Confine: Fine-grained System Call Filtering for Container Attack Surface Reduction

Maryam Rostamipoor, Seyedhamed Ghavamnia, and Michalis Polychronakis

Stony Brook University

Abstract. Reducing the attack surface of the OS kernel is a promising defense-in-depth approach for mitigating the fragile isolation guarantees of container environments. In contrast to hypervisor-based systems, malicious containers can exploit vulnerabilities in the underlying kernel to fully compromise the host and all other containers running on it. Previous container attack surface reduction efforts have relied on dynamic analysis and training using representative workloads to limit the set of system calls exposed to containers. These approaches, however, do not capture exhaustively all the code that can potentially be needed by future workloads or rare runtime conditions, and are thus not appropriate as a generic solution.
Aiming to provide a practical solution for the protection of arbitrary containers, in this paper we present a generic approach for the automated generation of restrictive system call policies for Docker containers. Our system, named *Confine*, uses static code analysis to inspect the containerized application and all its dependencies, identify the superset of system calls required for the correct operation of the container, and generate both a container-wide and application-specific Seccomp system call policy that can be readily enforced while loading the container and launching the main program. We also show that further attack surface reduction is possible by enforcing fine-grained system call policies that do not only consider the system calls used by the target application, but also their argument values.
The results of our experimental evaluation with a set of 27 Docker images show that applying container-wide filtering disables more than 145 system calls on average across the entire container, and application-specific filtering increases the number of filtered system calls by 25% on average, as many system calls used exclusively by utilities and scripts during the container's initialization phase can be safely removed afterwards.

Keywords: Debloating, Attack surface reduction, Program analysis, System call filtering, Argument concretization

1 Introduction

The convenience of running containers and managing them through orchestrators, such as Kubernetes (kub, c2023), has popularized their use by developers and organizations, as they provide both lower cost and increased flexibility. In contrast to virtual machines, which run their own operating system (OS), multiple tenants can launch containers on top of the same OS kernel of the host. This makes containers more lightweight compared to VMs, and thus allows for running a higher number of instances on the same hardware (Butler, 2016).

The performance gains of containers, however, come to the expense of weaker isolation compared to VMs. Isolation between containers running on the same host is enforced purely in software by the underlying OS kernel. Therefore, adversaries who have access to a container on a third-party host can exploit kernel vulnerabilities to escalate their privileges and fully compromise the host (and all the other containers running on it).

The trusted computing base in container environments essentially comprises the entire kernel, and thus all its entry points become part of the attack surface exposed to potentially malicious containers. Despite the use of strict software isolation mechanisms provided by the OS, such as capabilities (lin, c2023a) and namespaces (lin, c2023c), a malicious tenant can leverage kernel vulnerabilities to bypass them. For example, a vulnerability in the `waitid` system call (cve, 2017) allowed malicious users to run a privilege escalation attack (Shapira, 2017) and escape the container to gain access to the host.

At the same time, the code base of the Linux kernel has been expanding to support new features, protocols, and hardware. The increase in the number of exposed system calls throughout the years is indicative of the kernel's code "bloat." The first version of the Linux kernel (released in 1991) had just 126 system calls, whereas version 5.4 (released in 2019) supports 335 system calls. As shown in previous works (He et al., 2007; Kurmus et al., 2013; Lee et al., 2004; Zhongshu Gu and Xu, 2014), different applications use disparate kernel features, leaving the rest unused—and available to be exploited by attackers. Kurmus et al. (Kurmus et al., 2013) showed that each new kernel function is an entry point to accessing a large part of the whole kernel code, which leads to attack surface expansion.

As a countermeasure to the ever expanding code base of modern software, *attack surface reduction* techniques have recently started gaining traction. The main idea behind these techniques is to identify and remove (or neutralize) code which, although is part of the program, it is either i) completely inaccessible (e.g., non-imported functions from shared libraries), or ii) not needed for a given workload or configuration. A wide range of previous works have applied this concept at different levels, including removing unused functions from shared libraries (Mishra and Polychronakis, 2018; Mulliner and Neugschwandtner, 2015; Quach et al., 2018) or even removing whole unneeded libraries (Koo et al., 2019); tailoring kernel code based on application requirements (Kurmus et al., 2013; Zhongshu Gu and Xu, 2014); or limiting system calls for containers (doc, 2023; Rastogi et al., 2017a,b; Wan et al., 2017). In fact, one of the suggestions in the NIST container security guidelines (Murugiah Souppaya, 2017) is to reduce the attack surface by limiting the functionality available to containers.

Despite their diverse nature, a common underlying challenge shared by all these approaches is how to accurately identify and maximize the code that can be *safely* removed. On one end of the spectrum, works based on static code analysis follow a more conservative approach, and opt for maintaining compatibility in the expense of not removing all the code that is actually unneeded (i.e., "remove what is not needed"). In contrast, some works rely on dynamic analysis and training (doc, 2023; Kurmus et al., 2013; Rastogi et al., 2017a,b; Wan et al., 2017; Zhongshu Gu and Xu, 2014) to exercise the system using realistic workloads, and identify the actual code that was executed while discarding the rest (i.e., "keep what is needed"). For a given workload, this approach maximizes the code that can be removed, but as we show in Section 3, it does not capture exhaustively *all* the

code that can *potentially* be needed by different workloads—let alone parts of code that are executed rarely, such as error handling routines.

Given that previous efforts in the area of attack surface reduction for container environments have focused on dynamic analysis (doc, 2023; Rastogi et al., 2017a,b; Wan et al., 2017), in this work we aim to provide a more generic and practical solution that can be readily applied for the protection of any container without the need for training. Another shared characteristic of most previous works (doc, 2023; Rastogi et al., 2017a,b; Wan et al., 2017) is that they consider the entire container as a single entity, and generate system call policies for its whole lifetime, as opposed for the final target application. In addition, they follow an all-or-nothing approach to system calls, with each system call being either allowed or denied, missing out on the opportunity to further restrict the allowed interactions with the OS by partially blocking some system call functionality.

In this paper, we present an automated technique for generating restrictive system call policies for arbitrary containers, with the goal of limiting the exposed interface of the underlying kernel that can be abused. By relying on static code analysis, our approach inspects *all* execution paths of the containerized application and all its dependencies, and identifies the superset of system calls required for the correct operation of the container. Our system, named *Confine*, improves upon the state of the art in three main ways. First, it uses static (instead of dynamic) analysis to build a sound profile for the entire container. Second, based on the observation that containers typically host a single, long-running *target* application, Confine creates a second *application-specific* filter that is installed right before the execution of the target application. This increases significantly the number of filtered system calls compared to container-wide system call filtering, as many system calls used exclusively by utility programs during the container's initialization phase can be safely filtered afterwards. Third, for system calls that cannot be filtered, Confine *concretizes* (in a conservative, best-effort way) their arguments according to the application's needs. This prohibits the use of certain flag and constant values, which in many cases prevents the exploitation of kernel vulnerabilities associated with non-filtered system calls.

We experimentally evaluated Confine with a set of 27 publicly available Docker images, and demonstrate its effectiveness in deriving strict system call policies without breaking functionality. In particular, for about half of the containers, Confine disables 144 or more system calls (out of 335) by applying container-wide filtering. In addition to container-wide system call filtering, Confine's application-specific filtering increases the number of filtered system calls by 25% on average. On top of system call filtering, argument concretization results in the mitigation of a total of 28 Linux kernel CVEs across all tested container images.

Confine is publicly available as an open-source project at: `https://github.com/shamedgh/confine`.

2 Background

The attack surface of the OS kernel used by containers can be reduced by *restricting* the set of system calls available to each container. In this section, we describe how Linux containers provide isolation to different "containerized" processes, and how Seccomp BPF (sec, c2023) can be used to reduce the kernel code exposed to containers.

2.1 Linux Containers

Linux containers are an OS-level virtualization approach that can be used to execute multiple userlands on top of the same kernel. The Linux kernel uses Capabilities (lin, c2023a), Namespaces (lin, c2023c) and Control Groups (cgroups) (lin, c2023b) to provide isolation among different containers.

Namespaces are a kernel feature that virtualizes *global* system resources (specifically: mount points, process IDs, network devices and network stacks, IPC objects, hostnames, user and group IDs, and cgroups), providing the "illusion" of exclusive use of these resources to processes within the same namespace.

Control Groups allow processes to be organized into hierarchical groups, whose usage of various types of resources (e.g., CPU time, memory, disk space, disk and network I/O) can be limited, accounted, or prioritized accordingly. Containers use cgroups to provide "fair" usage of resources.

Docker (doc, c2023a) is a platform that employs the software-as-a-service and platform-as-a-service models for developing, deploying, and running containers. Every Docker container launched is based on a Docker image, which is a file built in layers, encapsulating the entire environment (including a whole Linux distribution, libraries, and support utilities) required to execute the containerized application(s). The specification of the Docker image is described in a text file, called *Dockerfile*. The Dockerfile essentially contains all the commands required to assemble the respective image. Docker uses Linux namespaces and cgroups to provide isolation between containers.

Docker Hub (doc, c2023a) is a central repository of both community-based and official Docker images, which has drastically popularized container use among system administrators. More importantly, by building streamlined services with a minimal code base, Docker has enabled corporations to increasingly switch to the use of microservices. Each microservice can be configured as a Docker image once, and then multiple instances of it can be launched.

2.2 Seccomp BPF

User-space applications communicate with the OS kernel through the provided set of *system calls* , i.e., a pre-defined API that allows access to specific kernel functionalities programmatically. Most applications, however, typically need only a *subset* of the available system calls to function properly, i.e., most applications do not make use of all the provided system calls.

Processes can invoke any of the system calls provided by the OS, as long as they have the necessary privileges to perform the requested operation, regardless of their needs. This can be viewed as a violation of the principle of least privilege, as allowing a program to invoke more system calls than those it actually requires may aid an attacker in two main ways: i) a compromised (i.e., previously benign) process can invoke the extra system calls to perform operations that the original program never intended to perform; and ii) a malicious process may exploit kernel vulnerabilities associated with those extra system calls, which otherwise would have been inaccessible.

Seccomp BPF (sec, c2023) is a subsystem of the Linux kernel that allows processes to limit the set of system calls available to them. The filters are defined using the BPF

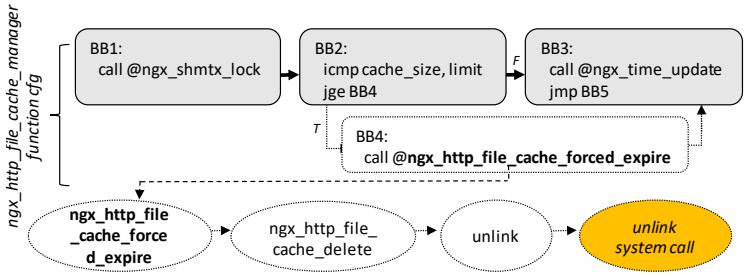

Fig. 1: Example of a control flow in Nginx that is missed by dynamic analysis. Ovals represent functions, while rectangles represent basic blocks. Dashed branches and blocks are not executed during the training phase.

(Berkeley Packet Filter) language, and can allow, deny, or log invocations of specific system calls. To prevent time-of-check-time-of-use (TOCTOU) attacks, Seccomp BPF programs cannot dereference pointers (sec, c2023). Consequently, arguments passed to system calls can be restricted only based on their absolute values.

2.3 Threat Model

We consider attacks against the OS kernel (and other containers running on top of it) by an adversary who has gained access to a container by exploiting a vulnerability in the container's target application. The goal of our work is to reduce the adversary's chances of escaping from the container. We do not focus on preventing the exploitation of applications running in the container. Any additional defenses implemented in the application or the OS are orthogonal to our approach, as we do not rely on any other protection mechanism to be in place (except Seccomp BPF).

Using Confine, attackers are restricted to a smaller set of system calls with fewer features, which in turn limits the functionality of exploit code. More importantly, kernel vulnerabilities that would have been exploitable through the invocation of certain system calls with certain arguments, now become unreachable. If the system call that triggers a certain vulnerability is filtered, privilege escalation (Li et al., 2017) and other attacks can be averted.

3 The Need for Static Analysis

Previous works (doc, 2023; Rastogi et al., 2017a,b; Wan et al., 2017) have used dynamic analysis to derive the list of system calls used by a container. However, dynamic analysis is not sound, and thus can miss system calls along execution paths that were not exercised during the training phase. To demonstrate this issue, we manually analyzed Nginx and discovered three examples of system calls that would be missed if only dynamic analysis were used. For our evaluation, we use Nginx with the Cache Management and Auto Index features enabled.

Nginx spawns a separate *cache-manager* process to handle cache management. This process clears the older cached files when the cache is full using the `unlink` system

call. Dynamically analyzing Nginx would capture the initialization of the cache-manager process, but would likely fail to capture the *deletion* of older cached files, and therefore fail to capture the use of the `unlink` system call. As the `unlink` system call is not invoked anywhere else during the normal execution of the program, relying on training alone would cause it to be marked as unused. Moreover, extending the training phase for a longer duration would not solve the problem because the deletion of older files is triggered only when the cache is full. Training would need to request enough *new* files to fill up the cache. Correctly setting up the training process to handle such situations is thus challenging. Figure 1 shows the parts of the control flow that are not discovered during training.

Another example of failure to capture a system call is the use of `lstat` when displaying directory listings. Apart from this functionality, `lstat` is not used in any other part of Nginx. As listing a directory is usually triggered by users who manually type a URL, and not by following any existing URL on a website, it is unlikely that a training-based approach would be able to capture this system call.

In yet another case, the Nginx binary can be updated with a newer version without dropping client connections. The system calls `getsockopt` and `getsockname` are used to hand over the existing socket connections to the new process, and are not used anywhere else in the code, making it challenging for dynamic analysis to discover them.

The above examples are indicative of the trade off between fragility and overapproximation faced by dynamic and static analysis. Relying on dynamic analysis alone would require the training to be comprehensive enough to anticipate and capture all above corner cases. In contrast, static analysis results are guaranteed to be sound, but may include system calls that are never invoked by certain workloads. As we aim for a practical and generic solution, we opt for using static analysis to capture the superset of system calls used by an application.

4 Design

Our goal is to reduce the kernel attack surface available to an attacker of a container service by limiting the number of system calls available to each container, which can potentially be of use for malicious purposes (as a gateway to exploiting kernel vulnerabilities). To achieve this, Confine "hardens" the container image once it has been fully configured by the user, by limiting its system call access at three levels. First, Confine generates a container-wide system call filter that is applied to *all* programs launched in the container. Then, given the fact that most containers execute a single long-running program, it creates an application-specific system call filter that removes all system calls needed solely during the initialization phase of the container. In its final step, Confine further restricts the remaining system calls needed by the main program by limiting their argument values.

Identifying the system calls and their argument values that are necessary for the correct execution of the container and its main program requires addressing the following requirements: 1) identify all applications that may run on the container; 2) identify all library functions imported by each application; 3) map library functions to system calls; 4) extract direct system call invocations from applications and libraries; and 5) extract hardcoded argument values for identified system calls.

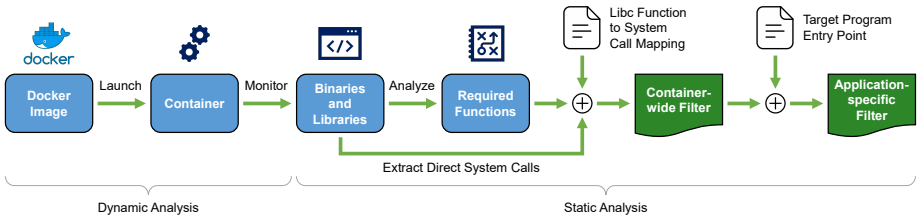

Fig. 2: Overview of Confine's main processing steps. A one-time dynamic analysis phase that does not require any application-specific workloads is used for the sole purpose of identifying the applications running in the container. Each application is then statically analyzed to identify all the library functions that it uses, and the system calls it relies on, to generate a container-wide filter. Finally, a more restrictive application-specific filter is generated for the container's target application.

Figure 2 presents a high-level overview of Confine's main processing steps, discussed in detail in the rest of this section. Confine currently supports the popular Docker containers running on a native Linux-based host, but similar analysis could be performed for other container environments and operating systems.

4.1 Identifying Running Applications

Although containers are usually specialized to run a single application or service, they typically invoke many other utility and support programs prior to executing the main program. For example, the default MongoDB Docker image (mon, c2023) invokes the following supporting programs to set up the environment: `bash`, `chown`, `find`, `id`, and `numactl`. To generate container-wide system call policies, we must thus identify all programs that can potentially run during the lifetime of a container.

We can use different techniques to identify the programs running in a container: 1) Static analysis: extract all programs available in the Docker image; 2) Dynamic analysis: launch container and identify the programs it executes; and 3) Manual: the developer/end-user provides the list of programs. If we use the first method and consider *all* the programs in a Docker image, the generated system call profile would suffer severely from overapproximation (i.e., filtering less than 80 system calls). Strictly relying on the developer to provide the list of binaries also seems unnecessary since a container typically executes the programs it requires during its initialization phase. Therefore, Confine relies on limited dynamic analysis to capture the list of processes created on the system. A profiling tool records every application launched within a configurable time period (30 seconds by default) since the creation of the container—long enough to capture both system initialization, as well as the "stable" state of the system. The obtained set of applications is then used to derive the corresponding system call policy.

However, since most programming languages give the programmer the ability to launch applications using special library calls (e.g., `execve`) and those invocations might not occur during our monitoring window, Confine may fail to analyze any executables launched in this way. For these cases, currently, the developer is expected to provide a list of binaries executed using such library calls.

Our approach is different from previous works that rely on dynamic training using various workloads to derive a list of allowable system calls (Wan et al., 2017). In our approach, the goal of the dynamic analysis is merely to identify the set of binary executables to be analyzed—the system calls invoked by these programs are then derived statically.

The above dynamic analysis is meant to be a convenient and automated way to carry out the batch analysis of multiple container images. For containers that may include applications that are not launched from the beginning, our system supports manually provided external lists of executables that should be included in the analysis.

4.2 Static Analysis

Dynamic analysis often fails to exercise all possible code paths, especially when comprehensive workloads are not available during training. To ensure complete code coverage, once we have the list of applications that are executed on the container, we perform static analysis to extract the system calls that are needed for the correct execution of each application. We combine the system call requirements of all these identified applications to generate a container-wide system call profile. Then, we use the extracted system calls for the main program to generate an application-level system call filter, and finally, restrict the system calls required by the main program even further by limiting their argument values.

System Call Identification

Libc Function to System Call Mapping User programs typically invoke system calls through the libc library, which provides corresponding wrapper functions (e.g., the libc function `read` invokes the system call `SYS_read`). Confine analyzes the source code of libc to derive a mapping between exported functions and the system calls they invoke. For the rest of the programs and libraries on a given container, however, Confine only needs to analyze their *binaries*.

A libc function may have multiple control flow paths to the actual system call. To correctly identify which system calls are invoked by a given libc function, we thus need to analyze these control flow paths. To that end, Confine statically analyzes the source code of libc to derive its full callgraph, and accurately map each function to its respective system calls.

Function pointers are used widely in libc. However, performing accurate points-to analysis has significant scalability and performance issues (Andersen, 1994; Hind, 2001). To avoid having to perform points-to analysis, we follow a more conservative approach and retain all system calls that are invoked through any function that has its address taken. In Section 5.1 we discuss the technical challenges we encountered during this process.

Having an accurate mapping between libc functions and system calls, it is then straightforward to analyze each program (main executable and libraries), identify all imported libc functions, and derive the set of all possible system calls the program may invoke. It is important to stress that this phase is performed only *once* per libc *version*—the derived mapping is then saved and used across all containers.

Direct System Call Invocation In addition to using libc wrappers, applications and libraries may also invoke system calls directly using the `syscall()` function, or using the `syscall`

assembly instruction. Although the number of applications and libraries which use this approach are limited, for the sake of completeness, we use binary code disassembly to extract any directly invoked system calls. We describe in detail this process in Section 5.2.

System Call Argument Concretization We further reduce the exposed kernel attack surface by restricting the values that may be passed as arguments to system calls that cannot be filtered. To derive concrete argument values, Confine performs intra-procedural reaching definitions data flow analysis, starting from each argument at each invocation site. When the value passed to a given argument can be identified across *all* its invocation sites of the respective system call, then Confine *concretizes* this argument by allowing only the given value (or set of values) to be passed. As discussed in Section 2.2, Seccomp BPF does not support pointer dereferencing, which prevents us from concretizing pointer arguments. Given this limitation, Confine strives to restrict flags and constant arguments, which fortunately are quite common.

For direct system call invocations, reaching definitions analysis is performed on each of the six CPU registers that may be used to pass constant arguments, starting right before the `syscall` assembly instruction.[1] For libc function call sites, we identify two cases. While an important subset of libc functions are wrappers, acting as a simple interface for invoking individual system calls, other more complex libc functions (e.g., `printf()`) internally invoke several system calls to carry out the intended operation.

Libc Wrapper Functions Most libc wrapper functions merely copy their arguments (from the wrapper's call site) and the corresponding system call number to the appropriate registers, and then invoke the system call. Reaching definitions analysis is thus performed at the call sites of wrapper functions to identify their argument values. In most cases, there is a one-to-one mapping between the arguments of a wrapper and the arguments of the system call it internally invokes, but this is not always the case. To accurately derive this mapping, Confine performs (a one-time) intra-procedural data flow analysis to identify how libc wrapper arguments flow into the internal system call invocation site.

Among the exceptions we identified as a result of this analysis, the `clone()` libc function modifies the order of the arguments prior to invoking the respective system call, while `fork()` invokes the `clone` system call with hardcoded argument values. The `mmap()` function invokes its respective system call twice, once with a set of hardcoded values, and a second time with the values passed to the wrapper function.

Complex Libc Functions Some libc functions lead to the invocation of several system calls, the arguments of which often do not depend on the function's arguments. For example, `fgets()` internally invokes several system calls, including `mmap`, `write`, and `openat`. To handle these cases, Confine could treat libc as any other opaque binary executable and identify the hardcoded arguments of these internal system calls irrespectively of whether a given control flow path will actually be executed. This would lead to a less restrictive argument-level filter due to the inclusion of control flow paths that invoke system calls with concretized arguments, which though are *only* accessible from libc functions that are *not imported* (i.e., used) by the application or its libraries.

[1] Confine currently supports only the x86-64 ABI, in which arguments are passed through registers.

Recall though that for libc we do have a more accurate callgraph, extracted at the source code level as a result of system call identification (Section 4.2). Confine thus leverages this callgraph to increase the accuracy of argument concretization by identifying the exported functions from which these control flow paths can actually be executed, and includes their concretized system call arguments *only* in case those functions are imported by the main application or its libraries.

For each internal path leading to a system call (either direct or through a wrapper function), we traverse the path to identify the function that invokes the system call. After identifying this function, we perform the same reaching definitions analysis starting from its call site to identify the passed argument values. Although we do assume that the source code of libc is available (for callgraph extraction), our data flow analysis implementation operates at the binary level, and thus Confine performs the reaching definitions analysis of complex libc functions at the binary level—implementing the same analysis at the source code level would not offer any significant advantage in terms of accuracy. In most cases, the values of constant arguments are hardcoded at these internal call sites, and can be easily identified by our analysis. Therefore, Confine considers these concretized argument values only if the respective libc function is actually called by the target application.

As an example, the following path extracted from the callgraph shows how the `fgets` function internally calls the `__mmap` wrapper function: `fgets` → `__libgcc_s_init` → `__libc_fatal` → `_libc_message` → `__mmap` → `__mmap64` → `mmap`. Confine is able to extract concrete values for the non-pointer arguments of the `__mmap` wrapper function (`prot` and `flags`) by analyzing its call site within the `_libc_message` function (`__mmap` is a weak alias of the `mmap` wrapper, while `__mmap64` is inlined; `__libgcc_s_init`, `__libc_fatal`, and `_libc_message` are internal, non-exported functions).

4.3 Hardening the Container Image

Container-wide Filter Enforcement Docker containers support the use of Seccomp filters to limit the system calls accessible from the container. The user can launch the container with a custom ruleset which specifies the system calls that can be accessed by the container. This ruleset can be either in the form of a deny-list or an allow-list of system calls prohibited or permitted. For Confine, we use an allow-list of system calls that the container is permitted to invoke, blocking the rest.

Based on the analysis performed in Sections 4.1 and 4.2, we use an automated script to derive the list of prohibited system calls, and construct the corresponding Seccomp profile.

Application-specific Filter Enforcement While it is common for a container to execute different programs to prepare the environment, it finally ends up running a single target application that carries out the container's main task. For example, as shown in Figure 3, the MySQL container (doc, c2023a) eventually runs `mysqld` after executing `mkdir`, `find`, `mktemp`, and several other programs. While previous works (Ghavamnia et al., 2020a; Wan et al., 2017) generate a container-wide Seccomp profile for the entire container, we show that many system calls are only required for setting up the container, and can be safely filtered once the main program is ready to be launched.

In its final step, Confine generates a Seccomp filter tailored to the container's target application. Since Docker does not provide a mechanism to install new filters after the

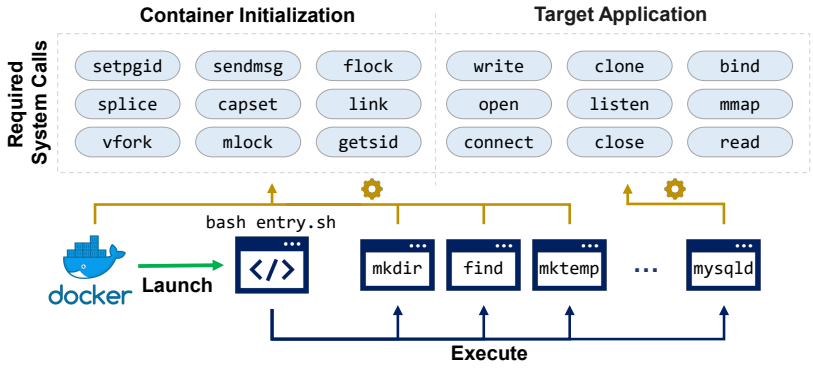

Fig. 3: Many of the system calls required during a container's initialization phase are not needed by the main long-running application (`mysqld` in this example) and thus can be filtered.

container is launched, we must use a non-intrusive mechanism to apply the application-specific filter. Although it is possible to change an installed Seccomp BPF program by modifying the Linux kernel (Lei et al., 2017), deploying such a technique would introduce significant deployment hurdles, and is therefore not ideal. Another approach is to install the application-specific filter upon launching the program by patching the target binary and direct its entry point to our added code that installs the application-specific filter. Another alternative would be `LD_PRELOAD` (ld-, c2023), which can also be used to hook the `main` function of the binary.

Instead of these more intrusive approaches, Confine uses an easy-to-deploy mechanism that installs the application-specific filter through a small "proxy" C program, which in turn executes the target program. To that end, Confine modifies the container initialization script to replace the main target program (e.g., `mysqld`) with Confine's proxy program, without altering the original initialization process. When the container attempts to execute the target program, Confine's proxy program is launched instead, at which point it installs the filter and then executes the actual target application. Using this approach, neither the program binary nor the kernel or the Docker daemon need to be modified.

5 Implementation

To capture a trace of all invoked executables, Confine leverages Sysdig (sys, c2023) to monitor the `execve` calls made during the initial 30 seconds (configurable value) of the container. After it generates the list of programs the container runs, Confine further performs static analysis to extract the list of system calls necessary for the correct execution of the container.

To generate an application-specific filter, Confine must separately analyze the system call requirements of a container's main long-running program. Confine currently relies on the user to provide the name of this program. The application name typically matches the name of the Docker image (e.g., the Nginx image runs Nginx as its main application). We plan to implement automated identification as part of our future work, as the main

application is the last one in the execution chain that remains running after the initialization phase, and can thus be easily identified in an automated way.

5.1 Mapping Libc Functions to System Calls

To ensure correctness, a precise libc function callgraph is required to identify and filter unused system calls. Based on our analysis of the top 100 most popular Docker images from Docker Hub (doc, c2023a), we found that all containers use the popular glibc library as their main user-space libc library. Therefore, we use glibc to build a mapping between its functions and system calls. This is a one-time effort, and after we build the mapping, Confine uses it to analyze the system call requirements of each container.

Glibc heavily relies on multiple GCC (gcc, c2023) features that are not implemented in LLVM. Due to this issue, we implemented a second analysis pass to extract the callgraph and system call information from glibc, based on GCC' register translation language (RTL) intermediate representation (IR). Our callgraph extraction implementation is based on the Egypt tool (Gustafsson), which operates on GCC's RTL IR. We discovered that there are three main mechanisms through which glibc invokes system calls, explained below.

System Call via Inline Assembly and Assembly Files This is the most straightforward mechanism for invoking system calls. Functions such as `accept4()`, which is responsible for accepting incoming socket connections, contain inline invocations using the x86-64 `syscall` instruction. Given the source code, the Egypt tool constructs the function callgraph for any given application or library. We augmented Egypt to iterate over every call instruction in the RTL IR and record any native x86-64 `syscall` instruction. Similarly, assembly files may also contain `syscall` instructions. Therefore, Confine performs code analysis at the assembly language level to identify all direct `syscall` instructions.

System Call Wrapper Macros In addition to directly using the `syscall` instruction, glibc also uses macro expansion to generate wrappers to system calls. Other glibc routines use these wrappers to invoke system calls. Because these wrappers are implemented as architecture-dependent (in our case x86-64) macros, they cannot be retrieved by analyzing the RTL IR. Moreover, the parameters to these macros are provided by a bash script during compilation time.

The `syscall-template.S` file contains the macros `T_PSEUDO`, `T_PSEUDO_NOERRNO`, and `T_PSEUDO_ERRVAL`, which define wrappers to system calls. The list of system calls to be generated, along with other information, such as symbol names and the number of arguments, are provided in the `syscalls.list` file. The Bash script `make-syscalls.sh` reads this file at compilation time, generates the correct macro definitions, and invokes the expansion of the macros in `syscall-template.S`. This script is invoked as part of the build process of glibc. During the compilation of glibc, we trace the execution of this script and record the relevant macro definitions observed during its execution. Using these macros and macro definitions, we derive the mappings between these wrappers and their respective system calls.

Weak Symbols and Versioned Symbols Glibc uses the `weak_alias` macro to define weak symbols for functions. GCC supports symbol versioning, and glibc uses this feature to support multiple versions of glibc. The versioned symbols are defined using the

macro `versioned_symbol`. Both `weak_alias` and `versioned_symbol` provide aliases for functions. Other functions within glibc, as well as the applications using glibc, can invoke these aliased functions either through the original function name or its alias. We analyze the C source code to extract these aliases, and add them to the callgraph.

5.2 Binary Analysis

Confine uses the Angr framework (Shoshitaishvili et al., 2016) to perform binary code analysis for two main purposes. First, Confine uses Angr to identify the libc functions that are imported by all the invoked programs and all their libraries. Second, Confine uses the CFGFast analysis of Angr to extract the control flow graph (CFG) of the executables and their libraries. Confine relies on this CFG to identify all the invocation sites of libc wrapper functions and direct system calls, and then to perform reaching definitions analysis on their arguments to concretize their values.

Data Flow Analysis Confine leverages the `Function Manager` object (created by Angr during function recovery) to extract the basic blocks of each function, and analyzes Angr's intermediate representation (VEX IR) to identify direct system call invocation sites and function call sites. The IR supports six types of jumps, represented by the `jumpkinds` enum. Among the six, Confine uses the two that correspond to function calls (`Ijk_Call`) and direct system call invocations (`Ijk_Sys`). From the identified function call sites, Confine only considers those that target a glibc wrapper function or its weak aliases. It selects them by comparing the call site target addresses with the glibc function addresses collected by the `Function Manager`.

After identifying all function call sites and system call invocation sites, Confine performs backwards intra-procedural data flow analysis to derive concrete values for the arguments passed at each call site. It employs the reaching definitions analysis pass of Angr for this purpose. This analysis requires an observation point to be defined for each register that is used for passing an argument value. The analysis then extracts the address where each register is defined (`CodeLoc`) along with its possible value(s). For example, to identify the system call number of a direct system call invocation, Confine defines an observation point for the `rax` register. We use the same technique to perform a one-time analysis of glibc for mapping the arguments of wrapper functions to their respective system call arguments, and for identifying system call argument values used by complex glibc functions.

It is worth noting that Angr had limited support for some instructions encountered in the programs of our data set (e.g., `cmpxchg`). We resolved this issue by extending its VEX execution engine and implementing separate handlers for these instructions.

Wrapper Function Argument Mapping Within a wrapper function, we must ensure that its arguments flow unmodified to the system call invocation site. Since wrapper functions directly invoke their respective system call (without calling any other internal functions), performing intra-procedural analysis for them is sufficient to ensure that the mapping holds.

Confine uses Angr to analyze glibc wrapper functions and identify any modifications to the passed argument values before they are supplied to the corresponding system call. An observation point is defined for each register used to pass a system call argument, on which

reaching definitions analysis is then performed. Although the analysis attempts to identify the final instruction that assigns a value to the register (as well as the value itself), the analysis may fail depending on whether the register is used or modified within the function.

If the register is not used nor modified in the function, Angr does not return a valid output, which means that the register will be assigned a value that flows in from the wrapper's caller. If the register is used but *not* modified in the function, the analysis returns `Undefined` as the value of the register, and `External` for `CodeLoc` (the address of the location where the register is assigned). This output indicates that although the register is *used* in the function (e.g., to perform a comparison), its value again is assigned outside the function. In both these cases the one-to-one mapping between the wrapper's and the system call's arguments holds.

If the register is modified inside the function, the returned `Codeloc` is an address inside the function, and the one-to-one mapping does not hold. In that case, a new observation point is defined on that address, and reaching definition analysis is performed again. This process continues recursively until either i) the value with which the register is initialized is reached, i.e., argument value is hardcoded within the function; or ii) the recursive analysis returns `Undefined`, which means that the value is passed to the wrapper by the caller, i.e., the order of the arguments is modified by the wrapper prior to invoking the system call. For example, the `clone()` wrapper function moves the value of the `r8` register (the fifth argument of the wrapper function) to the `rdx` register before invoking the `clone` system call. In such cases, Confine modifies the mapping accordingly.

Complex Glibc Functions As mentioned in Section 4.2, complex libc functions often internally invoke several system calls (direct or through libc wrappers). Confine uses the glibc callgraph to map each complex function to the system calls it may invoke, and then uses binary analysis to concretize their arguments. Since the glibc callgraph is generated by analyzing the source code, there are functions in the callgraph that do not exist in the final library due to optimizations (e.g., function inlining). To minimize this inconsistency, we compile glibc using the lowest optimization level possible, so that the resulting binary code has the highest correspondence with the callgraph.

To reduce Confine's analysis time and improve usability, we use caching of intermediate analysis results extensively. For each encountered glibc version used by a container, the derived mapping between glibc functions, their system calls, and their argument values, is generated only once, and is then stored for future use when the same glibc version is encountered in another container. This allows Confine to skip this part of the analysis most of the time once a diverse-enough set of glibc versions has been encountered.

Dynamically Loaded Libraries An issue that requires special consideration is dynamic loading, a mechanism through which applications can load modules on demand throughout their execution. The `dlopen()`, `dlsym()`, and `dlclose()` API functions are used to load a library, retrieve its symbols, and close it, respectively. Because these operations are performed at runtime, any libraries loaded in this way cannot be identified by looking at the application's ELF binary header. For instance, Apache Httpd uses this feature to load libraries based on the user-defined configuration. Quach and Prakash (Quach and Prakash, 2019) showed that only around 3% of the 3,174 programs and 2% of the 4,292

libraries analyzed in their dataset used these features, all of which loaded the required libraries during initialization.

To identify such dynamically loaded libraries, we monitor the list of libraries loaded by the application at runtime through the `/proc` virtual file system, which provides this information for every process.

One consideration is that if an application dynamically loads libc, we cannot identify the individual functions imported by the application, and would have to retain all system calls made by libc. However, it is unlikely that libc will be loaded in this fashion, as dynamic loading is used for modules that provide *additional* functionality to the application. We did not encounter any such case in our experiments.

5.3 Seccomp Profile Generation

Confine automatically generates Seccomp policies by classifying all system calls present on the final list of required system calls as "permitted," and assigning them to an allow list, denying anything not provided in this list. The Docker Seccomp ruleset requires the name of the allowed system calls, while our analysis of the containers generates system call numbers. Confine maps all the available system calls in the kernel to their respective number by using the symbol information related to the names of the system calls from the `procfs` pseudo-filesystem. Based on the `sys/syscall.h` header file, Confine maps the system call name to its number, and uses it to convert the permitted system call numbers to their names. Finally, Confine creates the Seccomp profile with an allow list containing these system calls and applies it to the container.

Container-wide Filter Enforcement Docker uses a JSON file to define the permitted system calls. Listing 1.1 shows a sample ruleset which only allows the *pwrite64* system call. The default action for this ruleset is to deny all system calls, except those specified under the `syscalls` tag. Each system call is specified by three arguments: its name, the action, and its arguments.

Listing 1.1: Example of a Docker Seccomp ruleset file.

```
{
    "defaultAction": "SCMP_ACT_KILL",
    "architectures": [
        "SCMP_ARCH_X86_64",
        "SCMP_ARCH_X86",
        "SCMP_ARCH_X32"
    ],
    "syscalls": [
        {
            "name": "pwrite64",
            "action": "SCMP_ACT_ALLOW",
            "args": []
        }
    ]
```

```
15 }
```

Application-specific Filter Enforcement As discussed in Section 4.3, we use a "proxy" C program to install Confine's application-level Seccomp profile. This program uses the `prctl` system call to install the updated filter on the current process, and then invokes the target binary as part of the same process using the `execve` system call. We compile the proxy program statically to avert any failures due to library dependencies not available in the container. This would mainly happen in Docker images in which the programs have been statically linked with glibc. Our proxy program itself requires a limited set of system calls to install the application-specific filter (`exit_group`, `brk`, `mmap`, `munmap`, `write`, `fstat`, and `execve`). Note that these system calls are required by most programs (24 containers in our dataset), and not filtering them does not reduce the security benefit. We provide a more detailed discussion regarding these system calls and how their inclusion affects the security benefit of Confine in Section 7.

After we generate our Seccomp installation program, we need to modify the container to execute it instead of the main application, preferably without rebuilding the Docker image. Each Docker image is built based on a *Dockerfile*. This file includes the information required to launch a container running the applications requested by the creator of the image. One of the variables specified in this file is the *entrypoint*, which specifies the program to be executed upon launching the container. Most Docker images typically use a bash script as their entrypoint. This script usually performs operations needed to prepare the filesystem or general settings (e.g., add a user, create a directory) for the correct execution of the final target program. We need to replace this script with our own custom-built script, which performs operations needed by Confine and then invokes the original script.

Confine's custom script performs the following operations: 1) makes a backup of the main target binary; 2) overwrites the original binary with our proxy program; and 3) executes the original entrypoint script. There are two ways to modify the entrypoint of a container: change the Dockerfile and rebuild the image, or change the entrypoint temporarily when the container is launched. We use the second option, which is less intrusive and can be easily applied to any Docker image, without having to worry about rebuilding the image itself. For example, the Nginx Docker image uses the `docker-entrypoint.sh` script as its entrypoint, which executes the arguments passed to the container in its final step. These arguments are usually the name of the main binary along with any arguments it may require. We modify the entrypoint at runtime to execute our specially crafted script and then execute the `docker-entrypoint.sh` script available in the original Docker image.

In cases where the Docker image does not have any bash program available to execute scripts (e.g., images based on Alpine Linux) and the main program is executed instead as the entrypoint, we modify the entrypoint (using the runtime option) to directly launch our proxy program.

6 Experimental Evaluation

To assess the security benefits of Confine, we used a base Linux system with kernel v5.4, which provides 335 system calls. We evaluated Confine with 27 publicly available Docker

container images available from Docker Hub (doc, c2023a). Due to the nature of our application-specific system call filtering technique, we did not consider Docker images which are meant to be used as vanilla installations for other applications to build upon. When launched, these containers either do not run any program at all, or only execute a shell. Instead, we only apply Confine to containers that run a target long-running application, in which the initialization and post-initialization phases actually differ. Also, we do not consider images in which the main application is a Golang binary, as they are statically linked, and as a result, the argument values cannot be extracted by performing our intraprocedural analysis at system call invocation sites.

We ended up with these 27 Docker images after evaluating the top-100 official Docker images available on Docker Hub, and selecting the ones that i) execute a single target application (other than bash); ii) do not require any manual registration or payment on Docker Hub; and iii) are compatible with Angr for extracting the CFG for the invoked programs and its libraries (we encountered just four images for which Angr failed to extract the CFG of the main executable).

We only analyzed the top-100 official Docker images available on Docker Hub because as we mentioned in Section 5, we require the developer to provide the name of the final binary to generate the application-specific filter. Therefore, we analyzed the top 100 Docker images to reduce the manual effort required.

To ensure that the generated system call policies do not break any functionality, we performed additional validation runs. First, we check if the container does not exit abrubtly when being launched with the specified Seccomp profile. As we mentioned in Section 5.3, the Dockerfile specifies the application the container must invoke upon launch. If this application exits (or crashes), the container exits, and thus we verify that this does not happen.

Even if the application remains running, however, it might still encounter errors. For example, it might encounter exceptions that are gracefully handled by the application, but still cause problems in its correct operation. To capture these cases, we check the log files generated by the container. Docker provides a streamlined process of reading the logs produced by the containerized application. We compare the logs produced by the hardened container with the default container. Because values in the logs, such as timestamps and process IDs, might differ between different executions, we ignore these values.

6.1 Filtered System Calls

First, Confine automatically analyzes each container and extracts the list of system calls required by its binaries based on the analysis described in Section 5.2. Then, it generates a Seccomp filter to prohibit the use of all remaining system calls. Finally, we run the container on a Docker Engine, along with our filter, to validate the correctness of our analysis.

We assess the effectiveness of our approach by measuring the number of filtered system calls per container. Each system call is an entry point to some kernel functionality, and thus completely disabling a system call is equivalent to preventing the exposure of vulnerabilities in all relevant code of that kernel functionality (in addition to prohibiting the use of that system call as part of malicious code)—we have measured the degree of attack surface reduction in terms of known CVEs that become neutralized and present our results in Section 6.3. We leave the actual *removal* of the kernel code related to each

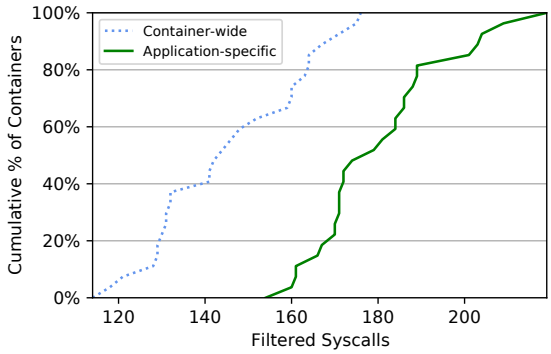

Fig. 4: Cumulative distribution of the number of filtered system calls as a percentage of all tested Docker containers for container-wide and application-specific system call filtering. Application-specific filtering increases the average number of filtered system calls by 25%.

system call as part of our future work, but the number of filtered system calls is indicative of the amount of kernel code that could potentially be removed.

Figure 4 shows the cumulative distribution of the number of removed system calls across all containers in our dataset for container-wide filtering and application-specific filtering. Confine can filter 144 system calls or more for half of the Docker images in our dataset across the entire container. By differentiating between a container's requirements during its initialization phase and after it starts executing the target application, Confine's application-specific filtering proves to be quite effective, increasing the number of filtered system calls further, by 25% on average. As an example, for the Nginx (doc, c2023c) and Apache Httpd (doc, c2023b) containers, applying container-wide filtering disables 159 system calls for Nginx and 176 for Apache, while application-specific filtering increases the number of filtered system calls to 204 and 203, respectively.

As shown in Figure 5 (bottom two segments of each bar), application-specific filtering is more effective for containers that run many programs during their initialization phase. For example, Postgres (doc, c2023e) and Percona (doc, c2023d) have the highest increase in the number of filtered system calls (63% and 51%), because both rely on many utility programs (e.g., `mkdir`, `awk`, `cat`) to set up the container environment for the correct execution of the main program. In contrast, Julia (doc, c2023a) does not have any initialization scripts and directly executes the main program. Its slight increase in the number of filtered system calls (4%) is due to the requirements of the Docker runtime itself, which still invokes a few system calls needed to launch the container that can be safely filtered afterwards.

Confine's application-specific filter generation is similar to applying previous works that generate system call filters for a binary (e.g., sysfilter DeMarinis et al. (2020)). However, sysfilter fails to analyze libraries that are not compiled with debug symbols. Since there were libraries in our dataset that did not contain debug symbols, we were not able to perform a complete comparison. Furthermore, sysfilter does not restrict the system call arguments which is one of the contributions of our work.

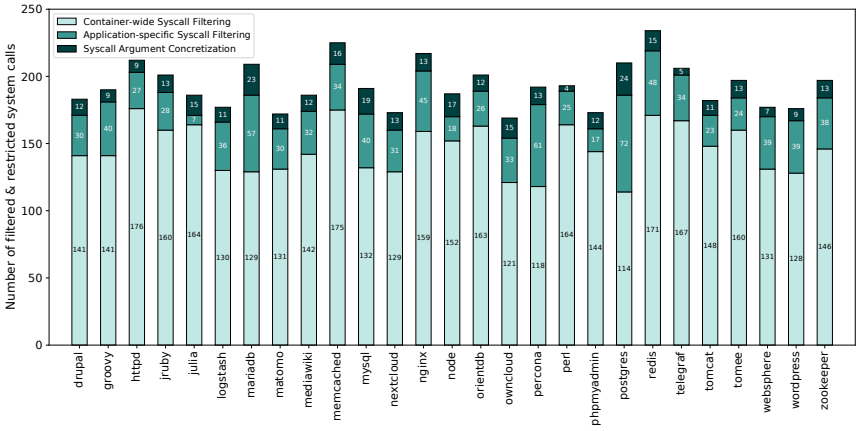

Fig. 5: Number of disabled system calls by container-wide (bottom part) and application-specific (middle part) filtering, and of restricted system calls by argument concretization (top part).

6.2 Restricted System Call Arguments

Restricting the allowable argument values of system calls that cannot be filtered entirely still contributes in disabling parts of a system call's crucial functionality. Currently, Confine attempts to restrict 145 different constant arguments across 118 system calls. Given that Confine's argument value inspection relies on conservative, best-effort static analysis at the binary level, only a subset of system calls can be handled—those that conform to the following conditions: i) have at least one constant argument; ii) remain unfiltered across the majority of containers; and iii) their arguments can be mapped to the corresponding Libc functions.

Going back to Figure 5, the topmost segment of each bar shows the number of restricted (but not filtered) system calls for which at least one argument can be concretized. Confine restricts 421 constant arguments across the 27 containers. For example, Confine limits the `request` argument of the `ioctl` system call in 24 containers. The Linux header file `<sys/ioctl.h>` contains several predefined values for the `request` argument. Concretizing this argument results in the neutralization of most kernel vulnerabilities related to built-in or additional devices. CVE-2019-6974 (cve, 2019) is such a use-after-free vulnerability in the KVM hypervisor that can be triggered by calling `ioctl` with `KVM_CREATE_DEVICE` as an argument—one of the argument values that Confine prohibits across all 24 containers.

Other examples of restricted arguments include: the `level` and `optname` arguments of `setsockopt` and `getsockopt` across 10 and 16 containers, respectively, and the `flags` argument of `clone` and `mmap` across 27 and 4 containers. More specifically, for `mmap`, the `flags` argument can take 22 different values, which can also be combined with each other through a bitwise OR. Among these values, Confine disables nine, and allows only some combinations of the remaining 13 values in the four containers where `mmap` is restricted. Furthermore, among the disabled protocols for the `socket` system call is IPsec, set by the `NETLINK_XFRM` flag, which is filtered across seven containers. This flag is associated with a heap buffer overflow flaw found in IPsec, which may lead to a local privilege escalation

attack (cve, 2012). Also, the `clone` system call accepts 53 different flags and signals, but the 27 evaluated containers use only three unique combinations of them. Consequently, most vulnerabilities related to `clone` (cve, 2012, 2013) can be neutralized as a result of Confine's argument concretization.

The Perl container has the lowest number of concretized arguments due to breadth and flexibility of the Perl API, as implemented in the `libperl.so` library. For example, `setsockopt` and `ioctl` cannot be restricted because their argument values cannot be identified in just one call site for each of them in this library. These call sites are located in the `Perl_pp_ssockopt` and `Perl_pp_ioctl` functions, respectively, the arguments of which will become available only at runtime, depending on the user program that will invoke them. Language-level analysis of Perl scripts could be used to identify those argument values, but this type of analysis is beyond the scope of our work. We further discuss the implications of performing this analysis for dynamically-typed languages in Section 7.

6.3 Security Evaluation

System calls are the main entry point into the kernel. While security-critical system calls are typically used as part of exploit payloads (Mishra and Polychronakis, 2020), *any* system call can be used to exploit a vulnerability in the kernel. Previous works (Kemerlis, 2015; Kemerlis et al., 2012, 2014; Pomonis et al., 2017) have shown that malicious users can attack the kernel and either leak sensitive data or perform privilege escalation. In most cases, these attacks are performed by exploiting vulnerabilities accessible through system calls. Consequently, when considering the kernel's attack surface, *all* system calls have potential for a malicious user.

To demonstrate the security benefit of Confine, we measure the additional Linux kernel vulnerabilities mitigated due to filtering or restricting system calls. We consider vulnerabilities instead of proof-of-concept (PoC) exploits, because each vulnerability can be used by different PoCs. Mitigating a single vulnerability breaks *all* PoCs that leverage it, while preventing any single PoC may not break the rest. To that end, we need to identify vulnerabilities that can be exploited through the invocation of one or more system calls, and whether any specific system call argument value is required for a successful exploitation. We built a two-level mapping between system calls and vulnerabilities as follows. First, to derive a coarse-grained mapping, we scrape patches assigned to CVEs to identify the functions that hold the root cause of a vulnerability. Then we use the Linux kernel callgraph to identify from which system calls the vulnerable code can be accessed. Second, we derive a fine-grained mapping that considers the system call argument values that must be used to exploit an accessible vulnerability.

Coarse-grained Syscall to CVE Mapping To perform our analysis, we crawled the CVE website (cve, c2023) for Linux kernel vulnerabilities using a custom automated tool. The tool parses each commit in the Linux kernel's Git repository to find the corresponding patch for a given CVE, and retrieves the relevant file and function that was modified by the patch. After mapping CVEs to their respective functions, we built the Linux kernel callgraph and analyzed which parts of it can be exclusively accessed by a given system call.

We constructed the Linux kernel's callgraph using KIRIN (Zhang et al., 2019). This allows us to map which functions in the kernel are invoked from which system call, and

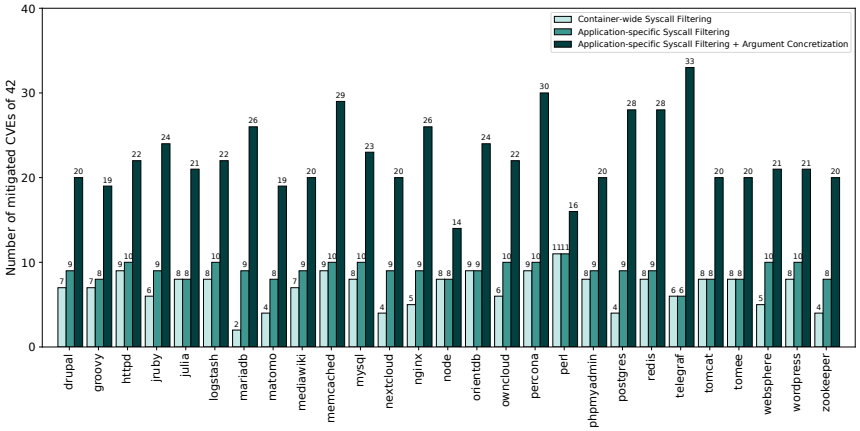

Fig. 6: Our fine-grained system call to CVE mapping shows that while application-specific filtering only slightly increases the number of mitigated kernel CVEs compared to container-wide filtering (left and middle bars), the addition of system call argument concretization in many cases more than doubles the number of mitigated CVEs per container (rightmost bars).

therefore reason about which part of the kernel's code will never be invoked when a set of system calls are filtered. We discovered that while there are only a few CVEs directly associated with the code of filtered system calls, many CVEs are associated with files and functions that are invoked exclusively by the code of filtered system calls. By matching the CVEs to the callgraph created by KIRIN, we were able to pinpoint all the vulnerabilities that are related to the set of system calls filtered by a given container. This provides us with a quantifiable property to assess the attack surface reduction achieved by our method, i.e., the number of CVEs that would have been neutralized for a given container, if the respective system call policy generated by Confine was applied.

Based on our coarse-grained analysis, in addition to the 25 CVEs mitigated by Docker's default Seccomp policy, 20 CVEs across all studied containers are effectively removed (i.e., the respective vulnerabilities cannot be triggered by the attacker) by applying our generated policies. One example is the recently disclosed "Dirty Pipe" vulnerabilty (cve, 2022) which is mapped to the `splice` and `pipe` system calls. An unprivileged local user could use this flaw to write to the page cache of a read-only file (e.g., `/etc/passwd`) to escalate their privileges in the container. Then they could misuse this privilege to escape the container and gain access to the underlying host. Confine increases the number of containers where this system call is filtered from 19 in container-wide filtering (Ghavamnia et al., 2020a) to 26, by applying application-specific filtering.

Fine-grained Syscall to CVE Mapping The coarse-grained CVE-to-system call mapping is not enough for assessing the added benefit of system call concretization. To reason whether disabling *part* of a system call's functionality affects the exploitability of a given vulnerability, we gathered at least one proof-of-concept exploit per CVE, and manually studied how the essential system calls *and their arguments* affect the successful operation

of the exploit. We further expanded the set of kernel CVEs with more recent exploits for which publicly available exploits are available.

Using this process, we were able to map 42 CVEs to the crucial system calls *and* arguments necessary for their successful exploitation. As an example of our mapping, CVE-2016-3134 (cve, 2016) is a vulnerability in the netfilter subsystem of the Linux kernel, which can only be triggered by invoking the `setsockopt` system call with `IPT_SO_SET_REPLACE` as its third argument (`optname`).

Given this set of 42 CVEs, Confine mitigates at least one of them in a container after applying argument concretization, and 28 of which cannot be mitigated by container-wide filtering. As shown in Figure 6, application-specific filtering offers only a slight advantage compared to container-wide filtering, but the addition of argument concretization offers a significant improvement. In the best case, the number of mitigated CVEs for Telegraf increases from six to 32, while in the worst case only five additional CVEs are mitigated for Perl. These CVEs, that are mitigated due to applying argument concretization, cannot be mitigated by other system call filtering approaches (doc, 2023; Canella et al., 2021; DeMarinis et al., 2020; Ghavamnia et al., 2020a; Lei et al., 2017; Wan et al., 2017) that do not restrict the system call arguments, because the container or application uses the respective system call.

Table 1 provides a description of the CVEs blocked by Confine across all containers, as well as the number of containers in which a given CVE is blocked. It is worth noting that CVEs related to the `setsockopt`, `socket`, and `ioctl` system calls cannot be mitigated by system call filtering alone. Confine manages to mitigate these CVEs across 10, 14, and 24 containers, respectively, as a result of argument concretization. Overall, Confine mitigates 276% more CVEs per container (22 CVEs on average) compared to container-wide filtering (seven CVEs on average).

7 Discussion and Limitations

In this work, we mainly focus on a container's execution phases to show that even without complex analysis of the target program (e.g., temporal system call specialization (Ghavamnia et al., 2020b)), we can still generate restrictive system call filters and enforce them using a ready-to-use, non-intrusive technique. It is possible to further restrict the final application by applying temporal system call specialization (Ghavamnia et al., 2020b) after the binary enters its own serving phase. Applying configuration-based software debloating (Ghavamnia et al., 2022; Koo et al., 2019) could also result in even more restrictive filters by removing unnecessary features depending on the selected configuration options. We leave the integration of such techniques into Confine as part of our future work.

As we discussed in Section 6, extracting system call filters for dynamically-typed languages (e.g., Perl, Python) is more complicated. These languages typically have an interpreter which provides APIs for accessing the underlying OS features and system calls. Confine currently generates a system call profile for these languages by considering *all* system call requirements of the interpreter. Our approach suffers from overapproximation, because the actual program may not require all the features of the interpreter. We can generate a more restrictive system call profile by analyzing the interpreter and its APIs. Saphire (Bulekov et al., 2021) builds more restrictive system call filters for PHP programs

Table 1: Number of containers for which a given CVE is blocked due to W: container-wide filtering, S: application-specific filtering, A: application-specific filtering with argument concretization.

CVE ID	System Calls	Description	W	S	A
2022-27666	socket, setsockopt	A heap buffer overflow flaw was found in IPsec can cause privilege gain via overwrite kernel heap objects	0	0	7
2022-0847	splice, pipe	A local privilege escalation vuln. that allows an attacker to overwrite data in arbitrary read-only files	19	26	26
2019-6974	ioctl	A use-after-free vuln. triggered by calling ioctl with kvm_ioctl_create_device as its argument	0	0	24
2012-2127	clone	Remote attackers can cause DOS via calling CLONE_NEWPID clone system call	0	0	27
2013-1858	clone	Improper handling in a combination of the CLONE_NEWUSER and CLONE_FS flags of clone system call allows local users to gain privileges	0	0	27
2009-4131	ioctl	The EXT4_IOC_MOVE_EXT implementation in the ext4 FS allows overwriting of files	0	0	24
2009-4141	ioctl	Privilege gain using UAF vuln. via vectors involving enabling O_ASYNC on a locked file	0	0	23
2010-0437	ioctl, socket	DoS through an IPv6 TUN network interface	0	0	24
2010-4077	ioctl	Access sensitive information from kernel memory via a TIOCGICOUNT ioctl call	0	0	24
2010-4158	socket, setsockopt	Access sensitive information from kernel memory via a crafted socket filter	0	0	1
2012-0207	socket	Remote attackers can cause a DoS via IGMP packets	0	0	4
2014-2851	socket	Integer overflow in ping_init_sock function can cause a DoS or privilege gain	0	0	4
2014-5207	mount	A vuln. in namespace.c can cause privilege gain or DoS via a remount of a bind mount using MS_BIND flag	10	25	25
2016-3134	setsockopt	A vuln. can cause privilege gain or DoS via an IPT_SO_SET_REPLACE setsockopt call	0	0	7
2009-1337	clone, execve	A vuln. allows local users to send signal to a process and then launch a setuid app.	0	0	27
2010-3081	setsockopt, getsockopt	A stack underflow vuln can lead to privilege gain by using the compat_mc_getsockopt function	0	0	7
2010-4258	clone, splice	A vuln. can cause privilege gain via the clear_child_tid feature and the splice syscall	19	26	27
2009-0676	getsockopt	Potentially sensitive information can be obtained from kernel memory via an SO_BSDCOMPAT getsockopt request	0	0	7
2016-1583	mmap, clone	Crafted mmap syscall can lead to privilege gain and DoS	0	0	27
2016-5195	madvise, mmap	Race condition causes privilege gain due to incorrect handling of COW which allows write to read-only memory	0	0	6
2016-4578	ioctl	Sensitive information can be obtained from kernel memory via crafted use of the ALSA timer interface	0	0	24
2016-4997	setsockopt, socket	Vulnerability in the IPT_SO_SET_REPLACE and IP6T_SO_SET_REPLACE setsockopt implementations can lead to privilege gain or DoS.	0	0	7
2016-8655	setsockopt	Bug in the packet_set_ring and packet_setsockopt functions can lead to privilege gain or DoS	0	0	3
2016-9793	setsockopt	The sock_setsockopt function can lead to DoS by a crafted setsockopt syscall	0	0	5
2017-15649	setsockopt, socket	Vulnerability in af_packet.c leads to privilege gain via crafted syscalls	0	0	10
2017-16939	setsockopt, socket	SO_RCVBUF setsockopt syscall and XFRM_MSG_GETPOLICY Netlink msg leads to privlege gain or DoS	0	0	4
2017-2671	socket	Access to the protocol value of IPPROTO_ICMP in a socket syscall can cause DoS	0	0	4
2010-4165	setsockopt, socket	Improper handling of TCP_MAXSEG values in do_tcp_setsockopt causes DoS via a setsockopt call	0	0	2
2019-9213	mmap	Improper check of mmap min. address in expand_downwards allows exploitation of null ptr derefs	0	0	4
2009-2767	clock_nanosleep	DoS or privilege gain due to CLOCK_MONOTONIC_RAW clock_nanosleep call	3	8	8
2012-0957	uname, personality	The override_release function allows information to be obtained from kernel memory via a uname and a UNAME26 personality syscall	27	27	27
2014-3631	add_key	Multiple keyctl newring operations followed by a keyctl timeout operation can cause DoS	19	27	27
2016-0728	keyctl	The join_session_keyring function can lead to privilege gain or DoS via crafted keyctl commands	24	27	27
2017-7533	inotify_add_watch, inotify_init1	Race condition leads to privilege gain or DoS by leveraging simultaneous execution of the inotify_handle_event and vfs_rename functions	23	25	27
2019-11599	mmap, ioctl	Race condition allows information leak or DoS by mmget_not_zero or get_task_mm calls	0	0	24
2012-3375	epoll_ctl	Improper handling of EPOLL_CTL_ADD operations can lead to DoS	1	1	1
2011-1082	epoll_create	Crafted application that makes epoll_create and epoll_ctl syscalls can cause DoS	1	1	1
2014-4014	chmod	Vulnerability allows bypass of intended chmod restrictions	1	2	3
2014-7822	splice	Crafted splice syscall can cause DoS	19	26	26
2017-11176	socket	Attackers can cause DoS by using a user-space close of a Netlink socket	0	0	4
2017-5123	waitid	Insufficient data validation in waitid allowed an user to escape sandboxes on Linux	20	22	22
2009-1527	ptrace	Race condition in the ptrace_attach function can lead to privilege gain via a PTRACE_ATTACH ptrace call	24	24	24

by analyzing the interpreter. This work shows performing more fine-tuned system call filtering on dynamically-typed languages requires analyzing the interpreter.

Furthermore, as we discussed in Section 4, among the approaches we could use to install the application-specific filter, we choose the "proxy" program technique due to its low intrusiveness. However, this approach requires the `execve` system call to remain

non-blocked. In cases that the target program does not need `execve` (only 3 among our containers), this becomes a security limitation of the proxy program approach, as `execve` could otherwise have been blocked. The binary patching approach (as mentioned in Section 4.3), however, circumvents this issue at the cost of being more intrusive, requiring modification of the binary. Switching among these two approaches is a matter of engineering effort, and we plan to support both as part of our future work.

We considered the top-100 official container images and selected only those that meet specific criteria, including the ability to execute a single target application, without requiring manual registration or payment on Docker Hub, and compatibility with Angr for extracting the control flow graph (CFG) of the invoked programs and their libraries. Therefore, we ended up with the 27 Docker images mentioned in the section 6. Among the top-100 official container, there were 11 that executed more than one application in their post-initialization phase. We did not apply our technique to these Docker images, because they required more tuning to generate separate post-initialization Seccomp profiles for the different applications that remain running. These were mainly containers which executed programs developed using the Erlang (doc, c2023a) programming language.

Among the rest, the Cassandra image (doc, c2023a) is the only one that required some manual analysis of a script executed during its post-initialization phase, which performs another set of operations to prepare the environment before executing the target application. Confine could still harden the image by manually modifying the final script to execute our specially-crafted C program instead of the main application. Additionally, we encountered only four images for which Angr was unable to extract the control flow graph (CFG) of the main executable. Of the remaining Docker images, they were intended as vanilla installations for other applications to build upon, and thus did not meet our specific criteria for evaluation.

8 Related Work

System call policy generation through static source code analysis has been a widely used approach in the fields of host-based intrusion detection (Feng et al., 2004; Forrest et al., 1996; Jain and Sekar, 2000; Kruegel et al., 2005; Parampalli et al., 2008; Wagner and Dean, 2001) and sandboxing (Garfinkel et al., 2004; Rajagopalan et al., 2005). However, these previous works target programs from more than twenty years ago which were less complicated and mostly used static linking. Furthermore, they either do not restrict the argument values passed to system calls at all, or use dynamic analysis and anomaly detection to extract the used values of the system call arguments. This can cause soundness issues and break the target program.

Given that the main focus of our work is on attack surface reduction, and that programs nowadays have become much more complicated, we discuss more recent related works in this context.

8.1 Container Security and Debloating

Given the increased use of containers, previous works have focused on evaluating the security of container environments (Combe et al., 2016; Lin et al., 2018; Shu et al., 2017)

and reducing attack surface by applying more restrictive security policies (Findlay et al., 2021; Loukidis-Andreou et al., 2018), or splitting them into smaller containers (Rastogi et al., 2017a,b).

Generating system call policies for containers has also been explored by prior work. Wan et al. (Wan et al., 2017) and DockerSlim (doc, 2023) apply dynamic analysis to container hardening. They profile applications running in the container to extract the set of system calls they use and generate corresponding Seccomp filters. DockerSlim also removes files which are unused during the profiling.

Speaker (Lei et al., 2017) is another system that relies on dynamic analysis to extract a container's system calls requirements. Similarly to Confine, Speaker considers the two execution phases of a container, and generates a boot-time and runtime Seccomp profile. The main difference compared to our work is that Confine uses static analysis to identify the required system calls, and more importantly, Speaker uses a more intrusive mechanism to apply the application-specific Seccomp profile, by modifying the Linux kernel.

In general, all the works mentioned in this section rely solely on dynamic analys to extract system calls, while our approach generates system call policies using static code analysis. In addition, Confine also performs argument concretization, a feature that to the best of our knowledge is not supported by any previous container hardening approach. Therefore, comparing our approach with these related works would not be fair to directly compare them.

8.2 Application Debloating

Many of the prior works on software debloating have focused on removing excessive code from individual processes. Various static and dynamic code analysis techniques have been proposed for debloating of software developed in different programming languages, including C/C++ (Agadakos et al., 2019; Alhanahnah et al., 2021; Ghaffarinia and Hamlen, 2019; Hashim Sharif and Zaffar, 2018; Heo et al., 2018; Mulliner and Neugschwandtner, 2015; Porter et al., 2020; Qian et al., 2019, 2020; Quach et al., 2018; Song and Xing, 2018), Java (Jiang et al., 2016; Suparna Bhattacharya and Nanda, 2013; Yufei Jiang and Liu, 2016), and PHP (Amin Azad et al., 2019).

Similar to Confine, Shredder (Mishra and Polychronakis, 2018) restricts system calls by limiting their arguments, but focuses on Windows applications and modifies the user-space system call interface to enforce the restriction. Saffire (Mishra and Polychronakis, 2020) also applies argument-level filtering by generating specialized versions of library functions. In contrast to Confine, it needs the source code of the application to perform its analysis.

Sysfilter (DeMarinis et al., 2020) and Chestnut (Canella et al., 2021) both apply binary analysis to identify the system call requirements of a given application. However, neither apply any restrictions for the system call arguments. Temporal system call specialization (Ghavamnia et al., 2020b) takes the execution phase of server applications into account and creates two separate filters for each phase. However, it requires the source code of the application to perform pointer analysis and apply pruning on the application callgraph.

8.3 Kernel Debloating

There have been several works that focus on minimizing the kernel's footprint and customizing its code according to user requirements. KASR (Zhang et al., 2018) and FACE-CHANGE (Zhongshu Gu and Xu, 2014) use dynamic analysis to generate kernel profiles based on the requirements of a single application. Then they use virtualization mechanisms to limit each application to its pre-generated profile. Shard (Abubakar et al., 2021) combines static and dynamic analysis to generate kernel profiles per application and per system call. Kurmus et al. (Kurmus et al., 2013) propose a system for the automated generation of kernel configuration files for tailoring the Linux kernel to special workloads.

9 Conclusion

Our work was motivated by the lack of a generic solution for the automated generation of restrictive system call policies for container environments—one that does not rely on training with realistic workloads, which is a cumbersome and error-prone method. Furthermore, our work shows that further attack surface reduction is possible by i) moving from container-wide to application-specific system call policies, and ii) filtering not only at the system call level, but also at the system call argument level. Deferring the installation of the filter from the launch time of the container to the launch time of the container's target application increases the number of system calls that can be filtered, as those that are required during the container's initialization phase but are not needed by the target application can be safely removed right before the application is launched. In addition, the functionality of the remaining system calls can be limited by concretizing when possible the arguments of system calls according to the values passed to these system calls by the program.

We have implemented these capabilities in Confine, which automates the whole process by scanning a container image and generating a container-wide system call policy and an application-specific system call policy that is transparently enforced, without the need to modify the kernel or the target application. The results of our experimental evaluation show that moving from container-wide to application-specific filtering increases the number of filtered system calls by 25% on average, while argument concretization results in the neutralization of more Linux kernel CVEs compared to plain system call filtering.

Acknowledgments This work was supported by the Office of Naval Research (ONR) through award N00014-17-1-2891, the National Science Foundation (NSF) through award CNS-1749895, and the Defense Advanced Research Projects Agency (DARPA) through award D18AP00045, with additional support by Accenture. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the ONR, NSF, DARPA, or Accenture.

References

National Vulnerability Database, CVE-2012-2127, 2012. `https://nvd.nist.gov/vuln/detail/CVE-2012-2127`.

National Vulnerability Database, CVE-2013-1858, 2013. https://nvd.nist.gov/vuln/detail/CVE-2013-1858.
National Vulnerability Database, CVE-2016-3134, 2016. https://www.cvedetails.com/cve/CVE-2016-3134/.
National Vulnerability Database, CVE-2017-5123, 2017. https://www.cvedetails.com/cve/CVE-2017-5123/.
National Vulnerability Database, CVE-2019-6974, 2019. https://nvd.nist.gov/vuln/detail/CVE-2019-6974.
National Vulnerability Database, CVE-2022-0847, 2022. https://nvd.nist.gov/vuln/detail/CVE-2022-0847.
Slim.AI, slimtoolkit. https://dockersl.im, 2023.
CVE details, common vulnerabilities and exposures database. https://www.cvedetails.com, c2023.
Docker Hub. https://hub.docker.com, c2023a.
Docker Hub, Apache httpd. https://hub.docker.com/_/httpd, c2023b.
Docker Hub, Nginx. https://hub.docker.com/_/nginx, c2023c.
Docker Hub, Percona. https://hub.docker.com/_/percona, c2023d.
Docker Hub, Postgres. https://hub.docker.com/_/postgres, c2023e.
GNU Compiler Collection, c2023. https://gcc.gnu.org.
Production-Grade Container Orchestration, Kubernetes, c2023. https://kubernetes.io.
Linux manual page, ld.so(8). https://man7.org/linux/man-pages/man8/ld.so.8.html, c2023.
Linux Programmer's Manual, Capabilities(7), c2023a. http://man7.org/linux/man-pages/man7/capabilities.7.html.
Linux Programmer's Manual, Cgroups(7), c2023b. http://man7.org/linux/man-pages/man7/cgroups.7.html.
Linux Programmer's Manual, Namespaces(7), c2023c. http://man7.org/linux/man-pages/man7/namespaces.7.html.
Docker Hub, MongoDB, c2023. https://hub.docker.com/_/mongo/.
Seccomp BPF (SECure COMPuting with filters), c2023. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.
Sysdig, c2023. https://sysdig.com/.
Muhammad Abubakar, Adil Ahmad, Pedro Fonseca, and Dongyan Xu. Shard: Fine-grained kernel specialization with context-aware hardening. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2435–2452, 2021.
Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P Kemerlis, and Georgios Portokalidis. Nibbler: debloating binary shared libraries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.
Mohannad Alhanahnah, Rithik Jain, Vaibhav Rastogi, Somesh Jha, and Thomas Reps. Lightweight, multi-stage, compiler-assisted application specialization, 2021.
Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *Proceedings of the 28th USENIX Security Symposium*, 2019.
Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

Alexander Bulekov, Rasoul Jahanshahi, and Manuel Egele. Saphire: Sandboxing php applications with tailored system call allowlists. In *Proceedings of the 30th USENIX Security Symposium (USENIX Security 21)*, pages 2881–2898, 2021.

Brandon Butler. Which is cheaper: Containers or virtual machines? `https://www.networkworld.com/article/3126069/which-is-cheaper-containers-or-virtual-machines.html`, September 2016.

Claudio Canella, Mario Werner, Daniel Gruss, and Michael Schwarz. Automating seccomp filter generation for linux applications. In *Proceedings of the 2021 on Cloud Computing Security Workshop*, pages 139–151, 2021.

Theo Combe, Antony Martin, and Roberto Di Pietro. To Docker or not to Docker: A security perspective. *IEEE Cloud Computing*, 3(5):54–62, 2016.

Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated system call filtering for commodity software. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020.

Henry Hanping Feng, Jonathon T Giffin, Yong Huang, Somesh Jha, Wenke Lee, and Barton P Miller. Formalizing sensitivity in static analysis for intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 194–208, 2004.

William Findlay, David Barrera, and Anil Somayaji. Bpfcontain: Fixing the soft underbelly of container security. *arXiv preprint arXiv:2102.06972*, 2021.

Stephanie Forrest, Steven A Hofmeyr, Anil Somayaji, and Thomas A Longstaff. A sense of self for Unix processes. In *Proceedings of the IEEE Symposium on Security & Privacy (S&P)*, pages 120–128, 1996.

Tal Garfinkel, Ben Pfaff, and Mendel Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2004.

Masoud Ghaffarinia and Kevin W. Hamlen. Binary control-flow trimming. In *Proceedings of the 26th ACM Conference on Computer and Communications Security (CCS)*, 2019.

Seyedhamed Ghavamnia, Tapti Palit, Azzedine Benameur, and Michalis Polychronakis. Confine: Automated system call policy generation for container attack surface reduction. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, 2020a.

Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *Proceedings of the 29th USENIX Security Symposium*, 2020b.

Seyedhamed Ghavamnia, Tapti Palit, and Michalis Polychronakis. C2C: Fine-grained configuration-driven system call filtering. In *Proceedings of the 29th ACM SIGSAC Conference on Computer and Communications Security*, pages 1243–1257, 2022.

Andreas Gustafsson. Egypt. `https://www.gson.org/egypt/egypt.html`.

Ashish Gehani Hashim Sharif, Muhammad Abubakar and Fareed Zaffar. Trimmer: Application specialization for code debloating. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2018.

Haifeng He, Saumya K Debray, and Gregory R Andrews. The revenge of the overlay: automatic compaction of OS kernel code via on-demand code loading. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 75–83, 2007.

Kihong Heo, Woosuk Lee, Pardis Pashakhanloo, and Mayur Naik. Effective program debloating via reinforcement learning. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, 2018.

Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, pages 54–61, 2001.

Kapil Jain and R Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2000.

Yufei Jiang, Can Zhang, Dinghao Wu, and Peng Liu. Feature-based software customization: Preliminary analysis, formalization, and methods. In *Proceedings of the 17th IEEE International Symposium on High Assurance Systems Engineering (HASE)*, 2016.

Vasileios P. Kemerlis. *Protecting Commodity Operating Systems through Strong Kernel Isolation*. PhD thesis, Columbia University, 2015.

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kguard: Lightweight kernel protection against return-to-user attacks. In *Proceedings of the 21st USENIX Conference on Security Symposium*, Security'12, pages 39–39, Berkeley, CA, USA, 2012. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=2362793.2362832`.

Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: Rethinking kernel isolation. In *Proceedings of the 23rd USENIX Security Symposium*, pages 957–972, 2014.

Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, 2019.

Christopher Kruegel, Engin Kirda, Darren Mutz, William Robertson, and Giovanni Vigna. Automating mimicry attacks using static binary analysis. In *Proceedings of the USENIX Security Symposium*, 2005.

Anil Kurmus, Reinhard Tartler, Daniela Dorneanu, Bernhard Heinloth, Valentin Rothberg, Andreas Ruprecht, Wolfgang Schroder-Preikschat, Daniel Lohmann, and Rudiger Kapitza. Attack surface metrics and automated compile-time OS kernel tailoring. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2013.

Chi-Tai Lee, Jim-Min Lin, Zeng-Wei Hong, and Wei-Tsong Lee. An application-oriented Linux kernel customization for embedded systems. *J. Inf. Sci. Eng.*, 20(6):1093–1107, 2004.

Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. SPEAKER: Split-phase execution of application containers. In *Proceedings of the 12th Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, pages 230–251, 2017.

Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2017.

Xin Lin, Lingguang Lei, Yuewu Wang, Jiwu Jing, Kun Sun, and Quan Zhou. A measurement study on Linux container security: Attacks and countermeasures. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, pages 418–429, 2018.

F. Loukidis-Andreou, I. Giannakopoulos, Doka K., and N. Koziris. Docker-sec: A fully automated container security enhancement mechanism. In *Proceedings of the 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1561–1564, 2018.

Shachee Mishra and Michalis Polychronakis. Shredder: Breaking Exploits through API Specialization. In *Proceedings of the 34th Annual Computer Security Applications Conference (ACSAC)*, 2018.

Shachee Mishra and Michalis Polychronakis. Saffire: Context-sensitive function specialization against code reuse attacks. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2020.

Collin Mulliner and Matthias Neugschwandtner. Breaking payloads with runtime code stripping and image freezing, 2015. Black Hat USA.

Karen Scarfone Murugiah Souppaya, John Morello. Application Container Security Guide, 2017. `https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-190.pdf`.

Chetan Parampalli, R Sekar, and Rob Johnson. A practical mimicry attack against powerful system-call monitors. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security (ASIACCS)*, pages 156–167, 2008.

Marios Pomonis, Theofilos Petsios, Angelos D. Keromytis, Michalis Polychronakis, and Vasileios P. Kemerlis. kRˆX: Comprehensive Kernel Protection against Just-In-Time Code Reuse. In *Proc. of EuroSys*, pages 420–436, 2017.

Chris Porter, Girish Mururu, Prithayan Barua, and Santosh Pande. Blankit library debloating: Getting what you want instead of cutting what you don't. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–180, 2020.

Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. RAZOR: A framework for post-deployment software debloating. In *Proceedings of the 28th USENIX Security Symposium*, 2019.

Chenxiong Qian, Hyungjoon Koo, ChangSeok Oh, Taesoo Kim, and Wenke Lee. Slimium: Debloating the chromium browser with feature subsetting. In *In the proceedings of the 27th ACM SIGSAC Conference on Computer and Communications Security*, pages 461–476, 2020.

Anh Quach and Aravind Prakash. Bloat factors and binary specialization. In *Proceedings of the 3rd ACM Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 31–38, 2019.

Anh Quach, Aravind Prakash, and Lok Yan. Debloating software through piece-wise compilation and loading. In *Proceedings of the 27th USENIX Security Symposium*, pages 869–886, 2018.

Mohan Rajagopalan, Matti Hiltunen, Trevor Jim, and Richard Schlichting. Authenticated system calls. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, pages 358–367, 2005.

Vaibhav Rastogi, Drew Davidson, Lorenzo De Carli, Somesh Jha, and Patrick D. McDaniel. Cimplifier: automatically debloating containers. In *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, 2017a.

Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2nd Workshop on Forming an Ecosystem Around Software Transformation (FEAST)*, pages 51–56, 2017b.

Daniel Shapira. Escaping Docker container using waitid() – CVE-2017-5123, 2017. `https://www.twistlock.com/labs-blog/escaping-docker-container-using-waitid-cve-2017-5123/`.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on Docker Hub. In *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 269–280, 2017.

Linhai Song and Xinyu Xing. Fine-grained library customization. In *Proceedings of the 1st ECOOP International Workshop on Software Debloating and Delayering (SALAD)*, 2018.

Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. Combining concern input with program analysis for bloat detection. In *Proceedings of the ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

David Wagner and Drew Dean. Intrusion detection via static analysis. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 156–168, 2001.

Zhiyuan Wan, David Lo, Xin Xia, Liang Cai, and Shanping Li. Mining Sandboxes for Linux Containers. In *Proceedings of the 10th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 92–102, 2017.

Dinghao Wu Yufei Jiang and Peng Liu. Jred: Program customization and bloatware mitigation based on static analysis. In *Proceedings of the 40th Annual Computer Software and Applications Conference (ACSAC)*, 2016.

Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. PeX: A permission check analysis framework for linux kernel. In *Proceedings of the 28th USENIX Security Symposium*, pages 1205–1220, 2019.

Zhi Zhang, Yueqiang Cheng, Surya Nepal, Dongxi Liu, Qingni Shen, and Fethi Rabhi. KASR: A reliable and practical approach to attack surface reduction of commodity OS kernels. In *Proceedings of the International Conference on Research in Attacks, Intrusions, and Defenses (RAID)*, pages 691–710, 2018.

Xiangyu Zhang Zhongshu Gu, Brendan Saltaformaggio and Dongyan Xu. Face-change: Application-driven dynamic kernel view switching in a virtual machine. In *Proceedings of the 44th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2014.