e., directories and permissions) utilizing only commonly-used primitives (i.e., read and writing to a file). However, it allows complex functionality to safely run untrusted user programs on vulnerable code. This ensures that an attacker will be unable to trigger bugs in less commonly-used kernel paths. As a result, bugs or failures will be contained within the restricted environment.

Using this design, we develop a prototype virtualization system for running untrusted user programs on vulnerable kernel code. Dubbed Lind, the prototype adapts two existing technologies- Native Client (NaCl) [**?**] and Restricted Python (Repy) [**?**], to handle distinctly different tasks. NaCl serves as a computational module that isolates binaries, providing memory safety for a legacy

program like Tor while passing the calls to the operating system interface. Repy is a sandbox technique we used to build our operating system interface called SafePOSIX. SafePOSIX is isolated within Repy to restrict the interface to invoke only common code paths. It reimplements complex operating system functionality in an isolated environment provided by a small (8K LOC) Repy sandbox kernel. This provides straightforward access to the required common kernel paths while allowing more complex functionality to be built. As a result, Lind can offer enhanced security without sacrificing basic functionality.

To demonstrate the effectiveness of Lind, we replicated 35 kernel bugs in the Linux kernel version 3.14.1. We exploited these kernel bugs in eight different virtualized environments, including commercial systems VirtualBox, VMWare Workstation, Docker, LXC, KVM, QEMU, and research prototypes Graphene and Lind. Our results show that applications run in Lind were the least likely to trigger kernel bugs, with only one out of the 35 (2.9%) kernel vulnerabilities being found. Furthermore, our results show that virtualization systems that use commonly executed kernel paths have better resilience to vulnerabilities in the underlying kernel.

In summary, the main contributions of this paper are as follows:

- We propose and verify a novel metric for quantitatively measuring and evaluating the security of privileged code, such as in an OS kernel.
- We validate the hypothesis that commonly-used kernel paths contain fewer bugs, as predicted by our metric.
- We apply the metric to a secure design that accesses only commonly-used kernel paths. System calls from uncommon paths are reimplemented in a restricted environment.
- We develop a prototype secure virtualization system called Lind that encompasses the reimplementation design and a strictly controlled access to the kernel through a very small trusted computing base.
- We evaluate Lind and find it triggers only one (2.9%) of the kernel vulnerabilities we examined. This result is about an order of magnitude better than the seven other virtualization systems that we tested.

The remainder of this paper is organized as follows. Section **??** describes our key hypothesis and how we tested our metric. Section **??** presents our goals for using the metric to guide the design of effective security systems. A study of kernel protection strategies, including our newly proposed design is discussed in Section **??**. Section **??** describes the implementation of Lind. In Section **??** the security and efficiency of Lind is tested against other virtualization systems. Section **??** outlines the limitations of Lind and possible future initiatives. Finally, Section **??** reviews existing techniques that share Lind's security techniques and goals, and we share some concluding thoughts in Section **??**.

2 Goals and Threat Model

Goals. Our goal is to design and build secure virtualization systems, that allow untrusted programs to run on an unpatched and vulnerable host OS (Linux OS in this study), without causing damages to other parts of the system. [Yanyan: what are the other parts of the system?] We focus on providing a solution when a host OS contains vulnerabilities that an attacker could exploit. To combat this, untrusted programs are executed in a secure virtualization system, such as a guest OS VM, a system call interposition module, or a library OS. We are trying to design such a secure virtualization system that could protect the underlying host OS and all the programs running in the host OS, when facing malicious attackers.

In the following, we define the level of security that we intend to enforce, and the possible attacks that we desire to protect against. We assume that an attacker is able to run an attack program in our secure virtualization system, attempting to exploit an unpatched vulnerability in the host OS. Our goal is to prevent the attack program from triggering a vulnerability in the host OS. To trigger a vulnerability, the attacker could have two approaches. First, the attack program could exploit a flaw in the host OS by performing an action in the secure virtualization system that triggers a flaw in the host OS. Effectively, the attacker is taking advantage of the secure system utilizing paths in the host OS kernel that have a vulnerability. Second, the attack program could exploit a flaw in the secure virtualization system to escape its containment. [Yanyan: maybe explain what is to escape its containment?] Once the attacker has exploited this type of flaws, the attack program will be able to run arbitrary code, and therefore can make system calls in the host OS directly, which allows the attacker to trigger a known exploit. We try to prevent both attacks.

It should be noted that several solutions exist for building a virtualization system for running untrusted programs. However, the following techniques are out of scope as they have different hardware and software requirements. We do not compare with solutions that do not run on a full-fledged privileged operating system, such as virtualization system that uses a bare-metal hypervisor [**?**] or hardware-based virtualization solution [**?**]. These solutions have substantial dependency on the underlying hardware, which is different from our goal. In addition, they do not interact with a privileged OS kernel, which makes it very difficult to directly compare with our solution that accesses and relies on the OS

kernel. [Yanyan: I feel this make it sound like "accesses and relies on the OS kernel" constrains our method.]

Threat model. To summarize, our threat model makes the following assumptions.

- The attacker possesses knowledge about one or more unpatched vulnerabilities in the host OS.
- The attacker is permitted to execute any code in the secure virtualization system.
- If the attack program can trigger a vulnerability in any privileged code, whether in the host OS or the secure virtualization system, the attacker is then able to compromise the system.

To achieve our goals, we need to have a better understanding about how vulnerabilities are triggered by attack programs. It is therefore critical to know where vulnerabilities are located in an OS kernel. Hence, we design a security metric that measures how bugs and vulnerabilities are distributed in the kernel.

3 Developing and Assessing a Quantitative Evaluation Metric for Kernel Security

Developing a quantitative metric that could more accurately identify portions of the OS kernel that had the highest potential to unleash inherent bugs begins with a hypothesis based on observation and common sense. In this section, we show how we went about documenting the accuracy of the statement below.

Key Hypothesis: Kernel paths that are executed by common applications during everyday use are less likely to contain security flaws.

This key hypothesis posits that by understanding how and when a line of code in the kernel is used, we can predict its likelihood to contain a security flaw. The intuition is that these code paths are very well-tested due to their constant use, and thus it is much less likely that security bugs will occur in these lines of code.

3.1 Experiment Setup

To test our hypothesis we performed an analysis of two different versions of the Linux kernel, 3.13.0 and 3.14.1. Our findings for these versions are quantitatively and qualitatively similar, so we report the results for 3.13.0 in this section and use 3.14.1 in Section **??**. To trace the kernel, we used `gcov` [**?**]. A standard utility with the GNU compiler collection (GCC) suite, `gcov` is a program profiling tool that indicates which lines of kernel code are executed while an application runs.

Commonly-used kernel paths. To capture the commonly-used kernel paths, we used two strategies concurrently. First, we attempted to capture the normal usage behavior of popular applications. To do this, two students used applications in the 50 most popular Debian packages [**?**] (omitting libraries) for Debian 7.0. Each student used 25 applications for their designed tasks (i.e., writing, spell checking, and then printing a letter in a text editor, or recoloring and adding a caption to a picture in image processing software). In instances where there were two applications that performed a similar task (i.e., Mozilla Firefox and Google Chrome), both programs were used. These tests were completed over 20 hours of total use over 5 calendar days.

The second strategy was to try to capture the total range of usages for a specific computer user. Hence the students used the workstation as their desktop machine for a one week period. They did their homework, developed software, communicated with friends and family, etc., using this system. Software was installed as needed.

Using these two strategies, we obtained a profile of the lines of kernel code (publicly available at [**?**]), that indicate a set of commonly-used kernel paths.

Locating bugs. Having identified the kernel paths used during application execution, we next addressed how bugs are distributed in these paths. We collected a list of severe kernel bugs from the National Vulnerability Database [**?**]. For each bug, we found the patch that fixed the problem and identified which lines of kernel code were modified to remove the bug. For the purpose of this study, a user program that can execute a line of kernel code changed by such a patch is considered to have the *potential to exploit that flaw*. Note, it is possible that in some situations this may overestimate the ability for an attacker to exploit a flaw, since it may be possible that additional lines of code must also be executed.

3.2 Results and Analysis

We now examine our hypothesis given traces for the commonly-used kernel paths and the set of lines that were patched to fix bugs. We found that only one of the 40 kernel bugs falls within the commonly-used paths, despite the commonly-used kernel paths making up 12.4% of the kernel. To verify that bugs are more likely to appear in certain parts of the kernel, we performed the following analysis.

We assume that kernel bugs appear at an average rate proportional to the number of lines of kernel code, and independently of the time since the last bug occurrence. Therefore, the rate of defect occurrence per LOC follows a Poisson distribution [**?**]. This is consistent with the work of Mayer, et. al. [**?**]. Our hypothesis is that bugs occur at different rates in different parts of the kernel, i.e., the risky portion has more bugs. Without loss of generality, we assume that the kernel can be divided

into two sections, A and B, where bugs occur at rates λ_A and λ_B, and $\lambda_A \neq \lambda_B$. Given the null-hypothesis that the rate of defect occurrences is the *same* in set A and B (or bugs in A and B are drawn from the same Poisson distribution), we used the Uniformly Most Powerful Unbiased (UMPU) test [**?**] to compare unequal-sized code blocks. At a significance level of $\alpha = 0.01$, the test was significant at $\rho = 0.0015$, rejecting the null-hypothesis. The test also reported a 95% confidence interval that $\lambda_A/\lambda_B \in [0.002, 0.525]$. This indicates that ratios between bug-rates in each set are well below 1, and B is the risky set that tends to have more bugs.

Comparison with other metrics. We are not the first to propose a metric for which kernel code may be buggy. Many metrics work at a coarser granularity (e.g., at file level) than our work that focuses on individual lines of code. This is particularly key because at a file granularity, we found that commonly used programs used parts of 32 files that contained flaws. In fact, common programs executed 36 functions that later were patched to fix security flaws, indicating the need to better localize bugs.

Earlier work by Ozment, et al. [**?**] demonstrated that code that had been around longer in the BSD kernel tended to have fewer bugs.

Chou, et al. [**?**] showed that certain parts of the kernel were more vulnerable than others. In particular, device drivers have much higher error rates than those in other parts of the kernel.

However, this led us to consider that perhaps since drivers are not used in many scenarios, code that is unreachable in some situations may have a different vulnerability profile. To test this, we further examined the reachable lines of code within the kernel using two techniques. First, we performed system call fuzzing experiments with the Trinity system call fuzz tester [**?**]. These included 16 child processes (Trinity workers) executing each Linux system call with 1 million iterations. Second, we used the Linux Test Project (LTP) [**?**], a test suite written using detailed kernel knowledge. This test suite is meant to exercise the existing Linux system call interface to test its correctness, robustness, and performance impact.

The (primarily) black box fuzzing technique from Trinity and test suite of LTP combine to reach 44.6% of the kernel, including all 12.4% of common paths. The security in the reachable portion is actually slightly higher than the unreachable portion. This is true despite approximately 1/3 of this code, the commonly-used paths, only containing a single flaw. This means that the rate of bug occurrence in reachable, but not commonly-used kernel paths is actually higher than that in unused code. We speculate that this may be because of a higher rate of bug discovery in code that is available to execute in diverse configurations.

To summarize, we demonstrated that the metric of looking at commonly-used kernel paths provides a statistically significant ($\alpha = 0.01$, $\rho = 0.0015$) means for predicting where in the kernel exploitable flaws will be found in the future. For the remainder of the paper, we will focus on using this result to build more secure systems.

3.3 Risk Metrics for Identifying Kernel Flaws

Even though there is no widely accepted method for quantifying the safety (or risk) of privileged code, there have been a number of attempts to measure flaws in both software and operating systems. In this section, we briefly summarize a few of these approaches to provide information on past studies of potential flaws.

Other researchers have attempted to study the lifecycle of vulnerabilities when they are initiated and how long they last. Ozment and Schechter, who studied vulnerabilities in the code base of an OpenBSD operating system, determined that a significant extent (61%) of the reported vulnerabilities were "foundational," meaning they were introduced prior to the initial version studied. They also reported these vulnerabilities have a median lifetime of at least 2.6 years.

These past methods can provide some insight on where bugs may lie in kernel and when they may develop. However, most have relied on statistical methods, such as negative binomial regression model [**?**]. The metric we set out to develop was to be based on empirical study of the most critical bugs. This ad-hoc security metric can be used for quantitative measurement and evaluation of the kernel bugs, and other vulnerabilities at the level of lines of code by checking against known kernel bugs.

We believe our metric will provide better understanding of kernel security. As a result, a secure interface with an isolation mindset can be designed to be exposed to the userspace.

Other work has generated vulnerability signatures [**?**], which match all exploits of a given flaw. Most of this work is based on a mix of static and dynamic analysis, constraint solving and symbolic execution [**?**].

4 Design Options for Secure Virtualization Systems

Providing essential system functionality without exposing privileged code is one of the primary challenges. Currently, there are two basic approaches. One, called "System Call Interposition (SCI)," checks and passes

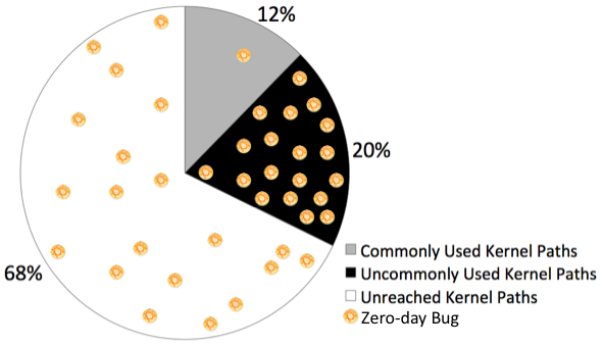

Figure 1: Percentage of different kernel areas that were reached during LTP and Trinity system call fuzzing experiments, with the zero-day kernel bugs identified in each area.

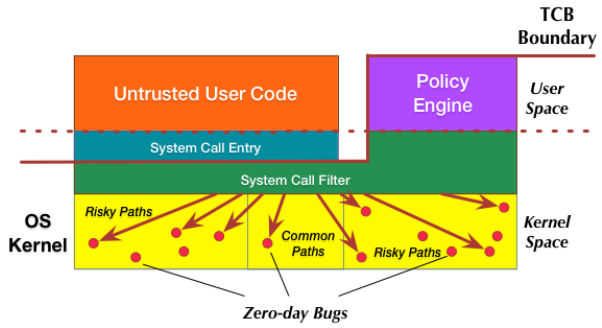

Figure 2: Schematic of System Call Interposition.

system calls through to the underlying kernel. The other, known as "functionality reimplementation," requires rebuilding system functionality with new code. In the following, we show that both methods are limited in their ability to prevent attacks in the kernel. With our metric in Section **??**, we propose a new design scheme named "Lock-in-Pop" that combines safer reimplementation within a secure environment, and accesses only popular code requests through a very small trusted computer base.

4.1 System Call Interposition (SCI)

SCI is the long-standing idea behind sandboxing systems like Janus [**?**, **?**]. It relies on the underlying kernel to provide system functionality. To prevent attackers from undermining the system, SCI uses a system call filter module to mediate requests from untrusted user code instead of allowing it to go directly to the kernel. The filter will check a predefined security policy to decide which system calls are allowed to pass to the underlying kernel, and which ones must be stopped. Figure **??** illustrates the general design of a system call interposition system.

[Lois: Yiwen, remember to address Yanyan's comment here about the placement of the TCB boundary in Figure 2 being above the uaer space]

System call interposition was once a popular approach to the design of secure virtualization systems because it gave developers the power to set and enforce security policies. [Lois: I think I asked this during the ;last revision. You say "was" a popular approach. Is it not a popular approach anymore?] However, this design is limited by its overly complicated approach to policy decisions and implementation. To make a policy decision, the system needs to obtain and interpret the OS state associated with the programs it is monitoring. [Lois: Can you clarify the above sentence? What do you mean by the OS state?] The complexity of OS states makes this process difficult and can lead to inaccurate policy decisions. Second, there are many indirect paths in the kernel that can be accessed. Overlooking those paths would render the system call interposition policy ineffective, because attackers will be able to bypass the imposed security checks.[Lois: Who or what is "overlooking those paths"] Moreover, many side-effects of blocking certain system calls could affect the function of desired system calls. It is difficult for developers to fully understand the side-effects of all the system calls in an interface as complex as the UNIX API. [Lois: can you either define or give an example of these "side-effects"] This makes it challenging to design and build a secure virtualization system using system call interposition alone.

4.2 Functionality Reimplementation

Systems such as Drawbridge [**?**], Bascule [**?**], and Graphene [**?**] can provide richer functionality and run complex programs than most systems built with SCI alone. These systems have their own system interfaces and libraries. Some virtualization systems, such as OS VM systems VirtualBox, and VMWare Workstation, even have the full functionality of an OS reimplemented in their codebase. We call such a design "functionality reimplementation."

The key idea of this design is to not fully rely on the underlying kernel for system functions. Instead, critical OS functions are re-written with new code. As illustrated in Figure **??**, this design scheme reimplements its own system functionality to provide to user code. [Lois: Please check the previous sentence. I re-wrote it because it was a bit awkward, but I may have changed the meaning] When it has to communicate with the kernel to access resources like memory, CPU, and disk storage, the system will do so with its underlying TCB code, which can access the kernel directly. For example, Graphene [**?**] reimplements its own Linux system calls in the `libLinux.so` module. When it needs to acquire

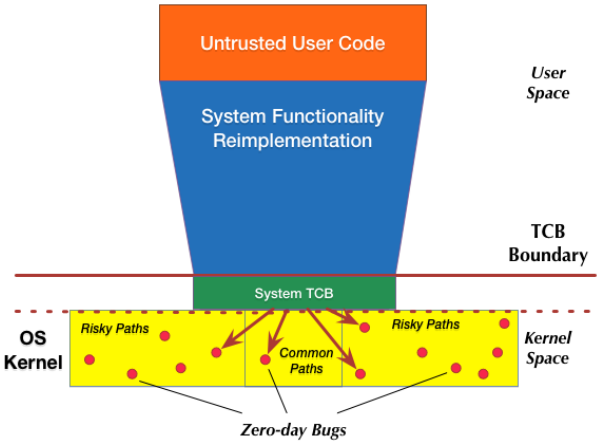

Figure 3: Schematic of Functionality reimplementation System.

resources from the kernel, it uses a Platform Adaptation Layer (PAL) module that has access to the kernel and provides basic ABI functions to the OS library.

Functionality reimplementation provides a more realistic solution to building virtualization systems than earlier efforts, and offers rich functionality. However, hundreds of vulnerabilities in existing virtualization systems have still been reported [**?**].[Lois: Do we have a time frame here? Over what period of time were those vulnerabilities reported] One shortcoming of such a system is the size of the implementated components. To provide rich functions, systems using this design have introduced larger codebases and increased the size of their TCBs. In addition, the complexity of OS functions can easily result in bugs and vulnerabilities in reimplementation [Lois: Complexity in what sense?]. Some vulnerabilities will directly cause a privilege escalation, which allows attackers to escape the sandbox and execute arbitrary code on the host OS. [Yanyan: give an example.]

Even operations that are considered legal and harmless in the guest OS may open up system call paths into the underlying host OS and cause a problem. The second type of problem[Lois: What does the "second type" refer to?] can be fatal, as it can reach and trigger vulnerabilities in the underlying OS kernel.

4.3 Lock-in-Pop: Staying on the Beaten Path

A common weakness of both of these previous approaches is the inevitable contact between the privileged code of the kernel and the untrusted application. By leveraging our key observation that "popular, frequently-used kernel paths contain fewer bugs" we propose a design in which all code *including the complex part of the operating system interface* should access only popular kernel paths through a small TCB. As it "locks" all functionality requests into only the "popular" paths, we thus dubbed the design "Lock-in-Pop."

In addition, the system is entirely in the userspace, and both the size of the sandbox kernel and its access to the OS kernel is restricted (Figure **??**). Any complex or possibly risky system functions that require contact with the OS kernel is reimplemented using memory-safe code and is contained within a sandbox. This approach has advantages over the others that require modifications to the OS kernel (Section **??**). The isolation provided by placing Lock-in-Pop in the userspace is also an added protection over functionality reimplemention. If a modified module is in the OS kernel and is compromised, it exposes kernel privileges that could allow attacks in the underlying system and any applications in the userspace.

As shown in Figure **??**, the Lock-in-Pop design has two components. The first is a computational module that enables the system to support and run legacy applications, and execute binary code compiled from unmodified source code on popular hardware architecture. This module's main responsibility is to provide an execution environment that can run unmodified source code. It also performs functions like type checking, object creation, and garbage collection. [Yanyan: I don't see the computational module in the figure.] The second is a library OS that serves requests to access the OS kernel. [Yanyan: why in the figure it's called "library OS sandbox"?] The system invokes the computation module to perform its operations, and, if there are riskier calls, the computation module directs those requests to the library OS.

In turn, the library OS responds to the system call requests and returns results to the user code, if the requests are granted. It is comprised of two parts: a sandbox kernel that provides access to basic but critical system calls, and a system API safe reimplementation that implements more complex calls, such as file system calls. [Yanyan: in the figure it's called "system functionality safe reimplementation".] The sandbox kernel forms the only TCB of our library OS, and is kept extremely small and simple so that it is easy to verify its security. The sandbox code provides an API that performs a few critical system calls with the most basic parameters, such as writing data to the file system and communicating with the network. The kernel utilizes the most simplistic calls possible with the most basic arguments. [Yanyan: this paragrah talks about sandbox kernel, but never metioned safe reimplementation.]

With the computation module and the library OS module, unmodified user code is able to run on top of our designed system. It is important to note that our design does not rely on any specific technique or tool, and it is possible to choose from several different techniques that

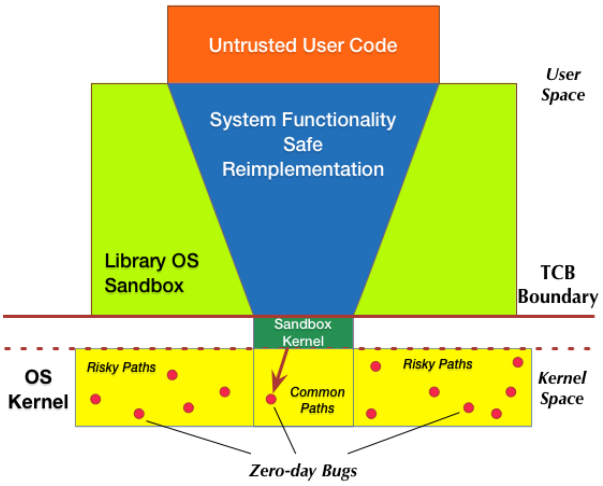

Figure 4: "Lock-in-Pop"System ensures safe execution of untrusted user code despite existing potential zero-day bugs in the OS kernel.

fit specific security requirements. The following section describes one implementation of our design called Lind.

5 Implementation of Lind

To test the feasibility of our Lock-in-Pop design in Section **??**, we used it to implement a secure virtual machine called Lind[1]. As in Section **??**, the Lock-in-Pop design utilizes both a computational module and a library OS module. In Lind, Native Client (NaCl) serves as the former, while Restricted Python (Repy), and a POSIX interface built into it, perform the task of the latter. Below we present a brief description of these components and how they were integrated into Lind, followed by a brief example of how the system works. [Lois: I am not sure how to format it, but insert a subhead here 5.1 Primary Components]

Native Client. We use NaCl to isolate the computation of the user application from the kernel. NaCl allows Lind to work on most types of legacy code. It compiles the programs to produce a binary with software fault isolation. This prevents the majority of the application from performing system calls or executing arbitrary instructions. To perform a system call, the application will call into a small privileged part of the NaCl TCB that forwards system calls, usually to the OS for processing. To build Lind, we changed the NaCl TCP to forward these calls to the library OS that we call SafePOSIX (details below) for processing.

[1]An old English word for a shield constructed from two layers of linden wood, providing strength in a lightweight form, the name is appropriate for a virtual machine that adapts two technologies–yet still protects vulnerable OS kernel code from exploitation by untrusted user programs.

Seattle's Repy. To build an API to access the safe parts of the underlying kernel, we need two things. First, we need a restricted sandbox that isolates computation and only allows access to commonly used kernel paths. We used Seattle's Repy [**?**] sandbox to perform this task. Second, we need to build a POSIX implementation to run within that sandbox. [Yanyan: why have to be POSIX?]

The Repy Sandbox Kernel. As the only piece of code in contact with the system call paths of the TCB, the sandbox kernel's security is of paramount concern. We used Seattle's Repy system API due to its tiny sandbox kernel (comprised of around 8K LOC), and its ability to provide straightforward access to the minimal set of the system call API needed to build general computational functionality. Repy allows access only to the safe portions of the OS kernel with 33 basic API functions, including 13 network functions, 6 file functions, 6 threading functions, and 8 miscellaneous functions [**?**, **?**]. The code is written using style guidelines designed to ease security auditing of the code [**?**]. Most of these functions are simple and regularly used system calls that access the commonly used kernel paths.

The Repy kernel code provides a solid foundation for our secure virtual machine. It has been audited by a professional penetration tester and, since 2010, there has also been a bug bounty program for security flaws in the sandbox. The code is deployed in daily use across thousands of devices, including on the Seattle testbed [**?**], and has been examined by hundreds of parties. To date no security flaws have been found in the sandbox kernel. Having a small, easily auditable piece of code thus helps to reduce the risk of such occurrence.

However, Lind's security does not rely solely on Repy's safety record. A reconfigured POSIX interface, and the isolation of the NaCl module within the dual sandbox design enhance the protection it can provide.

[Lois: I am not sure how to format it, but insert a subhead here "5.2 Enhanced Safety in Call Handling] The kernel interface is extremely rich and hard to protect. The dual sandbox Lock-in-Pop design used to build Lind provides enhanced safety protection through both isolation and a POSIX interface that reformulates risky system calls to provide sufficient API for legacy applications, with minimal impact on the kernel.

An example system call execution. In Lind, a system call issued from user code are received by NaCl, and then redirected to our system API module, which includes a POSIX API to serve those requests. A POSIX API is a set of standard operating system interfaces that provide necessary operating functionality. A standard POSIX API is large and complex enough to make it difficult to ensure its implementation is secure and bug-free. Lind takes advantage of the fact that Repy is a programming language sandbox to construct a variation on the

POSIX API. Following the Lock-in-Pop design, to service a system call in NaCl, a server routine in Lind marshals its arguments into a text string, and sends the call and the arguments to the Repy sandbox, where the "reimplemented" system call, marshals the result and returns it back to NaCl. Eventually, the result is returned as the appropriate native type to the calling program.

In the Lock-in-pop scheme, the file system API only need to provide functionality of writing data to storage. [Yanyan: why only discuss file system? how about network, threading, etc?] It eliminates the need of having a direct abstraction, the concept of file permissions, links, or even the concept of multiple files. [Lois: I moved this copy from Section 4. Please check that it makes sense where placed. Also, as Yanyan notes, there should be an actual example here.] The system API safe reimplementation is a set of more complicated system calls derived from functions in the sandbox kernel. We reimplement those system calls because we do not want this potentially risky user code to have direct access to the underlying OS kernel. Instead, our reimplementation layer serves as a mediator between the user code and the OS kernel. The reimplementation is safe because the reconstructed calls are isolated in a sandbox, and the code for the reimplementation is written in a memory-safe programming language.

Here is an example of how this reimplementation would work with the symbolic link function. If there is a bug in this function, rather than rely on the kernel code paths for symbolic links, Lind will implement the incorrect behavior. [Yanyan: what is the incorrect behavior?] This denies the code the privileged access to the system the OS kernel does. As a result, instead of creating a security issue, the application be denied access to the file system.

Our design could include more than two sandboxes, e.g., by sandboxing the sandbox kernel. However, in a secure system, the lowest level sandbox eventually must have some fundamental, even if limited, access to system resources, such as memory, and storage, threads. Even if we were to sandbox the sandbox kernel and have additional sandboxes, the one at the bottom level will still access the OS kernel in a similar way. Thus, having multiple sandboxes does not provide any extra security benefits.

6 Evaluation

In order to evaluate Lind's effectiveness in containing untrusted code and protecting OS kernel, we compared its performance against seven other existing virtualization systems–VirtualBox, VMWare Workstation,Docker, LXC, QEMU, KVM and Graphene. This section describes the setup of our experiments, and presents and discusses our experiment results. We chose these seven systems because they are representative of the most commonly-used design models in different markets. Among them, VirtualBox and VMWare Workstation are commercial OS VM products, while LXC and KVM are Linux kernel modules. Docker,QEMU, and Graphene are well-known open source projects. Lastly, we also compared against Native Linux, which serves as a baseline for comparison.

Our tests were designed to answer four fundamental questions:

How does Lind compare to other virtualization systems in protecting against zero-day Linux kernel bugs? (Section **??**)

How much of the underlying kernel code is exposed, and is thus vulnerable in different virtualization systems? (Section **??**)

If Lind's SafePOSIX reimplementation has bugs, how much of a threat does this pose? (Section **??**)

In the Lind prototype, what is the expected overhead in real-world applications?

6.1 Linux Kernel Bug Test and Evaluation

Claim To evaluate how well each virtualization system protects the Linux kernel against reported zero-day bugs.

Setup. We identified and examined a list of 69 historical bugs that have specifically targeted Linux kernel 3.14.1 [**?**]. By analyzing security kernel patches for those bugs, we identified the lines of code in the kernel that correspond to each one.

In order to test if a bug could be triggered, we created or located C code capable of exploiting each of the kernel bugs [**?**]. We were only able to trigger and obtain results for 35 out of the 69 bugs in our experiments, either because of a difficulty in clearly determining if triggering had occurred, or an inability, at this time, to find code to trigger them. We decided to focus our study on only these bugs and leave the other, more complex ones, to future work and analysis.

We compiled and ran the exploit C code under each virtualization system to obtain their kernel traces, and then used our kernel trace safety metric to determine if a specific bug was triggered.

Results. We found that a substantial number of bugs could be triggered in existing virtualization systems, as shown in Table **??**. A full 35 out of 35 (100%) bugs were triggered in Native Linux, while the other programs had lower rates: 14/35 (40%) in VirtualBox, 11/35 (31.4%) in VMWare Workstation, 8/35 (22.9%) in Docker, 12/35 (34.3%) in LXC, 5/35 (14.3%) in QEMU, 5/35 (14.3%) in KVM, and 8/35 (22.9%) bugs in Graphene. In comparison, only 1 out of 35 bugs (2.9%) was triggered in

Lind.

To better understand these results, we take a closer look at four vulnerabilities from Table **??**. These short case studies show how different system design philosophies can have different security impacts.[Lois: Per Yanyan's comments in the revision plan. A few short sentences to explain how you selected these categories is needed]

All systems vulnerable. Representative bug: CVE-2014-4171.[Lois: whatever you choose to change in this paragraph it MUST be broken up into at least two. The paragraph is way too long as is.] This is the only vulnerability in our test that was triggered in every system, including Lind. It resides inside the `mm/shmem.c` kernel path and can be triggered by using `mmap()` system call to access a hole in the memory. The `mmap()` call then invokes `shmem_fault()`, which will cause contention on `i_mmap_mutex`, and lead to a serious starvation of memory resource. The reason that Lind triggered this bug is because `mmap()` cannot easily be safely reimplemented inside our SafePOSIX API. The call sets up a memory region where the OS will later intervene and automatically convert all accesses into ones that reach the underlying file. The code does not explicitly make system calls, and as a result, with Lind's design we cannot intercept those accesses and call through the Repy sandbox kernel. Thus, Lind allows `mmap()` calls to directly access the kernel, which opens the chance to trigger this vulnerability. In the library OS system Graphene, similar to Lind, it does not reimplement the `mmap()` system call in its library module `libLinux`. Instead, when doing memory allocation operations like `malloc()` and `brk()`, Graphene will pass down those system calls to the underlying kernel, and rely on the kernel to perform the `mmap()` system function. KVM and LXC both reside inside the Linux kernel and inherit the `mmap()` system function from the kernel. QEMU is built on KVM, and relies on the KVM module to perform `mmap()` function. Docker is built on LXC, which relies on the LXC module to perform `mmap()` function. In VirtualBox and VMWare Workstation, the OS VM has its own mechanism to do memory management. During the initial setup, the OS VM needs to request memory resource by making `mmap()` to the host kernel. In addition, during the runtime of the VM process, when there is excessive memory usage, or reconfiguration of memory allocation, the VM kernel module will attempt to call `mmap()` to the host kernel. Thus, this vulnerability related with `mmap()` is exploitable under the virtualization systems that we tested.

Only Native Linux vulnerable. Representative bug: CVE-2014-5045. This vulnerability was only triggered inside Native Linux. It resides in the `fs/namei.c` kernel path and was triggered because the `mountpoint_last()` function does not properly maintain a reference count during attempts to use the `umount()` system call, in conjunction with a symbolic link. Unmounting from a symbolic link could block another unmount operation, and allow attackers to cause a denial of service or deploy use-after-free exploitation. Lind does not implement symbolic link, but even similar functionality that might be risky is implemented entirely within its SafePOSIX module. The bug would (at most) enable an attacker to execute code within the Repy sandbox and would not harm other parts of the system. Graphene does not implement symbolic link in its Linux system call API, and thus avoids this problem. OS VM like VirtualBox and VMWare Workstation have their own metadata to maintain their file directories and symbolic links, which do not directly rely on the host OS. Furthermore, symbolic links in those systems will be contained within the virtualization system's image, and will not be able to reach the underlying host OS. In this case, those virtualization systems provide enough isolation to prevent this bug. Docker/LXC and QEMU/KVM happen to avoid this bug because their kernel module modified the Linux kernel that fixed this problem. It should be noted though that by their design model, they both tend to suffer from the same vulnerabilities that exist inside the Native Linux kernel.

Some systems safe, some systems vulnerable. Representative bug: CVE-2014-8086. This vulnerability was not triggered inside Graphene and Lind, but was triggered inside VirtualBox, VMWare workstation, Docker, and Native Linux. It resides in the `fs/ext4/file.c` kernel path, and can be triggered by a file system write function call, made together with `fcntl` function call with argument `F_SETFL`, and `O_DIRECT` flag. If triggered, it could allow attackers to cause a denial of service (file unavailability). Both Lind and Graphene prevented this bug, but for different reasons. Lind implements `fcntl` in SafePOSIX, so the underlying kernel is not called. Graphene checks and blocks certain system calls, including a `fcntl` system call with the `O_DIRECT` flag. Other systems like VirtualBox, VMWare Workstation, Docker, and Native Linux, all suffer from this vulnerability because calls go directly into the host OS kernel.

Only Lind safe. Representative bug: CVE-2015-5706. As shown in our results, this vulnerability was triggered in every virtualization system we tested, except Lind. This vulnerability is closely related to file system calls and file flags, resides in the `fs/namei.c` kernel path, and can be triggered by making a `path_openat()` function call with file flag `O_TMPFILE`. `path_openat()` will jump to the wrong place after `do_tmpfile()`, and do `path_cleanup()` twice. This would allow local users to perform use-after-free exploitation to cause a denial of service. This bug was not triggered in Lind, because

Table 1: Linux Kernel Bugs, and Vulnerabilities in Different Virtualization Systems (✓: vulnerability triggered; ✓: vulnerability triggered with Guest Additions; ✗: vulnerability not triggered).

Vulnerability	Native Linux	VirtualBox	VMWare	Docker	LXC	QEMU	KVM	Graphene	Lind
CVE-2015-5706	✓	✓	✓	✓	✓	✓	✓	✓	✗
CVE-2015-0239	✓	✓	✓	✗	✓	✓	✓	✗	✗
CVE-2014-9584	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-9529	✓	✓	✗	✗	✓	✗	✗	✗	✗
CVE-2014-9322	✓	✓	✗	✓	✓	✓	✓	✓	✗
CVE-2014-9090	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-8989	✓	✓	✓	✓	✓	✓	✓	✓	✗
CVE-2014-8559	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-8369	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-8160	✓	✓	✓	✗	✓	✗	✗	✗	✗
CVE-2014-8134	✓	✓	✓	✗	✓	✗	✗	✓	✗
CVE-2014-8133	✓	✓	✗	✗	✗	✗	✗	✗	✗
CVE-2014-8086	✓	✓	✓	✓	✓	✗	✗	✗	✗
CVE-2014-7975	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-7970	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-7842	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-7826	✓	✓	✓	✗	✗	✗	✗	✓	✗
CVE-2014-7825	✓	✓	✓	✗	✗	✗	✗	✓	✗
CVE-2014-7283	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-5207	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-5206	✓	✗	✓	✓	✓	✗	✗	✗	✗
CVE-2014-5045	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-4943	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-4667	✓	✗	✗	✗	✗	✗	✗	✓	✗
CVE-2014-4508	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-4171	✓	✓	✓	✓	✓	✓	✓	✓	✓
CVE-2014-4157	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-4014	✓	✗	✓	✓	✓	✗	✗	✗	✗
CVE-2014-3940	✓	✓	✗	✓	✓	✗	✗	✗	✗
CVE-2014-3917	✓	✓	✗	✗	✗	✗	✗	✗	✗
CVE-2014-3153	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-3144	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-3122	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-2851	✓	✗	✗	✗	✗	✗	✗	✗	✗
CVE-2014-0206	✓	✗	✗	✗	✗	✗	✗	✗	✗
Vulnerabilities Triggered	**35/35 (100%)**	**14/35 (40.0%)**	**11/35 (31.4%)**	**8/35 (22.9%)**	**12/35 (34.3%)**	**5/35 (14.3%)**	**5/35 (14.3%)**	**8/35 (22.9%)**	**1/35 (2.9%)**

it does not support the use of the `O_TMPFILE` file flag. In fact, the only call in the Repy sandbox that opens a file does not take an argument for flags or other operations. The only arguments it takes are a filename (that must consist of a small number of highly restricted characters) and a flag to indicate whether a file should be created if one does not exist. Other virtualization systems allow more complex configuration of flags to pass through to the underlying OS kernel. In this case, the `O_TMPFILE` file flag was allowed in Native Linux, VirtualBox, VMWare Workstation, Docker, and Graphene.

Our initial results, as illustrated by the above case studies, suggest that bugs are usually triggered by complex system calls,or basic system calls with complicated or rarely used flags. The Lock-in-Pop design on which Lind is based, which provides strictly controlled access to the kernel, poses the least risk of triggering bugs. Our next step was to compare exactly how much kernel accessand therefore, potential risk, was possible in systems not using our design metric.

Table 2: Reachable kernel trace analysis for different virtualization systems.

Virtualization system	Kernel trace		
	Compared to native Linux	In common paths	In risky paths
VirtualBox	78.8 %	46.5 %	53.5 %
VMWare Workstation	72.6 %	50.2 %	49.8 %
Docker	61.3 %	58.4 %	41.6 %
LXC	65.6%	55.7%	44.3%
QEMU	70.1%	61.2 %	38.8%
KVM	75.4%	57.0%	43.0%
Graphene	49.2 %	65.1 %	34.9 %
Lind	36.2 %	100 %	0 %

6.2 Comparison of Kernel Code Exposure in Different Virtualization Systems

Claim Controlled kernel access, such as that found in Lind, reduces the risk of triggering zero-day bugs **Setup.** To analyze the reachable kernel paths for each virtualization system, we conducted system call fuzzing with Trinity (similar to our approach in Section **??**) to obtain the kernel trace in each system. All experiments were conducted under Linux kernel 3.14.1.

Results. We obtained the total reachable kernel trace for

each tested system, and further analyzed the components of those traces. These results are shown in Table **??**.

As shown in the table, Lind accessed the least amount of code in the OS kernel. More importantly, all the kernel code it accessed was in the "safe" portion, the commonly used kernel paths. A large portion of the kernel paths accessed by Lind lie in `fs/` to perform file system operations. In Lind, only basic function calls, like `open()`, `close()`, `read()`, `write()`, `mkdir()`, `rmdir()`, are allowed. In addition, only commonly used flags are allowed. For example, only `O_CREAT`, `O_EXCL`, `O_APPEND`, `O_TRUNC`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR` are permitted for `open()`. As a result, the reachable kernel trace we obtained with Lind is from the safe portion of the kernel, which contains fewer bugs as verified in Section **??**.

The other virtualization systems all accessed a substantial number of code paths in the kernel, and they all accessed a larger section of the risky portion. This is because they have more dependence on many complex system function calls, and allow extensive use of complicated flags. For example, Graphene provides a complex system call API that allows `fork()` and `signals`, which can access many risky lines of code. VirtualBox, VMWare Workstation, and Docker have even larger code base and more complicated system functions. They allow rarely used flags, such as `O_TMPFILE`, `O_NONBLOCK`, and `O_DSYNC`, which can reach potentially dangerous lines of code. Based on our hypothesis, many historical bugs, as well as undetected zero-day bugs, could be located in the uncommonly used kernel paths. Thus, accessing the risky portion without restriction is dangerous, and leads to potential kernel bug exploitation. The results in Table **??** verify our hypothesis.

To summarize, our analysis signals that Lind triggers the fewest kernel bugs because it has better control over the access to the OS kernel.

6.3 Impact of Potential Vulnerabilities in Lind's SafePOSIX reimplementation

Claim To understand what potential risks could be posed if Lind's TCB has vulnerabilities. **Setup.** Since we needed to see what portion of the kernel could be reached by leveraging the Repy sandbox kernel, we conducted system call fuzzing with Trinity to obtain the reachable kernel trace in Linux kernel 3.14.1.

Results. An important question about Lind's security properties is what would happen if there is a bug or a failure in Lind's TCB, the Repy sandbox kernel?

The results are shown in Table **??**. The trace of Repy is slightly larger (5.8%) than that of Lind. Repy's design can not allow attackers or bugs to have more access to the risky paths in the OS kernel, and only a small amount (5.8%) of additional common paths in the OS kernel might be open. Those new kernel paths are added because some functions in Repy have more capabilities for message sending and network connecting than the system call interfaces provided by Lind. For example, in Repy, `sendmessage()` and `openconnection()` functions could reach out to more lines of code when fuzzed. However, the kernel trace of Repy still lies completely within the safe portion of the OS kernel. Since the safe portion contains fewer kernel bugs, the Repy sandbox kernel will have a very slim chance to trigger OS kernel bugs.

Table 3: Comparison of reachable kernel traces within Repy sandbox dernel and Native Linux.

Sandbox	Kernel trace			
	Compared to Lind	Compared to native Linux	In common paths	In risky paths
Repy	105.8 %	38.3 %	100 %	0 %

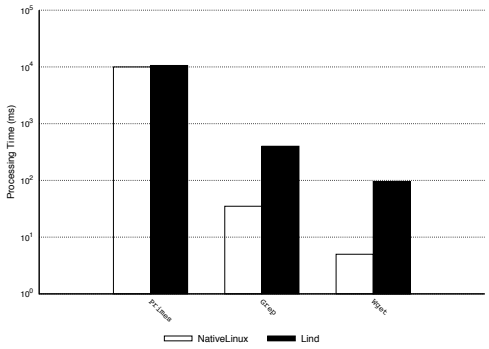

Figure 5: Overhead from applications run in Native Linux vs. Lind

The results explained above shows that even if our Repy sandbox kernel has a bug or failure inside, it only slightly increases the amount of OS kernel paths open to attacks, and all these paths accessed are still inside the safe portion. Therefore, Repy will not grant attackers more opportunities to trigger OS kernel bugs. Since Repy, arguably the main security weakness of the system, can be considered safe through our analysis, it shows that Lind has the potential to provide strong security to run user applications.

6.4 Performance Evaluation

We evaluated Lind to see how its performance compared to other systems in a real-world application. Note that we did not optimize the performance of Lind in any way before running these tests.

Setup. To test the overhead in Lind for running real-world applications, we first compiled and ran three

widely used applications: a prime number calculator, GNU `grep`, and GNU `wget`. All ran unaltered and correctly inside Lind. The source code of each of the applications remained unmodified. To run the applications, it was sufficient to just recompile the source code using NaCl's compiler and Lind's `glibc` to call into SafePOSIX.

Next, we ran the Tor 0.2.3 in Lind. Tor simply needs to be recompiled to run in Lind. We used the benchmarks that come with Tor to test its common operations.

Results. Figure **??** shows the runtime performance results for running the primes calculator, GNU `grep`, and GNU `wget`. The primes application run in Lind has a 6% performance overhead compared to Native Linux. CPU bound applications, like the primes, engender little overhead, because they run only inside the NaCl computation sandbox. No system calls are required, and there is no need to go through the safe POSIX interface. The small amount of overhead is generated by NaCl's instruction alignment at building time. Another reason for the overhead is that the instructions built by NaCl have a higher rate of cache misses, which can slowdown the program. We expect other CPU bound processes to behave similarly. `grep` experienced roughly 11x slowdown over Native Linux, while `wget` slowdown was roughly 19x. Since they are both I/O heavy applications, each repeatedly calls into the SafePOSIX code which then reimplements the call. The additional computation of SafePOSIX produced the additional overhead. We have not conducted any performance optimization for Lind. As we focus on the security aspect first, we leave that to future work.

A summary of the results for Tor is shown in Table **??**. The benchmarks focus on cryptographic operations, which are CPU intensive, but also make system calls like `getpid`, and reads to `/dev/urandom`. The digest operations time the access of a map of message digests. The AES operations time AES encryptions of several sizes and message digest creation. Cell processing executes full packet encryption and decryption. In our test, Lind slowed down these operations by 2.5x to 5x. We believe these slowdowns are due to the increased code size produced by NaCl, and the increased overhead from Lind's safe POSIX system call interface.

As shown above, Lind generally incurs some performance overhead. It should be noted that, we have not yet attempted to optimize its performance. However, since an attack on the kernel can have devastating consequences, a tradeoff between security and performance could be justified. The fact that Lind is able to run legacy applications suggests that it is a positive step towards building new secure systems.

Table 4: Performance results on Tor's built-in benchmark program: Native Linux vs. Lind.

Benchmark	Native Code	Lind	Impact
Digest Tests:			
Set	54.80 nsec/element	176.86 nsec/element	3.22x
Get	42.30 nsec/element	134.38 nsec/element	3.17x
Add	11.69 nsec/element	53.91 nsec/element	4.61x
IsIn	8.24 nsec/element	39.82 nsec/element	4.83x
AES Tests:			
1 Byte	14.83 nsec/B	36.93 nsec/B	2.49x
16 Byte	7.45 nsec/B	16.95 nsec/B	2.28x
1024 Byte	6.91 nsec/B	15.42 nsec/B	2.23x
4096 Byte	6.96 nsec/B	15.35 nsec/B	2.21x
8192 Byte	6.94 nsec/B	15.47 nsec/B	2.23x
Cell Sized	6.81 nsec/B	14.71 nsec/B	2.16x
Cell Processing:			
Inbound	3378.18 nsec/cell	8418.03 nsec/cell	2.49x
(per Byte)	6.64 nsec/B	16.54 nsec/B	-
Outbound	3384.01 nsec/cell	8127.42 nsec/cell	2.40x
(per Byte)	6.65 nsec/B	15.97 nsec/B	-

7 Discussion

One of our challenges in conducting this study was deciding where to place the limits of its scope. The literature documents a variety of strategies that, over the years, have been used to secure the OS kernel. To explore any one strategy in depth, we felt it was necessary to intentionally exclude consideration of other valid approaches. In this section, we acknowledge some of the alternative approaches that we chose not to investigate, as well as a few issues that we plan to address in the future.

Exclusions. In designing our study, we intentionally decided that we would not include any discussion of native or bare-metal hypervisors that run directly on server hardware. Though these devices, also called Type I hypervisors, have been used for decades, we chose to exclude them because our initial emphasis in designing Lind was to come up with something small and light that did not require any alterations to the kernel. As we continue to develop our Lind prototype, we may choose to study current examples of these native hypervisors to either directly compare security and performance attributes, or to see if our design metric could be adapted to these machines.

A second exclusion stems from our criteria for locating bugs. At the beginning of our study, we identified a set of common but seriously dangerous zero day bugs and then went looking for them within our obtained kernel traces. By looking only for a specific subset of bugs, we acknowledged that we might be limiting our ability to find a broader spectrum of kernel vulnerabilities. For example, bugs caused by a race condition, or that involve defects in the internal kernel data structures, or that require complex triggering conditions across multiple kernel paths, may not be immediately found using our metric. As we continue to refine our metric, we will look to also evolve our evaluation criteria to allow us to find and protect against more complex brands of bugs. In

the meantime, we feel that avoiding the triggering of this initial set of bugs through the use of our "Lock-in-Pop" design scheme can still address the security needs for a significant segment of users.

Future work. While our experiments were limited to the Linux kernel 3.14.1, our future work will include testing its applicability to other operating systems, such as Windows and Mac OS. Since the Lock-in-Pop design scheme is not dependent on the use of any specific hardware, we believe it has the flexibility to be adapted to these other widely-used systems.

The other challenge facing wide-scale adoption of Lind is improving its performance in terms of bandwidth and other overhead factors. As addressed in Section 6.4, Lind does incur some performance overhead. Since our initial tests were focused on security issues, we did not take any steps to optimize its performance before running these tests. Future work will focus on identifying the factors that contribute to this overhead, and the best ways to make Lock-in-Pop a cost-effective alternative to other virtual machine designs.

Due to the nature of bugs, our proposed metric cannot assure whether a portion of code is an absolute safe or risky area in kernel. For example, bugs that are caused by a race condition cannot be identified by directly checking if certain lines of code have been executed. For complicated bugs that involve defects in the internal kernel data structures, or require complex triggering conditions across multiple kernel paths, our metric will not be accurate. In these cases, more complex metrics might be needed.

Lind has not been optimized for performance. We would like to explore what existing OS VM optimizations can be safely applied and their impact.

We would like to test our metric in other operating systems, such as Windows and Mac OS. Our experiments were limited to Linux kernel 3.14.1 and some of the typical virtualization systems that existed in Linux. It would be interesting if similar tests could be run in other widely-used operating systems.

8 Related Work

A substantial amount of effort has gone into building secure systems, including many that This section summarizes a number of previous security approaches that aim to ensure the safety of privileged code in user space and kernel space. These approaches include designing safety metrics to quantify the risk in an OS kernel, various virtualization techniques, etc.

Safety metrics for kernel code risk. Early research has suggested a number of approaches to quantifying risk in the OS kernel. One such approach are studies that identify a correlation between certain kernel directories or modules and the presence of kernel faults. For example, Palix found that the `drivers` directory in the Linux kernel contains the most faults, in addition to HAL and `fs` [**?**]. Similarly, Chou [**?**] showed that device drivers have much higher error rates than other parts of the kernel, and are thus more vulnerable. However, level of information is [Yanyan: xxx (less accurate or something) compared to Lind.]

Alhazmi [**?**] used metrics like defect density and fault exposure ratio, etc., to describe the vulnerability discovery process. In [**?**], Zimmermann descibed metrics like code complexity, code churn (the total added/modified/deleted lines of code) to predict vulnerabilities. Kim in [**?**] analyzed the correlation between shared code size and shared vulnerabilities in successive versions of a software system. While these metrics can be useful for determining readiness for release, and evaluating the risk of vulnerability exploitation, they do not provide insights for secure system design.

Prior work has also examined the evolution of kernel defects. For example, Ozment [**?**] and Li [**?**] have both studied how the numbers of security issues evolved over time. Ozment reported a decrease in the rate at which new vulnerabilities are reported by studying the age of vulnerabilities in OpenBSD. Li reported an increase in security bugs by studying the number and relative percentage of bugs. However, neither of these two approaches allowed an investigation on how vulnerabilities are distributed. [Yanyan: In contrast, Lind...(and related to the new results).]

Much work has been done in analyzing system faults using techniques in software engineering. For example, in [**?**] Engler analyzed system code by static analysis to look for contradictions, such as acquired locks that are not released. Coverage-based fault localization assumes that execution events that occur mostly in failing test cases, but rarely in passing test cases, are more *suspect* of being the fault [**?**]. Further research lead to measuring the set difference of the statements covered by passing and failing test scenarios with run-time events [**?**, **?**, **?**]. Ostrand in [**?**] predicted the number of faults in a system based on fault and modification history from previous releases. [Yanyan: While safety may be the eventual end goal, the authors of these SE papers are more interested in understanding the code than using it as a security tool. I wonder if this paragraph should be cut.]

Virtualization. *Language-based virtualization.* Programming languages can provide safety through virtualization like Java, JavaScript, Lua [**?**], and Silverlight [**?**]. Though many sandboxes implement the bulk of standard libraries in memory-safe languages like Java or C#, flaws in this code can still pose a threat [**?**, **?**]. Any bug or failure in a programming language virtual machine is usually fatal. In contrast, Lind with a very small TCB en-

hances security of privileged code due to fewer LOC, which in turn reduces the number of potential bugs or weaknesses. [Lois: OK, what does the TCB have to do with these languages?]

OS virtualization techniques include bare-metal hardware virtualization, such as VMware ESX Server, Xen, LXC [**?**], BSD's jail, Solaris zones, and Hyper-V, and hosted hypervisor virtualization such as VMware Workstation, VMware Server, VirtualPC and VirtualBox. Security by isolation [**?**, **?**, **?**, **?**] provides safe executing environments through containment for multiple user-level virtual environments sharing the same hardware. However, there are limitations of this approach due to the large attack vectors against the hypervisors, including vulnerabilities of software and configuration risk. Lind avoids such restrictions due to a smaller TCB.

Library OSes allow applications to efficiently obtain the benefits of virtual machines. Drawbridge [**?**] is a library OS that uses picoprocesses (lightweight containers), a security monitor (to enforce rules), and a library OS to present a Windows persona for Windows applications. Similar to Lind, it restricts access from user-mode to host OS through a number of operations that pass through the security monitor. Bascule [**?**], an architecture for library OS extensions based on Drawbridge, allows application behavior to be customized by extensions loaded at runtime. Graphene [**?**] is a library OS system that executes both single and multi-process applications with low performance overheads. Haven [**?**] uses a library OS to implement shielded execution of unmodified server applications in an untrusted cloud host.

While the library OS technique relies extensively on the underlying kernels to perform system functions, Lind reimplements complex OS functions through memory-safe Repy code, relying on only a limited set of system functions from the Repy sandbox. Unlike Lind, Drawbridge, Bascule or Haven do not have the sandbox environment that properly contains buggy or malicious behavior, and therefore could allow attackers to trigger zero-day vulnerabilities in the underlying kernel.

System Call Interposition (SCI) provides secure execution of applications by exposing a minimal kernel surface. This approach usually ensures the userspace policies through a monitor process that interacts with the system call execution engine [**?**]. Issues in SCI include the difficulty of appropriately replicating OS semantics, multi-threaded applications that cause race conditions, and indirect paths to resources that are often overlooked. In addition, system calls that are denied can cause inconsistencies [**?**]. SCI is similar to the Lind isolation mechanism. However, a key difference is the actual execution of a system call through the Lind's reimplementation to ensure safe POSIX API for the userspace applications.

Software Fault Isolation (SFI) provides sandboxing in which native instructions can only be executed if they do not violate the sandbox's constraints [**?**], and security policies are enforced through machine-level code analysis. In SFI, memory writes are protected and code jumps cannot access predefined memory of other programs to execute their code. Native Client (NaCl) [**?**] is an SFI system for the Chrome browser that allows native executable code to run directly in a browser. NaCl prevents suspicious code from memory corruption or direct access to the underlying system resources.

9 Conclusion

In this paper, we proposed a new metric based on quantitative measures derived from the execution of kernel code when running user applications. We verified the hypothesis that commonly used kernel paths contain fewer bugs. Our metric was used to implement a new virtualized security system called Lind. Designed with a minimized TCB and interacting with the kernel in only commonly used paths, Lind addresses the need to support risky system calls by securely reconstructing complex, yet essential OS functionality inside a sandbox. Evaluation results have shown that Lind is the least likely to trigger zero-day Linux kernel bugs, when compared to seven other virtualization systems, such as VirtualBox, VMWare Workstation, Docker, LXC, QEMU, KVM and Graphene.

All of the data and source code for this paper is available at the Lind website [**?**].