Confine: Automated System Call Policy Generation for Container Attack Surface Reduction

Seyedhamed Ghavamnia
Stony Brook University

Tapti Palit
Stony Brook University

Azzedine Benameur
Accenture

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 realistic 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 a corresponding Seccomp system call policy that can be readily enforced while loading the container. The results of our experimental evaluation with 150 publicly available Docker images show that Confine can successfully reduce their attack surface by disabling 145 or more system calls (out of 326) for more than half of the containers, which neutralizes 51 previously disclosed kernel vulnerabilities.

1 Introduction

The convenience of running containers and managing them through orchestrators, such as Kubernetes [31], 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 [4].

The performance gains of containers, however, come to the expense of weaker isolation compared to VMs. Container isolation 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) [56].

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 containers. Despite the use of software isolation mechanisms provided by the OS, such as capabilities and namespaces, a malicious tenant can leverage kernel vulnerabilities to bypass them. For example, a vulnerability in `waitid` [8] allowed malicious users to run a privilege escalation attack [59] 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 had just 126 system calls, whereas version 4.15.0-76 supports 326 system calls. As shown in previous works [20,32,33,68], different applications use disparate kernel features, leaving the rest unused—and available to be exploited by attackers. Kurmus et al. [32] 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 that 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 [41,43,52] or even removing whole unneeded libraries [28]; tailoring kernel code based on application requirements [32,68]; or limiting system calls for containers [12,53,54,64]. In fact, one of the suggestions in the

NIST container security guidelines [44] 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. In contrast, some works rely on dynamic analysis and training [12,32,53,54,64,68] to exercise the system using realistic workloads, and identify the actual code that was executed while discarding the rest. For a given workload, this approach maximizes the code that can be removed, but as we show in Section 4, 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 [12, 53, 54, 64], in this work we aim to provide a more generic and practical solution that can be readily applied on any container without requiring any training. To that end, we present an automated technique for extracting restrictive system call policies for arbitrary containers, and limiting the exposed interface of the underlying kernel that can be potentially 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 fully automated system, named *Confine*, takes a container image as its input and generates a customized system call policy. Containers, once initialized, run a single application for their entire execution time. We use dynamic analysis to capture all binary executables that might be invoked during container initialization. This initial limited dynamic analysis phase does not depend on the availability of any workloads, and just pinpoints the set of executables that are invoked in the container, which are then statically analyzed. We have chosen Docker as the main supported type of container images, as it is the most widely used open-source containerization technology.

We experimentally evaluated our prototype with a set of 150 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 145 or more system calls, while at least 100 or more system calls are disabled in the worst case and 219 in the best case. This is in stark contrast to Docker's default list of 49 (plus four partially) disabled system calls. More importantly, disabling these system calls effectively *neutralizes 51 previously disclosed kernel vulnerabilities*, in addition to the 25 vulnerabilities mitigated by Docker's default Seccomp policy.

The main contributions of our work include:

- We propose a generic approach for the automated generation of restrictive, ready-to-use Seccomp system call policies for arbitrary containers, without depending on the availability of source code for the majority of the target programs.
- We performed a thorough analysis of Linux kernel CVEs, mapping them to functions in the kernel code. We identified which system calls can be used to exploit each CVE, and used this mapping as the basis for evaluating the effectiveness of our approach.
- We examined more than 200 of the most popular publicly available Docker images from DockerHub and present an analysis of their characteristics.
- We experimentally evaluated our system with the above images and demonstrate its effectiveness in generating restrictive system call policies, which neutralize a set of 51 previously disclosed kernel vulnerabilities.

Our Confine prototype is publicly available as an open-source project from `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 SECure COMPuting with filters (Seccomp BPF) [13] can be used to reduce the kernel code exposed to containers.

2.1 Linux Containers

Linux containers are an *OS-level virtualization approach*, which can be used to execute multiple userlands on top of the same kernel. The Linux kernel uses Capabilities [37], Namespaces [39] and Control Groups (cgroups) [38] 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 [30] 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.

DockerHub [10] 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 throuth the provided *system call* set, i.e., a pre-defined API that allows access to specific kernel functionalities programmatically. More importantly, however, applications 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.

Nevertheless, although a program may not use all of the provided system calls, the complete set is available to all processes. This modus operandi has two issues: (1) a compromised application may use additional system calls (from what the author of the application originally intended) to carry out malicious operations that require access to system resources (e.g., filesystem, network) that the application never meant to access; and (2) a malicious (or compromised) application may invoke unused system calls to exploit underlying kernel vulnerabilities (typically related to the implementation of a given system call) for privilege escalation [26, 27], thereby gaining access to every process and container on the host.

Seccomp BPF [13] is a mechanism for restricting the set of system calls that are accessible by a given application. Specifically, Seccomp BPF uses the Berkeley Packet Filter language [40] for allowing developers to write arbitrary programs that act as system call filters, i.e., BPF programs that inspect the system call number (as well as argument values, if needed) and allow, log, or deny the execution of the respective system call. Docker containers can be executed with Seccomp BPF profiles, allowing users to provide white/black lists of permitted/prohibited system calls. The specified white/black list is applied to the entire process namespace, limiting all processes executed inside the respective container. We use this mechanism to reduce the kernel code available to each container.

3 Threat Model

We consider a local adversary who has full access to a container running on a third-party host. This access may be granted either legitimately (e.g., as a regular user of a cloud service), or as a result of compromising a vulnerable process running on the container. Potential victims include the OS kernel of the host, as well as any other containers running on it. We specifically focus on preventing the attacker from escaping a container—exploiting an application running on a container is not the focus of our work. Any exploit mitigations and defenses deployed on the host or individual containers are orthogonal to our approach, as it does not rely on any additional protection mechanisms being in place at user or kernel space.

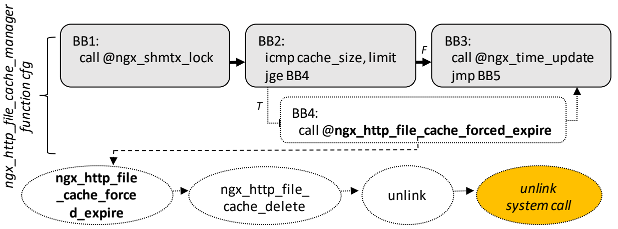

ngx_http_file_cache_manager function cfg
BB1:
call @ngx_shmtx_lock
BB2:
icmp cache_size, limit
jge BB4
F
BB3:
call @ngx_time_update
jmp BB5
T
BB4:
call @ngx_http_file_cache_forced_expire
ngx_http_file_cache_forced_expire
ngx_http_file_cache_delete
unlink
unlink system call

Figure 1: Example of 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 training.

Confine limits the set of system calls an attacker can invoke. This means that exploit code (e.g., shellcode or ROP payload) or malicious programs run by the attacker will have more limited capabilities, as they cannot rely on system calls that are not needed by the container. More importantly, by preventing access to less-frequently used and less-tested system calls—the kernel code of which may contain vulnerabilities that can lead to privilege escalation [35]—an attacker cannot trigger those vulnerabilities to compromise the kernel, as the respective system calls cannot be invoked in the first place.

4 The Need for Static Analysis

Previous works [64] 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. 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 used anywhere else in the code, 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, and not simply request the same set of files repeatedly. 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 is the use of `lstat` when displaying directory listings. Apart from this use, `lstat` is not used in any other part of Nginx. As listing a directory is usually triggered by users manually entering URLs, 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 Nginx, making it challenging for dynamic analysis to discover them.

The above examples are indicative of the trade off between fragility vs. 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.

5 Design

Our goal is to reduce the kernel attack surface available to a malicious tenant of a container service by limiting the number of system calls available to each container, which can potentially be of use for malicious purposes (either as part of exploit code, or 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 access to only those system calls that are actually needed for the proper operation of the container.

Identifying the system calls that are necessary for the correct execution of the container 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; and 4) extract direct system call invocations from applications and libraries.

Figure 2 presents a high-level overview of our approach, which, given a container image, automatically generates Seccomp rules that limit the system calls that may be invoked. Confine currently supports Docker containers running on a native Linux-based host, but similar analysis could be performed for other container environments and operating systems.

5.1 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 [11] invokes the following supporting applications to set up the environment: `bash`, `chown`, `find`, `id`, and `numactl`. To generate system call policies, we must thus identify all programs that can potentially run during the lifetime of a container. 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. We further discuss the completeness of the derived list in Section 8.

Our approach is different from previous works that rely on dynamic training using various workloads to derive a list of allowable system calls [64]. 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.

For containers that may include applications that are not launched from the beginning, our system supports external lists of executables that can be included in the analysis.

5.2 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.

Libc 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 just analyzes 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 call graph, 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 [3, 22]. 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 6.1 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

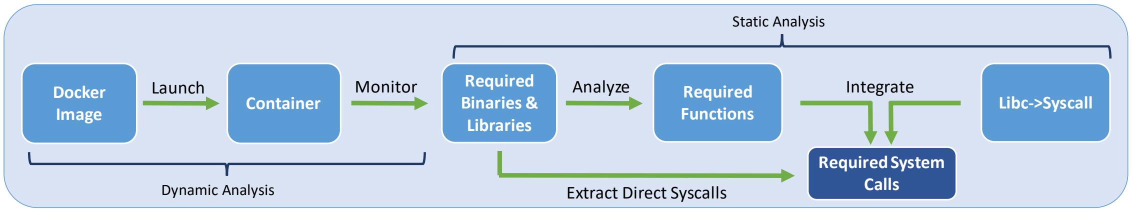


Figure 2: Overview of Confine's system call extraction process. A one-time dynamic analysis phase that does not require any application-specific workloads is used for the sole purpose of identifying the applications running in the container. Each application is then statically analyzed to identify all the library functions that it uses, and the system calls it relies on.

(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 6.2. Some applications developed in languages other than C/C++ also have special considerations which we discuss in Section 6.3.

5.3 Hardening the Container Image

Once we have generated the list of system calls needed to run the container, we can proceed to harden the container image. 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 blacklist or a whitelist of system calls prohibited or permitted. For Confine, we use a blacklist of system calls that the container is not allowed to invoke.

On the basis of the analysis performed in Sections 5.1 and 5.2, we use an automated script to derive the list of blacklisted system calls, and construct the corresponding Seccomp profile. If any new application needs to be executed on the container after this process, the administrator must run the analysis on the application to update the Seccomp profile.

6 Implementation

6.1 Mapping Libc Functions to System Calls

To ensure correctness, a precise function call graph is required to identify and filter unused system calls. Based on our analysis of more than 200 popular Docker images from DockerHub [10], we found that even though most containers use the popular glibc library as their main user-space libc library, musl-libc [45] was also used in 12 occasions. Although both musl-libc and glibc provide implementations of the C standard library functions, and applications should be able to use both interchangeably, we discovered that the system calls used by standard libc functions some times differ between musl-libc and glibc. To maximize compatibility, we analyzed both libraries independently to extract their call graphs and their corresponding function-to-system-call mapping. Moreover, due to certain differences between glibc and musl-libc, which we discuss next, we had to use a different toolchain for the analysis of each of these libraries.

Musl-Libc Musl-libc [45] is a lightweight C standard library which has a smaller codebase compared to glibc. For our analysis, we compiled musl-libc with the LLVM compiler toolchain and implemented an LLVM pass to extract the complete call graph. This pass operates on the intermediate representation (IR) of the code and records each function call. To identify system calls, in addition to recording each function call, we make special note of calls to the `syscall` function. Using the extracted call graph, we create a map between each exported function in musl-libc and the system calls it invokes. We modified the compiler toolchain to invoke the pass before any optimization to prevent the loss of precision due to optimizations and code transformations.

Musl-libc uses a `weak_alias` macro to define weak symbols for functions. Weak symbols can be overridden by strong symbols having the same name, without name collision errors. Our LLVM pass keeps track of these aliases as well.

Glibc Glibc is the most popular libc implementation used in most containers. Glibc heavily relies on multiple GCC features which are not implemented in LLVM. Due to this issue, we implemented a second analysis pass to extract the call graph and system call information from glibc, based on the GCC RTL (Register Translation Language) Intermediate Representation. Our call graph extraction implementation is based on the Egypt [18] tool, which operates on GCC's RTL IR. We discovered that there are three main mechanisms through which glibc invokes system calls.

System Call via Inline Assembly and Assembly Files: This is the most straightforward mechanism for invoking system calls. Functions such as `accept4()` contain inline invocations using the x86-64 `syscall` instruction. Given the source code, the Egypt tool constructs the function call graph 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 also contain `syscall` instruction. Therefore, we analyze the assembly files and extract all `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 compile time. The `syscall-template.S` file contains the macros `T_PSEUDO`, `T_PSEUDO_NOERRNO`, and `T_PSEUDO_ERRVAL`, that 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 compile time, generates the correct macro definitions, and invokes the expansion of the macros in the `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: Similarly to musl-libc, 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 call graph.

6.2 Binary Analysis

To capture a trace of all invoked executables, we leverage Sysdig [23] to monitor the `execve` calls made during the initial 30 seconds (configurable value) of the container. After we generate the list of programs the container runs, we further perform static analysis to extract the list of system calls necessary for the correct execution of the container.

System Call Invocation Through Libc After extracting the list of binaries, we recursively find any other libraries (except libc) that are loaded by them, and then use `objdump` to extract the superset of imported functions across all main executables and libraries. This analysis gives us the list of libc (glibc or musl-libc) functions that are imported by an application and its libraries. Then, using the libc-to-syscall map generated as described in Section 6.1, we derive the list of system calls required by the application. In addition to these, the Docker framework itself needs certain system calls to run. Consequently, after deriving the required system calls for all the programs of a container, we combine them with the list of system calls which Docker requires by default to launch the container.

Direct System Call Invocation We further encountered a limited number of libraries and applications that invoke system calls directly through either the libc `syscall()` interface, or the native `syscall` assembly instruction. Analyzing such invocations requires deriving the values of the arguments being passed to the system call. Fortunately, extracting the first argument, which specifies the system call number, is straightforward, as it is typically set by the (few) instructions preceding the `syscall` instruction or the `syscall()` function invocation. We therefore use binary code disassembly to identify the system call number by extracting the values assigned to the `RAX/EAX` register for the `syscall` instruction, and the `RDI/EDI` register for the `syscall()` function. Confine currently supports only x86-64, but adding support for other platforms is straightforward by following their calling conventions.

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 run time, 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 in [51] have shown that only around 3% of the 3174 programs and 2% of the 4292 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 run time through the `/proc` virtual file system, which provides this information for every process. In Section 6.3 we discuss how we use the same technique of monitoring the `procfs` to detect the list of libraries used by Java applications.

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.

6.3 Language-specific Considerations

Different languages have different software stacks, and therefore different analysis techniques to extract the system call policies. The programming language of the containerized application has an important effect on the analysis methods used to identify the system calls required by a given application. In this section, we describe the different techniques we used to handle applications written in programming languages other than C/C++, which we encountered during our study. In Section 7 we present statistics on the usage of these languages across the container images studied.

Go Applications written in Go consist of *command* packages and utility non-main packages. Go applications can be compiled into executables using two build-modes: default, and c-shared. When compiled with the *default* build mode, all main packages are built into executables, and all non-main packages are built into a static `.a` archive that is linked statically with the executables. Go applications use system call wrappers provided by Go's `syscall` and `runtime` packages to invoke the system calls. When compiled with the *c-shared* build-mode, the main package relies on the standard libc library to invoke system call wrapper functions.

The analysis of our dataset shows that most of the Go applications in the studied containers are built using the default build mode. Therefore, unlike C/C++ applications which rely on glibc, these applications use the Go core packages, `syscall` and `runtime`, to make system calls. Consequently, for containers that include Go applications, we require the source code of all running Go applications to identify their system calls. We use the `callgraph` tool [17] to build the call graph of Go applications and all their dependencies, which we have extended to record all calls to the system call wrappers specified in the `syscall` and `runtime` packages.

Java/NodeJS Both the Java and NodeJS runtime applications use libc as a shared library to invoke system calls. The Java compiler compiles Java source code into Java bytecode and uses its own virtual machine (JVM) to interpret the bytecode. Java programs are not compiled to machine code and a binary is thus not generated. The interpreter and JVM is provided by the `java` binary, which is launched via an `execve` system call. To find the system calls of a container that hosts a Java type application, in addition to analyzing all other running binaries, we also analyze the *java* binary that contains this JVM, and any other libraries that are dynamically loaded, as described above. Similarly, we handle the system calls invoked by the `node-js` runtime.

Purely Interpreted Languages Scripting languages, such as Python and Perl, are purely interpreted and require the respective interpreter to run. We extract the required system calls for these types of containers using the same approach used for other binaries, by applying our method to the interpreter binary and all its required libraries.

6.4 Seccomp Profile Generation

We automatically generate Seccomp policies by classifying all system calls not present on the final list of required system calls as "not-permitted," and assigning them to a blacklist. The Docker Seccomp ruleset requires the name of the filtered system calls, while our analysis of the containers generates system call numbers. We map 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, we map the system call name to its number, and use it to convert the blacklisted system call numbers to their names. We create the Seccomp profile with a blacklist containing these system calls and apply it to the container.

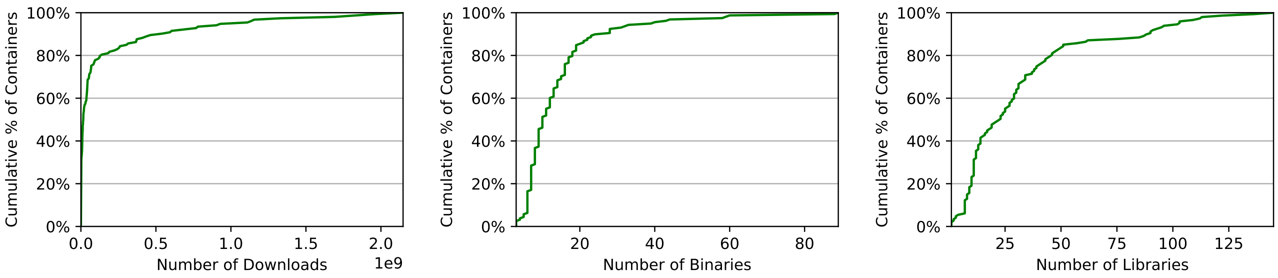
Docker uses a JSON file to define the permitted system calls. Listing 1 shows a sample ruleset which only blacklists the *pwrite64* system call. The default action for this ruleset is to allow all system calls, except those specified under the `syscalls` tag. Each system call is specified by three arguments: its name, the action, and its arguments.

Listing 1: Example of a Docker Seccomp ruleset file.

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

7 Experimental Evaluation

We evaluated Confine with a set of 150 publicly available Docker container images available from DockerHub [10]. We conducted experiments to get insight into the characteristics of popular container images, assess the effectiveness of our system call policy extraction approach and evaluate its security benefits in terms of attack surface reduction.



(a) Container popularity.

(b) Executed binaries per container.

(c) Loaded libraries per container.

Figure 3: Cumulative distribution of number of downloads, executed binaries, and loaded libraries per container.

7.1 Analysis of Publicly Available Containers

7.1.1 Dataset Collection

DockerHub [10] provides a set of 153 “official” container images that are maintained by the service, along with many other community-maintained images. For our experimental evaluation, we collected a data set comprising i) the 153 official DockerHub images, and ii) the top-200 most popular community-maintained images. Due to the overlap between the two lists, our initial data set consists of 209 unique Docker images.

Out of these 209 images, 193 are available for free and can be used without any paid licensing requirements—we exclude the rest from our data set. Out of the remaining 193 images, 43 require some form of manual effort to setup, such as launching prerequisite containers (e.g., *rocket-chat*), or specifying complex configuration settings. We leave these images out of the scope of our work due to the complexity of the manual steps required to run them. Our final data set thus comprises 150 Docker images (122 official and 28 unofficial) that can be automatically processed by our system.

7.1.2 Container Statistics

DockerHub assigns a popularity metric to each of the official images based on its number of pull requests (downloads). We retrieve this number through the official DockerHub API for the 153 official images (the returned value is zero for most of the non-official images). Figure 3a shows the popularity distribution in terms of number of downloads for the 153 official DockerHub images. We can see that 15% of the Docker images account for most of the downloads, while the rest are much less popular. This implies that including more images to the evaluation set would not increase the significance of the dataset in terms of container popularity.

In Section 6.3 we discussed how we tackle applications written in languages other than C/C++. To gain some insight about how commonly these languages are used, we gathered some statistics about the programming languages used by the containers in our dataset. The most common programming languages used in containerized applications include C/C++, Java and Go. As shown in Table 1, many well-known server applications (e.g., Nginx, Apache Httpd, and MySQL) fall into the C/C++ category, which comprises 61% of the containers. There are, however, 22% images hosting Java-based applications, such as Apache Cassandra (NoSQL database), Apache Solr (open-source search engine platform), and Apache Tomcat. Since Go has been used to develop many of the management tools used in the Docker ecosystem, Go-based containers also account for a considerable number of images (14%).

We classify containers as *Java-based* or *Python/Perl-based* depending on the presence of the corresponding language runtime in the container. Since most containers contain small utility tools, such as `sed`, `grep`, and `find`, which would cause a container to fall into the C/C++ category, Confine only classifies a container in the C/C++ category if it does not run any other program written in some other programming language. For example, the Cassandra [9] container invokes both `sed` (a C program) and the `java` application. Instead of classifying this container in both the C/C++ and Java categories, we only consider it as a Java container for the results of Table 1.

As discussed in Section 6.2, we extract the list of executed binaries in each container and retrieve their loaded libraries. Figures 3b and 3c show the distribution of the number of binaries executed and libraries loaded by the tested containers. About 75% of the containers execute fewer than 16 binaries, and 75% require fewer than 47 libraries.

7.2 System Call Filtering

We used the set of 150 Docker images to evaluate the effectiveness of our approach in filtering unused 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 6.2. 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 disabling a system call is equivalent to preventing

Table 1: Breakdown of Docker images according to the programming language used for the main hosted application.

Language	# Img.	Example Applications
C/C++	92	Nginx, Apache Httpd, MongoDB, MySQL
Java	34	Cassandra, Solr, Tomcat, Sonarqube, JRuby
Go	21	Traefik, Registry, Telegraph, Metricbeat
JavaScript	6	Kibana, Ghost, Hipache
Python	5	Celery, Plone, Hylang, Hipache, ROS
Perl	2	Nuxeo, GitLab Community Edition

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 7.3. We leave the actual *removal* of the kernel code related to each 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 filtered system calls for the 150 successfully analyzed containers. For about half of the containers, Confine disables 145 or more system calls (out of the 326 currently available in Linux). Even in the worst case (leftmost part of x axis), 100 or more system calls are removed. This means that *at least twice or more* system calls can be disabled compared to Docker's default Seccomp filter, which includes 49 system calls.

Confine filtered 148 system calls on average for the top-15 ranked Docker images that were successfully analyzed. The list includes many popular applications, such as Nginx, PostgreSQL, MySQL, and MongoDB. For Nginx, we observe that 160 system calls are required for its proper operation (166 are filtered out of 326), whereas Wan et. al [64] identified only 76 through dynamic analysis. The complete list is shown in Table 2.

7.2.1 Validation Methodology

General Validation First, we check if the container is launched properly with the specified Seccomp profile. Unless we filter system calls that are required by the Docker framework itself, this step will succeed.

The Docker image is specified using a configuration file, called *Dockerfile*. The *Entrypoint* attribute in 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.

Table 2: Number of filtered system calls for the top-15 images with the highest number of downloads.

Image Name	Filtered Syscalls	Popularity Rank
oracle-database-enterprise-edition	138	1
oracle-serverjre-8	190	2
mysql-enterprise-server	128	3
couchbase	140	5
db2-developer-c-edition	100	6
oracle-instant-client	190	7
redis	171	8
ibm-security-access-manager	110	9
mongo	143	10
ubuntu	184	11
busybox	142	12
node	150	13
postgres	133	14
nginx	166	15
mysql	135	16

In-Depth Validation We further validated the soundness of the profiles generated for some of the *ready-to-use* Docker images, by testing them with available benchmark suites for the corresponding applications.

For this validation effort, we focused on the most popular Docker images due to the manual effort required. We collected a set of available benchmarking tools, from various sources, which are applicable to 10 of the top-50 Docker images. These include domain-specific benchmarking tools, Selenium [57] scripts, and the CloudSuite benchmarks [46]. We applied the benchmarks to the following Docker images: MongoDB, PostgreSQL, MySQL, Redis, Wordpress, PHP, Memcached, Nginx, Apache Httpd and MediaWiki.

The web-serving benchmark of CloudSuite [46] is based on the open-source social networking site Elgg [15]. Elgg is a PHP-based application that uses MySQL as the database server. It also provides a media-streaming server running on the Nginx server. Finally, it also provides a benchmark for Memcached. We derived system call policies for containers running these benchmarks and verified that the benchmarks successfully ran.

For MongoDB, we used `mongo-perf` [42], and applied all the test cases in the *simple_insert* and *simple_query* test suites on one thread for 10 seconds. For PostgreSQL, we used `pgbench` [48], which first creates a new database and then prepares and runs the test cases. For MySQL, we used the `sysbench` tool [29]. After initializing a new database, we ran the OLTP read and write tests using 16 threads on 10 tables.

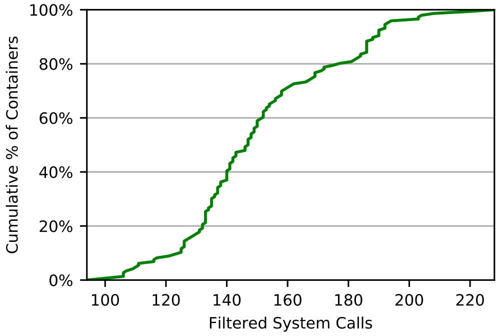


Figure 4: Cumulative distribution of the number of filtered system calls, as a percentage of all tested containers.

For Redis, we used `redis-benchmark` [55], and ran multiple tests changing the type and number of requests, number of clients and size of data being loaded in each test.

Wordpress and Mediawiki run on Apache Httpd. We ran a hardend MySQL container as their database and used Selenium [57] to create content through automated user interaction. We applied scripts created based on online tutorials prepared and released by Azad et al. [2]. Through these scripts, different operations, such as creating users, posts, applying images and modifying them were performed. All operations were performed successfully and no irregular impact was identified through the logs or the script outputs.

Validation Results Based on the above process, we verified that 146 out of the 150 containers run successfully, while 4 failed for the following reasons. First, Confine failed to extract system calls for three Go-based images, because the Go call graph tool (discussed in Section 6.3) could not be applied on the source code of Influxdb [24] and Chronograf [5] due to numerous dependencies, while the source code of Sematext [58] was not available.

Second, Java provides the option of accessing OS interfaces directly through its Java Native Interface. We would need to analyze the Java code of each program to extract JNI-invoked system calls. Elasticsearch [14] is the only example we saw in our Java dataset which used this feature, and thus we did not invest the time to implement this capability.

7.3 Security Evaluation

Previous works in the area of software debloating [21, 28, 52] mostly focus on the amount of code (and number of ROP gadgets) removed as the main measure of improvement. In contrast, our approach does not remove any code, but merely restricts the system calls that a malicious container can invoke, i.e., it reduces the attack surface of the host's kernel by reducing the number of system calls it exposes to (potentially malicious) applications.

To demonstrate the effectiveness of our approach in reducing the attack surface, we evaluate how removing unused system calls can reduce the risk of privilege escalation attacks. To that end, our starting point was the vulnerabilities used in a previous study by Lin et al. [36]. These vulnerabilities are exploitable despite the use of container isolation mechanisms, such as namespaces,cgroups, and capabilities. To gain a better understanding of the impact that filtering individual system calls has in neutralizing potential kernel vulnerabilities, we mapped each CVE to its corresponding system calls.

7.3.1 Mapping Kernel CVEs to System Calls

To perform our analysis, we crawled the CVE website [7] 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 call graph and analyzed which parts of it can be exclusively accessed by a given system call.

We constructed the Linux kernel's call graph using KIRIN [66]. This allows us to map which functions in the kernel are invoked from which system call, and 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 call graph 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.

7.3.2 CVEs Mitigated by Confine

Our results are summarized in Table 3. Linux kernel CVEs are assigned a category depending on how their exploitation can affect the underlying system. While privilege escalation has the most severe outcome, others are important to consider as well. Using a denial-of-service attack, the attacker can disrupt the functionality of *all* containers and applications running on the same host. The "Bypass a Restriction" category includes attacks which allow the attacker to directly or indirectly bypass isolation mechanisms. Similarly, exploiting a vulnerability in the "Obtain Information" category could cause the leakage of sensitive kernel data which endangers the isolation guarantees provided to other containers.

Based on our analysis, in addition to the 25 CVEs mitigated by Docker's default Seccomp policy, 51 CVEs across all studied containers are effectively removed (i.e., the respective vul-

Table 3: CVEs mitigated by removing unneeded system calls.

System Call(s)	# CVEs	CVE Examples	CVE Type	# Imgs.	Docker Image Examples
set_thread_area	1	CVE-2014-8133	B	146	Nginx, MongoDB, Apach Httpd, MySQL
mq_notify	1	CVE-2017-11176	D	146	Redis, CouchDB, Apache Httpd, MySQL
sched_getattr	1	CVE-2014-9903	I	146	Postgresql, Nginx, Memcache
io_submit	1	CVE-2010-3066	D	142	Rethinkdb, Apache Httpd, Nginx, Redis
rt_(tg)sigqueueinfo	1	CVE-2011-1182	Other	140	MongoDB, Nginx, Apache Httpd, MySQL
clock_nanosleep	1	CVE-2018-13053	O	140	MongoDB, Nginx, Apache Httpd, MySQL
ioprio_get	1	CVE-2016-7911	P,D	131	Redis, Nginx, Apache Httpd, MySQL
waitid	2	CVE-2017-14954, CVE-2017-5123	B,P,I	114	Nginx, MongoDB, CouchDB, MySQL
inotify_init1	1	CVE-2010-4250	D	101	Nginx, Apache Httpd, MySQL
semctl	1	CVE-2010-4083	I	97	Nginx, CouchDB, Redis
inotify_add_watch	1	CVE-2019-9857	D	77	Nginx, Apache Httpd, MySQL
shmctl	2	CVE-2009-0859, CVE-2010-4072	I,D	71	Iojs, Hylang, Rethinkdb
semget, msgget, shmget	1	CVE-2015-7613	P	62	Julia, Iojs, Clearlinux
splice	1	CVE-2009-1961	D	57	MongoDB, Rethinkdb, Oraclelinux
epoll_ctl	1	CVE-2012-3375	D	48	Crux, IBM-db2-warehouse-cc, Adminer
setsockopt	5	CVE-2016-4997, CVE-2016-8655	P,M,O,D	22	Euleros, Clearlinux
([f,l]remove,[f,l]set)xattr	1	CVE-2011-1090	D	22	Clearlinux, Fluentd
ioctl	26	CVE-2010-2478, CVE-2009-0745	I,P,B,O,D	1	Nats
madvise	2	CVE-2012-3511, CVE-2017-18208	D	1	Busybox

I: Obtain Information, P: Gain Privileges, B: Bypass a Restriction, O: Overflow, D: Denial of Service, M: Memory Corruption

nerabilities cannot be triggered by the attacker) by applying our generated policies. These include CVEs that an attacker could exploit to perform denial-of-service attacks against the kernel (CVE-2012-3375, CVE-2016-7911, and CVE-2017-11176), perform privilege escalation attacks (CVE-2017-5123, CVE-2016-7911, and CVE-2015-7613), or leak sensitive kernel information (CVE-2017-14954 and CVE-2014-9903). Of these 51 CVEs, seven were removed in more than 130 containers.

8 Discussion and Limitations

As shown in Table 3, the system calls filtered by our technique are not very commonly used, but at the same time can lead to mitigating a large number of kernel vulnerabilities. We must emphasize that although system calls such as `execve` and `mmap` are used as part of user-space exploits, *any* system call associated with a kernel CVE can be used to exploit the kernel. For an attacker seeking to escape a container, exploiting commonly used system calls such as `execve` or `mmap` provides no additional benefit over exploiting system calls such as `waitid`, which are used less frequently.

In addition to launching applications from scripts and the command line, most programming languages give the programmer the ability to launch applications using special library calls, such as `execve`. As it is not guaranteed that such invocations will occur within our monitoring window, our approach may fail to analyze any executables launched in this way. Currently, the developer is expected to provide a list of binaries executed using such library calls. Our approach provides an initial list of applications for the user to build upon, which can further reduce the manual effort required.

A better alternative would be to statically analyze the source code of all invoked applications to identify process creation events. This can easily be done for applications written in interpreted languages, as they typically supported by many static analysis tools (e.g., php-ast [47] for PHP, or the built-in AST [49] functionality for Python). We executed php-ast on the Wordpress Docker image and validated the correctness of extracting paths of binaries which could be passed to any *exec*-like function (e.g., php_exec, shell_exec). This could easily be extended to applications written in different languages. We leave the full implementation of such a capability as part of our future work, since it only requires engineering effort.

Although not recommended, some Docker images use `cron` jobs to run periodic tasks in the container. In these cases we expect the user to provide the list of programs which can be executed through `cron`, although again such cases could be automatically handled by parsing the `crontab` file.

9 Related Work

Static source code analysis for deriving system call policies has been a widely used approach in the field of host-based intrusion detection [63]. Our work mainly falls in the area of software debloating attack surface reduction, and we thus discuss related works in this context.

Container Security and Debloating. Wan et al. [64] use dynamic analysis to profile the running applications on a container and generate the Seccomp ruleset for containers. DockerSlim [12] is also an open source tool which relies on dynamic analysis to generate Seccomp profiles and remove unnecessary files from a docker image. However, as shown in Section 7.2.1, the required system calls can't be reliably extracted through dynamic analysis alone – especially for cases that handle exceptions and errors, which are typically not part of the usual execution. Therefore, dynamic analysis can't guarantee completely covering all the system call requirements of each application. Our system, on the other hand, provides a more comprehensive static analysis mechanism for extracting the system calls used by a container.

Speaker [34] separates the required system calls into two main phases, booting and runtime. It dynamically extracts the required system calls for each phase and filters them based on the necessity of each state. Cimplifier [53] splits containers running multiple applications into multiple single-purpose containers using dynamic analysis. Rastogi et al. [54] suggest improvements to Cimplifier [53] through symbolic execution.

Previous works have also focused on container security with regards to its software protection mechanisms and vulnerabilities. Lin et al. [36] provide a dataset of security vulnerabilities and exploits which can potentially bypass the software isolation provided by the Linux Kernel. One of their recommendations is the use of stricter Seccomp policies for containers. Shu et al. [60] have created a framework to perform vulnerability scanning on images found on the Docker hub. Combe et al. [6] explored the security implications of using containers, by considering adversary models which assume complete access of an adversary to one container on a host.

Application Debloating. Most of the prior works in the area of debloating have focused on removing unnecessary code from individual processes. Mulliner and Neugschwandtner [43] proposed one of the first approaches for library specialization, which identifies and removes all non-imported library functions at load time. Quach et al. [52] developed a modified loader and compiler to perform shared library specialization by removing unnecessary functions extracted through call dependency and function boundary identification at compile time. Agadakos et al. [1] perform similar library specialization, but at the binary level. Song et al. [61] used data dependency analysis to show the potential of fine-grained library customization of statically linked libraries. Shredder [41] limits arguments passed to critical system API functions to legitimate values extracted from each application's source code. Qian et al. [50] use training and heuristics to identify basic blocks which can be removed from a binary, while Ghaffarinia and Hamlen [16] restrict the control flow of the binary, instead of removing the extra code using a similar training approach.

Other works explore the potential of debloating software based on predefined features. CHISEL [21] is a framework for shrinking software using a reinforcement learning approach based upon test cases provided by the user. It is a test case driven approach, which reduces the code size, such that none of the test cases fail. TRIMMER [19] uses inter-procedural analysis to find unnecessary parts of code based upon user-defined configuration data.

Other works in this area have also focused on different programming languages [25, 62, 65]. Jred [65] performs static analysis on Java code to remove unused methods and classes. Jiang et al. [25] propose feature-based debloating for Java programs using data flow analysis.

Kernel Debloating. Several works focus on debloating the kernel and customizing it according to user requirements. KASR [67] and FACE-CHANGE [68] use dynamic analysis to identify unused parts of the kernel and use virtualization mechanisms to limit each application to its profile. Kurmus et al. [32] propose a method to tailor the Linux kernel for special workloads through automatic generation of kernel configuration files.

10 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. We believe that the results of our experimental evaluation demonstrate the practicality of the proposed system, as it managed to disable (without breaking any functionality) 145 or more system calls for more than half of the analyzed containers, neutralizing this way 51 previously disclosed kernel vulnerabilities. As part of our future work, we plan on exploring the generation of more fine-grained system call policies.

References

[1] 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*, pages 70–83. ACM, 2019.

[2] Babak Amin Azad, Pierre Laperdrix, and Nick Nikiforakis. Less is more: Quantifying the security benefits of debloating web applications. In *28th USENIX Security Symposium (USENIX Security 19)*, Santa Clara, CA, 2019. USENIX Association.

[3] Lars Ole Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, University of Cophenhagen, 1994.

[4] 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.

[5] Chronograf. Chronograf - Docker Hub. `https://hub.docker.com/_/chronograf`, 2019.

[6] 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.

[7] CVE-Details. Common vulnerabilities and exposures database. `https://www.cvedetails.com`, 2019.

[8] CVE Details. CVE Details (CVE-2017-5123), 2017. `https://www.cvedetails.com/cve/CVE-2017-5123/`.

[9] Docker. cassandra - Docker Hub, May 2019. https://hub.docker.com/_/cassandra/.

[10] Docker. Docker Hub Images. https://hub.docker.com, 2019.

[11] Docker. mongoDB - Docker Hub, May 2019. https://hub.docker.com/_/mongo/.

[12] docker slim. Docker slim - minify and secure docker containers (free and open source!). https://dockersl.im.

[13] The Linux Kernel Docs. Seccomp BPF (SECure COMPuting with filters), 2019. https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[14] Elasticsearch. Elasticsearch - Docker Hub. https://hub.docker.com/_/elasticsearch, 2019.

[15] Elgg. Elgg. https://elgg.org/, 2019.

[16] Masoud Ghaffarinia and Kevin W Hamlen. Binary control-flow trimming. In *Proceedings of the cnum26th ACM Conference on Computer and Communications Security nymCCS. forthcoming*, 2019.

[17] Golang. Package Callgraph - GoDoc, May 2019. https://godoc.org/golang.org/x/tools/go/callgraph.

[18] Andreas Gustafsson. Egypt Tool, 2011. https://www.gson.org/egypt/egypt.html.

[19] 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.

[20] 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. ACM, 2007.

[21] 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.

[22] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '01, pages 54–61, New York, NY, USA, 2001. ACM.

[23] Sysdig Inc. Sysdig Tool, May 2019. https://github.com/draios/sysdig.

[24] Influxdb. Influxdb - Docker Hub. https://hub.docker.com/_/influxdb, 2019.

[25] 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.

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

[27] 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.

[28] Hyungjoon Koo, Seyedhamed Ghavamnia, and Michalis Polychronakis. Configuration-driven software debloating. In *Proceedings of the 12th European Workshop on Systems Security*, page 9. ACM, 2019.

[29] Alexey Kopytov. Sysbench. https://github.com/akopytov/sysbench, 2019.

[30] Petros Koutoupis. Everything You Need to Know about Linux Containers, Part II: Working with Linux Containers (LXC), 2018. https://www.linuxjournal.com/content/everything-you-need-know-about-linux-containers-part-ii-working-linux-containers-lxc.

[31] Kubernetes. Kubernetes - Production-Grade Container Orchestration, 2019. https://kubernetes.io.

[32] 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.

[33] 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.

[34] Lingguang Lei, Jianhua Sun, Kun Sun, Chris Shenefiel, Rui Ma, Yuewu Wang, and Qi Li. Speaker: Split-phase execution of application containers. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 230–251, 2017.

[35] 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.

[36] 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*, pages 418–429. ACM, 2018.

[37] Linux Manpages. Capabilities(7) - Linux Programmer's Manual, 2019. `http://man7.org/linux/man-pages/man7/capabilities.7.html`.

[38] Linux Manpages. Cgroups(7) - Linux Programmer's Manual, 2019. `http://man7.org/linux/man-pages/man7/cgroups.7.html`.

[39] Linux Manpages. Namespaces(7) - Linux Programmer's Manual, 2019. `http://man7.org/linux/man-pages/man7/namespaces.7.html`.

[40] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, 1993.

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

[42] MongoDB. Mongo-perf. `https://github.com/mongodb/mongo-perf`, 2019.

[43] Collin Mulliner and Matthias Neugschwandtner. Breaking payloads with runtime code stripping and image freezing, 2015.

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

[45] Musl. Musl Libc. `https://www.musl-libc.org`, 2019.

[46] Tapti Palit, Yongming Shen, and Michael Ferdman. Demystifying cloud benchmarking. In *2016 IEEE international symposium on performance analysis of systems and software (ISPASS)*, pages 122–132. IEEE, 2016.

[47] Nikita Popov. Php abstract syntax tree. `https://github.com/nikic/php-ast`, 2019.

[48] PostgreSQL. pgbench. `https://www.postgresql.org/`, 2019.

[49] Python. Python ast. `https://docs.python.org/3/library/ast.html`, 2019.

[50] 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.

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

[52] 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.

[53] 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)*, 2017.

[54] Vaibhav Rastogi, Chaitra Niddodi, Sibin Mohan, and Somesh Jha. New directions for container debloating. In *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation*, pages 51–56. ACM, 2017.

[55] Redis. Redis-benchmark. `https://redis.io/topics/benchmarks`, 2019.

[56] Henrik Rosendahl. Containers vs Virtual Machines (vms) – A Security Perspective, 2016. `https://neuvector.com/container-security/containers-vs-virtual-machines-vms`.

[57] Selenium. Selenium. `https://selenium.dev/`, 2019.

[58] Sematext. Sematext Agent Monitoring and Logging - Docker Hub. `https://hub.docker.com/_/sematext-agent-monitoring-and-logging`, 2019.

[59] 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/`.

[60] Rui Shu, Xiaohui Gu, and William Enck. A study of security vulnerabilities on docker hub. In *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*, pages 269–280. ACM, 2017.

[61] Linhai Song and Xinyu Xing. Fine-grained library customization. In *Proceedings of the ECOOP 1st International Workshop on SoftwAre debLoating And Delayering (SALAD)*, 2018.

[62] Kanchi Gopinath Suparna Bhattacharya and Mangala Gowri Nanda. Combining concern input with

program analysis for bloat detection. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, 2013.

[63] David Wagner and R Dean. Intrusion detection via static analysis. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*, pages 156–168. IEEE, 2000.

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

[65] 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.

[66] Tong Zhang, Wenbo Shen, Dongyoon Lee, Changhee Jung, Ahmed M. Azab, and Ruowen Wang. Pex: A permission check analysis framework for linux kernel. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1205–1220, Santa Clara, CA, August 2019. USENIX Association.

[67] 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 *International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 691–710. Springer, 2018.

[68] 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.