UnPAK: Using a New Approach to Kernel Tailoring to Take Action Against Risky Code Exposed to Containers

Abstract

One reason 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. Yet, most security measures designed for containers do not take into account how to protect against the zero-day kernel bugs inside the kernel itself. Containers are vulnerable to these bugs because they allow access to rarely executed paths where such vulnerabilities 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 the risk of triggering these vulnerabilities.

In this paper, we present a multi-step methodology for improving container security by leveraging data about popular paths into a kernel tailoring strategy. It starts with a systematic approach to identifying and capturing the popular paths data for widely-used container applications. By evaluating this data at different levels of granularity (line, function, and file), users can get fresh insights into which parts of the kernel are being used, and so make informed decisions about which code is safe to execute. The data also guided the implementation of a tailored kernel called the ***Un-Popular Action Kernel (UnPAK)***. **UnPAK** registers when a potentially risky path is reached and can respond with a number of actions, from issuing warning messages to denying execution of commands. When testing containers using UnPAK, we found that they were able to run their default workload using just the popular paths more than 99.9% of the time. Furthermore, the kernel was able to operate with only minor increments (0.1% on average) in runtime overhead, and only around 0.37% extra cost in memory space. Most importantly, since only 6% of 50 Linux kernel CVE security vulnerabilities we examined were present in the popular paths, utilizing the UnPAK system offers a way to reduce the risk of triggering bugs without sacrificing operational efficiency. Lastly, UnPAK compares favorably with three other current kernel tailoring strategies, as it is more adaptable to a wider variety of applications, and can reduce the largest amount of attack surface.

1 Introduction

Containers continue to grow in popularity, with one industry survey suggesting the technology could become a $2.7 billion market by 2020 [26]. This widespread adoption of containers, such as Docker [33] and LinuxKit [44], can be attributed to several factors, including portability and elimination of the need for a separate operating system [56]. In addition, the perceived isolation and security they provide to the programs running inside of them is reassuring to users looking to protect their data and operations. However, several recent incidents in which zero-day kernel bugs have been triggered from inside of a container [36, 51] suggest this perception may not be completely true. 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, allowed attackers to escape from a Docker container and access files on the host system [3]. 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 was being exploited in more than 1200 malicious Android apps, affecting users in at least 40 countries [5].

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 makes containers more efficient and lightweight to use—features that make them so attractive to end-users. Unfortunately, this access also exposes containers to zero-day bugs within the kernel itself [51]. And, with one OS kernel servicing a number of containers, it is common for its code to become bloated, a risk to both security and efficiency. While a number of research initiatives have looked at code debloating [37, 40, 54, 55], most of these efforts are not broadly applicable, and perform inconsistently, Furthermore, they do not eliminate bloat at the line of code level, nor can these approaches broadly identify which parts of the kernel are less likely to host zero day bugs. To improve security, the issue is not just reducing the total amount of code, but figuring out which parts of the kernel should be targeted in such a reduction.

Over the years, several researchers [30, 50] have proposed metrics that point to where buggy code might be within the kernel, and these strategies have provided some useful design guidance. One such study [43], released three years ago, suggested a powerful correlation between kernel lines accessed by widely-used programs and their likelihood to contain security flaws. Furthermore, the study [43] 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?

In this paper, we study the feasibility of using the popular paths metric to secure the LinuxKit container toolkit. The study required a number of steps, starting with finding a systematic way to gather data on popular paths, and affirming that this data is representative of hundreds of widely-used containers from Docker Hub [34]. By analyzing this data at three different levels of granularity—file, function, and lines of code—we found that each can provide data values that collectively form a hierarchy of security options from which users can choose, based on the security sensitivity of the application. A user can choose to simply analyze the data at the file level to quickly eliminate a significant portion of all security risks, or use data gathered at the lines of code level to obtain a broader picture of where bugs are located, and how to avoid them.

Using the raw popular paths data, we were also able to design the ***UnPopular Action Kernel (UnPAK)***, a modified version of the Linux kernel which is tailored to register any attempt to access infrequently used paths. Developing and testing UnPAK contributed to our study in two ways. Firstly, it provided usable data on how often applications accessed risky paths, so we could determine how essential such access is for their functionality. And, secondly, it offered a chance to program a number of responses into the operations of the kernel, such as a warning system. While initially UnPAK was instrumented to warn users of potentially dangerous behaviors so they can make informed decisions on whether or not to execute it, it could also be programmed for other actions, such as automatically refusing such an execution.

When tested at all three of the granularity levels mentioned above, we confirmed that UnPAK can effectively prevent most kernel bugs from being triggered when running Docker containers. Only three of the 50 CVE kernel vulnerabilities checked for were found in those LinuxKit paths. Even at the file level, our tests showed that security can be improved by simply removing unloaded / unused drivers, and unused architectures, as more than half of the kernel bugs reside in these files. Better still, an evaluation of our findings affirmed that frequently used Docker containers experience no loss of functionality when using just the popular paths. Indeed, we found these paths were used more than 99.9% of the time to run the official Docker containers' default workload. In addition, UnPAK compared favorably as a debloating strategy when run against instantiations of three current kernel tailoring strategies [8, 28, 42]. It reduced the largest amount of attack surface while offering similar runtime performance. UnPAK also offers more flexibility, as many of the existing strategies are tied to a specific application, or to addressing a particular subset of bugs.

Lastly, in our performance evaluation, we found that running UnPAK as currently designed only incurred about 1% of runtime overhead, while the memory space overhead was only about 0.37%. Thus, reduced exposure to bugs could be achieved with very little perceived difference in operation efficiency or cost.

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

- We develop a methodology to systematically identify and capture the "popular paths" data for widely-used container applications, and verify that the technique works for the most downloaded Docker containers run inside of the LinuxKit VM.
- We evaluate popular paths data at different levels of granularity (line, function, and file) and find that, at each level, following the popular paths metric can provide opportunities to identify and eliminate kernel vulnerabilities.
- We use the popular paths data to design and implement UnPAK, a modified version of the Linux kernel. This instrumented kernel logs attempts to trigger unpopular paths and can respond with a number of defensive actions, from sending warning messages to denying execution of the program.
- We demonstrate that frequently-used Docker containers were able to run their default workload using the popular paths more than 99.9% of the time, with only a negligible (less than 1%) performance overhead.
- We compare the UnPAK strategy to three other kernel tailoring approaches and find that, as currently configured, it can reduce the largest amount of attack surface, and is more adaptable to a wider variety of applications.

2 Background and Motivation

To understand the central hypothesis of our work, that containers can be designed to run with only limited access to the OS kernel, it is important to understand the design idea on which it was built. Various metrics have been proposed over the last few decades as a way to predict where bugs may lie. Chou et al. [30] focused on a particular part of the code, the device drivers, and argued that this component was the most likely place to find bugs. Ozment and Schechter [50] examined the age of code to see if that worked as a predictor of vulnerability in the OpenBSD [22]. Other researchers have also considered software engineering elements, such as the complexity of the code or how often a developer rewrites the code, as factors tied to bugs [27, 31, 39, 57, 61]. In terms of our current research, none of the above options have explored the feasibility of reducing the attack surface of the host OS kernel for containers. To enhance container security effectively, we need a security metric that could help identify where zero-day vulnerabilities are located in the kernel.

The Popular Paths metric described in [43] serves as a solid basis for us to protect containers from a vulnerable host OS kernel. It takes a quantitative approach, and evaluates security at the fine-grained line-of-code level. It posits that lines of code in the kernel used to run popular software programs are less likely to contain security bugs. The study involved asking subjects to run daily tasks, such as writing, spell checking, printing in a text editor, sending and receiving emails, and using an image processing program, on applications from the 50 most popular packages in Debian 7.0. Tests were conducted over a period of 5 calendar days for 20 hours of total use. In addition, the authors also had students use the workstation as their desktop machine for a one-week period to do their homework, develop software, communicate with friends and family, and so on. These combined experiments documented that only 2.5% of zero-day bugs were found in these popular paths, demonstrating that this is indeed the safest part of the kernel.

In addition, the study documented in [43] that in practice, it is possible to design and build a virtualization system that can run large and complex legacy programs using only the popular paths. The system instantiated in the study was tested on Tor and Apache, as well as on widely used tools, such as GNU Grep, GNU Wget, GNU Coreutils, GNU Netcat, and K&R Cat. Results from the paper [43] shows that the incurred performance overhead was around 2.5x to 5x, or about the same magnitude as existing state-of-the-art library OS systems.

Most significantly, the study [43] showed that leveraging the popular paths metric to design and build new architecture can lead to a more secure virtualization environment. This newer work explores the feasibility of utilizing this metric on an existing container system to improve security, and provide a solid standard for container security evaluation.

3 Design and Implementation

3.1 Design

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 ***UnPopular Action Kernel (UnPAK) system***, shown in Figure 1. In this design, various security actions could be placed to guard the unpopular paths. In this work, we implemented a kernel logging system to provide security warnings as one instance of the potential actions to protect the system (shown in Figure 2).

Identifying the popular paths for a given container system is handled by the system module we refer to as the Profiler. The Profiler sets out a map highlighting places in the kernel that are safe for an application to access. It does so by identifying the lines of code executed in the host kernel while running its default / regular workload, as defined in the configuration files (Dockerfiles) that come with the container images. We collected these lines of code, or the "popular paths kernel traces," as we labeled them, from a set of the containers we ranked as "most popular" based on the number of user downloads.

The second module, the Modifier, uses the map of safe access locations identified by the Profiler to find and adapt the rarely used paths to produce a tailored kernel, called the UnPAK. These risky paths are flagged to either generate security warning messages / logs, or to block code execution. (The process used to rework the kernel is described in more detail in the implementation section). The UnPAK can then be used to run containers without any required changes to the containers themselves.

3.2 Implementation

This section describes in detail how the modules in our system were implemented.

3.2.1 Implementing the Profiler

The Profiler is designed to collect the kernel trace of running user containers. The kernel trace refers to the lines of code in the underlying host operating system kernel that were executed when running the regular container workload. The process used to obtain the kernel trace can be broken down into the following steps:

1. The Linux kernel used to run the Docker containers is recompiled with the Gcov [13] kernel profiling feature enabled.
2. The Profiler 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 3.)
3. The LinuxKit virtual machine is booted and a script in the profiler automatically starts running the workload of the task container. The profiler collects the kernel trace data, in the form of gcda files generated by Gcov, stores them at `/sys/kernel/debug/gcov/`, and, by the end of the run, transfers the data to the host system for further use.
4. The profiler uses the lcov [15] tool to process the kernel trace data collected from the host system, and generates formatted data about which lines of code were executed in the kernel source files. This data will pinpoint which parts of the kernel are potentially risky and need

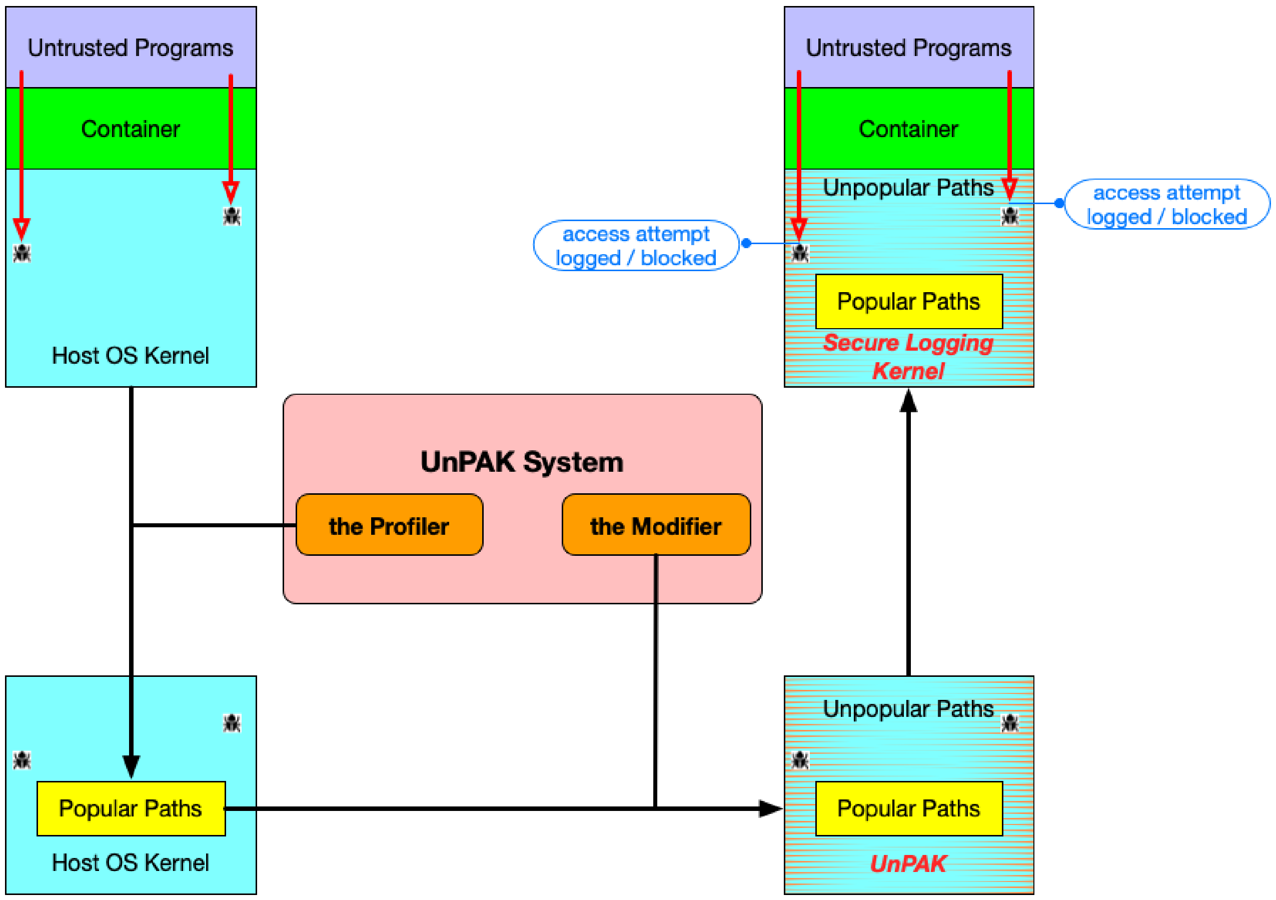
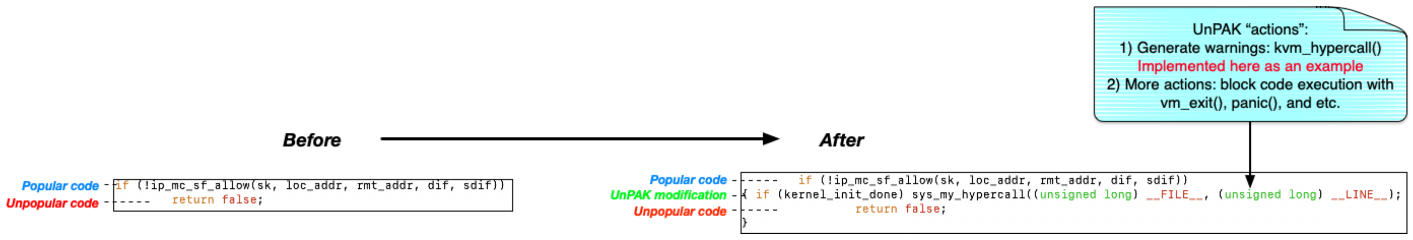


Figure 1. Design overview of the UnPAK system


UnPAK "actions":
1) Generate warnings: kvm_hypercall()
Implemented here as an example
2) More actions: block code execution with vm_exit(), panic(), and etc.
Before
After
Popular code
Unpopular code
Popular code
UnPAK modification
Unpopular code

```
if (!ip_mc_sf_allow(sk, loc_addr, rmt_addr, dif, sdif))
        return false;
```

```
        if (!ip_mc_sf_allow(sk, loc_addr, rmt_addr, dif, sdif))
{ if (kernel_init_done) sys_my_hypercall((unsigned long) __FILE__, (unsigned long) __LINE__);
                return false;
}
```

Figure 2. Example of UnPAK actions

to be marked. These sections will be instrumented by the Modifier to trigger an alert (or potentially another defensive action) if an application attempts to execute code in that path.

3.2.2 Implementing the Modifier

The Modifier has three main parts: the Clang compiler, the Clang analyzer, and the kernel instrumenting tool (shown in Figure 4). Its function is to identify the correct places in the source files to instrument, based on data extracted from the Profiler, and then to modify the code as needed.

The modifier operates in the following way:

1. The Clang C compiler from the Clang / LLVM project [20], along with a BEAR [11] tool compiles 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 will be used by our Clang analyzer to perform source code parsing in the next step.
2. The Clang analyzer performs static analysis on the kernel source code to obtain the control flow graph for each function in the files. The analyzer leverages Clang's LibTooling [12] to construct the AST tree from the kernel source code, and then obtain the control

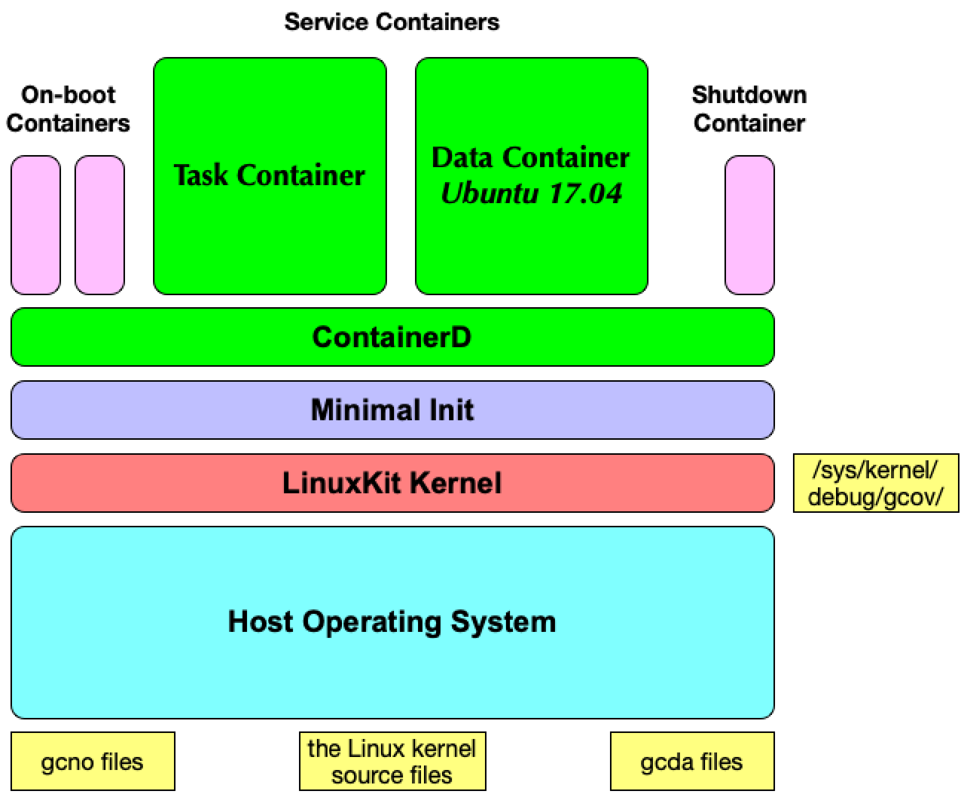


Figure 3. Setting up Gcov profiler in LinuxKit

Table 1. Instrumentation code added to UnPAK

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**

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 where it begins in order to alert users of any attempt to reach that code. This approach can optimize instrumentation by avoiding redundancy in code injection.

3. The instrumenting tool executes modifications based on both the basic blocks, and the collected popular paths data. The algorithm for kernel modification directs the user through the basic blocks in the kernel source files. If none of the lines in a block are in the popular paths data, the block is considered unpopular , and the instrumenting tool will add the security logging code, a kvm hypercall from the LinuxKit kernel into the host Linux kernel (as shown in Figure 5) at the beginning. 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. These markers can guarantee minimal effect on the LinuxKit kernel functionality, while still being able to generate secure logging whenever unpopular paths are reached.
4. The Modifier automatically inserts secure logging code at the beginning of the unpopular paths in the Linux kernel source files and then recompiles the kernel to produce UnPAK. For non-popular functions that contained zero lines of popular code, we inserted one line

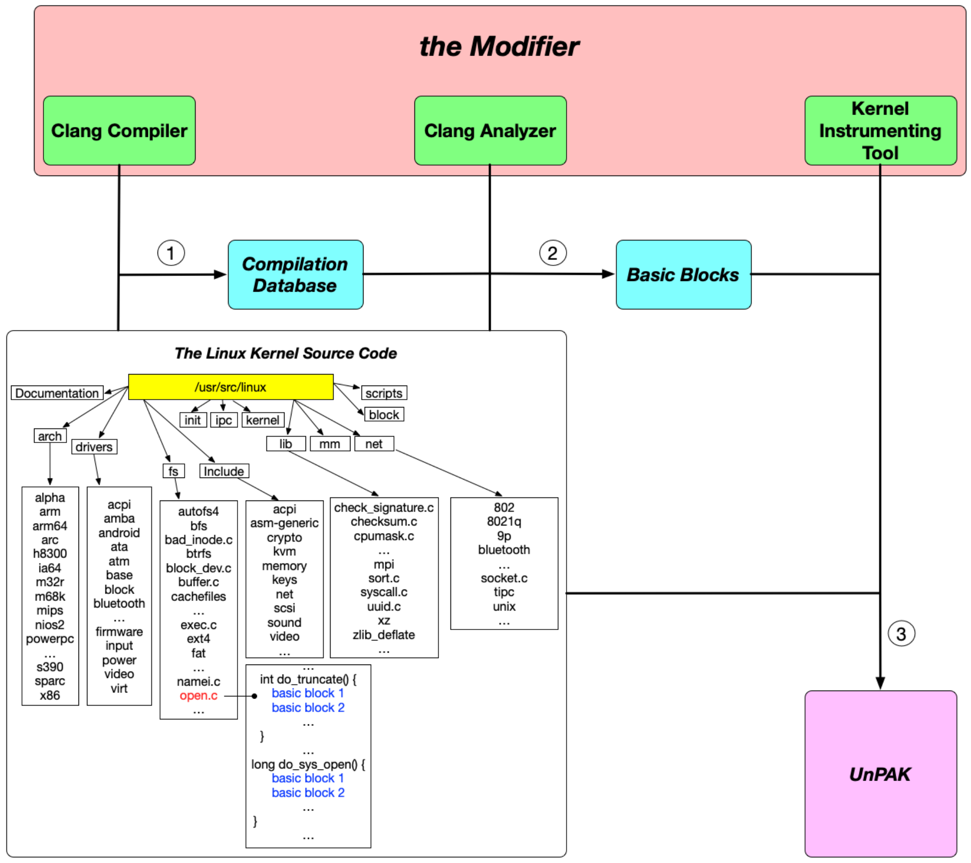

the Modifier
Clang Compiler
Clang Analyzer
Kernel Instrumenting Tool
1
Compilation Database
2
Basic Blocks
3
UnPAK
The Linux Kernel Source Code
/usr/src/linux
Documentation
scripts
block
init
ipc
kernel
arch
drivers
lib
mm
net
fs
Include
alpha arm arm64 arc h8300 ia64 m32r m68k mips nios2 powerpc ... s390 sparc x86
acpi amba android ata atm base block bluetooth ... firmware input power video virt
autofs4 bfs bad_inode.c btrfs block_dev.c buffer.c cachefiles ... exec.c ext4 fat ... namei.c open.c ...
acpi asm-generic crypto kvm memory keys net scsi sound video ...
check_signature.c checksum.c cpumask.c ... mpi sort.c syscall.c uuid.c xz zlib_deflate ...
802 8021q 9p bluetooth ... socket.c tipc unix ...
int do_truncate() { basic block 1 basic block 2 ... } ... long do_sys_open() { basic block 1 basic block 2 ... } ...

Figure 4. Implementation of the Modifier

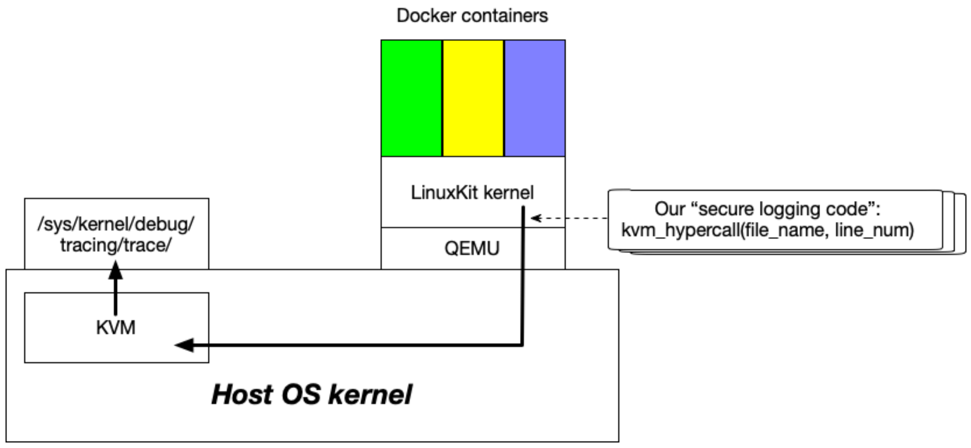

Docker containers
LinuxKit kernel
QEMU
Our "secure logging code": kvm_hypercall(file_name, line_num)
/sys/kernel/debug/tracing/trace/
KVM
Host OS kernel

Figure 5. the KVM_HYPERCALL implementation

of warning using the kvm hypercall at the beginning of each function. For popular functions, which have at least one line of popular code, we took the fine-grained approach to insert one line of warning code before each basic block that was unpopular. This tailored kernel can then be used to run Docker containers in the LinuxKit VM.

Table 1 presents an overview of the number of lines we inserted as security "warnings" in our kernel logging version implementation of UnPAK.

4 Evaluation

To demonstrate that our popular paths based approach is practical as well as effective in improving container security, we conducted a series of tests to answer the following questions:

- Can we obtain the popular paths data for containers in a systematic way? (Section 4.1)
- Can an analysis of the popular paths data for containers help developers and researchers achieve better security? (Section 4.2)
- Can real-world containers run only on the popular paths with no loss of functionality? (Section 4.3)
- What is the performance overhead of using UnPAK, as opposed to using the original Linux kernel? (Section 4.4)
- How does this work compare against current kernel tailoring methods in reducing the attack surface? (Section 4.5)

4.1 Can popular paths data for containers be collected in a systematic way?

Having a systematic way to obtain the popular paths data for containers is key to making its use scalable, reproducible, and accessible to developers and the research community. Therefore, the first step in our investigation was developing and testing such a standard procedure for containers. This procedure can be divided into three stages: selecting the **dataset**, assembling **a profiling framework**, and conducting the **experiment**.

The dataset: The dataset stage involved selecting the containers to use and the workload that would run inside of them. We wanted the defined dataset to be representative of the containers used by millions of clients, so we sourced a set accepted as widely used according to trustworthy real-world statistics. We selected 50 containers deemed most popular on Docker Hub according to its official number of pulls. Each selected container had more than 10 million pulls [34]. The workload run by each was the set of commands or operations defined in the official Dockerfile for each container image.

The profiling framework: This next stage involved establishing a running environment and capturing the actual data. The procedure required a robust framework, and ideally we wanted it to be portable so it could be used across different platforms. We selected the LinuxKit VM, which works with Linux, Windows, and Mac OS, and a Linux kernel with a Gcov profiling tool enabled to run the Docker containers. This framework works with other Linux kernel versions, which makes the procedure more widely applicable.

The experiment: The test itself consisted of running the containers from the dataset with their defined workloads in the profiling framework, and extracting the resulting data. The procedure automated the experimental runs in a way that allowed users to define key variables of the input, such as the exact version of the container image to use, and the number of iterations to run for each container. This design choice makes our procedure flexible, and better able to accommodate users with different needs and computing resources. In our experiment, we decided to do 10 iterations, because it provides a sufficient number of runs to make the data reliable while not consuming too much time and computing resources.

4.2 What security benefits can be obtained from using the popular paths data?

Using the procedures described above, we obtained data for some of the most frequently accessed containers on Docker Hub, and used it to check for security vulnerabilities. We started with a list of all 50 of the CVE kernel vulnerabilities for Linux kernel version 4.14.x that were available from the National Vulnerability Database when our study was conducted (July 2019) [49] and identified the kernel patch that fixed the bug. This allowed us to identify the source line numbers, functions and files that corresponded to it, and use this information to highlight where these bugs were located in the kernel.

It should be noted that 49 out of the 50 bugs tested in this experiment were discovered and confirmed after publication of the original popular paths study [43]. This shows that the metric can indeed effectively predict where bugs are likely to occur.

The raw data generated by Gcov in our profiling framework was at the line-of-code level, as this is the level at which our initial popular path study was done. We then processed it to generate additional information at the function and file level. These three different levels of granularity provide a full map of how the kernel code was used by containers and where the vulnerabilities might lie in each.

Popular paths data at the file level

For developers and security teams, the first thing to identify is which files are popular, as this can be a first step to secure the kernel without needing to deal with the complex logic and branches inside of each file. We deemed a file as "popular" if any line of code inside of it was in the raw popular paths data we obtained.

From our data (shown in Table 2), we can see that the all-popular files, or files in which every line is popular, contain no bugs and thus are the safest ones to use. Alternatively, the unpopular files contain more than half of the bugs. The higher bug density of the unpopular files suggests that removing these files could be an easy first step to securing the kernel.

The files designated as "partially popular," which accounted for about 30% of the bugs investigated, present a trickier problem to resolve. The presence of some popular lines means these items are being used and can not simply be removed. If stricter security requirements are desired, we need to go

Table 2. CVE bugs located in Unpopular and Popular kernel files

kernel dir	unpop (CVEs)	pop (CVEs)	all-pop (CVEs)	partial-pop (CVEs)	total
arch	133(3)	272(1)	81	191(1)	405
block	10	44	0	44	54
crypto	53(1)	89(2)	0	89(2)	142
drivers	161(11)	543(3)	5	538(3)	704
fs	52(6)	176(2)	5	171(2)	228
ipc	1	9	0	9	10
kernel	40(2)	183	2	181	223
mm	16(1)	59(4)	0	59(4)	75
net	266(5)	482(3)	0	482(3)	748
security	6	17	0	17	23
total	738(29)	1874(15)	93	1781(15)	2612
percent	28.3% (66%)	71.7% (34%)	3.6% (0%)	68.1% (34%)	100%

unpop: *contain 0 line of popular code.*
all-pop: *every line is popular code.*
partial-pop: *contain at least 1 line of popular code and 1 line of unpopular code.*
pop: *all-pop + partial-pop.*

Table 3. CVE bugs located in Unpopular and Popular kernel functions

kernel dir	unpop (CVEs)	pop (CVEs)	all-pop (CVEs)	partial-pop (CVEs)	total
arch	1528(7)	968(1)	312	656(1)	2496
block	777	264	68	196	1041
crypto	527(6)	121	36	85	648
drivers	6258(20)	2640	562	2078	8898
fs	1888(13)	1653(3)	559	1094(3)	3541
ipc	138	102	33	69	240
kernel	3562(8)	2280	793	1487	5842
mm	976(2)	889(6)	281	608(6)	1865
net	3703(9)	2405(3)	695(1)	1710(2)	6108
security	179	201	104	97	380
total	19536(65)	11523(13)	3443(1)	8080(12)	31059
percent	62.9% (83.3%)	37.1% (16.7%)	11.1% (1.3%)	26% (15.4%)	100%

inside of these files to look at a finer granularity, which is the function level.

Popular paths data at the function level

Using the raw data, shown in Table 3, we pinpoint which file functions were used by popular containers. Our data shows that functions deemed unpopular contain 5x more bugs than popular functions, while the all-popular functions (every line in the function is popular) contain only one bug.

As with the file analysis above, having this information can help security teams eliminate some risks by just avoiding the less-used functions. Unfortunately, that still leaves the partially-popular functions, which contain 12 bugs. As with the partially-popular files, the presence of some frequently used functions in this group means they cannot simply be removed at this level. While analyzing each function at the line-of-code level is the most labor intensive of the methods elaborated here, it offers the best opportunity to remove risky code. Below we look at vulnerabilities revealed by the popular paths data at this most fine-grained level.

Popular paths data at the line level

An analysis of the gathered data at the line level shows that the popular lines make up about 20% of the kernel codebase, and that only 6% (or three) of the CVE bugs were detected in those lines (Table 4.) 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 more detail below.

It is important, to note, however, that two of these bugs (CVE-2018-15594 and CVE-2018-15572) are Spectre-related hardware vulnerabilities, based on fundamental flaws in CPU's data cache and speculative execution. These flaws affect many modern processors [69], and software tools and compartmentalization techniques are not designed to prevent them. The only software-related bug is CVE-2018-1120, which is related to the `mmap()` system call, and regularly accessed by virtualization systems.

[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, making 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. This occurs 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, an essential function used to support 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 userspace-userspace spectreRSB attacks. As this piece of code in the `spectre_v2_select_mitigation(void)` function would be used by every program as the kernel attempts to mitigate potential Spectre attacks, it explains 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). It appears in the popular paths because `mmap()` is a commonly used system call. Furthermore,

Table 4. Evaluation of the CVE vulnerabilities at the line level

#	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	✗

this vulnerability involves `proc_pid_cmdline_read()` and `environ_read()` functions that are commonly invoked by virtual machines and user programs. According to CVSS 3.0 [32] metrics, this bug has "high" attack complexity, suggesting that it is very hard to exploit it in practice. Therefore, the negative impact of this bug's existence is low.

While a potential user of the popular paths metric must be aware of these bugs, limiting the risks to just these three greatly reduces the odds of having to deal with repeated and costly incidents from zero-day vulnerabilities. In addition, the actions instrumented into UnPAK were able to catch any attempt to reach these bugs, giving users alerts and options to terminate or block the risky programs trying to trigger the bugs.

An important result of this multi-level data analysis, was the discovery that users actually had a hierarchy of security choices, rather than one "all or nothing" option. Based on the sensitivity of the application and the resources available, users can actually choose the appropriate tradeoff of effort vs.security appropriate for them. File level information helps identify unused portions of the kernel at a glance, and can eliminate a large number of kernel vulnerabilities in a quick first step, with little expenditure of time and effort. The function level information enlarges the scope by focusing on less-used functions and suggesting the value of eliminating the bugs inside these functions. Less security sensitive applications could consider stopping at either one of these first two levels. But, for those applications where security is critical, moving on to the line level will give the most accurate information about where bugs are located and the most precise instructions for modifying the kernel.

Are there sufficient commonalities across different containers to form a valid dataset for the most frequently used containers?

To ensure our findings could be applicable to a broad variety of containers, we needed to establish that the data we found shared common ground in their kernel footprints. Were there enough commonalities to form a realistic popular paths dataset to support multiple popular containers in real practice? If so, we could use the kernel trace, or the record of all kernel code executed, from just a sample of containers to represent the trace of many more.

To answer this question, 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, point 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 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 frequently used Docker containers, we identified eight pieces of infrequently executed kernel code common among them. These traces are presented graphically in Figure 7.

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, despite the fact that they are infrequently executed, this code performs essential kernel functions and is used by multiple containers. Therefore, they should still be considered popular paths. On closer examination, we learn they are not always executed during each run because, as race-condition related codes, they depend on locks, and system conditions that may vary during different runs. Being able to capture these examples of infrequently-executed code and include them into our popular paths data is important, because it makes our UnPAK more reliable.

4.3 Can real-world containers run only on the popular paths with no loss of functionality?

Based on the popular paths data, UnPAK can initiate multiple types of actions to enhance container security. While our initial tests documented here focused just on logging and warning, UnPAK actions could also include kernel panic, VM (Virtual Machine) exit, and code removal. Each of these actions has its own pros and cons. In this section we first

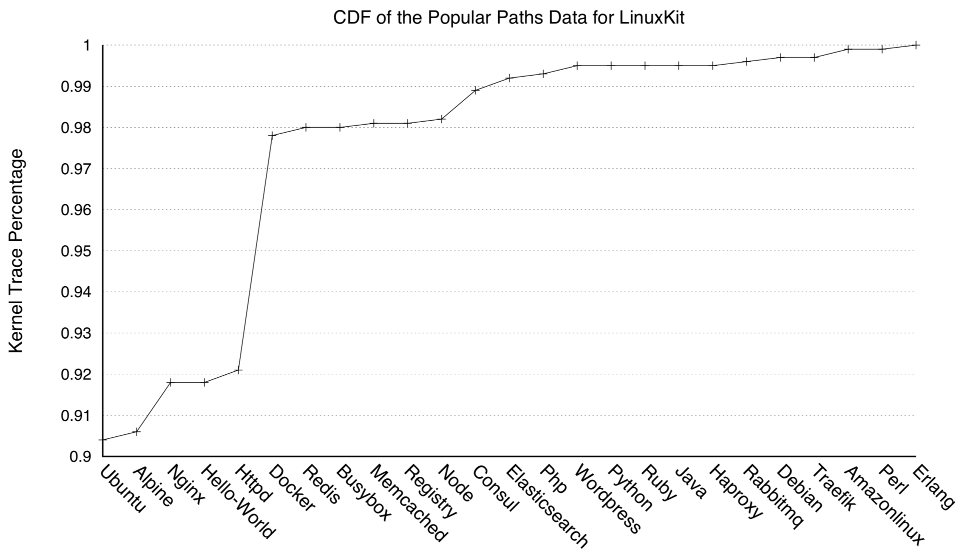


Figure 6. CDF of the popular paths of Docker containers showing most share the same kernel footprint

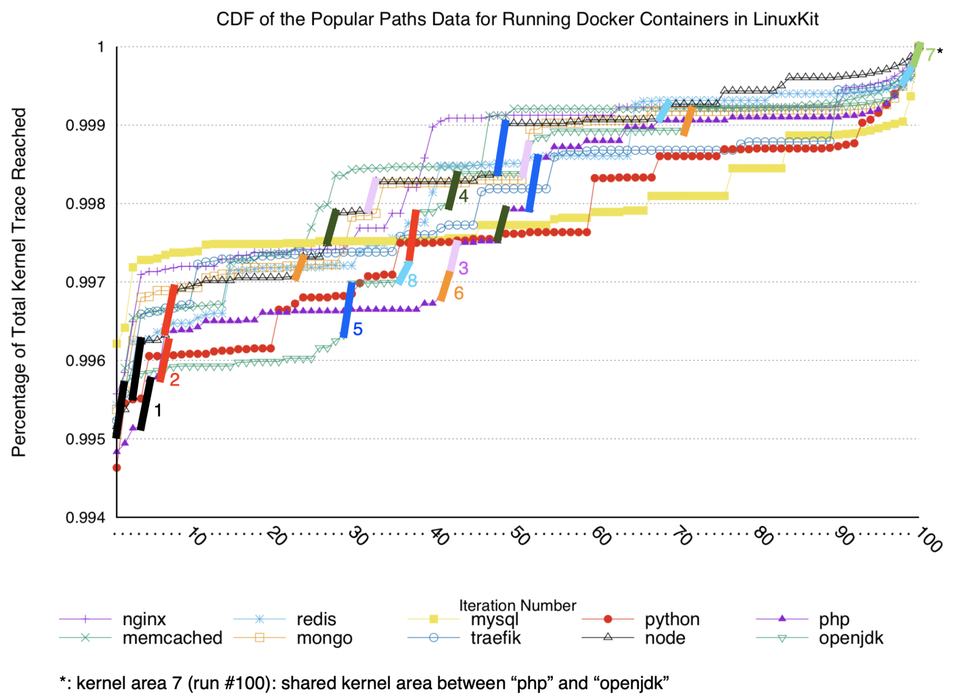


Figure 7. Common kernel code identified across different containers

present the results of functionality tests on UnPAK's current instantiation, and then discuss alternative actions, and how they might affect operations

4.3.1 Functionality evaluation of UnPAK with logging and warning actions

The next set of tests were to determine if running applications in UnPAK, as currently instantiated, will cause any loss of functionality, as compared to using the original Linux kernel. The first study used the Linux Testing Project (LTP) [19] test suites—an open source test project designed to validate the reliability, robustness, and stability of Linux— to verify that UnPAK functions as anticipated. The second test involved running a collection of the most popular containers from Docker Hub to verify if they were able to function correctly if they accessed only the popular paths.

Table 5. LTP test results

	Original Linux kernel	UnPAK
Expected Success	673	673
Expected Failures	105	105
Skipped	20	20
Total	798	798

Testing functionality with the LTP test suites
Experimental Setup: We used the Dockerfile provided by LinuxKit [44] 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 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 served as the local hypervisor.

Results: UnPAK was able to run all of the 798 test cases, and as shown in Table 5, produced the same results as the original unmodified Linux kernel. Among all these tests, 673 of them generated "success" as their expected output on both kernels. We observed 105 test cases returned "failures" as expected, mainly because of restrictions imposed by LinuxKit. For example, `mem01` test failed because the malloc function failed to allocate 3056 MB memory due to the default memory restriction; the `gf01` test failed due to "no space left on device" in LinuxKit. the `swapon01` test failed due to swapfile not available, and more. There were 20 tests skipped due to certain required functions being unavailable in the LinuxKit VM version we tested. For example, the swap file was not accessible in LinuxKit, therefore the related `swapoff` test iterations were skipped intentionally.

In summary, UnPAK yields the exact same output as the unmodified Linux kernel when running the 798 test cases in the LTP test suites. This shows that our logging implementation, using kvm hypercalls, can guarantee functionality correctness. It should be noted that a naive approach of logging using the kernel's `printk()` function did not work properly, because loops in some test cases will invoke a large number of `printk()` calls and result in a time-out error. For example, the `msgctl08` test case would invoke massive repeated `printk()` calls inside `ipc/util.c` and `ipc/msg.c` files. In addition to providing the correct functionality, UnPAK was able to capture and record all the unpopular paths accessed by the LTP test suites. This provides data that can help users understand the kernel footprint of different test cases.

Testing functionality on real-world container applications
Experimental Setup: To confirm that UnPAK could run applications in real-world practice, we identified the 100 most downloaded containers from Docker Hub, and ran them in the same version of the LinuxKit virtual machine used in the test described in 4.3.1. Each container was run in the LinuxKit VM using the commands from its official Docker image. To take into account any potential variances, each container was run 10 times in the exact same environment.

Results: We verified that the 100 containers run using UnPAK were able to finish their normal workloads with, on average, less than 1% of runtime overhead. In doing so, the containers used less than 0.1% of the unpopular paths. As this data proves real-world containers can run on only the popular paths in most (>99.9%) cases, it strongly argues the feasibility of creating a popular-paths based host kernel to run Docker containers.

4.3.2 Discussion on alternative actions

We implemented a version of UnPAK with logging and warning capabilities as one instance of the type of actions this tailored kernel could perform. While we established in this study that logging and generating security warnings via UnPAK worked with real-world containers, we believe the kernel could be instrumented to take other actions to improve security. In this section, we discuss three alternative actions for UnPAK: kernel panic, VM (Virtual Machine) exit, and code removal.

Inserting the kernel `panic()` function in front of each unpopular code block could prevent containers from triggering zero-day bugs in the risky unpopular paths. In this modification, whenever a container attempts to reach unpopular paths, the host kernel will "panic" and stop the execution. We tried out this strategy and found it was not ideal. We tried adding `panic()` calls at locations related to existing CVE bugs, and verified that the kernel did successfully prevent the bug from being triggered. However, adding a `panic()` call to every unpopular path will incorrectly render the kernel function. Some kernel code under `init/` directory is required during the boot stage, and cannot be modified with `panic()` calls. In addition, the kernel code needed by the `panic()` call itself cannot be instrumented without breaking the function. As causing the host kernel to panic is not an elegant way to end programs, and may not be suitable for many users, this strategy is not a practical solution.

A better way to stop code execution on risky unpopular paths is to invoke a VM exit, since this gives users more control over what to do next and avoids a kernel crash. With our current UnPAK implementation, we can directly deploy the VM exit action if desired. With the kernel logs provided by UnPAK, we can simply run a script from the host to monitor and check if there is any log entry showing an attempt to reach unpopular paths. If so, the script just needs to issue a command to let the LinuxKit VM shutdown. The disadvantage with this is that we would have to shut down the entire VM and every container running inside. If there are multiple containers running at the same time and only one of them exhibits suspicious behavior, we do not at this time have a reliable way to figure out which container was trying to reach the unpopular paths. But, it may be possible to infer that information from the memory. This would require a bit more effort, but would be an interesting future work project.

Lastly, removing code on unpopular paths would be the most direct action, but this solution lacks flexibility. One possible action is to remove all the unpopular paths from the source code, which could cause trouble if some of it was needed by specific programs or in rare situations the kernel has to handle. Some unpopular code may also be needed by kernel internal functions, so it might not be a good idea to simply remove it. We have not implemented this action yet, but are interested in exploring it to see how well it works.

4.4 Performance Evaluation

Adoption of UnPAK is dependent on its real-world viability. So, the next round of tests were to ensure it 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: We compared data from containers running on the original Linux kernel and on the popular-paths based kernel and measured the overhead costs. We used the exact same running environment and configuration to ensure a fair comparison. In each test, the container finished its workload as defined in the official Dockerfile. We then measured the runtimes for both kernels and compared them.

Results: On average, the results show running containers on UnPAK incurred about 0.5% to 1% of extra performance overhead, as compared to the original 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, UnPAK is sized at 163,622,440 bytes. Thus the extra memory space added by our modified kernel was only about 0.37%. In addition, for both runtime and memory space UnPAK has negligible overhead.

4.5 Comparison with existing kernel tailoring approaches

Lastly, we needed to compare our kernel tailoring approach with similar strategies dedicated to eliminating inefficient and unnecessary code that can lead to vulnerabilities. We selected three current kernel tailoring strategies and examined how our popular paths metric stacked up in terms of flexibility of application, ability to reduce the size of the attack surface, and other security and functionality issues. To do so, we utilized both published data and results of replication tests on available datasets.

Table 6. Existing Kernel Tailoring Approaches

Work	Main Goals	Main Approach
UnPAK (this paper)	Reduce the kernel attack surface for a container application	Dynamic analysis / profiling + source code modification
Alharhi et al. [28]	Reduce the number of reported kernel CVE bugs	Tailoring the kernel configuration based on vulnerability dependencies
Kurmus et al. [42]	Reduce the kernel attack surface for a given user program	Kernel automatically configured based on specific user application workloads
Kang et al. [8]	Reduce the kernel attack surface	Automatic kernel configuration based on profiling and fixing configs for boot

4.5.1 Goals and approaches of current strategies

While all three works propose kernel tailoring strategies similar in some ways to our own, the main goals of these strategies and their implementations vary. We start by summarizing these similarities and differences, as shown in Table 6.

The work of Alharhi et al. [28] seeks to improve security by eliminating a selected group of reported CVE kernel bugs. Using a static-analysis strategy, this research team aims to establish a dependency between the CVE kernel bugs and particular kernel configuration options. Based on these dependencies, they were able to tailor the kernel to reduce the number of CVE bugs exposed . The key difference between their approach and the work presented in this paper is that Alharhi et al. explicitly designed their kernel tailoring strategy to find existing CVE bugs. As such, its usefulness may expire as new bugs are added. One advantage of our approach is that it is effective in predicting future bugs. In fact, in the evaluation above (Section 4.2), we have shown that the popular paths approach was able to help predict and avoid 94% of the zero-day bugs reported after the metric was published [43].

Alternatively, Kurmus et al. [42], and Kang et al. [8] both aimed to reduce the kernel attack surface, but utilized different techniques. Kurmus et al. modified its kernel configuration by disabling certain options and thus removing unneeded code. The researchers used profiling and dynamic analysis to identify which options to disable. Later, Kang et al. [8], also attempted to reduce the attack surface, while making the kernel perform with greater stability. Their work began with an existing tool called undertaker-tailor [42], which was not very stable and, in some cases, produced kernel images that could not boot or function properly. By fixing

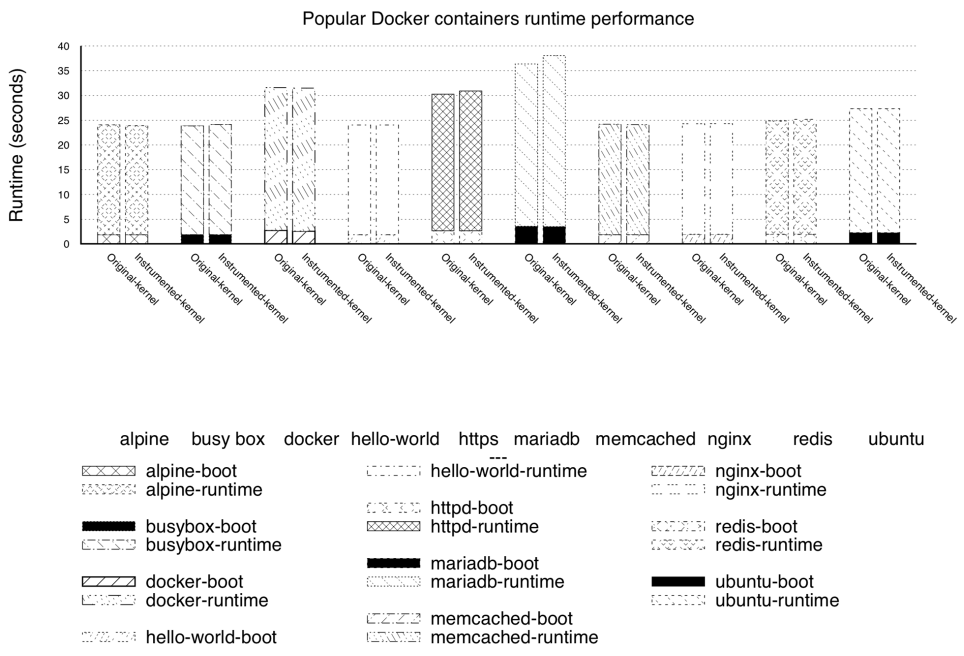


Figure 8. Runtime performance comparison for the top 10 containers

configuration options needed at the booting stage, their tool, called Linux Kernel Tailoring Framework [8], was able to produce tailored kernels that were more consistently reliable. Both of these research strategies required a mapping between the profiled kernel footprint and the kernel configuration, and the kernel was tailored through configuration options. In contrast, the kernel code tailoring in the popular paths related work was directly based on the data gathered from the popular paths.

4.5.2 How do the tailored kernels compare by percent of tailoring and reduction of attack surface?

We studied each of the current kernel tailoring approaches described above, and compared them with our work. Specifically, we utilized two criteria in our comparison: how much code in the kernel was tailored (modified) in each case, and how much of the attack surface could be reduced (as shown in Table 7). For the Linux Kernel Tailoring Framework [8], we were able to access the data from their open source online repository [8] and so we ran the code and workload on our lab machine. The numbers used for the other initiatives were obtained from their conference presentations [42].

The comparison shows that all tools were able to tailor the kernel code by 50% or more, except in certain instances in Alharhi's [28] results. UnPAK tailored the largest percentage number of all four tools at 80% of the kernel code. Furthermore, UnPAK achieved the largest reduction in attack surface.

Table 7. Comparison of the Tailored Kernels

Work	Code tailored by size (percent)	Attack surface reduction (percent)
UnPAK	80%	94%
Alharhi et al. [28]	34%-74%	89%
Kurmus et al. [42]	50%	50%-85%
Kang et al. [8]	78%	N/A

From a security standpoint, this is a more important and urgent factor than the code size reduction, as it means fewer chances for attackers to exploit zero-day vulnerabilities.

4.5.3 Are there differences in limitations?

In practice, there may be conditions that prevent a tailored kernel from being useful. We extracted a few identified limitations in the past strategy examples to see if UnPAK would also be vulnerable to these conditions. The limitations are:

- **Coarse-grained tracing:** Some approaches only work at the function, file, or library level, which, as discussed in Section 4.2, offer less flexibility in how a kernel can be tailored. This limitation could result in some user applications not being supported, or in missing opportunities for further code reduction that could be done at a finer level.
- **Kernel trace is not generic:** An approach that only works for specific target applications could seriously affect the usefulness of the tool.

Table 8. Limitations Comparison

Work	Coarse-grained	Kernel trace not generic	Not bootable or working stably	Runtime performance overhead
UnPAK	✗	✗	✗	✗ negligible overhead (0.1% runtime; 0.37% memory in its current configuration)
Alharhi et al. [28]	✓	✓	✗	N/A
Kurmus et al. [42]	✓	✓	✓ (not stable)	✗ (no overhead)
Kang et al. [8]	✓	✓	✗	✗ no overhead, and a little better than the original kernel in some cases

✓: has this limitation; ✗: does not have this limitation

- **Kernel is not bootable or able to work in a stable manner:** Some approaches may result in an unstable kernel that is not able to boot or function correctly every time. This will severely limit its usage.
- **Large runtime performance overheads:** If the runtime overhead is too large, users are less likely to use the strategy, even if it can effectively reduce attack surfaces .

Through this comparison (Table 8), UnPAK suffers the least number of limitations, and is thus a more practical solution.

Kernel tailoring strategies are important tools to improve system security. Through the comparison of UnPAK and three other strategies, we found that the popular-paths based approach was able to achieve the largest reduction in the kernel attack surface with little overhead in runtime performance and memory usage in its current configuration. Perhaps more importantly, it is a strategy that is not tied to a specific application or use case. Therefore, we believe that the UnPAK approach provides a more general and practical container security solution to users and developers. to secure the running environment of their container applications.

5 Limitations

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

Another factor that could have affected our results 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. If we had run these additional images using the same methodology, it might have yielded a broader spectrum of results.

Our implementation is currently focused only on warning users whenever there is a potentially dangerous attempt to trigger an unpopular line of code. There are certainly more ways we could have explored to deal with risky attempts to reach unpopular lines. For example, we could have instrumented the kernel to try to exit the VM and block the container from executing more code to avoid any further security breach. Another possible strategy is to first monitor any attempt to access the unpopular lines, and then dynamically remove the unpopular code binary from the memory at runtime to prevent potential exploitation of bugs. We leave evaluation of these options as future work.

Lastly, all the research documented in this paper was conducted only with Linux kernel version 4.14.24. For future work, we plan to extend our system so that it can automatically work with other kernel versions. This will generalize the application of our work, and potentially help a lot more users secure their containers.

6 Related Work

In this paper, we seek to enhance the security of containers running on top of a host kernel by reducing exposure to the rarely used risky code it contains. When developing our strategy, we consulted previous studies in four areas: 1) metrics for vulnerability prediction, 2) techniques for enhancing kernel security, 3) efforts to debloat operating system kernels, and 4) alternative approaches for securing containers. Below, we summarize a few of the relevant studies in each area and discuss how they compare to our recent work in container security and vulnerability detection.

Estimating and predicting vulnerabilities. Metrics that can identify the code most likely to contain vulnerabilities help developers and researchers to prioritize their efforts to fix bugs. Towards this end, a number of prediction methods have been put forward in the last decade or so. Chou et al. [30] 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 other components. Ozment and Schechter [50] examined the age of code as a predictor of vulnerability in the OpenBSD [22] operating system, and generally confirmed the finding that the rate at which bugs are found goes down over time, meaning older code is less risky than new code. In a more recent work [43], Li et al. directly compared both of these metrics [30, 50] to the popular paths metric, and found that neither was as predictive of vulnerabilities in the Linux kernel.

Shin et al. [58] 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 project codebases they studied. Similarly, Chowdhury et al. [31] proposed a metric that validates a correlation between vulnerabilities and a software entity's CCC(Complexity, Coupling, and Cohesion), or the combined measures of these variables. Imtiaz et al. [39] proposed a metric that can automatically predict vulnerabilities early on in the design process from publicly available data sources, such as issue tracking systems and vulnerability databases. Other researchers have used software engineering related metrics to train and build classification models that can predict defects in software components and modules. Zhang et al. [61] used three code-level complexity measures—lines of code, McCabe's cyclomatic complexity, and Halstead's volume—to assess classification models as predictors of code quality. Alenezi et al. [27] used software metrics, process and product metrics to predict vulnerabilities in PHP files. While these techniques are helpful for identifying potentially defective code areas, they worked at the coarse level of files, functions, classes, modules, etc. In practice, it is usually unideal or impossible to remove an entire function or file. Though our recent study proved there is value in extracting security data at coarser granularities, being able to predict vulnerabilities at the lines of code level is more precise, and therefore more effective.

Operating System kernel security enhancement. One common research focus in the area of OS kernel security studies has been developing methods to protect kernel memory [1, 2, 38, 52]. The deployment of standard kernel-hardening schemes, such as address space layout randomization (KASLR) [1] and non-executable memory [2], has prevented attackers from injecting legacy code in the kernel space. Giuffrida et al. [38] proposed a fine-grained address space randomization technique that enhances security by randomizing the location where system executables are loaded into memory. Pomonis et al. [52] proposed a kernel hardening scheme that uses execute-only memory and code diversification techniques to protect the kernel against JIT (Just-In-Time) code reuse attacks. These memory protection strategies can effectively protect the kernel against certain attack schemes in practice, but can not defend against security bugs within the kernel itself.

Some research has focused on leveraging hardware features to enhance kernel security. Azab et al. [29] presented a solution that routes functions through the ARM TrustZone isolated environment for inspection before being executed, rather than allowing them to execute directly in the kernel [10]. This can provide extra security since the security monitor is isolated and protected by the TrustZone. However, it also requires special hardware that may not be available to every user.

A number of prior research efforts in enhancing kernel security required refactoring or modification of the kernel. Engler et al. [35] introduced the Exokernel architecture, which allowed applications more direct control of hardware resources through an architecture that exports these resources to a library OS. More recently, but in a similar spirit, unikernels, such as Mirage [46] run each application as its own operating system inside of a virtual machine, thus customizing the "kernel" for each application. Each of these systems, however, requires significant changes to existing applications. For example, Mirage requires applications to be rewritten in the OCaml programming language [21]. Other library OSes are written to run existing applications. Drawbridge [53] and Graphene [60] support security in unmodified Windows and Linux applications, respectively, by implementing an OS "personality" as a support library in each address space. Unfortunately, reimplementation of OS functionality incurs a performance overhead. Containers can mostly avoid this type of overhead, since they allow contained applications to make system calls directly. Thus, our work focused on the container-based approach.

Prior research has also looked at improving security by reducing the kernel attack surface. Kurmus et al. [41] has proposed finding and removing unused kernel code sections by binary instrumentation. The main difference from our work is that this approach only profiles kernel code usage at the function level, while we examine the lines of code inside of a function for better precision. The way to reduce the unused kernel code also differs. While they employ binary instrumentation, we use source code modification.

Operating system kernel debloating. Modern operating system kernels have grown in both size and complexity, a condition that affects operating speed and increases the potential for bugs. Previous research work [55] has attempted to debloat OS kernels to reduce boot time and memory usage, as well as to reduce the size of attack surfaces. Kurmus et al. [42] found that more than half of the total attack surfaces for the Linux kernel could be reduced by automating the kernel configuration to a particular workload and hardware. Alharthi et al. [28] investigated including only kernel modules necessary for the target applications, and reported that such an approach could avoid known security vulnerabilities in the Linux kernel by around 35% to 75%. While these approaches provide some help for specific user applications, they tend to be ad hoc and are not necessarily applicable on a larger scale.

Many researchers [4, 6–8, 47, 59] have explored a configuration-driven (also known as feature-driven) approach to debloating the OS kernel. This method decides which parts of code are to be compiled and built by selecting existing kernel configuration options to support corresponding features. While this approach has the advantage of being able to produce stable kernels, it could be argued that relying entirely on this method may mean missing some opportunities to reduce the code. Such an approach may allow code containing rarely used flags and modes not needed by most applications. This method tends to be more coarse than our strategy, because it works at the configuration-option level, and therefore is less flexible in making decisions on which lines could be reduced within functions.

Container security enhancement. Existing security mechanisms available for containers usually leverage host kernel features. Namespaces [18] is one mechanism that can provide isolation between processes, and has been adopted by Docker. Linux capabilities [17] 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 they want to use, such as file permissions. Control groups [16],, a key Linux feature, can be used to limit, account for, and isolate important system resources, such as memory, CPU, and disk I/O, in each container. More recently, various kernel hardening systems have been developed to provide access control mechanisms for better safety, such as AppArmor [9], SELinux [25], GRSEC [14], and PAX [23]. 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 [24]. 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 keep 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 in the host kernel than they actually need, including potentially risky, rarely used code.

Researchers have attempted to use the above existing security mechanisms to design and build container security systems. Loukidis-Andreou et al. [45] proposed a framework that enforces access control rules by dynamically adding access rules during container runtime to restrict its behavior. Mattetti et al. [48] presented a framework that traces a container's execution and profiles its kernel security modules. Based on this data, AppArmor rules can be automatically constructed to restrict container capabilities and prevent attack from malicious containers.

While these prior systems can construct security rules and restrictions to better define what operations containers can do, we can recommend what code the operating system kernel should offer. This extra step improves 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 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. To do so, we developed a methodology to systematically profile and obtain the popular paths data of widely-used container applications. Using this data, we were not only able to create UnPAK, a modified Linux kernel that could alert users whenever potentially dangerous unpopular lines are about to be triggered, but also identify ways to improve security at various levels of granularity. By allowing users to improve container security with steps as simple as eliminating unused files, it means that those who can not do the more labor intensive work of eliminating bugs at the line of code level do not need to remain completely vulnerable. Security thus stops being an "all or nothing" proposition, but instead can be tailored to the security sensitivity of the application and the available resources of the user.

We were also able to verify that, consistent with previous findings, the popular paths in the host kernel for frequently used Docker containers do contain fewer security vulnerabilities, and thus are inherently more secure. Equally important, usingUnPAK, we were able to prove that these containers were able to run their default workload using the popular paths more than 99.9% of the time, with only a negligible (less than 1%) performance overhead based on its current action configuration. Compared with three other current kernel tailoring strategies, we demonstrate that UnPAK was an efficient solution in reducing the kernel attack surface, with relatively fewer limitations.

References

[1] 2013. Kernel address space layout randomization. https://lwn.net/Articles/569635/
[2] 2015. Kernel W xor X improvements in OpenBSD. https://www.openbsd.org/papers/hackfest2015-w-xor-x.pdf
[3] 2016. Dirty COW - (CVE-2016-5195) - Docker Container Escape. https://blog.paranoidsoftware.com/dirty-cow-cve-2016-5195-docker-container-escape/
[4] 2017. A different approach to kernel configuration. https://lwn.net/Articles/733405/
[5] 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/
[6] 2017. A Practical Approach of Tailoring Linux Kernel. https://ossna2017.sched.com/event/BCsG/a-practical-approach-of-tailoring-linux-kernel-junghwan-kang-national-security-rese
[7] 2018. An Empirical Study of an Advanced Kernel Tailoring Framework. https://ossna18.sched.com/event/FAN5/an-empirical-study-of-an-advanced-kernel-tailoring-framework-junghwan-kang-nati

[8] 2018. Linux Kernel Tailoring Framework. https://github.com/ultract/linux-kernel-tailoring-framework
[9] 2019. AppArmor. https://wiki.ubuntu.com/AppArmor
[10] 2019. ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone
[11] 2019. Build EAR. https://github.com/rizsotto/Bear
[12] 2019. Clang LibTooling. https://clang.llvm.org/docs/LibTooling.html
[13] 2019. Gcov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[14] 2019. GRSEC. https://wiki.debian.org/grsecurity
[15] 2019. Lcov. http://ltp.sourceforge.net/coverage/lcov.php
[16] 2019. Linux control groups. http://man7.org/linux/man-pages/man7/cgroups.7.html
[17] 2019. Linux kernel capabilities. http://man7.org/linux/man-pages/man7/capabilities.7.html
[18] 2019. Linux Namespaces. http://man7.org/linux/man-pages/man7/namespaces.7.html
[19] 2019. Linux Testing Project (LTP). https://github.com/linux-test-project/ltp
[20] 2019. The LLVM Compiler Infrastructure Project. https://llvm.org
[21] 2019. OCaml programming language. https://ocaml.org
[22] 2019. OpenBSD. https://www.openbsd.org
[23] 2019. PAX. https://pax.grsecurity.net
[24] 2019. Red Hat Linux. https://www.redhat.com/en/technologies/linux-platforms/enterprise-linux
[25] 2019. SELinux. https://selinuxproject.org/page/Main_Page
[26] 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
[27] Mamdouh Alenezi, Ibrahim Abunadi, and S. Arabia. 2015. Evaluating Software Metrics as Predictors of Software Vulnerabilities. *International journal of security and its applications* 9 (2015), 231–240.
[28] Mansour Alharthi, Hong Hu, Hyungon Moon, and Taesoo Kim. 2018. On the Effectiveness of Kernel Debloating via Compile-time Configuration. In *Proceedings of the 1st Workshop on SoftwAre debLoating And Delayering (SALAD'18)*.
[29] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-Time Kernel Protection from the ARM TrustZone Secure World. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. Association for Computing Machinery, New York, NY, USA, 90–102. https://doi.org/10.1145/2660267.2660350
[30] 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
[31] Istehad Chowdhury and Mohammad Zulkernine. 2010. Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?. In *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*.
[32] CVSS 2020. Common Vulnerability Scoring System. Retrieved September, 2020 from https://www.first.org/cvss/
[33] Docker 2019. Enterprise Container Platform for High Velocity Innovation. Retrieved April, 2019 from https://www.docker.com
[34] Docker Hub 2019. A Library and Community for Container Images. Retrieved April, 2019 from https://hub.docker.com
[35] 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
[36] 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/
[37] Masoud Ghaffarinia and Kevin W. Hamlen. 2019. Binary Control-Flow Trimming. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz (Eds.). ACM, 1009–1022. https://doi.org/10.1145/3319535.3345665
[38] Cristiano Giuffrida, Anton Kuijsten, and Andrew S. Tanenbaum. 2012. Enhanced Operating System Security Through Efficient and Fine-grained Address Space Randomization. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 475–490. https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/giuffrida
[39] S. Imtiaz and T. Bhowmik. 2018. Towards data-driven vulnerability prediction for requirements. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2018).
[40] Hsuan Chi Kuo, Jianyan Chen, Sibin Mohan, and Tianyin Xu. 2020. Set the Configuration for the Heart of the OS: On the Practicality of Operating System Kernel Debloating. In *Abstracts of the 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems*. Association for Computing Machinery, Inc, 87–88. https://doi.org/10.1145/3393691.3394215 2020 SIGMETRICS/Performance Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2020 ; Conference date: 08-06-2020 Through 12-06-2020.
[41] Anil Kurmus, Alessandro Sorniotti, and Rüdiger Kapitza. 2011. Attack Surface Reduction for Commodity OS Kernels: Trimmed Garden Plants May Attract Less Bugs. In *Proceedings of the Fourth European Workshop on System Security (EUROSEC '11)*. Association for Computing Machinery, New York, NY, USA, Article 6, 6 pages. https://doi.org/10.1145/1972551.1972557
[42] Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, and Daniel Lohmann. 2013. Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*.
[43] 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.
[44] LinuxKit 2019. A toolkit for building secure, portable and lean operating systems for containers. Retrieved April, 2019 from https://github.com/linuxkit/linuxkit
[45] F. Loukidis-Andreou, I. Giannakopoulos, K. Doka, and N. Koziris. 2018. Docker-Sec: A Fully Automated Container Security Enhancement Mechanism. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 1561–1564.
[46] 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
[47] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than Your Container. In *Proceedings of*

the 26th Symposium on Operating Systems Principles (SOSP '17). Association for Computing Machinery, New York, NY, USA, 218–233. https://doi.org/10.1145/3132747.3132763

[48] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini. 2015. Securing the infrastructure and the workloads of linux containers. In *2015 IEEE Conference on Communications and Network Security (CNS)*. 559–567.

[49] NVD 2019. National Vulnerability Database. Retrieved April, 2019 from https://nvd.nist.gov

[50] 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

[51] 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

[52] Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. 2019. Kernel Protection Against Just-In-Time Code Reuse. *ACM Trans. Priv. Secur.* 22, 1, Article 5 (Jan. 2019), 28 pages. https://doi.org/10.1145/3277592

[53] 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

[54] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. 2019. RAZOR: A Framework for Post-deployment Software Debloating. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 1733–1750. https://www.usenix.org/conference/usenixsecurity19/presentation/qian

[55] Anh Quach, Aravind Prakash, and Lok Yan. 2018. Debloating Software through Piece-Wise Compilation and Loading. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 869–886. https://www.usenix.org/conference/usenixsecurity18/presentation/quach

[56] 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

[57] Y. Shin, A. Meneely, L. Williams, and J. A. Osborne. 2011. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering* 37, 6 (2011), 772–787.

[58] 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

[59] Reinhard Tartler, Anil Kurmus, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Daniela Dorneanu, Rüdiger Kapitza, Wolfgang Schröder-Preikschat, and Daniel Lohmann. 2012. Automatic OS Kernel TCB Reduction by Leveraging Compile-Time Configurability. In *Eighth Workshop on Hot Topics in System Dependability (HotDep 12)*. USENIX Association, Hollywood, CA. https://www.usenix.org/conference/hotdep12/workshop-program/presentation/Tartler

[60] 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

[61] H. Zhang, X. Zhang, and M. Gu. 2007. Predicting Defective Software Components from Code Complexity Measures. In *13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*. 93–96.