Enhancing Container Security by Logging the Unpopular Paths in the Operating System Kernel

Yiwen Li
New York University
liyiwen@nyu.edu

Brendan Dolan-Gavitt
New York University
brendandg@nyu.edu

Justin Cappos
New York University
jcappos@nyu.edu

Abstract

One of several reasons why containers, such as Docker and LinuxKit, are so widely used is because of the perceived isolation and security they provide against the potentially malicious or buggy user programs running inside of them. However, containers provide no protection from the zero-day kernel bugs inside the kernel itself, which, if triggered, could lead to such security problems as privilege escalation. Containers remain vulnerable because they allow access to rarely executed paths where such bugs can be found. Limiting kernel access to only frequently used ***popular paths***, which have previously been proven to contain fewer security bugs, would greatly reduce this risk. Therefore, if a container could run using only these paths, its security would be greatly enhanced.

In this paper, we propose a method to enhance the security of existing container designs by leveraging the popular paths metric. We designed and implemented a ***Secure Logging System***, which first systematically identifies popular paths data in widely-used containers, and uses this data to create a kernel that generates security logs and warning messages when a potentially risky path is reached. Users can then decide whether the potentially dangerous code execution should be stopped. We tested this modified kernel, which we named the ***Secure Logging Kernel***, to determine its ability to run containers. Our evaluation verified that, more than 99.9% of the time, containers were able to run their default workload using just the popular paths. Furthermore, only 6% of 50 Linux kernel CVE security vulnerabilities we examined were present in the popular paths on which these containers ran, greatly reducing any potential risk of triggering kernel bugs. Lastly, our performance evaluation shows that containers running our Secure Logging Kernel only incur about 0.1% runtime overhead on average, and around 0.37% extra cost of memory space. This means the Secure Logging Kernel can improve security with little or no loss of efficiency.

1 Introduction

Containers continue to grow in popularity, with one industry survey suggesting the technology could become a $2.7 billion market by 2020 [20]. This widespread adoption of containers, such as Docker [22] and LinuxKit [28], can be attributed to several factors, including portability and elimination of the need for a separate operating system [34]. In addition, the perceived isolation and security they provide to the user programs running inside of them is reassuring to users looking to protect their data and operation. However, this perception may be false, as several recent incidents have demonstrated it is possible to trigger zero-day kernel bugs from inside of a container [25, 32]. Furthermore, since the kernel is the critical and privileged code shared by containers and the host system, exploitation of such bugs could lead to severe security problems, such as privilege escalation. The famous "Dirty COW" vulnerability that emerged in 2016, reported as CVE-2016-5195, allow attackers to escape from a Docker container and access files on the host system [1]. This vulnerability affected all of the Linux-based operating systems that used older versions of the Linux kernel, including Android. One analysis, conducted a year after it was first reported, found the bug in more than 1200 malicious Android apps, affecting users in at least 40 countries [2].

The fundamental reason such exploits can occur is that existing containers are designed to directly access the underlying host operating system kernel on which they are run. This design, which makes containers more efficient and lightweight to use—features that make them so attractive to end-users— unfortunately exposes them to zero-day bugs within the kernel itself [32]. To this point, there has been little or no effort to limit such contact, in part because there was no efficient way to identify where these bugs might be. If one knew which part of the kernel was free of zero-day bugs — or at the very least was less likely to host this threat— then restricting a container to touch only this part of the kernel would greatly improve security.

Yet, targeting where the bugs are likely to be has been an elusive goal for many years. A number of researchers have promoted metrics [21, 31] that claim to be able to predict the presence of bugs, but most have proven only minimally effective. That is, until two years ago when a study [27] demonstrated a powerful correlation between the popularity of a kernel code path and its likelihood to contain a security flaw. Furthermore, the study [27] demonstrated that these frequently used kernel paths can be leveraged to design and construct secure virtualization systems. Building on these results, we ask the question, can we apply the popular paths metric to securing existing containers? And, if such an application is possible, could it provide stronger security and truer isolation than the existing design of container systems permits? To do so, we would first have to answer

the question: Can applications in containers run without access to the whole kernel?

In this paper, we document our efforts to answer those questions by studying the feasibility of using the popular paths metric to secure the LinuxKit container toolkit. This entailed first, finding a way to identify the parts of the underlying kernel that are less likely to contain security bugs. We were able to demonstrate a systematic way to gather data on popular paths, and affirmed that this data is representative of hundreds of popular Docker containers from Docker Hub [23]. Next, we produced a Secure Logging Kernel designed to log any attempt to access the unpopular paths, and to warn users away from code found in the less-used paths. The Secure Logging Kernel contributed to our study in two ways. Firstly, it provided usable data on how often applications used these risky paths, so we could determine whether such paths are essential to their functionality. And, secondly, it built into our instantiation of the modified kernel a way for users to detect potentially dangerous program behaviors and make decisions about whether or not to block executions in unpopular paths.

An evaluation of our findings affirmed that popular Docker containers experience no loss of functionality when using just the popular paths. Indeed, for the official Docker containers' default workload, the Secure Logging Kernel was able to run it smoothly. And our data shows that the popular paths were used more than 99.9% of the time. In the case of less than 0.1% of the time, unpopular lines were used, and were captured by the Secure Logging Kernel with security logs generated. Our security evaluation confirmed that using the popular paths to run Docker containers can effectively prevent most kernel bugs from being triggered, as only three of the 50 CVE kernel security vulnerabilities we checked for were found in those LinuxKit paths. And, in our performance evaluation, we found that running our Secure Logging Kernel only incurred about 1% of runtime overhead, while the memory space overhead was only about 0.37%. Thus, greater security could be achieved with very little perceived difference in operation efficiency or cost.

In summary, we make the following contributions in this paper:

- We systematically identify and capture the "popular paths" data from Docker containers running in the LinuxKit VM.
- We design and implement a Secure Logging System, that utilizes the popular paths data to create a modified Linux kernel (Secure Logging Kernel). This instrumented kernel outputs security warning messages whenever there is an attempt to reach and trigger unpopular paths.
- Using the Secure Logging Kernel, we demonstrate that popular Docker containers were able to run their default workload normally all the time with negligible (less than 1%) performance overhead. Popular paths were used more than 99.9% of the time. Unpopular paths were reached only less than 0.1% of the time, and were recorded by the Secure Logging Kernel with security logs to warn the users about potential malicious program behavior.

2 Background and Motivation

The concept of full virtualization, where a guest operating system runs programs in isolation, has been around for several decades [30]. Yet, the emergence of container programs [39], which directly utilize the host kernel code to perform OS functionality, and therefore have a lower overhead than full virtual machines, has spurred growth in container popularity. Docker [22], LinuxKit [28], and Kubernetes [26] have all seen wide adoption over the past few years. For example, Docker Hub [23] now holds more than two million container images, and the most popular ones have been downloaded by more than ten million users. Many users prefer using containers over traditional full-scale virtual machines, such as VMware workstation [38] and VirtualBox [37], because containers allow them to test and develop their software programs with relatively low overhead. In addition, the idea of running their programs in a "contained" environment tends to make developers feel they have sufficiently secured them against potential threats.

This perception has been bolstered over the years by the deployment of a number of security and isolation mechanisms designed to protect the programs running inside of them. Linux introduced Namespaces [11] to the kernel between versions 2.6.15 and 2.6.26, to provide a global system resource abstraction that helps achieve isolation between different containers. Docker and LXC [14] containers also use this technique. Another key Linux feature called control groups [9] can be used to limit, account for, and isolate important system resources, such as memory, CPU, and disk I/O, to each container. This feature not only controls resources, but also helps prevent denial-of-service attacks.

When Linux kernel capabilities [10] were introduced, container systems, such as Docker were also able to restrict permissions on specific resources, such as network and file system. Another option for enhancing kernel security has emerged, with the development of kernel hardening systems, such as AppArmor [3], SELinux [19], GRSEC [7], and PAX [17]. Acknowledging that buggy code is likely unavoidable, these systems look to make the kernel code resilient enough to attacks that any resident bugs cannot be exploited. GRSEC and PAX add safety checks at compile-time and run-time of the kernel. Docker ships a template that works with AppArmor, and also has SELinux policies working with Red Hat [18]. These kernel hardening systems are mostly access control mechanisms that provide safety in a way similar to that of capabilities.

These existing mechanisms do improve security to a certain degree, but do not address the aforementioned access problem. Unlike in a full-scale virtual machine, the host kernel is shared among all the containers and the host operating system. This magnifies the damage a kernel vulnerability could cause, in addition to, breaking the isolation that users expect from using a container.

The design initiative described in the next section was launched specifically to enhance container security by preventing zero-day kernel bugs from being triggered inside containers, while still allowing containers to perform their assigned workload. Having already established in a previous paper that leveraging the popular paths metric could identify and neutralize the threat of these bugs [27], we proposed that container security can be enhanced in the same manner. To prove this point, we designed and implemented the Secure Logging System.

3 Design and Implementation

Designing an implementation of the popular paths metric for use with containers required completion of two separate tasks. First, we had to find the popular paths for the underlying host operating system. And, second, once we knew where these paths were, we needed a way to log / block the unpopular paths to warn and keep containers away from these less-used and potentially buggy code paths. Our solution was a dual module approach we call the Secure Logging System (Shown in Figure 1).

3.1 Design of Our "Secure Logging System"

Identifying the popular paths for the given container system is handled by the system module we refer to as the Kernel Profiler. This module identifies the lines of code in the host kernel that are executed when running its default / regular workload. Such a workload is often defined in the configuration files (Dockerfiles) that come with the container images. We collected "popular paths kernel traces," as we named them, from a set of the containers, as ranked "most popular" by the number of user downloads.

In effect, the Profiler sets out a map highlighting places in the kernel that are safe for an application to access. The second module, the Kernel Modifier, uses this map to instrument the rarely used unpopular paths to either generate security warning messages / logs, or to block code execution on these paths. When both modules are implemented, the result is an instrumented Linux kernel with security monitors and checks inserted at the non-popular paths. This Secure Logging Kernel can then be used to run containers without any required changes to the containers themselves.

3.2 Implementation

To adapt our design for real-world containers and applications, it was necessary to build our own infrastructure to first systematically profile popular containers to obtain the popular paths data, and then use it to instrument the host Linux kernel to add the function of generating security logs and warning messages. This section describes in detail how the modules in our system were implemented.

3.2.1 Implementing the Kernel Profiler

The Kernel Profiler is designed to collect the kernel trace of running user containers. The popular paths kernel trace we are looking to identify refers to the lines of code in the underlying host operating system kernel that were executed when running the regular container workload. This workload is defined in the configuration files of the corresponding images from containers we labeled popular based on the number of user downloads. Our Kernel Profiler works in the following way:

1. The Linux kernel used to run the Docker containers is recompiled with the Gcov [6] kernel profiling feature enabled.
2. To run the Docker containers, the Kernel Profiler first automatically generates the configuration file for the Gcov-enabled LinuxKit. This file will define a task container, running the main testing workload, along with a data container responsible for collecting, storing, and transferring the kernel trace data. (shown in Figure 2)
3. When booting the LinuxKit virtual machine, a script in the profiler automatically starts running the workload of the task container. The profiler will collect the kernel trace data, in the form of gcda files generated by Gcov, store them at "/sys/kernel/debug/gcov/," and by the end of the run transfer the data from inside of the data container to the host system for further use.
4. Using the lcov [8] tool to process the kernel trace data collected from the host system, the profiler generates formatted data about which lines of code were executed in the kernel source files. These files will be used by the Kernel Modifier to identify the unpopular paths that must trigger an alert if used.

3.2.2 Implementing the Kernel Modifier

For the Kernel Modifier to insert security logging code at the unpopular paths in the Linux kernel, It needs to first identify the correct places in the kernel source files to be instrumented. It can then modify the code as needed. The Kernel Modifier has three main parts: the Clang compiler, the Clang analyzer, and the kernel instrumenting tool (Shown in Figure 3). It operates in the following way:

1. The Clang C compiler from the Clang / LLVM project [13], along with a BEAR [4] tool compiled the Linux kernel source. Using BEAR, we can generate a compilation database containing all the compile flags and options needed for the process. In turn, this database

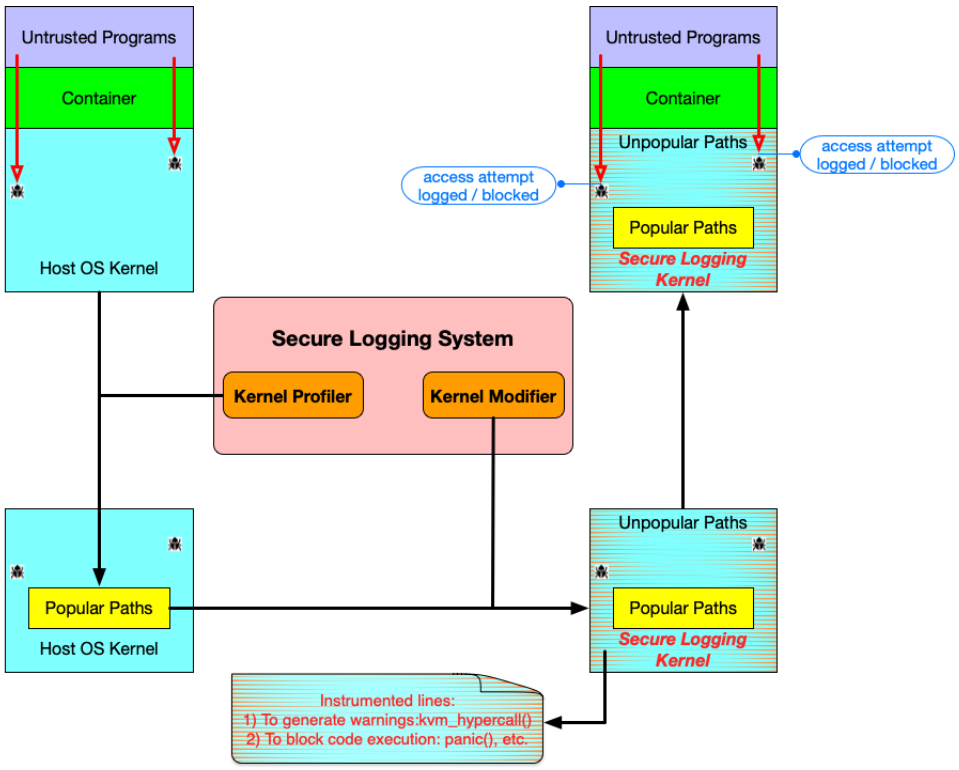

Figure 1. Design overview of our Secure Logging System

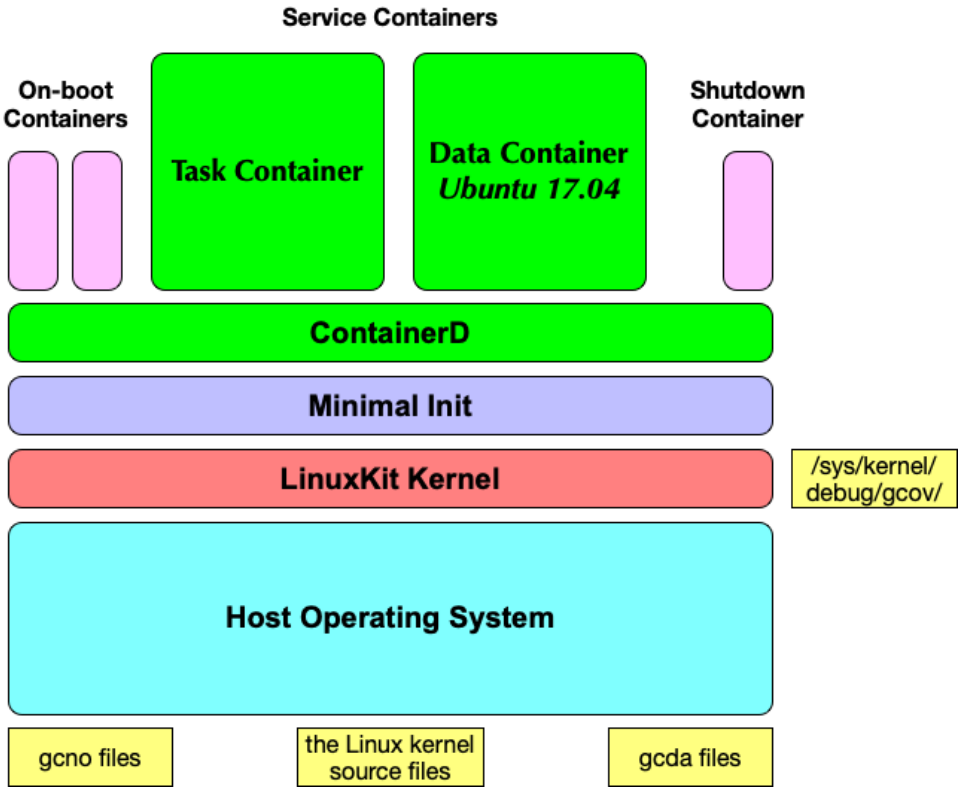

Figure 2. Implementation of the Kernel Profiler

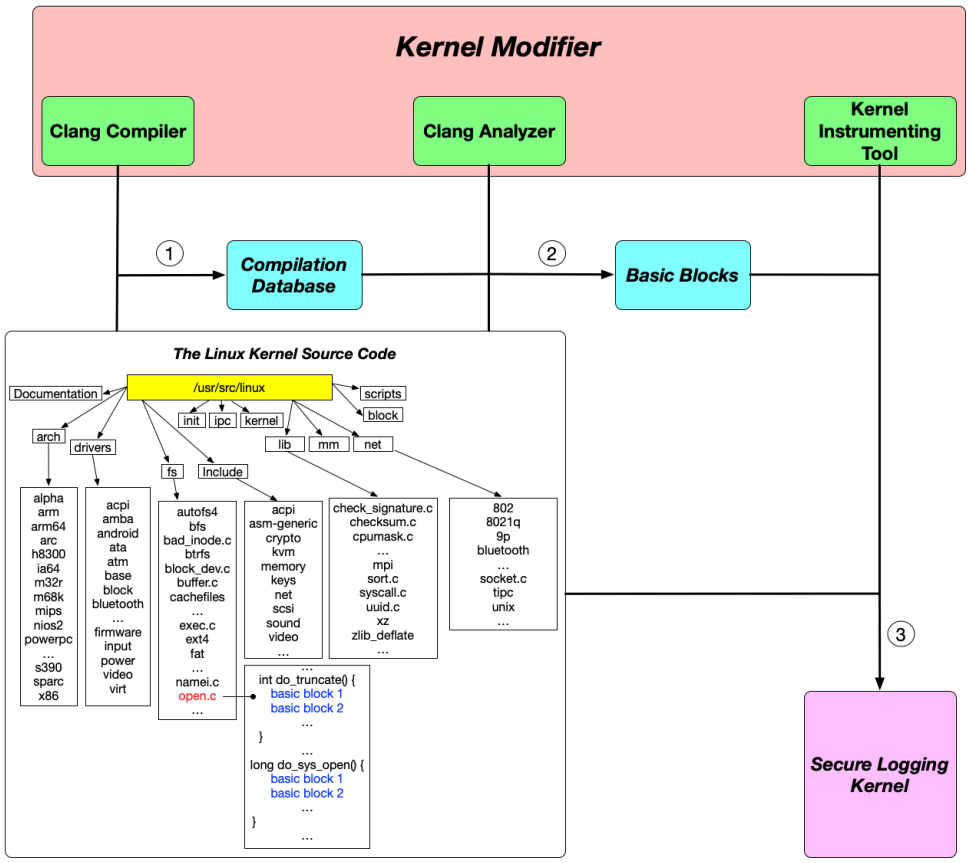

Figure 3. Implementation of the Kernel Modifier

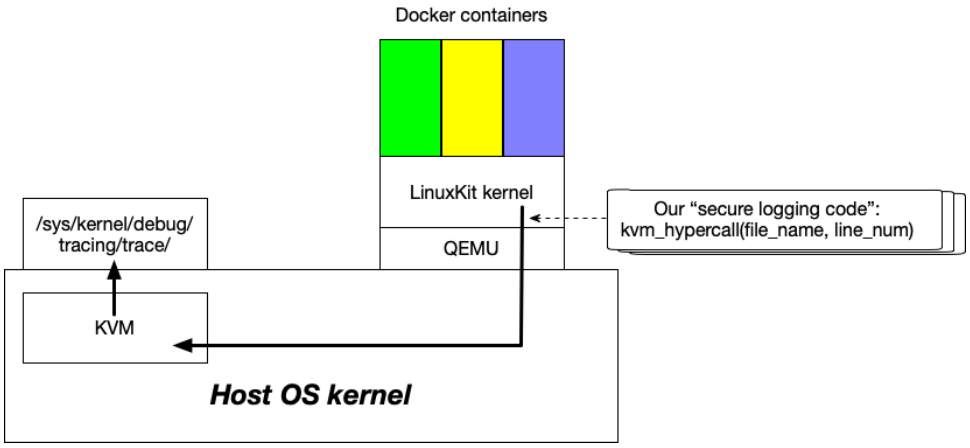

Figure 4. Our KVM_HYPERCALL kernel instrumentation

will be used by our Clang analyzer to perform source code parsing in the next step.

2. Once we have the compilation database for the Linux kernel, we use the Clang analyzer to perform static analysis on the kernel source code to obtain the control flow graph for each function in the files. The analyzer leveraged Clang's LibTooling [5] to construct the AST tree from the kernel source code, and then obtain the control flow graph. With this graph, we can identify the corresponding source line number for each basic block. Furthermore, the beginning lines of all the blocks in the source files are candidates for places to add security logging code. Here, a basic block refers to a straight-line code sequence with no branches in

except to the entry and no branches out except at the exit. Thus, if a basic block is unpopular, it is sufficient to insert only one piece of secure logging code at the beginning of it to monitor and warn the attempt to reach that code. This approach can optimize our instrumentation by avoiding redundancy in our code injection.

3. The next steps has the kernel instrumenting tool executing modifications based on both the basic blocks, and the collected popular paths data . The algorithm for our kernel modification directs us through the basic blocks in the kernel source files. If none of the lines in a block are in the popular paths data, we consider this an unpopular basic block, and our instrumenting tool will add the security logging code at the beginning of this block. If we discover that an entire function has no lines in the popular paths data, we consider this an unpopular function, and just add our security logging code once where it starts . This allows us to avoid adding redundant and unnecessary code. The secure logging code we inserted in front of the unpopular paths was a kvm hypercall from the LinuxKit kernel into the host Linux kernel (Shown in Figure 4). In this way, we can guarantee minimal affect on the LinuxKit kernel functionality, while still being able to generate security logging whenever unpopular paths were reached.
4. The Kernel Modifier automatically inserts the secure logging code at the beginning of each basic block in the unpopular paths data. Our kernel modification works at the source-code level. The kernel source files are directly modified, and then compiled to produce the Secure Logging Kernel, which can be directly used to run Docker containers in the LinuxKit VM.

4 Evaluation

To demonstrate that our popular paths based approach is practical and effective in improving container security, we first had to verify that real-world container applications could function correctly using the popular paths. In addition, we needed to prove that the modified kernel was actually more secure than existing options.

To test these assumptions, our evaluation set out to answer the following questions:

- Can real-world containers run on the popular paths with correct functionality? (Section 4.1)
- Does restricting access only to the popular paths improve the security of running containers? (Section 4.2)
- What is the performance overhead of adopting our popular paths based security strategy, the Secure Logging Kernel? (Section 4.3)

4.1 Functionality Evaluation

Answering our first question required a two-part functionality evaluation. First, we used the Linux Testing Project (LTP) [12] test suites—an open source test project designed to validate the reliability, robustness, and stability of Linux—on the Secure Logging Kernel to verify it functions as anticipated. Second, we ran a collection of the most popular Docker containers from Docker Hub to verify if they could function correctly accessing only the popular paths.

4.1.1 Testing functionality with the LTP test suites

Experimental Setup. We used the Dockerfile provided by LinuxKit [28] to create the container image that runs the LTP version 20170116. This test project offers a set of regression and conformance tests designed to let members of the open source community confirm the behavior of the Linux kernel. The test script we ran consisted of 798 test cases that verified the correctness of system functionalities, such as memory allocation, network connection, file system access, locking, and more. Using the test suites, we ran our experiments inside of a LinuxKit version 0.2+ virtual machine. The machine was built using Docker version 18.03.0-ce running on a host operating system of Ubuntu 16.04 LTS, with Linux kernel 4.13.0-36-generic. A QEMU emulator version 2.5.0 (Debian 1:2.5+dfsg-5ubuntu10.41) served as the local hypervisor, using the KVM Linux kernel modules with KVM acceleration enabled.

Results. The Secure Logging Kernel was able to complete all of the 798 test cases. Furthermore, compared to results from identical tests on the original unmodified Linux kernel, the Secure Logging Kernel produced the same output. (Shown in Table 1). Among all these 798 test cases, 673 of them passed with the expected return values. 20 tests were skipped due to certain required functions unavailable in the LinuxKit VM. For example, the swap file was not accessible in LinuxKit, therefore the related "swapoff" test iterations were skipped. 105 test cases failed due to restrictions or functionality issues of LinuxKit. For example, "mem01" test failed because the malloc function failed to allocate 3056 MB memory due to the default memory restriction. "gf01" test failed due to "no space left on device" in LinuxKit. "swapon01" test failed due to swapfile not available, and more. These skipped and failed tests were due to the inherent issues of the LinuxKit VM. Our Secure Logging Kernel didn't incur any additional functionality issues.

Table 1. LTP tests

	Original Linux kernel	Secure Logging Kernel
Total Tests	798	798
Total Skipped Tests	20	20
Total Failures	105	105

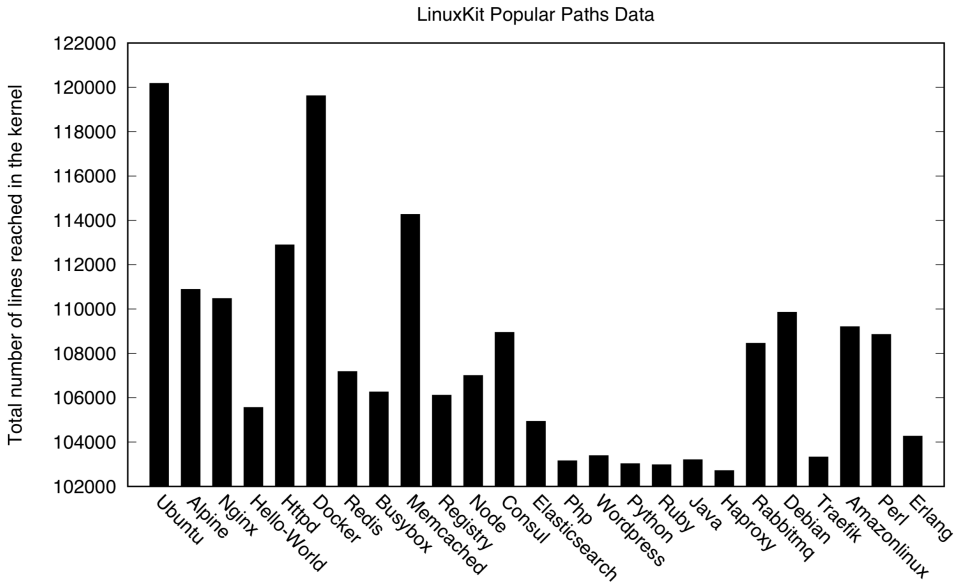

Figure 5. Popular paths for individual Docker containers

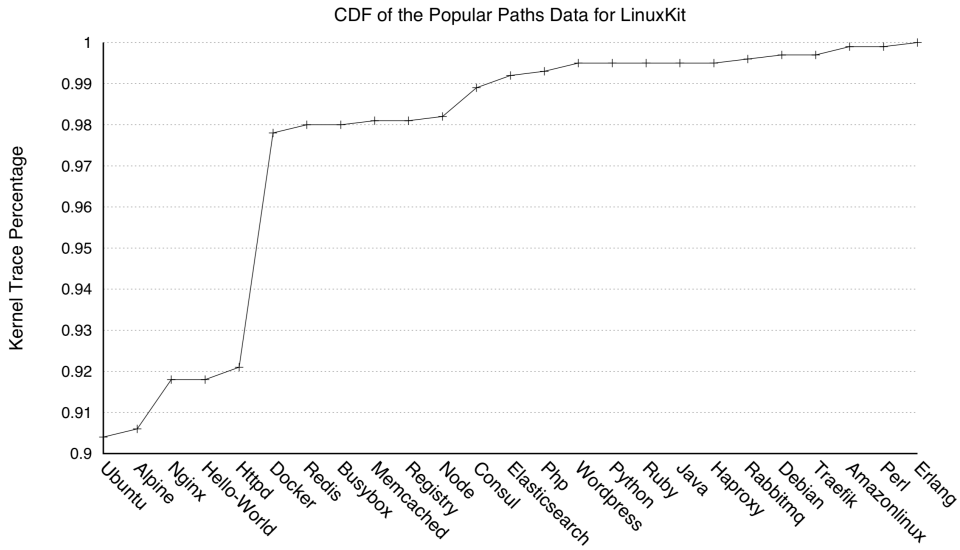

Figure 6. CDF of the popular paths of Docker containers showing most share the same kernel footprint

4.1.2 Testing functionality on real-world container applications

Experimental Setup. To confirm that our Secure Logging Kernel works in real-world practices, we tested the 100 most downloaded containers from Docker Hub. We ran the experiment in the same version of the LinuxKit virtual machine as in 4.1.1. Each Docker container was run in the LinuxKit VM using the commands in its Dockerfile from its official Docker image. To take into account any potential variances, each container was run in the exact same environment for 10 times.

Results. One of the more important results of this initial test was that it showed most containers shared the same kernel footprint. This means that the kernel trace, or the record of all kernel code executed, of a sample of containers can represent the trace of many more. We used the CDF (Cumulative Distribution Function) to analyze how soon the kernel trace of different containers would converge. Results of the CDF, as visualized in Figure 6, points out that the trace of the top six popular containers covered about 98% of the total kernel trace for 25 containers. This offers additional corroboration that the popular paths data we collected is

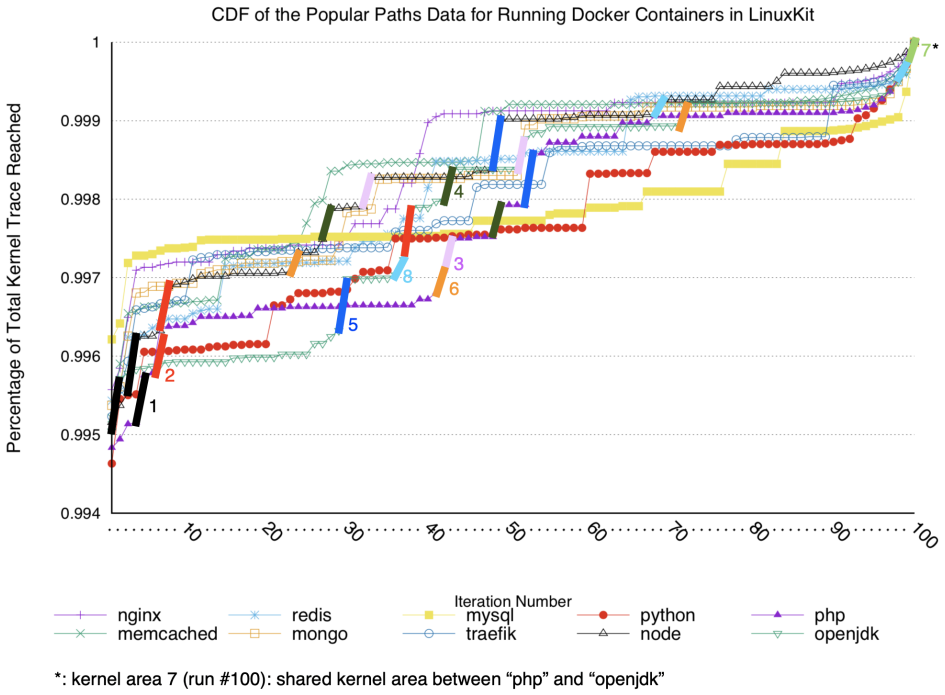

Figure 7. Common kernel code identified across different containers

valid not only for the few containers we ran, but also for hundreds of containers on Docker Hub.

In addition to the always used popular paths, which we identified as the common kernel lines that appeared each and every time, we also discovered certain lines of kernel code that showed up sporadically. For these lines, we looked at what activities they performed, and how common they were across different containers. In our analysis of the kernel trace of 10 popular Docker containers, we identified eight pieces of infrequently executed kernel code common among them. These traces are presented graphically in Figure 7. Each numbered line (from 1 to 8) in the graph represents a piece of infrequently executed code. The kernel function that corresponds with each of the eight pieces is explained below:

1. `kernel/cgroup/freezer.c`: a cgroup is freezing if any FREEZING flags are set.
2. `kernel/locking/rwsem-xadd.c`: waiting for a write lock to be granted.
3. `kernel/locking/rwsem-xadd.c`: waiting for the read lock to be granted.
4. `mm/filemap.c`: process waitqueue for pages and check for a page match to prevent potential waitqueue hash collision.
5. `fs/exec.c`: all other threads have exited, wait for the thread group leader to become inactive and to assume its PID.
6. `kernel/workqueue`: insert a barrier work in the queue. According to the comments from the kernel source code, the reason for inserting a barrier work seems to be to prevent a cancellation. This is because `try_to_grab_pending()` can not determine whether the work to be grabbed is at the head of the queue and thus can not clear LINKED flag of the previous work. There must be a valid next work after a work with a LINKED flag set.
7. `arch/x86/kernel/tsc.c`: try to calibrate the TSC against the Programmable Interrupt Timer and return the frequency of the TSC in kHz.
8. `kernel/locking/mutex.c`: hand off a mutex. Give up ownership to a specific task, when @task = NULL, this is equivalent to a regular unlock.

What this analysis tells us is that this infrequently used code performs essential kernel functions used by multiple containers, and therefore should still be considered popular paths, despite the fact that they are infrequently execute. On closer examination, we learn they are not always executed during each run because they are race-condition related code that depend on locks, and system conditions may vary during different runs. As the Secure Logging System profiles and identifies these infrequently-executed popular paths, as well as the always-executed ones, our tests show that the Secure Logging Kernel functions in a correct manner.

Based on the comprehensive popular paths data obtained, the Secure Logging System was able to create a customized

Table 2. Instrumentation code added in our Secure Logging Kernel

kernel dir	unpopular functions	popular functions	total lines inserted
arch	1502	931	3325
block	774	245	1035
crypto	527	121	656
drivers	6290	2584	13305
fs	2108	1404	4883
ipc	198	42	249
kernel	3721	2120	6335
mm	1037	792	2954
net	4107	1746	7510
security	200	180	551
total	**20464**	**10165**	**40803**

Secure Logging Kernel for the LinuxKit VM. Table 2 presents an overview of the required modifications.

Using the Secure Logging Kernel, we ran the 100 containers and verified that they were able to finish their normal workloads, as defined in their official Dockerfiles, with, on average, less than 1% of runtime overhead. In doing so, the containers used less than 0.1% of the unpopular paths. A total number of 50 lines of unpopular paths were reached and recorded by the Secure Logging Kernel. This data strongly argues the feasibility of creating a popular-paths based host kernel to run Docker containers. In other words, real-world containers can run on only the popular paths in most (>99.9%) cases.

4.2 Security Evaluation

The main advantage of running containers on the popular paths is to leverage the security benefits. The popular paths have previously been proven to contain fewer security vulnerabilities. We want to verify if this still applies to containers in the face of new zero-day kernel vulnerabilities. The methodology we selected for this evaluation was to examine how many CVE kernel security bugs resided in the LinuxKit popular paths. If, as the metric suggests, there were fewer bugs in these paths, then our popular paths based solution would by its very design improve the security of running containers.

Experimental Setup. We obtained a list of all the available (at the time of our study) CVE kernel vulnerabilities for Linux kernel version 4.14.x from the National Vulnerability Database. For each of these 50 vulnerabilities, we looked at the kernel patch that fixed the bug to identify the source line numbers that correspond to it. We then compared the line numbers against the popular paths kernel trace we had for LinuxKit to verify if the bug was present.

Results. Out of the 50 CVE kernel security bugs we searched for, we found only three in the LinuxKit popular paths (lines occurred in any of the training runs), as shown in Table 3. These three bugs were in commonly used kernel code that, to the best of our knowledge, cannot be avoided by any existing security systems. We describe these three bugs in detail below to explain why they cannot be avoided, and to demonstrate that our popular paths based strategy did its best to reduce the risk of triggering kernel bugs.

[CVE-2018-15594] This bug lies in `arch/x86/kernel/paravirt.c` in the Linux kernel before 4.18.1. The source code mishandles certain indirect calls, which makes it easier for attackers to conduct Spectre-v2 attacks against paravirtual guests. In our tests, this bug was found in the kernel trace data, and was reached frequently by programs running in LinuxKit, because the code inside of the `paravirt_patch_call()` function in `arch/x86/kernel/paravirt.c` is used to rewrite an indirect call with a direct call, which is an essential function used to support the Linux kernel paravirtualization.

[CVE-2018-15572] The `spectre_v2_select_mitigation` function in `arch/x86/kernel/cpu/bugs.c` in the Linux kernel before 4.18.1 does not always fill RSB upon a context switch. This makes it easier for attackers to conduct user-space-userspace spectreRSB attacks. This piece of code in `spectre_v2_select_mitigation(void)` function would be used by every program, as the kernel attempts to mitigate potential Spectre attacks. Therefore any program that tries to mitigate the original Spectre attack would use this piece of code that contains this new CVE vulnerability. That's the reason why it appeared in our LinuxKit popular paths.

[CVE-2018-1120] This vulnerability was found affecting the Linux kernel before version 4.17. by `mmap()`ing a FUSE-backed file onto the memory of a process containing command line arguments (or environment strings). This bug appeared in the popular paths because `mmap()` is commonly used by user programs. Furthermore, this vulnerability involves `proc_pid_cmdline_read()` and `environ_read()` functions, that are commonly invoked by virtual machines and user programs.

By limiting the potential risk of exposure to kernel code to just these three bugs, the popular path metric already

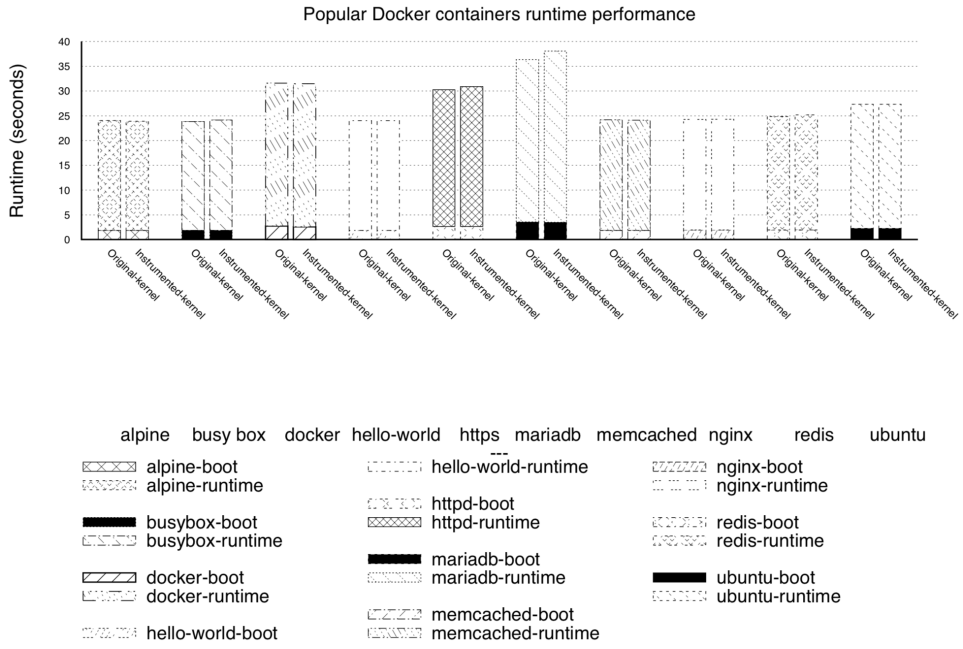

Figure 8. Runtime performance comparison for the top popular 10 containers

greatly reduces risks to container security. While a potential user of the popular paths metric must be aware of these bugs, the odds on having to deal with repeated and costly incidents from zero-day vulnerabilities become much more manageable.

It should also be noted that 49 out of the 50 bugs tested were discovered and confirmed after publication of the original popular paths study [27], which shows that the metric can indeed predict where bugs are likely to occur effectively.

4.3 Performance Evaluation

Lastly, we wanted to assure container users that utilizing our Secure Logging Kernel would not negatively affect performance overhead costs. To demonstrate this, we evaluated both the run-time performance and memory space overhead of the modified kernel.

Experimental Setup. In order to measure the overhead costs of our Secure Logging Kernel, we did a comparison between containers running on the original Linux kernel and ones running on our popular-paths based kernel. We used the exact same running environment and configuration to ensure a fair comparison. We ran the 100 most popular containers from Docker Hub on both the original kernel and our Secure Logging Kernel. In each test, the container finished its workload as defined in the official Dockerfile. We measured the runtimes for both kernels and compared them.

Results. Our results show that, on average, running containers on our popular-paths based kernel incurred about 0.5% to 1% of extra performance overhead as compared to the original Linux kernel. Results of the top 10 containers are shown in Figure 8.

The original Linux kernel image used for the LinuxKit test is sized at 163,012,008 bytes . In comparison, the popular-paths based kernel is sized at 163,622,440 bytes. Therefore, the extra memory space added by our modified kernel was only about 0.37%.

Based on our performance evaluation, the Secure Logging Kernel has negligible overhead in both runtime and memory space.

5 Limitations

While our results and analysis indicate that our dataset was representative of the workload users perform when using container applications, the sample size was somewhat limited and therefore, so was the data. Additional runs and a larger sample base could potentially capture the race-condition related kernel footprint more comprehensively, thus rendering more accurate popular paths data.

Another factor that could have affected our work was the type of images we used. We limited our selection to official container images from Docker Hub, yet there are a number of open source project images we did not access. The methodology used in this work run in these additional containers might have given us a broader spectrum of results.

Lastly, in this work, we ran our system and produced results with Linux kernel version 4.14.24. For future work, we plan to extend our system so that it can automatically work with different kernel versions. This will generalize the

application of our work, and potentially help a lot more users to secure their containers.

6 Related Work

In this paper, we seek to enhance the security of containers running on top of the host kernel by reducing exposure to the rarely-used risky code in the kernel. When developing our strategy, we consulted previous studies in three areas: 1) metrics for vulnerability prediction, 2) techniques for enhancing kernel security, and 3) approaches for enhancing container security.

Estimating and predicting vulnerabilities.

Metrics that can identify code most likely to contain vulnerabilities help developers and researchers prioritize their efforts to fix bugs and improve security. Chou et al. [21] looked at error rates in different parts of the operating system kernel, and found that device drivers had error rates up to seven times higher than the rest of the kernel. Ozment and Schechter [31] examined the age of code as a predictor of vulnerability in the OpenBSD [16] operating system, and generally confirmed the finding that the rate at which bugs are found goes down over time. In a recent prior work [27], Li et al. directly compared both of these metrics to the popular paths metric, and found that neither was as predictive of vulnerabilities in the Linux kernel.

Shin et al. [35] examined code churn, complexity, and developer activity metrics, finding that these metrics together could identify around 70% of known vulnerabilities in the two large open source projects codebase they studied. However, file granularity is too coarse to be useful when deciding which parts of the kernel need protection. Customizing the operating system kernel to work at the lines-of-code level, as the popular paths metric [27] does, can be more effective.

Operating System kernel security enhancement.

Prior research in enhancing kernel security requires refactoring or modifying the kernel. Engler et al. [24] introduced the Exokernel architecture, which allowed applications to directly manage hardware resources in the hope of gaining efficiency in accessing the hardware. This style of operating system design came to be known as a library OS. More recently, but in a similar spirit, unikernels, such as Mirage [29] run each application as its own operating system inside of a virtual machine, customizing the "kernel" for each application. Each of these systems, however, requires significant changes to existing applications (e.g., in the case of Mirage, applications must be rewritten in the OCaml programming language [15].) Other library OSes are written to run existing applications. Drawbridge [33] and Graphene [36] support unmodified Windows and Linux applications, respectively, by implementing an OS "personality" as a support library in each address space to improve security. However, user-space reimplementation of OS functionality incurs a performance overhead. Containers can mostly avoid this overhead, since they allow contained applications to make system calls directly. Thus, our work focused on the container-based approach.

Container security enhancement.

Existing security mechanisms available for containers usually leverage host kernel features. Namespaces [11] is one mechanism that can provide isolation between processes, and has been adopted by Docker. Linux capabilities [10] allow users to define and choose smaller groups of root privileges, and thus can reduce the number of entities that can exploit containers. Docker containers use Linux capabilities, and LinuxKit allows users to define and control the capabilities (such as file permissions) they want to use. Control groups [9], a key Linux feature, can be used to limit, account for, and isolate important system resources, such as memory, CPU, and disk I/O, to each container. More recently, various kernel hardening systems have been developed to provide access control mechanisms for better safety, such as AppArmor [3], SELinux [19], GRSEC [7], sand PAX [17]. GRSEC and PAX add safety checks at compile-time and run-time of the kernel. Docker ships a template that works with AppArmor, and also has SELinux policies working with Red Hat [18].

While these existing approaches do help improve security for containers, they also have limitations. First, it can be really hard to understand how to correctly configure the security settings for containers. In fact, many users just use the default configuration, which may not be the best choice if strong security is needed. Second, in order to run the applications a user wants, current systems often allow containers to access more code than they actually needed in the host kernel, including the potentially risky rarely used code. Our work improves the container security by imposing a fine-grained security monitor and control over access to the potentially risky kernel code.

7 Conclusion

In earlier work, we found that running applications using only frequently used popular paths could greatly improve the security of virtual machines. As a follow-up we wanted to test the feasibility of applying this popular paths metric to existing containers. We came up with a new strategy to enhance container security, leveraging the popular paths metric. First, our Secure Logging System obtains the popular paths of the LinuxKit virtual machine by profiling popular Docker containers, and then creates a Secure Logging Kernel that automatically logs an application's attempt to access any unpopular path. By using this reformulated kernel, we found out that less than 0.1% of the unpopular paths were accessed by the container applications tested. The result verifies that most of the workload can be managed without using the riskier paths. We also demonstrate that, consistent with previous findings, the popular paths in the host kernel do contain fewer security vulnerabilities, and thus

are inherently more secure. Given that hundreds of zero-day vulnerabilities are discovered every year, we believe that our popular paths based kernel will provide a practical solution for containers to avoid potential vulnerabilities, and thus enhance the security of running containers significantly.

References

[1] 2016. Dirty COW - (CVE-2016-5195) - Docker Container Escape. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/

[2] 2017. Dirty Cow vulnerability discovered in Android malware campaign for the first time. https://www.zdnet.com/article/dirty-cow-vulnerability-discovered-in-android-malware-campaign-for\-the-first-time/

[3] 2019. AppArmor. https://wiki.ubuntu.com/AppArmor

[4] 2019. Build EAR. https://github.com/rizsotto/Bear

[5] 2019. Clang LibTooling. https://clang.llvm.org/docs/LibTooling.html

[6] 2019. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html

[7] 2019. GRSEC. https://wiki.debian.org/grsecurity

[8] 2019. Lcov. http://ltp.sourceforge.net/coverage/lcov.php

[9] 2019. Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html

[10] 2019. Linux kernel capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html

[11] 2019. Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html

[12] 2019. Linux Testing Project (LTP). https://github.com/linux-test-project/ltp

[13] 2019. The LLVM Compiler Infrastructure Project. https://llvm.org

[14] 2019. LXC. https://linuxcontainers.org

[15] 2019. OCaml programming language. https://ocaml.org

[16] 2019. OpenBSD. https://www.openbsd.org

[17] 2019. PAX. https://pax.grsecurity.net

[18] 2019. Red Hat Linux. https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux

[19] 2019. SELinux. https://selinuxproject.org/page/Main_Page

[20] 451 Research 2017. Cloud-Enabling Technologies Market Monitor Report. Retrieved April, 2019 from https://451research.com/images/Marketing/press_releases/Application-container-market-will-reach-2-7bn-in-2020_final_graphic.pdf

[21] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An Empirical Study of Operating Systems Errors. In *Proceedings of the Eighteenth ACM Symposium on Operating Systems Principles (SOSP '01)*. ACM, New York, NY, USA, 73–88. https://doi.org/10.1145/502034.502042

[22] Docker 2019. Enterprise Container Platform for High Velocity Innovation. Retrieved April, 2019 from https://www.docker.com

[23] Docker Hub 2019. A Library and Community for Container Images. Retrieved April, 2019 from https://hub.docker.com

[24] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. 1995. Exokernel: An Operating System Architecture for Application-level Resource Management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/224056.224076

[25] Sergiu Gatlan. 2019. RunC Vulnerability Gives Attackers Root Access on Docker, Kubernetes Hosts. https://www.bleepingcomputer.com/news/security/runc-vulnerability-gives-attackers-root-access-on-docker-kubernetes\-hosts/

[26] Kubernetes 2019. An open-source system for automating deployment, scaling, and management of containerized applications. Retrieved April, 2019 from https://kubernetes.io

[27] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. 2017. Lock-in-Pop: Securing Privileged Operating System Kernels by Keeping on the Beaten Path. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC '17)*. USENIX Association, Santa Clara, CA, USA, 1–13.

[28] LinuxKit 2019. A toolkit for building secure, portable and lean operating systems for containers. Retrieved April, 2019 from https://github.com/linuxkit/linuxkit

[29] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[30] OS Level Virtualization 2019. Wikipedia. Retrieved April, 2019 from https://en.wikipedia.org/wiki/OS-level_virtualisation

[31] Andy Ozment and Stuart E. Schechter. 2006. Milk or Wine: Does Software Security Improve with Age?. In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15 (USENIX-SS'06)*. USENIX Association, Berkeley, CA, USA, Article 7. http://dl.acm.org/citation.cfm?id=1267336.1267343

[32] Vijay Pandurangan. 2016. Linux kernel bug delivers corrupt TCP/IP data to Mesos, Kubernetes, Docker containers. https://tech.vijayp.ca/linux-kernel-bug-delivers-corrupt-tcp-ip-data-to-mesos-kubernetes\-docker-containers-4986f88f7a19

[33] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. 2011. Rethinking the Library OS from the Top Down. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 291–304. https://doi.org/10.1145/1950365.1950399

[34] P. Rubens. 2017. What are containers and why do you need them? https://www.cio.com/article/2924995/what-are-containers-and-why-do-you-need-them.htm

[35] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. 37, 6 (Nov./Dec. 2011), 772–787. https://doi.org/10.1109/TSE.2010.81

[36] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. 2014. Cooperation and Security Isolation of Library OSes for Multi-process Applications. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*. ACM, New York, NY, USA, Article 9, 14 pages. https://doi.org/10.1145/2592798.2592812

[37] VirtualBox 2019. An x86 and AMD64/Intel64 virtualization. Retrieved April, 2019 from https://www.virtualbox.org

[38] VMWare Workstation 2019. Running multiple operating systems as virtual machines (VMs) on a single Linux or Windows PC. Retrieved April, 2019 from https://www.vmware.com/products/workstation-pro.html

[39] W.G. Wong 2016. What's the Difference Between Containers and Virtual Machines? Electronic Design. Retrieved April, 2019 from https://en.wikipedia.org/wiki/OS-level_virtualisation

Table 3. Evaluation of the CVE vulnerabilities for the Linux kernel

#	CVE ID	CVSS Score	Description	Detected in the LinuxKit Popular Paths
1	CVE-2019-10124	7.8	denial of service, in mm/memory-failure.c	✗
2	CVE-2019-9213	4.9	kernel NULL pointer dereferences, in mm/mmap.c	✗
3	CVE-2019-9003	7.8	use-after-free	✗
4	CVE-2019-8956	7.2	use-after-free	✗
5	CVE-2019-8912	7.2	use-after-free	✗
6	CVE-2019-7308	7.5	out-of-bounds speculation on pointer arithmetic	✗
7	CVE-2019-3701	7.1	privilege escalation	✗
8	CVE-2018-1000204	6.3	copy kernel heap pages to the userspace	✗
9	CVE-2018-1000200	4.9	NULL pointer dereference	✗
10	CVE-2018-1000026	6.8	denial of service	✗
11	CVE-2018-20511	2.1	privilege escalation	✗
12	CVE-2018-20169	7.2	mishandle size checks	✗
13	CVE-2018-18690	4.9	unchecked error condition	✗
14	CVE-2018-18445	7.2	out-of-bounds memory access	✗
15	CVE-2018-18281	4.6	improperly flush TLB before releasing pages	✗
16	CVE-2018-18021	3.6	denial of service	✗
17	CVE-2018-16862	2.1	the cleancache subsystem incorrectly clears an inode	✗
18	CVE-2018-16658	3.6	local attackers could read kernel memory	✗
19	CVE-2018-16276	7.2	privilege escalation	✗
20	CVE-2018-15594	2.1	spectre-v2 attacks against paravirtual guests	✓
21	CVE-2018-15572	2.1	userspace-userspace spectreRSB attacks	✓
22	CVE-2018-14646	4.9	NULL pointer dereference	✗
23	CVE-2018-14634	7.2	integer overflow, privilege escalation	✗
24	CVE-2018-14633	8.3	stack buffer overflow	✗
25	CVE-2018-14619	7.2	privilege escalation	✗
26	CVE-2018-13406	7.2	integer overflow	✗
27	CVE-2018-12904	4.4	privilege escalation	✗
28	CVE-2018-11508	2.1	local user could access kernel memory	✗
29	CVE-2018-11412	4.3	ext4 incorrectly allows external inodes for inline data	✗
30	CVE-2018-10940	4.9	incorrect bounds check allows kernel memory access	✗
31	CVE-2018-10881	4.9	denial of service	✗
32	CVE-2018-10880	7.1	denial of service	✗
33	CVE-2018-10879	6.1	use-after-free	✗
34	CVE-2018-10878	6.1	denial of service	✗
35	CVE-2018-10074	4.9	denial of service	✗
36	CVE-2018-10021	4.9	denial of service	✗
37	CVE-2018-8781	7.2	code execution in kernel space	✗
38	CVE-2018-6555	7.2	denial of service	✗
39	CVE-2018-6554	4.9	denial of service	✗
40	CVE-2018-5390	7.8	denial of service	✗
41	CVE-2018-1130	4.9	NULL pointer dereference	✗
42	CVE-2018-1120	3.5	denial of service	✓
43	CVE-2018-1118	2.1	kernel memory leakage	✗
44	CVE-2018-1068	7.2	write to kernel memory	✗
45	CVE-2017-1000410	5.0	leaking data in kernel address space	✗
46	CVE-2017-1000407	6.1	denial of service	✗
47	CVE-2017-1000405	6.9	overwrite read-only huge pages	✗
48	CVE-2017-18224	1.9	race condition, denial of service	✗
49	CVE-2017-18216	2.1	NULL pointer dereference, denial of service	✗
50	CVE-2015-5327	4.0	out-of-bounds memory read	✗