Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path

Abstract

Virtual machines (VMs) are widely used in practice, in part for their ability to isolate potentially untrusted code from the rest of a system. Recently, library OSes and containers have also presented promising security options. However, it is often possible to trigger zero-day flaws in the host Operating System (OS) from inside of such virtualized systems. In this paper, we offer a new insight about where security bugs lie. By observing that the OS kernel paths accessed by popular applications in everyday use contain fewer security bugs than less-used paths, we devise a design that allows applications to run more securely in VMs on top of a vulnerable host OS. Furthermore, We leverage this observation to devise the *Lock-in-Pop* design, which ***locks*** an application, and the POSIX implementation that services it, into accessing only the well-used ***popular*** portion of the kernel. Using the *Lock-in-Pop* model, we implement a library OS virtual machine called Lind. We compare Lind and three other virtualized systems that were available at the release of Linux kernel version 3.14.1, and evaluate their effectiveness in containing the zero-day kernel bugs that have been discovered since then. Our results show that Lind can prevent the triggering of zero-day kernel bugs significantly better than an existing library OS (Graphene) and containers such as Docker and LXC.

1 Introduction

The number of attacks in which zero-day vulnerabilities have been exploited has more than doubled from 2014 to 2015 [52]. Skilled hackers can find a security flaw in a system and use it to hold the system's users hostage , e.g., by gaining root access and compromising the host [24]. Similarly, zero-day vulnerabilities can be exploited [17] or held back [29] by government agencies, thus rendering millions of devices vulnerable.

In theory, running a program in an operating-system-level virtual machine (OSVM) like Docker or LXC should prevent it from triggering bugs in the host OS kernel. However, to be effective, the isolation provided by such systems must meet two challenging criteria. First, the OSVM's software must not contain any bugs that could allow the program to escape the machine's containment and interact directly with the host OS. Unfortunately, these issues are very common in OSVMs. Virtualbox reports more than 40 such vulnerabilities [14] and more than 100 bugs have been found in VMware Workstation [13]. Given the large amount of complex code needed for such a system, it is understandable that flaws could occur, leaving tens of millions of user machines at risk [24].

Secondly, isolation will not work if a malicious program can access a portion of the host OS's kernel that contains a zero-day flaw. This may occur even if the containment of the OSVM is working as designed. Many system calls made within an OSVM eventually result in calls in the host OS (e.g., network I/O from the guest OSVM results in network I/O by the host OS kernel). If one of these paths in the OS kernel contains a zero-day security bug, the attacker may be able to trigger and exploit it.

With this paper, we move one step closer to designing secure OSVM systems that are resilient to zero-day flaws. We start with the proposition that kernel code found in popular paths, associated with frequently-used programs, has less potential risk of bugs than code in less-used parts of the kernel. Our intuition behind this proposition is that bugs in the popular paths are more frequently found in software testing, because of the numerous times they are executed by diverse pieces of software. We performed a quantitative analysis of resilience to flaws in two versions of the Linux kernel (version 3.13.0 and version 3.14.1), and found that only about 3% of the bugs were present in popular code paths, despite these paths accounting for about one third of the total reachable kernel code. Therefore, OSVMs that only use

these kernel paths will greatly increase resilience to zero-day bugs in the host OS kernel.

Guided by this knowledge, we propose a new design scheme for a secure virtual machine that accesses only popular code lines through a very small trusted computing base. We name it *Lock-in-Pop*, as it locks kernel access to only code associated with popular programs. However, there are many system calls outside of the popular paths that need to be supported to run applications [43]. Recognizing this, *Lock-in-Pop* re-creates needed-but-risky OSVM functionality (e.g., POSIX) in a sandbox that only allows access to popular kernel paths. In this way a bug inside of the OSVM implementation does not result in a system compromise so long as the minimal sandbox remains uncompromised.

To demonstrate the viability of *Lock-in-Pop*, we use it to implement a prototype library OS virtual machine that can offer enhanced security without sacrificing basic functionality. Dubbed Lind, it pairs two components – Google's Native Client (NaCl) and Seattle's Repy. NaCl serves as a computational module that isolates binaries, providing memory safety for legacy programs running in our OSVM. It also passes system calls invoked by the program to the operating system interface, called SafePOSIX. SafePOSIX re-creates the broader POSIX functionalities needed by applications, while being contained within the Repy sandbox. An API in the sandbox only allows access to popular kernel paths, while the small (8K LOC) sandbox kernel of Repy isolates flaws in SafePOSIX to prevent them from allowing direct access to the host OS kernel.

To test Lind's effectiveness, we replicated 35 kernel bugs that had been discovered in Linux kernel version 3.14.1. We attempted to trigger those bugs in three other virtualized environments, including Docker, LXC, and Graphene. Our results show that applications in Lind were substantially less likely to trigger kernel bugs.

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

- We propose a quantitative metric that evaluates security at the line-of-code level, and test its effectiveness against other proposed metrics, such as the age of code [31], or the increased risk in driver code [10]. We find that choosing popular kernel paths is a much more effective metric for identifying lines of code that are unlikely to contain flaws.
- Based on the metric, we postulate a new approach for securing privileged code, and develop a new design scheme called *Lock-in-Pop*. It accesses only popular code paths through a very small trusted computer base. The need for complex functionality is addressed by re-creating riskier system calls in a memory-safe programming language within a secure sandbox.
- We build a prototype library OS virtual machine, Lind, using the *Lock-in-Pop* design, and test its effectiveness against three other security systems. We find that Lind exposes 8-12x fewer risky (unpopular) kernel code paths containing bugs than prior production and academic systems.

The remainder of this paper is organized as follows. Section 2 presents the scope of our study and precisely describes our threat model. Earlier kernel protection metrics and how they performed against our newly proposed metric are discussed in Section 3. We discuss handling bugs in the OSVM software in Section 4, while focusing on design strategies for preventing exploitation of zero-day kernel bugs in the host OS. Section 4 also describes our *Lock-in-Pop* design scheme. In Section 5 we discuss the construction of the Lind prototype, while Section 6 provides a quantitative security analysis of Lind compared to other virtualization systems. Section 7 outlines limitations of our study. Finally, Section 8 reviews existing work relevant to Lind's security goals, while the cogent points relayed in the paper are reviewed in Section 9.

2 Goals and Threat Model

In this section, we define the scope of our efforts. We also briefly note why this study does not evaluate a few existing design schemes.

Goals. Our goal is to design and build a secure virtualization system that allows untrusted programs to run on an unpatched and vulnerable host OS (Linux OS in this study), without triggering vulnerabilities that attackers could exploit. Developing effective defenses for the host OS kernel is essential as kernel code can expose privileged access to attackers that could lead to a system take-over.

To combat the threat from zero-day vulnerabilities, untrusted programs are often executed in a secure virtualization system, such as a guest OSVM, a system call interposition module, or a library OS system. Our intent is to build such a system capable of protecting a vulnerable underlying host OS, while running untrusted user programs.

Threat model. When an attack attempt is staged on a host OS in a virtualization system, the exploit can be done either directly or indirectly. In a direct exploit, the attacker accesses a vulnerable portion of the host OS's kernel using a crafted attack code. In an indirect exploit, the attacker first takes advantage of a vulnerability in the virtualization system itself (for example, a buffer-over-flow vulnerability) to escape the VM's containment. Once past the containment, the attacker would be able to

run arbitrary code in the host OS. The secure virtualization system design we propose in Section 4 can prevent both types of attacks effectively.

Based on the goals mentioned above, we make the following assumptions about the potential threats our system could face:

- The attacker possesses knowledge of one or more unpatched vulnerability in the host OS.
- The attacker can 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 considered successful in compromising the system.

Exclusion. It should be noted that our study intentionally excludes a comparison with solutions that do not run on top of a full-fledged privileged OS, such as a bare-metal hypervisor [4, 46] or hardware-based virtualization [3, 22]. While our techniques can potentially apply to those systems, a direct comparison is not possible since they have different ways of accessing hardware resources, and require different measuring approaches.

In addition, we exclude evaluation and direct comparison with full virtualization virtual machines, such as VirtualBox [45], VMWare Workstation [47], and QEMU [36]. Such systems simulate hardware to allow an unmodified guest OS to run. The goal of our design is to substitute the large and complex TCB required for a guest OS, with a single-process program with a small TCB and a secure isolated environment. With different goals, our proposed design is a fundamentally different approach from full virtualization. As a result, direct measurement and comparison between full virtualization and our design is beyond the scope of this work.

To build a VM to resist zero-day vulnerabilities, we need to know which portions of the kernel may be more prone to exploitation. Our first step is to define and test a security metric that can quantitatively measure how bugs and vulnerabilities are distributed in the host OS kernel.

3 Developing a Quantitative Metric for Evaluating Kernel Security

If we knew which lines of code in the kernel were likely to contain zero-day bugs, we could try to avoid using them in an OSVM. In this section, we formulate and test a quantitative evaluation metric that can indicate which lines of code are likely to contain bugs. This metric is based on the idea that kernel paths executed by popular applications during everyday use are less likely to contain security flaws. The rationale is that these code paths are well-tested due to their constant use, and thus fewer bugs can go undetected. Our initial tests yielded promising results. Additionally, when tested against two earlier strategies for predicting bug locations in the OS kernel, our metric compared favorably.

3.1 Experimental Setup

We used two different versions of the Linux kernel in our study. Since our findings for these versions are quantitatively and qualitatively similar, we report the results for 3.13.0 in this section and use 3.14.1 in Section 6. To trace the kernel, we used `gcov` [19], a standard program profiling tool in the GCC suite. The tool indicates which lines of kernel code are executed when an application runs.

Popular kernel paths. To capture the popular 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 for Debian 7.0 , a widely-used and popular open source project, that Popularity Contest [1] had deemed the 50 most popular Debian packages (omitting libraries, which get included automatically by packages that depend on them). Each student used 25 applications for their tasks (i.e., writing, spell checking, printing in a text editor, or using an image processing program). These tests were completed over 20 hours of total use over 5 calendar days.

The second strategy was to capture the total range of applications an individual computer user might regularly access. 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, and so on, using this system. Software was installed as needed. From these two strategies, we obtained a profile of the lines of kernel code that defined our popular kernel paths. We make this trace publicly available to other researchers [redacted], so they may analyze or replicate our results.

Reachable kernel paths. There are certain paths in the kernel, such as unloaded drivers, that are unreachable and unused. To understand which paths are unreachable, we used two techniques. First, we performed system call fuzzing with the Trinity system call fuzz tester [41]. Second, we used the Linux Test Project (LTP) [25], a test suite written with detailed kernel knowledge.

Locating bugs. Having identified the kernel paths used in popular applications, we then investigated how bugs are distributed among these paths. We collected a list of severe kernel bugs from the National Vulnerability Database [30]. For each bug, we found the patch that fixed the problem and identified which lines of kernel code were modified to remove it. For the purpose of this study, a user program that can execute a line of kernel

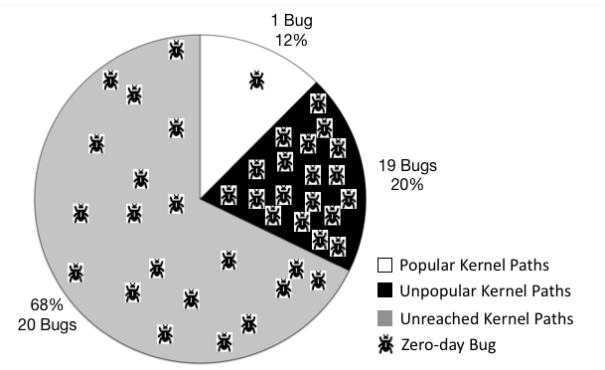

Figure 1: Percentage of different kernel areas that were reached during LTP and Trinity system call fuzzing experiments, with the zero-day kernel bugs identified in each area.

code changed by such a patch is considered to have the *potential to exploit that flaw*. Note that it is possible that, in some situations, this will over-estimate the exploitation potential because reaching the lines of kernel code where a bug exists does not necessarily imply a reliable, repeatable capability to exploit the bug.

3.2 Results and Analysis

Bug distribution. The experimental results from Section 3.1 show that only one of the 40 kernel bugs tested for was found among the popular paths, even though these paths make up 12.4% of the kernel (Figure 1).

To test the significance of these results, we performed a power analysis. We assume that kernel bugs appear at an average rate proportional to the number of lines of kernel code. Therefore, consistent with prior research [28], the rate of defect occurrence per LOC follows a Poisson distribution [34]. The premise we tested is that bugs occur at different rates in different parts of the kernel, i.e., the less popular kernel portion has more bugs.

We first divided the kernel into two sets, A and B, where bugs occur at rates λ_A and λ_B, and $\lambda_A \neq \lambda_B$. In this test, A represents the popular paths in the kernel, while B addresses the less commonly-used paths. 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 [38] 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 that $\lambda_A/\lambda_B \in [0.002, 0.525]$. This indicates that the ratio between the bug rates is well below 1. Since B has a bug rate much larger than that of A, this result shows that popular paths have a much lower bug rate than unpopular ones.

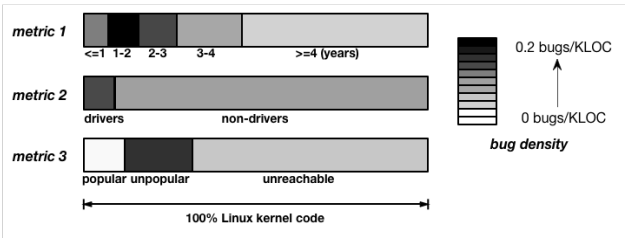

Figure 2: Bug density comparison among three metrics.

Comparison with other security metrics. Ozment, et al. [31] demonstrated that code that had been around longer in the Berkeley Software Distribution (BSD) [7] kernel tended to have fewer bugs (metric 1). To test Ozment's metric using our Linux bug dataset, we separated the code into five different age groups. Our results (Figure 2) showed a substantial number of bugs located in each group, and not just in the newer code. Therefore, buggy code in the Linux kernel cannot be identified simply by this age-based metric. In addition, this metric would seem to have limited use for designing a secure virtualization system, as no system could run very long exclusively on old code.

Another metric, reported by Chou, et al. [10], showed that certain parts of the kernel, device drivers in particular, were more vulnerable than others (metric 2). Applying this metric on our dataset, we found that the driver code in our version of the Linux kernel accounted for only 8.9% of the total codebase, and contained merely 4 out of the 40 bugs (Figure 2). One reason for this is that, after Chou's study was published, system designers focused efforts on improving driver code. For example, Palix [32] found that `drivers` now have lower fault rate than other directories, such as `arch` and `fs`.

Additionally, there are other security metrics that operate at a coarser granularity, e.g., the file level. However, when our kernel tests were run at a file granularity, we found that even popular programs used parts of 32 files that contained flaws. Yet, only one bug was triggered by those programs. In addition, common programs tested at this level also executed 36 functions that were later patched to fix security flaws, indicating the need to localize bugs at a finer granularity.

To summarize, we demonstrated that previously proposed security metrics have weak correlation between the occurrence of bugs and areas of code they identify. In contrast, our metric (metric 3) provides an effective and statistically significant means for predicting where in the kernel exploitable flaws will likely be found. For the remainder of the paper, we will focus on using this result to design and build secure virtualization systems.

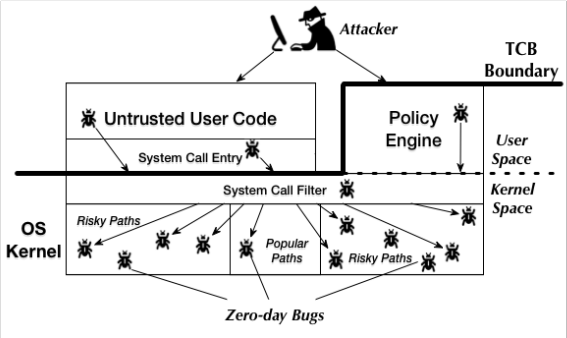

Figure 3: Schematic of how System Call Interposition functions.

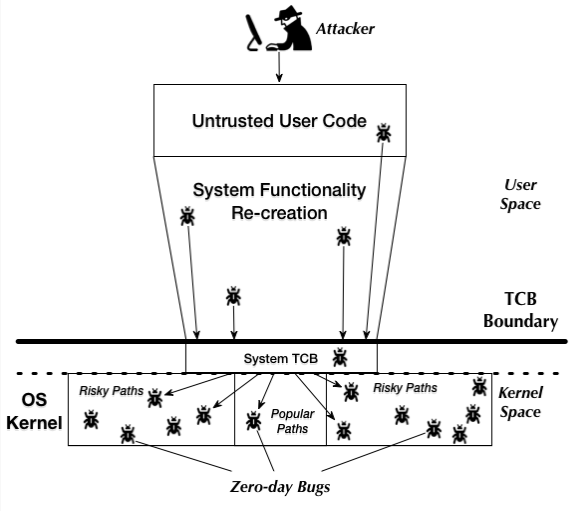

Figure 4: Schematic of a Functionality Re-creation System.

4 Design Options for Secure Virtualization Systems

Providing essential system functionality without exposing privileged code is a critical challenge in the design of secure systems. Currently, there are two basic approaches. One, known as system call interposition (SCI), checks and passes system calls through to the underlying kernel. The other, which we call "functionality re-creation," requires rebuilding system functionality with new code. In this section, we show that both methods are limited in their ability to prevent attacks in the kernel. Using our metric described in Section 3, we then propose a new design scheme named *Lock-in-Pop*, which accesses only popular code paths through a very small trusted computer base, and utilizes functionality re-creation within a secure environment for complex implementations.

4.1 System Call Interposition (SCI)

SCI systems [20, 48] filter system calls to mediate requests from untrusted user code instead of allowing it to go directly to the kernel. The filter checks a predefined security policy to decide which system calls are allowed to pass to the underlying kernel, and which ones must be stopped. Figure 3 illustrates how system call interposition works. System administrators have direct access to a policy engine that sets and changes security policies.

SCI was once popular because it gave developers the ability to set and enforce security policies. 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 (e.g., permissions, user groups, register flags) associated with the programs it is monitoring. The complexity of OS states makes this process difficult and can lead to inaccurate policy decisions. In addition, there are many indirect paths in the kernel that can be accessed. If security policy makers overlook those paths, it renders the policy ineffective, as attackers will be able to bypass security checks. Moreover, blocking certain system calls could affect necessary functionality. 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. For example, many applications that rely on `setuid` fail to check its return value. If `setuid` fails, these applications will continue to function in a compromised state, with incorrect permissions and privileges. The above limitations make it very challenging to design and build a secure virtualization system using system call interposition alone.

4.2 Functionality Re-Creation

Systems such as Drawbridge [35], Bascule [5], and Graphene [42] can provide richer functionality and run more complex programs than most systems built with SCI alone because they have their own interfaces and libraries. We label such a design as "functionality re-creation."

The key to this design is to not fully rely on the underlying kernel for system functions. As illustrated in Figure 4, this design re-creates its own system functionalities to provide to user code. When it has to access resources like memory, CPU, and disk storage, the system accesses the kernel directly with its underlying TCB code. For example, Graphene [42] re-creates its own Linux system calls in `libLinux.so`. When it needs to acquire resources from the kernel, it uses a Platform Adaptation Layer (PAL) with access to the kernel, and provides basic API functions to the OS library.

Functionality re-creation provides a more realistic so-

lution to building virtualization systems than earlier efforts. However, functionality recreation has two pitfalls: first, if the recreated functionality resides in the TCB of the virtualization system, then vulnerabilities there can expose the host OS to attack as well. For example, hundreds of vulnerabilities have been reported in existing virtualization systems such as QEMU and VMWare over the past ten years [30]. In addition, the complex semantics of OS functions can easily lead to the emergence of bugs during the re-creation process. Some of these vulnerabilities can directly lead to privilege escalation, which allows attackers to escape the sandbox and execute arbitrary code on the host OS. For example, a vulnerability in VMWare's codebase caused by buffer overflows in the VIX API allowed local users to gain privilege to execute arbitrary code in the host OS [11].

Second, functionality recreation may assume that the underlying host kernel is correct. As we have seen, this assumption is often incorrect; host kernels may have bugs in their implementation that leave them vulnerable to attack. Thus, to provide the greatest assurance that the host kernel will not be exposed to malicious user programs, a functional recreation should try to avoid kernel APIs that are likely to contain flaws. We discuss this approach in detail next.

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

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

At the lowest level of the design (interfacing with the host OS) is the sandbox kernel (① in Figure 5). The sandbox kernel's main role is to ensure that only popular paths (② in Figure 5) of the host OS's kernel can be accessed. The sandbox kernel could thus function as a very granular system call filter, or as the core of a programming language sandbox. Note that the functionality provided by the sandbox kernel is (intentionally) much less than what an application needs. For example, an application may store files in directories and set permissions on those files. The sandbox kernel may provide a much simpler abstraction (e.g., a block storage abstraction), so long as the strictly needed functionality (e.g., persistent storage) is provided.

The application is provided more complex functionality due to the SafePOSIX re-creation (③ in Figure 5).

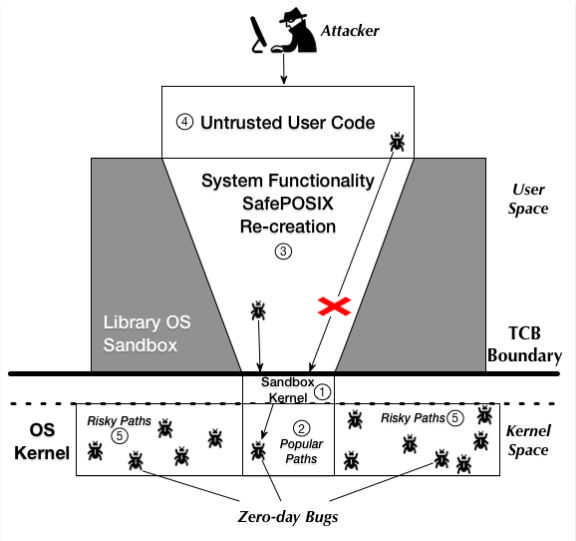

Figure 5: *Lock-in-Pop* system ensures safe execution of untrusted user code despite existing potential zero-day bugs in the OS kernel.

SafePOSIX has the needed complexity to build the more convenient higher-level abstractions using the basic functionality the sandbox kernel provides. The SafePOSIX re-creation is itself isolated within a library OS sandbox, which forces all system calls from through the sandbox kernel. So long as this is performed, all calls from SafePOSIX re-creation will only touch the permitted (popular) kernel paths in the underlying host OS.

Similarly, untrusted user code (④ in Figure 5) also must be restricted in the way in which it performs system calls. System calls must go through the SafePOSIX re-creation, into the sandbox kernel, and then to the host OS. This is done because if user code could directly make system calls, it could access any paths in the host OS's kernel desired and thus exploit bugs within them.

Note that it is expected that bugs will occur in many components. We expect that bugs will occur in the non-popular (risky) kernel paths (⑤ in Figure 5), bugs will exist in the SafePOSIX re-creation, and the user program will be buggy or even explicitly malicious (created by attackers). Since the remaining components (① and ② in Figure 5) are small and/or well tested, this leads to a lower risk of compromise.

5 Implementation of Lind

To test our *Lock-in-Pop* design, we used it to implement a secure library OS virtual machine called Lind[1]. Lind is divided into a *computational module* that enforces soft-

[1]Lind is an Old English word for a lightweight, but still strong shield constructed from two layers of linden wood.

ware fault isolation (SFI) and a library OS that safely reimplements OS functionality needed by user applications. We use a slightly modified version of Native Client (NaCl) [51] for the computational module; the library OS is implemented using Restricted Python (Repy) [8]. Finally, to support complex user applications without exposing potentially unsafe kernel paths, we provide a safe POSIX implementation, which is built on top of Repy.

In this section we provide a brief description of these components and how they were integrated into Lind, followed by an example of how the system works.

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 applications from performing system calls or executing arbitrary instructions. Instead, the application will call into a small, privileged part of NaCl that forwards system calls. In NaCl's original implementation, these calls would usually be forwarded to the host OS kernel. In Lind, we modified NaCl to instead forward these calls to our library OS, SafePOSIX (described in detail below).

Seattle's Repy. To build an API that can access the safe parts of the underlying kernel while still supporting existing applications, we need two things. First, we need a restricted sandbox that only allows access to commonly-used kernel paths. We used Seattle's Repy [8] sandbox to perform this task. Second, we have to provide complex system functions to user programs, for which we implemented the widely accepted standard POSIX interface on top of Repy.

Because the sandbox kernel is the only code that will be in direct contact with host system calls, it should be small (to make it easy to audit), while providing primitives that can be used to build more complex functionality. We used Seattle's Repy system API due to its tiny (around 8K LOC) sandbox kernel, and its minimal set of system call APIs needed to build general computational functionality. Repy allows access only to the popular portions of the OS kernel through 33 basic API functions, including 13 network functions, 6 file functions, 6 threading functions, and 8 miscellaneous functions (Table 1) [8, 37].

Repy Function	**Available System Calls**
Networking	*gethostbyname, openconnection, getmyip, socket.send, socket.receive, socket.close, listenforconnection, tcpserversocket.getconnection, tcpserversocket.close, sendmessage, listenformessage, udpserversocket.getmessage, and udpserversocket.close.*
File System I/O Operations	*openfile(filename, create), file.close(), file.readat(size limit, offset), file.writeat(data, offset), listfiles(), and removefile(filename).*
Threading	*createlock, sleep, lock.acquire, lock.release, createthread, and getthreadname.*
Miscellaneous Functions	*getruntime, randombytes, log, exitall, createvirtualnamespace, virtualnamespace.evaluate, getresources, and getlasterror.*

Table 1: Repy sandbox kernel functions that support Lind's SafePOSIX re-creation.

5.2 Enhanced Safety in Call Handling with SafePOSIX Re-creation

The full 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 (SafePOSIX) that reimplements risky system calls to provide full-featured API for legacy applications, with minimal impact on the kernel.

In Lind, a system call issued from user code is received by NaCl, and then redirected to SafePOSIX. 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 SafePOSIX. The SafePOSIX re-creation serves the system call request, marshals the result, and returns it back to NaCl. Eventually, the result is returned as the appropriate native type to the calling program.

SafePOSIX is safe because of two design principles. First, its re-creation only relies on a small set of basic Repy functions (Table 1). Therefore, the interaction with the host OS kernel is strictly controlled. Second, the SafePOSIX re-creation is run within the Repy programming language sandbox, which properly isolates any bugs inside SafePOSIX itself.

We now offer a more detailed example of how SafePOSIX works by reviewing how it re-creates a file system. The core of the SafePOSIX 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.

When Lind starts, the file system does some pre-initialization. Using the Repy API, the SafePOSIX file system reads a file named "lind.metadata" from the local directory. This file contains packed metadata from previous runs of Lind, and is loaded into the runtime SafePOSIX file system data structures. There are three main data structures: a list of open file handles, 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 SafePOSIX receives the `open` system call, it parses the path, and traverses the path in the inode lookup table. When SafePOSIX finds the file, it uses 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 allow us to store the real file name in the metadata, a necessary step because of Repy's strict rules about the content of filenames. Finally, the call returns the index into the file handle table or, if an error was encountered, an error number to which the Unix `errno` value is set.

As described in the above example, the SafePOSIX recreation only uses a few Repy sandbox kernel functions to access the hardware. It creates and maintains its own metadata and data structures, using the Repy programming language sandbox.

6 Evaluation

To evaluate Lind's effectiveness, we compared its performance against three existing virtualization systems – Docker, LXC, and Graphene. We chose these three systems because they currently represent the most widely-used design models for securing the OS kernel. LXC is a well-known container designed specifically for the Linux kernel. Docker is a widely-used container that wraps an application in a self-contained filesystem, while Graphene is an open source library OS designed to run an application in a virtual machine environment.. Lastly, we also tested Native Linux to serve 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 6.1)

How much of the underlying kernel code is exposed, and is thus vulnerable in different virtualization systems? (Section 6.2)

If Lind's SafePOSIX construction has bugs, how severe an impact would this vulnerability have? (Section 6.3)

In the Lind prototype, what would be the expected performance overhead in real-world applications? (Section 6.4)

6.1 Linux Kernel Bug Test and Evaluation

Setup. To evaluate how well each virtualization system protects the Linux kernel against reported zero-day bugs, we examined a list of 69 historical bugs that had been identified and patched in version 3.14.1 of the Linux kernel [12]. By analyzing security patches for those bugs, we were able to identify the lines of code in the kernel that correspond to each one.

In the following evaluation, we assume that a bug is potentially triggerable if the lines of code that were changed in the patch are reached (i.e., the same metric described in Section 3). This measure may overestimate potential danger posed by a system since simply reaching the buggy code does not mean that guest code actually has enough control to exploit the bug. However, this overestimate should apply equally to all of the systems we tested, which means it is still a useful method of comparison.

Next, we sought out proof-of-concept code that could trigger each bug. We were able to obtain or create code to trigger nine out of the 69 bugs [16]. For the rest, we used the Trinity system call fuzzer [41] on Linux 3.14.1 (referred to as "Native" Linux in Table 2). By comparing the code reached during fuzzing with the lines of code affected by security patches, we were able to identify an additional 26 bugs that could be triggered.

We then evaluated the protection afforded by four virtualization systems (including Lind) by attempting to trigger the 35 bugs from inside each one. The host system for each test ran a version of Linux 3.14.1 with gcov instrumentation enabled. For the nine bugs that we could trigger directly, we ran the proof of concept exploit inside the guest. For the other 26, we ran the Trinity fuzzer inside the guest, exercising each system call 1,000,000 times with random inputs. Finally, we checked whether the lines of code containing each bug were reached in the host kernel, indicating that the guest could have triggered the bug.

Results. We found that a substantial number of bugs could be triggered in existing virtualization systems, as shown in Table 2. All (100%) bugs were triggered in Native Linux, while the other programs had lower rates: 8/35 (22.9%) in Docker, 12/35 (34.3%) in LXC, and 8/35

(22.9%) bugs in Graphene. In comparison, only 1 out of 35 bugs (2.9%) was triggered in Lind.

When we take a closer look at the results, we can see that these outcomes have a lot to do with the design principles of these virtualization systems and the way in which they handle system call requests. Graphene [42] is a library OS that relies heavily on the Linux kernel to handle system calls. Graphene's Linux library implements the Linux system calls using a variant of the Drawbridge [35] ABI, which has 43 functions. Those ABI functions are provided by the Platform Adaptation Layer (PAL), implemented using 50 calls to the kernel. It turns out that 8 vulnerabilities in our test were triggered by PAL's 50 system calls. On the contrary, Lind only relies on 33 system calls, which significantly reduces risks and avoids 7 out of the 8 bugs.

Graphene supports many complex and risky system calls, such as `execve`, `msgsnd`, and `futex`, that reached the risky (unpopular) portion of the kernel and eventually led to kernel bugs. In addition, for many basic and frequently-used system calls like `open` and `read`, Graphene allows rarely-used flags and arguments to be passed down to the kernel, which triggered bugs in the unpopular paths. In Lind, all system calls only allow a restricted set of simple and frequently-used flags and arguments. One example from our test result is that Graphene allows `O_TMPFILE` flag to pass down with `path_openat()` system call. This reached risky lines of code inside `fs/namei.c` in the kernel, and eventually triggered bug CVE-2015-5706. The same bug was triggered in the same way inside Docker and LXC, but was successfully prevented by Lind, due to its strict control on flags and arguments. In fact, the design of Graphene requires extensive interaction with the host kernel and, hence, has many risks. The developers of Graphene manually conducted an analysis of 291 Linux vulnerabilities from 2011 to 2013, and found out that Graphene's design can not prevent 144 of those vulnerabilities.

LXC [27] is an operating-system-level virtualization container that uses Linux kernel features to achieve containment. Docker [15] is a Linux container that runs on top of LXC. The two containers have very similar design features that both rely directly on the Linux kernel to handle system call requests. Since system calls inside Docker are passed down to LXC and then into the kernel, we found out that all 8 kernel vulnerabilities triggered inside Docker were also triggered with LXC. In addition, LXC interacts with the kernel via its `liblxc` library component, which triggered the extra 4 bugs.

It should be noted that although the design of Lind only accesses popular paths in the kernel and implements SafePOSIX inside of a sandbox, there are a few fundamental building blocks for which Lind must rely on the kernel. To be more specific, `mmap` and `threads` cannot be recreated inside SafePOSIX without interaction with the kernel, since there has to be some basic operations to access the hardware. Therefore, in our design of Lind, `mmap` and `threads` are passed down to the kernel, and any vulnerabilities related to them are unavoidable. CVE-2014-4171 is a bug triggered by `mmap` inside Lind. It was also triggered inside Docker, LXC, and Graphene, indicating that those systems rely on the kernel to perform `mmap` operations as well.

Our initial results suggest that bugs are usually triggered by extensive interaction with the unpopular paths in the kernel through complex system calls, or basic system calls with complicated or rarely used flags. The *Lock-in-Pop* design, and thus Lind, provides strictly controlled access to the kernel, and so, poses the least risk.

Vulnerability	Native Linux	Docker	LXC	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%)**	**8/35 (22.9%)**	**12/35 (34.3%)**	**8/35 (22.9%)**	**1/35 (2.9%)**

Table 2: Linux kernel bugs, and vulnerabilities in different virtualization systems (✓: vulnerability triggered; ✗: vulnerability not triggered).

6.2 Comparison of Kernel Code Exposure in Different Virtualization Systems

Setup. To determine how much of the underlying kernel can be executed and exposed in each system, we conducted system call fuzzing with Trinity (similar to

Virtualization system	# of Bugs	Kernel trace (LOC)		
		Total coverage	In popular paths	In risky paths
LXC	12	127.3K	70.9K	56.4K
Docker	8	119.0K	69.5K	49.5K
Graphene	8	95.5K	62.2K	33.3K
Lind	1	70.3K	70.3K	0

Table 3: Reachable kernel trace analysis for different virtualization systems.

our approach in Section 3) to obtain kernel traces. This helps us understand the potential risks a virtualization system may pose based upon how much access it allows to the kernel code. 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, shown in Table 3, affirm that Lind accessed the least amount of code in the OS kernel. More importantly, all the kernel code it did access was in the popular kernel paths that contain fewer bugs (Section 3.2). A large portion of the kernel paths accessed by Lind lie in `fs/` to perform file system operations. To restrict file system calls to popular paths, Lind allows only basic calls, like `open()`, `close()`, `read()`, `write()`, `mkdir()`, `rmdir()`, and permits only commonly-used flags like `O_CREAT`, `O_EXCL`, `O_APPEND`, `O_TRUNC`, `O_RDONLY`, `O_WRONLY`, and `O_RDWR` for `open()`.

The other virtualization systems all accessed a substantial number of code paths in the kernel, and they all accessed a larger section from the unpopular paths. This is because they rely on the underlying host kernel to implement complex functionality. Therefore, they are more dependent on complex system calls, and allow extensive use of complicated flags. For example, Graphene's system call API supports multiple processes via `fork()` and signals, and therefore accesses many risky lines of code. For basic and frequently-used system calls like `open`, Graphene allows rarely-used flags, such as `O_TMPFILE` and `O_NONBLOCK` to pass down to the kernel, thus reaching risky lines in the kernel that could lead to bugs. By default, Docker and LXC do not wrap or filter system calls made by applications running in a container. Thus, programs have access to basically all the system calls, and rarely used flags, such as `O_TMPFILE`, `O_NONBLOCK`, and `O_DSYNC`. Again, this mean they can reach risky lines of code in the kernel.

To summarize, our analysis suggests that Lind triggers the fewest kernel bugs because it has better control over the portions of the OS kernel accessed by applications.

Virtualization system	# of Bugs	Kernel trace		
		Total coverage (LOC)	In popular paths (LOC)	In risky paths (LOC)
Repy	1	74.4K	74.4K	0

Table 4: Reachable kernel trace analysis for Repy.

6.3 Impact of Potential Vulnerabilities in Lind's SafePOSIX Re-creation

Setup. To understand the potential security risks if Lind's SafePOSIX re-creation has vulnerabilities, we conducted system call fuzzing with Trinity to obtain the reachable kernel trace in Linux kernel 3.14.1. The goal is to see how much kernel is exposed to SafePOSIX. Since our SafePOSIX runs inside the Repy sandbox kernel, fuzzing it suffices to determine the portion of the kernel reachable from inside the sandbox.

Results. The results are shown in Table 4. The trace of Repy is slightly larger (5.8%) than that of Lind. This larger design does not allow attackers or bugs to access the risky paths in the OS kernel, and it leaves open only a small amount of additional popular paths. These are added because some functions in Repy have more capabilities for message sending and network connection than Lind's system call interface. 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 popular paths that contain fewer kernel bugs. Thus, the Repy sandbox kernel has only a very slim chance of triggering OS kernel bugs.

Since it is the direct point of contact with the OS kernel, in theory, the Repy sandbox kernel could be a weakness in the overall security coverage provided by Lind. Nevertheless, the results above show that, even if it has a bug or failure, the Repy kernel should not substantially increase the risk of triggering bugs.

6.4 Performance Overhead

Setup. We ran a few programs of different types to understand Lind's performance impact. All applications ran unaltered and correctly in Lind. To run the applications, it was sufficient to just recompile the unmodified source code using NaCl's compiler and Lind's `glibc` to call into SafePOSIX.

To measure Lind's runtime performance overhead compared to Native Linux when running real-world applications, we first compiled and ran six widely-used legacy applications: a prime number calculator Primes 1.0, GNU Grep 2.9, GNU Wget 1.13, GNU Coreutils 8.9, GNU Netcat 0.7.1, and K&R Cat. We also ran more extensive benchmarks on two large legacy applications, Tor 0.2.3 and Apache 2.0.64 in Lind. We used Tor's built-in

Application	Native Code	Lind	Impact
Primes	10000 ms	10600 ms	1.06x
GNU Grep	65 ms	260 ms	4.00x
GNU Wget	25 ms	96 ms	3.84x
GNU Coreutils	275 ms	920 ms	3.35x
GNU Netcat	780 ms	2180 ms	2.79x
K&R Cat	20 ms	125 ms	6.25x

Table 5: Execution time performance results for six real-world applications: 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	-

Table 6: Performance results on Tor's built-in benchmark program: Native Linux vs. Lind.

benchmark program and Apache's benchmarking tool `ab` to perform basic testing operations and record the execution time.

Results. Table 5 shows the runtime performance for the six real-world applications mentioned above. 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 in the NaCl sandbox. No system calls are required, and there is no need to go through the SafePOSIX interface. The small amount of overhead is generated by NaCl's instruction alignment at building time. Another contributor to 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.

The other five applications experienced slowdowns roughly ranging from 3x to 6x. All of them require repeated calls into SafePOSIX, and this additional SafePOSIX computation produced the additional overhead. Since total execution time was limited to the magnitude of 10,000 ms, the user experience is still reasonably efficient.

A summary of the results for Tor is shown in Table 6. 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 includes encryptions of several sizes and the creation of message digests. 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 SafePOSIX system call interface.

# of Requests	Native Code	Lind	Impact
10	900 ms	2400 ms	2.67x
20	1700 ms	4700 ms	2.76x
50	4600 ms	13000 ms	2.83x
100	10000 ms	27000 ms	2.70x

Table 7: Performance results on Apache benchmarking tool `ab`: Native Linux vs. Lind.

Results for the Apache benchmarking tool `ab` are presented in Table 7. In the set of experiments, Lind produced performance slowdowns around 2.7x. Most of the overhead was incurred due to system call operations inside the SafePOSIX re-creation.

As shown above, Lind generally incurs some performance overhead. However, performance slowdown is common in virtualization systems. For example, Graphene [42] also shows an overhead ranging from 1.4x to 2x when running applications such as the Apache web server and Unixbench suite [44]. In many cases, Lind shares the same magnitude of slowdown with Graphene. Since an attack on the kernel can have devastating consequences, a tradeoff between security and performance could be justified. The fact that Lind has shown an ability to run legacy applications suggests that it is worth continuing research to optimize these systems.

7 Limitation

One of our challenges in conducting this study was deciding where to place the limits of its scope. To explore any one strategy in depth, we felt it was necessary to intentionally exclude consideration of a few other valid approaches. These choices may have placed some limitations on our results.

One such limitation stems from our chosen criteria for locating bugs. At the beginning of our study, we identified a set of common, but dangerous, zero-day bugs and then we looked for them in our obtained kernel traces. By looking only for a specific subset of bugs, we might have limited 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 ones 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 find and protect against more complex types of bugs. In the meantime, avoiding the triggering

of this initial set of bugs through the use of our *Lock-in-Pop* design can address the security needs of a significant segment of users.

8 Related Work

This section summarizes a number of earlier initiatives to ensure the safety of privileged code. The literature referenced in this section includes past efforts to design and build virtualized systems, as well as background information on technologies incorporated into Lind.

Virtualization systems. Lind incorporates a number of existing virtualization techniques, which are described below.

System Call Interposition (SCI) tracks all the system calls of processes such that each call can be modified or denied. Goldberg, et al. developed Janus [20, 48], which adopted a user-level "monitor" to filter system call requests based on user-specified policies. Garfinkel, et al. proposed a delegating architecture for secure system call interposition called Ostia [18]. Their system introduced emulation libraries in the user space to mediate sensitive system calls issued by the sandboxed process. SCI is similar to the Lind isolation mechanism. However, SCI-based tools can easily be circumvented if the implementation is not careful [40].

Software Fault Isolation (SFI) transforms a given program so that it can be guaranteed to satisfy a security policy. Wahbe, et al. [49] presented a software approach to implementing fault isolation within a single address space. Yee, et al. from Google developed Native Client (NaCl) [51], an SFI system for the Chrome browser that allows native executable code to run directly in a browser. As discussed in Section 5, Lind adopts NaCl as a key component to ensure secure execution of binary code.

Language-based virtualization. Programming languages like Java, JavaScript, Lua [26], and Silverlight [39] can provide safety in virtual systems by "translating" application commands into a native language. 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 [21, 33]. Any bug or failure in a programming language virtual machine is usually fatal. In contrast, the main component of Lind is built using Repy, which is a programming language with a very small TCB, minimizing the chance of contact with kernel flaws.

OS virtualization techniques include bare-metal hardware virtualization, such as VMware ESX Server, Xen [4], and Hyper-V, container systems such as LXC [27], BSD's jail, and Solaris zones, and hosted hypervisor virtualization, such as VMware Workstation, VMware Server, VirtualPC and VirtualBox. Security by isolation [2, 9, 23, 50] provides safe executing environments through containment for multiple user-level virtual environments sharing the same hardware. However, this approach is limited due to the large attack vectors against the hypervisors.

Library OSes allow applications to efficiently obtain the benefits of virtual machines by refactoring a traditional OS kernel into an application library. Porter, et al. developed Drawbridge [35], a library OS that presents a Windows persona for Windows applications. Similar to Lind, it restricts access from usermode to host OS through operations that pass through the security monitor. Baumann, et al. presented Bascule [5], an architecture for library OS extensions based on Drawbridge that allows application behavior to be customized by extensions loaded at runtime. The same team also developed Haven [6], which uses a library OS to implement shielded execution of unmodified server applications in an untrusted cloud host. Tsai, et al. developed Graphene [42], a library OS that executes both single and multi-process applications with low performance overheads.

While typical library OSes trust the underlying host kernels to perform many system functions, Lind recreates complex OS functions through memory-safe Repy code, relying on only a limited set of system functions from the Repy sandbox. Furthermore, Drawbridge, Bascule or Haven do not have a sandbox environment to properly contain buggy or malicious behavior, while Lind can offer this more secure environment.

9 Conclusion

In this paper, we propose a new security metric based on quantitative measures of kernel code execution when running user applications. After determining that fewer bugs exist in popular paths associated with frequently-used programs, we devise a new design for a secure virtual machine called *Lock-in-Pop*. As the name implies, the design scheme locks away access to all kernel code except that found in paths frequently used by popular programs. We test the *Lock-in-Pop* idea by implementing a prototype system called Lind, which features a minimized TCB and prevents direct access to application calls from less-used, riskier paths. Instead, Lind supports complex system calls by securely re-creating essential OS functionalities inside a sandbox. In tests against Docker, LXC, and Graphene, Lind emerged as the most effective system in preventing zero-day Linux kernel bugs.

So that other researchers may replicate our results, we make all of the kernel trace data, benchmark data, and source code for this paper available [redacted].

References

[1] Debian Popularity Contest. http://popcon.debian.org/main/index.html. Accessed December 2014.

[2] Qubes OS. http://www.qubes-os.org. Accessed November 2015.

[3] Intel Virtualization Technology Specification for the Intel Itanium Architecture (VT-i), April 2005.

[4] BARHAM, P., DRAGOVICH, B., FRASER, K., HAND, S., HARRIS, T., HO, A., NEUGEBAUER, R., PRATT, I., AND WARFIELD, A. Xen and the art of virtualization. In *Proceedings of the SOSP'03* (2003), pp. 164–177.

[5] BAUMANN, A., LEE, D., FONSECA, P., GLENDENNING, L., LORCH, J. R., BOND, B., OLINSKY, R., AND HUNT, G. C. Composing os extensions safely and efficiently with bascule. In *Proceedings of the Eurosys'13* (2013).

[6] BAUMANN, A., PEINADO, M., AND HUNT, G. Shielding applications from an untrusted cloud with haven. In *Proceedings of the OSDI'14* (2014).

[7] Berkeley software distribution. http://www.bsd.org. Accessed September 2016.

[8] CAPPOS, J., DADGAR, A., RASLEY, J., SAMUEL, J., BESCHASTNIKH, I., BARSAN, C., KRISHNAMURTHY, A., AND ANDERSON, T. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the CCS'10* (2010).

[9] CHEN, X., GARFINKEL, T., LEWIS, E. C., SUBRAHMANYAM, P., WALDSPURGER, C. A., BONEH, D., DWOSKIN, J., AND PORTS, D. R. Overshadow: A virtualization-based approach to retrofitting protection in commodity operating systems. *SIGPLAN Not. 43*, 3 (Mar. 2008), 2–13.

[10] CHOU, A., YANG, J., CHELF, B., HALLEM, S., AND ENGLER, D. *An empirical study of operating systems errors*, vol. 35. ACM, 2001.

[11] CVE-2008-2100. VMware buffer overflows in the VIX API let local users execute arbitrary code in the host OS. `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-2100`, 2008.

[12] CVE Details Datasource. http://www.cvedetails.com/vulnerability-list/vendor_id-33/product_id-47/version_id-163187/Linux-Linux-Kernel-3.14.1.html. Accessed October 2014.

[13] CVE DETAILS. 109 VMWare Workstation Vulnerabilities Reported. https://www.cvedetails.com/vulnerability-list/vendor_id-252/product_id-436/Vmware-Workstation.html, 2016.

[14] CVE DETAILS. 40 Virtualbox Vulnerabilities Reported. https://www.cvedetails.com/vulnerability-list/vendor_id-93/product_id-20406/Oracle-Vm-Virtualbox.html, 2016.

[15] Docker. https://www.docker.com. Accessed September 2016.

[16] Exploit Database. https://www.exploit-db.com. Accessed October 2014.

[17] FBI Tweaks Stance On Encryption BackDoors, Admits To Using 0-Day Exploits. http://www.darkreading.com/endpoint/fbi-tweaks-stance-on-encryption-backdoors-admits-to-using-0-day-exploits/d/d-id/1323526. Accessed September 21, 2016.

[18] GARFINKEL, T., PFAFF, B., AND ROSENBLUM, M. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the NDSS'04* (2004).

[19] gcov(1) - Linux man page. http://linux.die.net/man/1/gcov. Accessed October 2014.

[20] GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A secure environment for untrusted helper applications (confining the wily hacker). In *Proceedings of the USENIX UNIX Security Symposium'96* (1996).

[21] Learn about java technology. http://www.java.com/en/about/.

[22] KELLER, E., SZEFER, J., REXFORD, J., AND LEE, R. B. Nohype: virtualized cloud infrastructure without the virtualization. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 350–361.

[23] LI, C., RAGHUNATHAN, A., AND JHA, N. Secure virtual machine execution under an untrusted management os. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on* (July 2010), pp. 172–179.

[24] Linux kernel zero-day flaw puts 'tens of millions' of PCs, servers and Android devices at risk. http://www.v3.co.uk/v3-uk/news/2442582/linux-kernal-zero-day-flaw-puts-tens-of-millions-of-pcs-servers-and-android-devices-at-risk. Accessed September 21, 2016.

[25] Linux Test Project. https://linux-test-project.github.io/. Accessed February 2015.

[26] The programming language Lua. www.lua.org. Accessed October 2015.

[27] Linux Container (LXC). https://linuxcontainers.org. Accessed September 2016.

[28] MAYER, A., AND SYKES, A. A probability model for analysing complexity metrics data. *Software Engineering Journal 4*, 5 (1989), 254–258.

[29] NSA Discloses 91 Percent Of Vulns It Finds, But How Quickly? http://www.darkreading.com/vulnerabilities—threats/nsa-discloses-91-percent-of-vulns-it-finds-but-how-quickly/d/d-id/1323077. Accessed September 21, 2016.

[30] National Vulnerability Database. https://nvd.nist.gov/. Accessed September 2015.

[31] OZMENT, A., AND SCHECHTER, S. E. Milk or wine: does software security improve with age? In *Usenix Security* (2006).

[32] PALIX, N., THOMAS, G., SAHA, S., CALVÈS, C., LAWALL, J., AND MULLER, G. Faults in linux: ten years later. In *ACM SIGARCH Computer Architecture News* (2011), vol. 39, ACM, pp. 305–318.

[33] PAUL, N., AND EVANS., D. Comparing java and .net security: Lessons learned and missed. In *Computers and Security* (2006), pp. 338–350.

[34] Poisson Distribution. https://en.wikipedia.org/wiki/Poisson_distribution.

[35] PORTER, D. E., BOYD-WICKIZER, S., HOWELL, J., OLINSKY, R., AND HUNT, G. C. Rethinking the library os from the top down. In *Proceedings of the ASPLOS'11* (Newport Beach, California, USA, 2011), pp. 291–304.

[36] Qemu. http://wiki.qemu.org/Main_Page. Accessed September 2016.

[37] Seattle's Repy V2 Library. https://seattle.poly.edu/wiki/RepyV2API. Accessed September 2014.

[38] SHIUE, W.-K., AND BAIN, L. J. Experiment size and power comparisons for two-sample poisson tests. *Applied Statistics* (1982), 130–134.

[39] Microsoft Silverlight. http://www.microsoft.com/silverlight/. Accessed October 2015.

[40] TAL GARFINKEL. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.

[41] Trinity, a Linux System call fuzz tester. http://codemonkey.org.uk/projects/trinity/. Accessed November 2014.

[42] TSAI, C. C., ARORA, K. S., BANDI, N., JAIN, B., JANNEN, W., JOHN, J., KALODNER, H. A., KULKARNI, V., OLIVEIRA, D., AND PORTER, D. E. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the EuroSys'14* (Amsterdam, Netherlands, 2014).

[43] TSAI, C.-C., JAIN, B., ABDUL, N. A., AND PORTER, D. E. A study of modern linux api usage and compatibility: what to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems* (2016), ACM, p. 16.

[44] Unixbench. https://github.com/kdlucas/byte-unixbench. Accessed September 2016.

[45] Virtualbox. https://www.virtualbox.org. Accessed September 2016.

[46] Vmware server. https://my.vmware.com/web/vmware/info?slug=infrastructure_operations_management/vmware_server/2_0.

[47] Vmware workstation. https://www.vmware.com/products/workstation. Accessed September 2016.

[48] WAGNER, D. A. Janus: An approach for confinement of untrusted applications. In *Tech. Rep. CSD-99-1056, University of California, Berkeley* (1999).

[49] WAHBE, R., LUCCO, S., ANDERSON, T. E., AND GRAHAM, S. L. Efficient software-based fault isolation. In *SIGOPS Oper. Syst. Rev. 27, 5* (1993), pp. 203–216.

[50] YANG, J., AND SHIN, K. G. Using hypervisor to provide data secrecy for user applications on a per-page basis. In *Proceedings of the Fourth ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments* (New York, NY, USA, 2008), VEE '08, ACM, pp. 71–80.

[51] YEE, B., SEHR, D., DARDYK, G., CHEN, J. B., MUTH, R., ORMANDY, T., OKASAKA, S., NARULA, N., AND FULLAGAR, N. Native client: A sandbox for portable, untrusted x86 native code. In *Proceedings of the IEEE Symposium on Security and Privacy* (Berkeley, CA, USA, 2009), pp. 79–93.

[52] 0-day exploits more than double as attackers prevail in security arms race. http://arstechnica.com/security/2016/04/0-day-exploits-more-than-double-as-attackers-prevail-in-security-arms-race/. Accessed September 21, 2016.