A Hybrid System Call Profiling Approach for Container Protection

Yunlong Xing, Xinda Wang, Sadegh Torabi, Zeyu Zhang, Lingguang Lei, and Kun Sun

***Abstract*—Over-privileged Linux containers might put the underlying OS at risk by permitting pointless system calls that could be exploited as entry points to the kernel. However, finding such security profiles is a difficult task as it demands examining the implementation/operation of containers in the absence of knowledge regarding its required system calls. In this article, we propose a hybrid approach to limit the system call usage during the execution of containers. Specifically, given an application container, we maintain an initial fine-grained whitelist by dynamic tracking to control the run-time security along with a complementary whitelist extracted via static analysis to maintain container's functionality while addressing the coverage limitation of dynamic analysis. Our method automatically analyzes the container behavior to identify three execution phases and dynamically enforce the corresponding fine-grained system call whitelists. The invoked system call will be compared with both whitelists to decide if it should be killed to guarantee the container security or logged for further analysis. Our evaluation results with 193 Docker images demonstrate the effectiveness of our approach in significantly reducing the required system calls during the applications' life-cycle. Furthermore, we discuss the reduced attack surface and demonstrate the efficiency of our approach through empirical analysis results.**

***Index Terms*—Container security, system call reduction, seccomp filter, static & dynamic analysis, Docker image.**

I. Introduction

LINUX containers are primarily used to provide operating system-level virtualization for running large-scale microservice-based applications in a cloud environment. In contrast to traditional virtual machines (VM), which require their own operating systems (OS), containers provide a way to virtualize an OS so that multiple workloads can run on a single OS kernel. This makes containers very efficient and scalable, as they can start up much faster while requiring fewer system resources. Moreover, the convenience and flexibility of software deployment using containers have made them hugely popular, with about 83% of companies adopting various forms of infrastructure using containers [1]. Additionally, several container management technologies have been introduced to build and manage containerized applications on servers and cloud (e.g., Docker).

Despite their advantages, containers could endanger the security of the OS kernel. This can be accomplished by using vulnerable containers as a means of exploiting kernel interfaces to escape into the host kernel space and compromise the OS or other containers running in the same environment. For instance, using `unshare` system call, attackers can escalate privileges and gain necessary capabilities to complete Kubernetes container escape [2]. Also, a container process may abuse the `bpf` system call [3] to execute arbitrary code in the context of the kernel. The design of the containers, which represent isolated user instances sharing the kernel and resources of the underlying host system, is primarily to blame for these problems. This makes the kernel the most vulnerable attack vector, particularly in the presence of malicious or compromised over-privileged tenants.

To address these security issues, Linux containers are implemented based on `namespace` [4] and `cgroups` [5] and offer various levels of protection through user/process isolation, hierarchical grouping, and resource usage control/monitoring. Moreover, Linux `capabilities` [6] were incorporated to provide fine-grained security mechanism by allowing a process to have a subset of the privileges typically associated with the superuser (root) account, without actually running as root. `AppArmor` [7] and `SELinux` [8] are two additional security modules to provide more access controls for the applications, processes, and files.

As system calls are the main entry points for container processes to exploit kernel vulnerabilities, `seccomp` [9] has been embraced as an effective tool to reduce the attack surface by restricting the available system calls for containers through security profiles. The seccomp profile represents an allowlist that by default restricts access to all remaining system calls. Nevertheless, identifying such a security profile that ensures correct and secure container operations is a challenging task due to (i) a lack of knowledge about the required system calls per container image and (ii) the increasing number of diverse container images that may require various system calls.

To address these challenges, previous work proposed utilizing static and/or dynamic analysis techniques for analyzing container images to determine the required system calls [10], [11], [12], [13]. Nevertheless, despite such efforts, these approaches have their own limitations in terms of the completeness and/or correctness. To further solve these problems, we propose a multi-level hybrid approach for maintaining two whitelists for any

Manuscript received 17 May 2022; revised 22 March 2023; accepted 10 April 2023. Date of publication 19 April 2023; date of current version 16 May 2024. This work was supported in part by NSF under Grant CNS-1815650. *(Yunlong Xing and Xinda Wang contributed equally to this work.) (Corresponding author: Lingguang Lei.)*

Yunlong Xing, Xinda Wang, Sadegh Torabi, Zeyu Zhang, and Kun Sun are with the Center for Secure Information Systems, George Mason University, Fairfax, VA 22030 USA (e-mail: yxing4@gmu.edu; xwang44@gmu.edu; storabi@gmu.edu; zzhang28@gmu.edu; ksun3@gmu.edu).

Lingguang Lei is with the Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100864, China (e-mail: leilingguang@iie.ac.cn).

Digital Object Identifier 10.1109/TDSC.2023.3268124

specific container, one obtained from static analysis to maintain all of the container's functionalities and the other captured by dynamic tracking to control security. Given a container, for each execution phase (i.e., booting, running, and shutdown), we first dynamically enforce the fine-grained profile filter collected by dynamic tracking to limit the required system calls. If the container invokes any system calls beyond this whitelist, we then compare it with the system call lists obtained via static analysis to decide if the container should be killed or if the system call needs to be logged for further analysis. This paper extends a preliminary conference paper [10] by expanding the dynamic tracking phases from two (booting and running) to three (booting, running, and shutdown), which guarantees the applicability of our system during the whole life cycle of a container. In addition, we propose a new static image analysis method for obtaining a complementary system call whitelist, which includes system calls that may not be identified by dynamic analysis methods due to potential coverage and execution limitations. While this improves the security of the executed containers, our approach can also improve the usability and coverage to address a number of special corner cases.

Our experimental evaluation results with 193 widely used Docker images from various categories (e.g., Database, DevOps, OS, etc.) demonstrate the feasibility and effectiveness of our approach to significantly reduce the number of required system calls during the applications' life-cycle.

In summary, we make the following contributions:

- We develop a hybrid container profiling approach that leverages a multi-level static and dynamic analysis methodology to determine the required system calls for executing Docker images while enforcing fine-grained system call whitelists during the three main execution phases of the container life cycle. Our approach utilizes the benefits of both static and dynamic analysis techniques to implement an effective, efficient, and automated approach for securing containers.
- We develop an automated system call extraction technique that leverages static analysis to parse Docker images and identify invoked system calls. Our approach relies on constructing the layer relation for a given Docker image while analyzing the target layer binaries for invoked system calls or functions that indirectly call system calls (e.g., `libc` wrapper functions). We evaluate the proposed technique using 193 Docker images from widely used application categories and highlight the significant reduction in the number of required system calls for the majority of the containers as compared to the default configuration. Additionally, while we compare the results of our system call reduction technique to the state-of-the-art, we discuss the reduced security implications through the elimination of possibly vulnerable system calls.
- We extend previous work [10] to propose an effective approach for the automatic identification of phase-splitting time points for the execution of a given application container. We leverage our technique to profile the behaviors of containers and identify the invoked system calls during three distinguished execution phases, namely the booting, running, and shutdown phases. Indeed, our approach can be used to effectively reduce the number of required system calls by limiting access to unnecessary system calls throughout the container's execution life cycle. Moreover, our empirical analysis demonstrates the efficiency of the implemented approach, which dynamically updates the identified system call profiles without disturbing the normal operations inside the container.

II. Background

Docker Containers. Docker containers represent software packages that support OS-level virtualization while facilitating software development, shipment, and deployment. These containers are hosted in a software called Docker Engine, which was first introduced in 2013 by Docker. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application, including code, runtime, system tools, system libraries, and settings. Dockerhub provides an online repository where developers can download a variety of widely used container applications [14].

System Calls and Libc Functions. System calls are the fundamental interface between applications and the Linux kernel. Programs interact with kernel services mainly through system calls, which can be invoked by passing a system call number to registers `eax` or `rax` and triggering a software interrupt to switch to kernel mode to complete related kernel services. Current systems typically support the `syscall` or `sysenter` instructions to invoke system calls. And in this paper, we mainly focus on the Intel architecture, so we will use `syscall` instruction to capture the related system calls. In most cases, system calls are not invoked directly. Instead, they have corresponding C library wrapper functions, which invoke the system calls indirectly. For instance, the C standard library (libc) [15] provides a number of functions that will wrap system calls. Hence, most applications and other library functions can call libc functions to access kernel services instead of directly invoking system calls.

Seccomp Profiles. Secure computing (seccomp) is a sandbox tool in the Linux kernel to restrict a process from invoking certain system calls. Since the system calls provide the processes in a container with entry points into the host kernel, a malicious application may misuse such system calls to disable all the security measures and escape out of the container [16]. Seccomp can be used to reduce the number of entry points into the kernel space, thereby reducing the kernel attack surface. Since Docker 1.11.0, the seccomp option (*–security-opt*) has been supported to set a seccomp profile when the container is launched. It allows the user to specify a security profile for the operation of the container, which is represented as a list of "allowed" system calls that can be invoked by the container. Note that Docker provides a moderately protective default seccomp profile, which just blocks 44 system calls out of more than 300 ones [17]. However, such default set enables too many system calls for a given container. As a result, the extra system calls can be a springboard to attack other containers or the underlying system.

Seccomp Filters. In practice, given a seccomp profile, we can implement seccomp filters for specifying the whitelisted

system calls. The filter is expressed as a Berkeley Packet Filter (BPF) program, as with socket filters. Each process has a `task_struct` structure that contains a `seccomp` structure, which defines the seccomp state of the process. If the process is protected by seccomp, the *filter* field of the `seccomp` structure would point to the first seccomp filter defined as a `seccomp_filter` structure. The *BPF_interpreter* running in the Linux kernel is in charge of enforcing the system call filtering. When the process invokes a system call, the *BPF_interpreter* goes through all the seccomp filters attached to this process to determine if such a system call is allowed for this process or not.

In the Linux kernel, two system calls, `prctl()` and `seccomp()`, are provided to change the seccomp filters of a process. Nevertheless, we cannot directly use them to dynamically modify the seccomp filters of one container. The reason is that once a seccomp filter is installed, it cannot be altered or removed while the process is running. Although we can use these two system calls to add new seccomp filters, we are unable to remove any existing filters. Therefore, we choose to locate the memory address of the seccomp filters and directly modify their contents. However, to this end, we must bridge the semantic gaps and recover the seccomp filter-related data structures. More details will be discussed in Section III-C2.

Seccomp Working Modes. Seccomp has three working modes: seccomp-disabled, seccomp-strict, and seccomp-filter. The seccomp-filter mode allows a process to specify a filter for incoming system calls. Seccomp also allows users to define the action that should be taken when a forbidden system call is invoked. For instance, in the Docker container with seccomp enabled, the action `SCMP_ACT_KILL` means to kill the process when a banned system call is invoked. Finally, the `SCMP_ACT_LOG` indicates that the system call should be handled normally but with logs. This feature can provide means for logging and analyzing the system calls invoked by a container throughout its life cycle.

III. Approach

In this paper, we propose a hybrid approach to analyze Docker container images and identify the minimum required system calls throughout the execution life cycle. As illustrated in Fig. 1, the implemented multi-level approach consists of two main components: (i) static profiling and (ii) dynamic profiling. The static analysis generates a complete list of system calls that enables all the functionality of a given application container. The dynamic profiling helps identify a set of most frequent used system calls during the normal execution, which is a subset of the system calls identified by the static analysis. The final system call whitelist for each execution phase is determined by both results of static and dynamic analysis. If a process invokes a system call out of the static analysis-based list (and dynamic analysis-based list), it will be killed. If an invoked system call is out of the dynamic analysis-based list but in the static analysis-based list, it will be logged for further inspection. In what follows, we elaborate on the design of each component, and how these components work together to guarantee the container security.

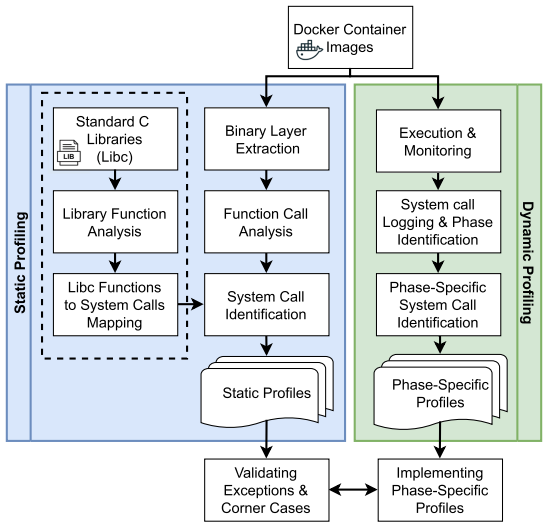


Fig. 1. Overall approach.

TABLE I
The Number of Extracted System Calls From Docker Images With *Core/All Syscalls* Extracted From the Binaries of the Core Application and All Layers, Respectively

Image	# of Core Syscalls	# of All Syscalls	Syscall Increase
redis	110	202	83.6%
couchdb	93	208	123.6%
bash	103	187	81.6%
ruby	80	192	140%
traefik	87	215	147.1%

A. Static Profiling

To automatically and efficiently identify a minimal but complete system call whitelist (i.e., static profile) for a given Docker container, we leverage static analysis techniques as elaborated in the following sub-sections.

1) Binary Layer Extraction: A Docker image contains multiple layers that are essentially files generated from commands when building it. Hence, the system calls required for running an image can be obtained by extracting executable files from all layers. However, it may introduce an extra set of system calls that are not necessarily needed in the running period, which can enlarge the attack surface of containers. The reason is that a Docker image encapsulates not only the core application like `redis` to provide core functionalities, but also general commands and programs like `gzip` and `more` to provide auxiliary functions. Nevertheless, the general commands and programs are typically not used during the running period of the core application. Table I shows examples of the number of system calls extracted from different images. If we extract system calls from binaries in all layers, the number of system calls can increase by more than 80% compared to the number of system calls required for running the core application. To exclude such system calls, we need to identify the target layer and extract its related binaries, which are analyzed further to identify required system calls.

To achieve this, we locate a text document named Dockerfile [18], which records a series of commands used to construct all layers within the image. Fig. 2 illustrates an example of a

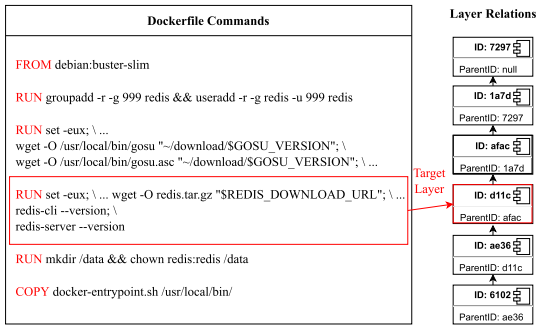


Fig. 2. Identified target layer using Dockerfile commands of `redis` image and the extracted layers' relations.

Dockerfile for the `redis` image, which consists of six layers of Dockerfile commands. Note that these layers are used to perform various functions. For instance, the first layer is used to build a Debian operating system with the buster-slim version. The second layer adds a user called `redis` to a group. Similarly, we identify the fourth layer as the target layer, where the RUN command is used to install the `redis` client and `redis` server applications.

Note that the organizational structure of a Docker image does not provide a clear relationship between the layers created in the image. Thus, to extract the target layer binaries, we need to construct the correct relations between all layers in a given image. To accomplish this, we apply the `docker save` command to the original image to create a static file system. The file system organizes files from each layer into folders, each with its own `json` configuration file containing the IDs of the current layer and its parent layer. By analyzing all the layer relationships from `json` files, we obtain a one-way acyclic chain relationship between different layers, with the initial layer having a 0 in-degree and the final layer having a 0 out-degree, respectively. An example of the constructed layer relationships for the `redis` image is presented in Fig. 2. Using such relationship information, we locate the binary files associated with the target layer and use them for further analysis.

2) Function Call Analysis: System calls are the fundamental interface between user programs and the Linux kernel. In most cases, system calls are not invoked directly. Instead, they rely on various wrapper functions from the Standard C Libraries (i.e., libc), which act as an interface between the user and the kernel spaces. Therefore, to identify the required system calls for a Docker image, we need to analyze its corresponding target binary files and identify all function calls and their relations to such interface functions in libc. Such calling relationships between library functions can be extremely complex, resulting in calling loopbacks. Thus, to model complex calling relations between functions, we construct a function call graph with direct calling relations between functions.

To achieve this, we leverage `objdump` to disassemble the target binary files into assembly code instructions and then analyze the code to determine the corresponding function calling relationships. As shown in Fig. 3, we extract the function call

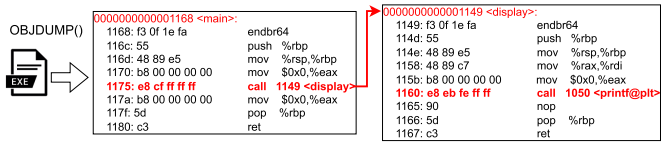


Fig. 3. An example of function call relation from binary.

relation from a binary executable representing a simple C++ program that uses the `main()` function to call the `display()` function, which calls the `printf()` function to display a simple message (e.g., "hello world") on the standard output. Note that the function calling relations have a fixed format in assembly codes (Fig. 3). As a result, we construct the complete direct calling relations graph for all functions by scanning the entire assembly code.

In addition to the identified libc function calls within the target binary, container images may leverage third-party libraries to indirectly call libc functions. To identify such indirect calling relations, we leverage the `ldd` tool to obtain all called libraries from a given target binary. Then, we follow the procedure described in the previous paragraph (Section III-A2) to analyze the identified binary files and construct the corresponding mapping and function call graphs. Finally, we leverage the Floyd–Warshall algorithm [19] to transit all constructed function call graphs for a given container image to identify the called libc functions, which are analyzed for identifying the invoked system calls.

3) System Call Identification: The operations and functions within containers invoke system calls indirectly by utilizing libc function calls. Therefore, to identify the system calls associated with the operations of a Docker image, we need to create a mapping between libc functions and system calls. Note that previous work [11] leveraged tools such as Egypt [20] to create a calling graph between library functions (e.g., Glibc) and system calls. However, these tools rely on compiling the source code of the library to generate the Intermediate Representation (IR), which is then utilized to extract the calling graph. However, in a Docker image, almost all the executable files and libraries represent as binaries, without source code. Even for open-source project, if we conduct source-code analysis, we have to obtain the configuration information, e.g., version, compilation option, and optimization level. In addition, many applications depend on ohter libraries, which call libc later, to complete the system call invocation, however, the relied libraries may not be open-sourced. Therefore, we create our function calling graphs by analyzing the binary code directly without the need for additional source code analysis (Section III-A2). This makes our calling graph construction approach more generalizable and applicable to the analysis of any given binary. In what follows, we provide further details about our approach towards mapping system calls to libc functions.

Libc Functions to System Calls Mapping. In general, a system call is invoked by passing its corresponding number, as presented in `unistd_64.h` header file, to the `eax` or `rax` registers,

```
bd404: b8 4e 00 00 00    mov    $0x4e,%eax
bd409: 41 55             push   %r13
bd40b: 41 54             push   %r12
bd40d: 55                push   %rbp
bd40e: 53                push   %rbx
bd40f: 48 89 f3          mov    %rsi,%rbx
bd412: 48 83 ec 08       sub    $0x8,%rsp
bd416: 0f 05             syscall
```

(a) Direct value passing (eax).

```
34678: b8 0e 00 00 00    mov  $0xe,%rax
3467d: 4c 89 44 24 48    mov  %r8,0x48(%rsp)
34682: 4c 89 44 24 50    mov  %r8,0x50(%rsp)
34687: 4c 89 44 24 58    mov  %r8,0x58(%rsp)
...
3469b: 4c 89 44 24 78    mov  %r8,0x78(%rsp)
346a0: 4c 89 84 24 80    mov  %r8,0x80(%rsp)
346a8: 0f 05             syscall
```

(b) Direct value passing (rax).

```
21ac2: ba 3c 00 00 00    mov  $0x3c,%edx
21ac7: 66 0f 1f 84 00    nopw (%rax,%rax,1)
...
21ad0: 31 ff             xor  %edi,%edi
21ad2: 89 d0             mov  %edx,%eax
21ad4: 0f 05             syscall
```

(c) Indirect value passing through multiple registers.

```
e46e0: 48 8d 05 71 f2    lea  0x27(%rip),%rax
e46e7: 8b 00             mov  (%rax),%eax
e46e9: 85 c0             test %eax,%eax
e46eb: 75 13             jne  e4700
e46ed: 31 c0             xor  %eax,%eax
e46ef: 0f 05             syscall
```

(d) Indirect value passing through XOR operation.

Fig. 4. Different methods for passing the system call numbers to eax/rax registers.

followed by a syscall instruction to complete the kernel operation. Based on this observation, we initially leverage the approach described in Section III-A2 to analyze the libc binary files and construct the corresponding call graphs for all functions and the target syscall instruction. Moreover, to identify the invoked system calls, we backtrack the values stored or passed into the eax and/or rax registers starting from the syscall instruction call. This is not a straight-forward task, as there are various complex ways for passing system call numbers to registers, which adds to the complexity of identifying the correct system call mapping.

Our analysis indicates that there are various common methods for assigning a system call number to the eax/rax registers. For instance, as illustrated in Fig. 4(a) and (b), constant values (i.e., 0x4e and 0xe) representing two system call numbers (getdents() and rt_sigprocmask()) are directly assigned to the eax and the rax registers. Additionally, the system call number could be passed to the eax/rax registers indirectly by going through multiple registers such as the edx, as illustrated in Fig. 4(c). Finally, system call numbers might be passed as the results of numerical calculations. For instance, as illustrated in Fig. 4(d), the xor operation is used to set the value of the eax to 0, representing the read() system call. Note that this technique is commonly used over the value assignment instructions (e.g., "mov $0x0,%eax") due to its efficiency.

Directly Invoked System Calls. In addition to the system calls invoked through the libc functions, our analysis of various Docker images showed a few cases where target binaries directly invoked system calls by passing their corresponding numbers to the eax/rax registers. In line with that, we leverage the register value backtracking method described in the previous sub-section to identify such cases and complement the list of system calls associated with the analyzed Docker images.

Environment-Related System Calls. In addition to the identified system calls for implementing the functionalities of a Docker image, there are other system calls required to create the execution environment for running containers. These additional system calls are mainly needed for creating the execution environment for programs (e.g., initializing the stack and allocating memory) and creating the running environment for containers through the runc process.

TABLE II
SYSTEM CALLS INVOKED TO CREATE THE EXECUTION ENVIRONMENT FOR PROGRAMS

Functionality	Execution Environment System Calls
Process-Related	arch_prctl, exit, exit_group, sched_yield, tgkill
Memory-Related	brk, mmap, mprotect, mremap, munmap
File-Related	close, fcntl, fstat, access, getcwd, getdents, lseek, lstat, newfstatat, openat, read, readlink, stat, write, writev
Signal-Related	rt_sigaction, rt_sigprocmask, rt_sigreturn
ID-Related	getegid, geteuid, getgid, getpid, gettid, getuid
System-Related	clock_getres, futex, ioctl, madvise, prlimit64, setitimer, sysinfo, time, uname

TABLE III
SYSTEM CALLS INVOKED TO CREATE THE RUNNING ENVIRONMENT FOR CONTAINERS

Functionality	Container Running Environment System Call
Capability-Related	capget, capset
File-Related	chdir, close, epoll_ctl, epoll_pwait, fchown, fcntl, fstat, fstatfs, getdents64, newfstatat, openat, read, write
Process-Related	execve, prctl, sched_yield
System-Related	futex, nanosleep
ID-Related	getppid, setgid, setgroups, setuid

As the environment-related system calls cannot be obtained by analyzing the target binaries, we leverage dynamic analysis methods to execute an empty program while applying the strace tool to monitor all invoked system calls. The assumption is that all invoked system calls are necessary for the execution of the program since the main() function in the program was cleared and did not contain any instructions. Table II lists the identified environment-related system calls. For evaluation purposes, we tested 100 different programs and found that these system calls are sufficient and consistent for creating the execution environment in all cases. Furthermore, to obtain the required system calls for creating the running environment for Docker containers, we monitor the startup process of a container to obtain the system calls invoked by the runC, which is a portable container runtime that includes all of the necessary code used by Docker to interact with system features related to containers. Indeed, our analysis of 100 different containers highlighted a fixed set of system calls that are necessary to run all containers, as summarized in Table III.

B. Dynamic Profiling

Although the static analysis generate a complete necessary system call list for the whole life-cycle of a container, it is hard to identify the required system calls during each execution phase. However, even for the same container, the invoked system calls in booting phase and running phase can be dramatically different. In our previous work [10], we tried to address this by profiling and generating individual system call whitelist for booting phase and running phase. In this paper, we extend such efforts by addressing the main limitations of previous work in terms of performance, accuracy, and automated phase detection and system call identification processes. Additionally, we introduce a three-phase realization of the execution life-cycle of a given container application by adding the "shutdown" phase to the typical "booting" and "running" phases for application

containers. In fact, our empirical analysis shows that the three-phase realization can provide more fine-grained system call profiling, which reduces the number of required system calls during the running phase while guaranteeing a graceful and secure shutdown for a container.

1) Execution and Monitoring: For a container, the booting phase is responsible for setting the container environment and initializing the service. In the running phase, the container service begins to accept service requests and send back responses. In the shutdown phase, certain critical jobs are performed, e.g., storing important data for future use and processing the outstanding requests.

Dynamic profiling is based on running a given container and monitoring its behavior in terms of invoked system calls throughout its life cycle. To this end, we examine database and web server containers as the two most popular application categories [1]. Note that our approach can be implemented with all container applications, however, we mainly focus on these two categories due to the availability of testing tools and benchmarking datasets. We analyze four popular database containers, such as MySQL [21], Postgres [22], MongoDB [23], and Redis [24]. We also analyze four popular web server containers in Docker Hub, namely, node.js, nginx, Tomcat, and httpd. To do this, we pull the following Docker images from Dockerhub (Wiki.js [25], DokuWiki [26], XWiki [27], and MediaWiki [28]), which represent the node.js, nginx, tomcat, and httpd web servers with an installed wiki software.

Triggering Booting & Shutdown Phases. We execute the specified containers and monitor the invoked system calls during the execution life-cycle. Note that invoking the required system calls for the booting and shutdown phases can be done by explicitly triggering the container booting and shutdown operations using `docker run` and `docker stop` commands. Nevertheless, the most challenging part is generating a complete workload for the running phase.

Executing Containers (Running Phase). To analyze database containers, we leverage the following three methods/tools to trigger the required workloads during the running phase: (i) we exercise the commands manually according to the official manual references of each data store [21], [22], [23], [24]; (ii) we utilize HammerDB [29] and Yahoo Cloud Serving Benchmark (YCSB) [30] to simulate the case when many users access the database server concurrently. Note that HammerDB is used to generate workload for Postgres, MySQL, and Redis, while YCSB is utilized for analyzing Redis and MongoDB, respectively; (iii) we use penetration testing tools to enumerate all the data in the data store platforms (e.g., databases and tables in MySQL). We leverage SQLMap [31] to generate normal user behaviors workload for SQL-based databases such as MySQL and Postgres. We also utilize NoSQL-Exploitation-Framework [32] and NoSQLMap [33] for analyzing Redis and MongoDB, respectively.

To analyze web servers, we consider the fact that the available system calls in the running phase might vary among different web server applications. Therefore, we perform evaluations on a httpd web server [34] with a popular bulletin board system with PHPBB3 deployed. To trigger the system calls during the running phase, we utilize the httperf [35] to access the web server continuously with increasing numbers of requests per second. We also scan the web servers using skipfish [36], which is a tool for conducting automated security checks for the web applications by recursive crawls and dictionary-based probes, hence exercising many corner cases. Besides, we manually access the websites to trigger as many functions as possible.

2) System Call Logging and Phase Identification: We propose a new approach to logging the invoked system calls during the execution of a given container application by leveraging the Linux audit system [37]. The idea is to configure a container with a seccomp filter that contains an empty list of allowed system calls, while using the SCMP_ACT_LOG option [9] to log all invoked system calls as violations in the audit log with their corresponding timestamps. After the container is stopped, we can obtain a list of all system calls invoked by a given container during its execution, which can be leveraged to segregate system call invocation during the three execution phases (booting, running, and shutdown). Now, the question is, *how can we identify the beginning and end times of each phase?*

To answer this question, we explore several techniques to investigate the application execution life-cycle. Normally, the life cycle of an application container can be divided into three phases: booting, running, and shutdown. To identify the beginning of the booting phase for all application containers, we trace the *docker run* command, which is used to initiate/start a container. Moreover, to determine the end of the booting phase, which is also the beginning of the running phase, we propose an approach that relies on the behavioral characteristics of the analyzed applications during idle running. Indeed, the analysis of various database and web server applications showed that they invoke a large number of system calls when booting and initializing the environment, while continuing with a relatively smaller set of repeatedly invoked system calls throughout the remaining idle running phase [10], [13]. Therefore, by monitoring the behaviors of various containers over time, we can empirically identify a grace period T_n (e.g., 100 seconds), which is long enough to ensure reaching the end of the booting phase while entering the running phase for the analyzed containers.

While adopting such a time-based approach has been shown to be effective [10], it has a number of limitations that hamper the efficiency and accuracy of the overall phase identification process. For instance, the time required to execute the booting phase may vary based on the underlying hardware specifications. Thus, in practice, we will need to set the grace period to a relatively longer period to avoid any discrepancies in the collected data. Consequently, setting a longer grace period may add some extra system calls from the running phase to the booting phase. To address the limitations of the time-based phase identification method, we rely on capturing the repeating behaviors in terms of the sequence of invoked system calls. To capture such repeating behavior (e.g., waiting for response from the user), we divide the execution time into equal units of $t_i = 1$ second. Then, for every execution time interval $t_i \in \{0, 1, 2, \ldots\}$, we record the set of invoked system calls x_i and compare it with the ones during previous time units (e.g., $\{x_{i-1}, x_{i-2}, \ldots,$ and $x_{i-N}\}$). Through our empirical analysis and testing, we found that the beginning

of the running phase can be identified when a repeating pattern is found during a minimum of five consecutive time periods (i.e., $N = 5$). Specifically, we use the Linux *strace* tool to dynamically trace the invoked system calls and implement a timer to record their time units for comparison. Since a container is a group of processes sharing the same set of kernel resources, a process inside a container is a normal process with a different PID when viewed from the host OS. For example, each container contains a process with PID 1, but the same process may have a different PID on the host OS. Therefore, instead of running *strace* to trace the processes inside one container, we can trace the same process on the host OS using its PID outside the container. To guarantee the completeness of the tracing results, we enable the `-f` option of *strace* to also trace all children of the processes currently being traced. In this manner, we can collect the set of a container's invoked system calls to identify the recurring pattern as the start of running phases.

To identify the beginning of shutdown phase, we trace the SIGTERM termination signal invoked by the Linux kernel when the *docker stop* command is used to shutdown the container. This is done by utilizing KProbes [38], [39] to trace a kernel function named *prepare_signal()*, which invokes SIGTERM by passing the signal number (i.e., 15 for SIGTERM) and the target application's PID as arguments (PID=1 for containers). Note that KProbes instruments two functions named *pre-handler* and *post-handler* before and after the execution of the probed instruction, such as the *prepare_signal()* function. While both handler functions can be used to probe the *prepare_signal()*, we leverage *pre-handler* to ensure we capture all invoked system calls using the seccomp filter before the termination of the container process.

3) Phase-Specific System Call Identification: As described in the previous subsection, we were able to log all invoked system calls during the execution of a given container. Furthermore, we were able to detect and record the timestamps of the beginning and end of the three main execution phases. As such, we leverage such information to identify the required system calls during each execution phase respectively. The outcome of this step represents three sets of phase-specific profiles, which enable the required system calls for the operations of the containers during the booting, running, and shutdown phases, respectively.

C. Enforcing Phase-Specific Profiles

1) Generating Phase-Specific Seccomp Profiles Considering Both Static and Dynamic Analysis: To generate the seccomp Profiles that restrict the usage of system calls for each execution phase of a container, we utilize the both the static and dynamic analysis results. Although the results from dynamic tracking may be a subset of the static analysis, we still need to consider both of them. The reason is twofold. First, the system call list generated from static analysis is required to enable the functionality during the whole life-cycle of a given container. To further narrow down the attack surface for each execution phase, dynamic tracking is leveraged to automatically detect the execution phase and record the corresponding system calls, respectively. Second, dynamic analysis may suffer from the coverage problem due to the limitations of the execution and bench-marking, which might miss on capturing some system calls for corner cases. In this case, we rely on the static profiles to produce a complementary whitelist.

Our generated system call whitelists (i.e., seccomp profiles) are composed of a set of system call numbers and their corresponding actions. Considering three possibilities during the specific execution phase for a system call invoked by a process within the container, we set their actions to implement a system call whitelist as following:

1) If the invoked system call is not found within the static and dynamic profiles, we kill the process by setting the seccomp action to SCMP_ACT_KILL [9].
2) If the invoked system call is found only in the static profile, then we assume that it could be related to a corner case within the normal functionalities. To deal with such cases, we let such execution to continue and log the system call invocation for further analysis by leveraging the SCMP_ACT_LOG seccomp action.
3) If the invoked system call is found within the dynamic profile (regardless of the static profiles), it will be normally executed.

Note that the above actions can be adjusted according to the specific usage scenarios. For instance, if a non-sensitive application is running, SCMP_ACT_LOG can be enforced for any cases to allow completion of all the executions temporarily and log for further analysis.

2) Dynamically Updating Phase-Specific Seccomp Profiles in Each Execution Phase: Given the generated system call lists for every execution phase, we aim to dynamically update the container seccomp filters when it enters each execution phase. In Linux, it is possible for a process to be attached with multiple seccomp filters, and all the seccomp filters are organized in a one-way linked list. Each seccomp filter is implemented as a program code composed of seccomp instructions, which is represented as a `bpf_prog` structure. Therefore, to implement the seccomp filters, we leverage the *libseccomp* library [40] to construct three `bpf_prog` structures in the memory for the booting, running, and shutdown phases. Particularly, we use the *seccomp_rule_add()* and *seccomp_export_bpf()* methods to add the required system calls and export the resulting *bpf* filter program, respectively.

Each seccomp filter structure has a *prog* pointer that points to the `bpf_prog` structure. Therefore, to implement a phase-specific seccomp profile, we dynamically redirect the *prog* pointer to the corresponding `bpf_prog` structure when the container enters a new phase. To do this, we locate the `task_struct` of the main process inside the container through its PID (i.e., PID=1). Since a parent process relies on creating new child processes to run the container, we leverage the inheritance attribute of the seccomp filter to enforce the new filters on all forked child processes.

We implement this using a kernel module to dynamically change the filter outside the container. The kernel module aims to obtain the *bpf* filter program generated in the user space and convert it into a seccomp filter in the kernel space. We install the kernel module with the `insmod` command, and the data content

TABLE IV
ANALYZED CONTAINER IMAGES

Categories	# Images (%)	Examples
Database	47 (**24.4**)	`redis`, `mongo`, `couchdb`
DevOps	45 (23.3)	`consul`, `jenkins`, `maven`
Infrastructure	41 (21.2)	`nginx`, `percona`, `tomcat`
Language	40 (20.7)	`python`, `perl`, `java`
OS	20 (10.4)	`ubuntu`, `centos`, `fedora`

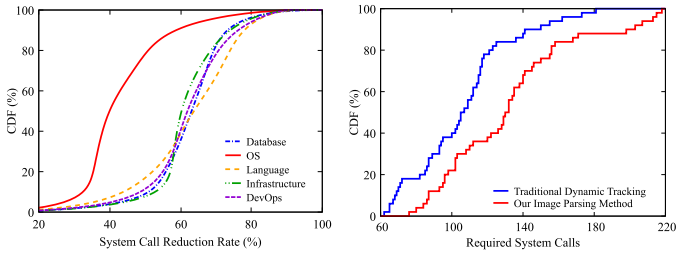
of the user space *bpf* filter can be passed into the kernel module as the kernel module parameter. Then, the module accepts the kernel buffer filter data to convert it into `bpf_prog` structure for changing the filter dynamically outside the container. To guarantee its security, the module does not export any APIs. Also, once the container enters the shutdown phase (i.e., the last execution phase of a container requiring the seccomp filter update) and the corresponding seccomp filter has been altered, the module can be safely removed. Therefore, it is unlikely for malicious processes within the container to misuse the module. Additionally, attackers within the container are hard to modify the seccomp filter via direct memory overwriting due to the difficulty of user processes in locating and understanding the seccomp filter-related structures in kernel space. Even if they could, these programs would be unable to write into the kernel space memory.

IV. Experimental Results and Evaluation

A. Static Profiling

We leverage Dockerhub [14] to obtain 193 popular official images with over 10 million downloads. As summarized in Table IV, these images can be generally classified into five categories in terms of their functionality, with the database images representing about 24% of the data, followed by the DevOps (23%) and Infrastructure (21%) images, respectively. Note that these images were not randomly selected. Instead, we use the same list as in previous work [11] to simplify evaluation and result comparison when necessary.

1) System Call Reduction: The static analysis results indicate a significantly smaller number of required system calls as compared to the available 300+ system calls, with an average of about 129 required system calls for the analyzed images ($min = 26$ for `hellow-world` and $max = 207$ for `fedora`). In general, the OS category images require a relatively larger number of system calls, ranging between 95 and 226. On the other hand, images from the remaining categories require between 80 and 180 system calls, with the majority of them requiring less than 145 system calls, respectively. Overall, our static analysis results indicate that we can significantly reduce the number of required system calls for all container images. This reduction represents about 78% system calls at most and 62% system calls on average for all 193 containers (Fig. 5(a)). Indeed, our analysis shows that we can reduce the number of necessary system calls by more than 60% for containers across all categories except the OS, which reached an average reduction of about 42% (min=35% and max=53%). This is typically due to the nature of OS container applications, which require a larger set of system calls to support a wide range of OS-related operations.



(a) Syscall reduction for different categories.

(b) Comparison between dynamic tracking and our static image parsing method.

Fig. 5. CDF analysis results for (a) system call reduction rate and (b) dynamic analysis.

It is worth noting that the required system calls of the containerized apps that are based on interpreters (e.g., Python, Node.js) depend on the executed code on top of the interpreter. Therefore, it is difficult to identify what behaviors these customized containers will produce until we load/execute them. To deal with such special cases, we can leverage our proposed hybrid approach to initially extract the required system calls for the interpreter applications using static/dynamic profiling. Furthermore, we can perform consequent analysis on the user codes (if available) to identify the additional system calls by the user codes to update the final allowlist.

2) Validation: To validate the results, we configure the analyzed containers using the obtained system call whitelists and execute the containers using various manually exercised inputs and states. Our objective is to check if the containers can boot and perform different functionalities without any issues or errors. We also leverage the approach described in Section III-B to record all invoked system calls during the experiments and use them for further analysis.

In general, our experimental results show that all containers boot successfully and operate without any issues. Additionally, as shown in Fig. 5(b), our static analysis presents a complementary system call list that includes ones that cannot be identified by the dynamic tracking. This is due to the limitations of the dynamic analysis technique, which cannot cover all execution branches for the analyzed containers, even when leveraging sophisticated fuzzing technique [41]. On average, the static analysis approach in our experiments detects 129 system calls, and the dynamic approach only identifies 105 out of them, which proves the necessity of considering both whitelists retrieved from static and dynamic analysis to guarantee the container's functionality while addressing the coverage limitation of dynamic analysis. Note that, since we cannot get the official description of the system calls required for each container (i.e., there is no ground truth on the required system calls for a container or publicly available evaluation dataset), we cannot adequately quantify the accuracy, false positive rate, and false negative rate of our approach.

3) Comparison With State of the Art: We evaluate the effectiveness of our approach by comparing the static analysis outcomes with previous work. To the best of our knowledge, Confine [11] represents the state-of-the-art work, which leveraged static analysis techniques to determine the required system

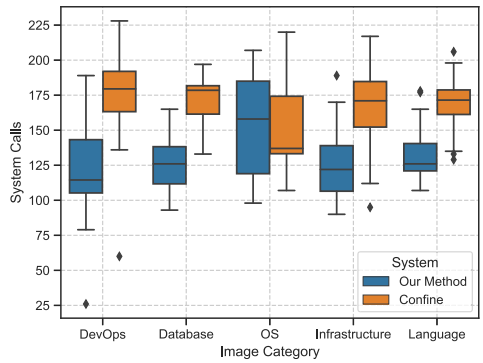


Fig. 6. Comparing the distribution of the required system calls per image category.

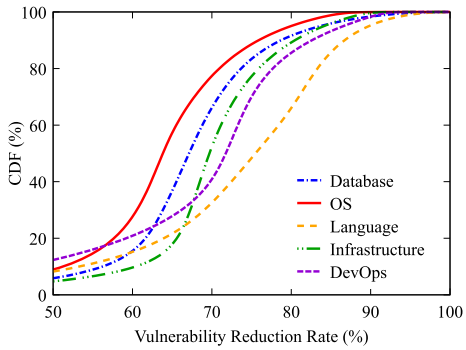


Fig. 7. CDF analysis results for vulnerability reduction rate.

TABLE V
COMPARISON WITH CONFINE [11] ON NUMBER OF FILTERED SYSTEM CALLS AND VULNERABILITY REDUCTION RATE FOR THE TOP 10 MOST DOWNLOADED IMAGES

Image*	Filtered System Calls		Vulnerability Reduction	
	Confine	Our Method	Confine	Our Method
couchdb	157	240 (+83)	50%	68% (+18%)
ubuntu	198	135 (-63)	68%	66% (-2%)
alpine	162	165 (+3)	65%	70% (+5%)
redis	179	223 (+44)	58%	62% (+4%)
node	170	192 (+22)	62%	64% (+2%)
postgres	141	213 (+72)	52%	66% (+14%)
mysql	149	183 (+34)	60%	64% (+4%)
nginx	177	188 (+11)	67%	68% (+1%)
mongo	152	193 (+41)	53%	60% (+7%)
python	154	177 (+23)	66%	70% (+4%)

*Images were ranked based on total downloads as of May 27, 2021.

calls for container images. Note that Confine is designed to extract the required system calls for containers following two main steps: (1) manually configure and run a container for a period of time to identify all called binaries and libraries that are required for initializing the execution of a container image; and (2) perform static analysis of the called binaries and libraries to extract the required system calls. The initial step of Confine [11] is to run the target container, hence the human involving, such as launching prerequisite containers (e.g., `rocket-chat`), or specifying complex configuration settings, is inevitable, which is the reason why 43 images (out of 193 images) are beyond their scope due to the manual effort to setup. In contrast, following our static analysis approach, we were able to analyze all 193 containers and provide the whitelisted system calls, respectively. Note that to ensure consistency, we excluded the 43 images that were not successfully analyzed by Confine from further analysis.

For the remaining 150 container images, the number of filtered system calls obtained by our approach shows a significant improvement over Confine, with a noticeable reduction in the number of whitelisted system calls across the majority of the analyzed images from all categories (Fig. 6). More specifically, our approach resulted in fewer system calls for about 94% of the analyzed images (141 out of 150), with an average of 36 fewer system calls ($\sigma = 17.4$, $min = 1$, $max = 90$). Indeed, by looking at the top 10 most popular container images listed in Table V, we observe that our approach can filter a relatively larger set of unnecessary system calls for almost all images (except `ubuntu`), as compared to Confine.

On the other hand, Confine identified a smaller number of system calls for 9 images, which are mainly related to the OS category (e.g., `ubuntu`). To investigate this, we review the design and implementation of Confine and find that it only records the called binaries and libraries during the initial stages of the container execution. This is problematic for containers such as those from the OS category, where they may rely on calling additional binary/library files during later stages of their execution life-cycle. Failing to consider such binaries will lead to an incomplete list of system calls, which will disable some functionalities of the container. For instance, in `ubuntu` container, a user can create a directory using `mkdir` command, which invokes `mkdir` system call. However, when Confine customizes the required system calls for `ubuntu`, since the `mkdir` command does not run during the startup period, Confine will block it and prevent users from using `mkdir` command [42].

Finally, our analysis of the source code of Confine [42] shows that it maintains a system call blacklist along with two except lists (i.e., `exceptList` and `javaExceptList`) for all containers. These two exception lists contain 87 system calls, which are enabled for all containers regardless of the need for such system calls to execute the containers. In fact, Confine does not justify the necessity of these 87 system calls. In contrast, our approach can extract the required system call based on the target binaries, which ensures the identification of a more precise list of system calls.

4) Vulnerability Reduction: We explore the security implications associated with implementing our static profiling technique to reduce the number of available system calls for a given container. We collect more than 70 vulnerabilities from Common Vulnerabilities and Exposures (CVE) [43] in the past six years. Moreover, we assume that removing the required system call associated with a known vulnerability will implicitly prevent it for this container, leading to attack surface reduction. We also define the *vulnerability reduction rate* as the number of vulnerabilities that can be mitigated for a given container by eliminating unnecessary system calls to the total number of vulnerabilities. As shown in Fig. 7, we summarize the vulnerability reduction rate for the tested containers from all categories by limiting the system calls that may trigger vulnerabilities. We

clearly observe that the vulnerability reduction rate is significant across all categories (between 60% and 80%). In other words, using our static profiling technique, we can generate system call whitelists that can mitigate more than 60% of the software vulnerabilities directly. Moreover, as compared to state-of-the-art approaches such as Confine, our analysis results demonstrate further vulnerability reduction by removing unnecessary system calls for the analyzed container images (Table V).

B. Dynamic Profiling

The analysis results discussed in Section IV-A demonstrate the effectiveness of our approach towards obtaining static profiles that can significantly reduce the number of required system calls for executing containers as compared to the default seccomp profile. In this section, we extend our static profiling approach by leveraging dynamic profiling methods to implement more fine-grained seccomp filters that limit access to unnecessary system calls throughout various execution phases of the container life-cycle. In what follows, we provide further results in terms of the identified phase-specific system calls and the performance evaluation when implementing our dynamic profiling approach for select database and web server container images.

1) Phase-Specific System Calls: Database Containers. Table VI shows the experimental results for the database containers. The number of system calls required for container booting ranges from 69 to 106, which is approximately less than one-third of the default seccomp filter. In addition, about half of the system calls invoked in the booting phase are not needed in the running phase, and the number of system calls for the shutdown phase further decreases by around half as compared to the running phase. Moreover, although the total number of identified system calls decreases during the execution phases of a container, we noticed that each phase introduces some additional system calls that were not included in the previous phases. This indicates that the required system call sets for different execution phases intersect but do not include each other. On the whole, compared with the available 300+ system calls, our approach reduces more than 84% system calls for the running phase, more than 69% system calls for the booting phase, and more than 90% system calls for the shutdown phase for the database containers.

Web Server Containers. The tracing results for the web server containers are summarized Table VI. We observe that the number of system calls invoked in the booting phase ranges from 70 to 101. Similar to the database containers, these numbers decrease when switching to the shutdown phase. As compared to the default list of available system calls, our approach can reduce more than 70% of the unnecessary system calls throughout the three execution phases of a web server container.

Moreover, although the same application (i.e., wiki) is deployed on all analyzed web servers, only 48 system calls are found to be identical during their running phases, which occupy about 69%, 44%, 55%, and 58% of the total system calls during the running phases of the analyzed containers, respectively. Despite such overlap, our analysis results indicate that the required system calls are implementation-specific. Thus, we cannot identify a one-size-fits-all system call whitelist for the running phase of various web server containers, even if they run similar services. On the other hand, our analysis of the httpd web server with two different deployed services such as wiki and phpBB resulted in identifying 67 common system calls during the running phase, which occupied about 82% and 74% of the total running phase system calls, respectively. Interestingly, this test demonstrates that the invoked system calls are not only implementation-specific, but are also determined by the deployed services and their supported functions.

TABLE VI
NUMBER OF IDENTIFIED SYSTEM CALLS DURING BOOTING, RUNNING, AND SHUTDOWN EXECUTION PHASE

	Image Name	Unique Syscalls (Static/Dynamic)	Boot.	Run.	Shutd.
Database	MySQL	150/109	106	47	32
	Postgres	120/101	97	53	31
	Mongo	146/103	96	48	27
	Redis	127/77	69	27	22
Web Server	Wiki.js (node.js)	141/82	70	70	8
	Dokuwiki (nginx)	145/114	73	110	52
	MediaWiki (Tomcat)	132/101	101	88	45
	XWiki (httpd)	134/105	74	82	16
	phpBB (httpd)	134/101	92	91	14

2) Comparison With Static Analysis: To evaluate the effectiveness of our dynamic analysis approach and its completeness in terms of the identified system calls, we compare the outcomes to the list of identified system calls using the static analysis approach (Section IV-A). As shown in Table VI, the dynamic analysis results represent a subset of the static analysis outcomes, with fewer whitelisted system calls. This is typical due to the nature of static analysis, which covers all branches/paths in the source code/binary file, while in practice, some of these paths might not be reachable during normal operations of a given container application.

3) Performance Overhead: We evaluate the performance impacts of our approach on the running containers, and the results show that our system introduces negligible overhead on system performance. In general, the overhead introduced by our approach is mainly caused by the time it takes to update the seccomp filters and/or the time *BPF_interpreter* spends going through the instructions in the `bpf_prog` structure. Since all the processes in one container share the same set of seccomp filters, when updating the system call whitelist, we only need to add one new `bpf_prog` struct to record *bpf* instructions and release the old `bpf_prog` struct. Because the seccomp filter is enabled by default for Linux application containers, the changes to the filtering rules can barely have an impact on application performance. For a list with 100 system calls, it only takes around 10 ms to change the seccomp filter, and it only occurs three times throughout the lifetime of a container. In addition, since the phase-based new `bpf_prog` structures contain fewer instructions than the `bpf_prog` structure generated from the Docker default seccomp profile, our approach will hardly introduce extra overhead when *BPF_interpreter* checks the new seccomp policy.

To evaluate the throughput results for database containers (Fig. 8), we leverage HammerDB [29] and YCSB [30] as load

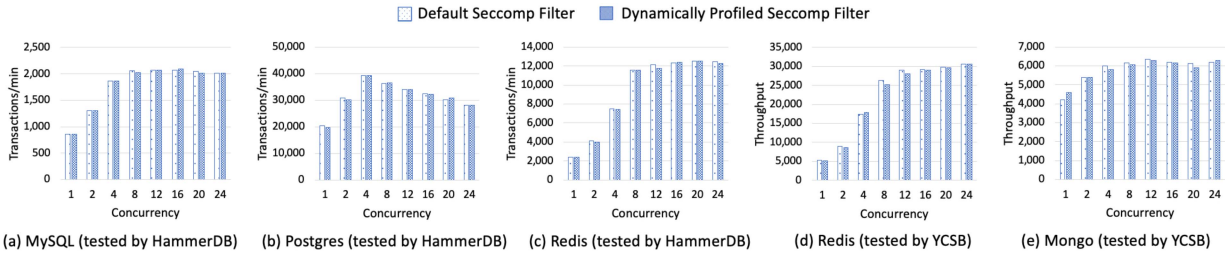


Fig. 8. Performance overhead for the database containers.

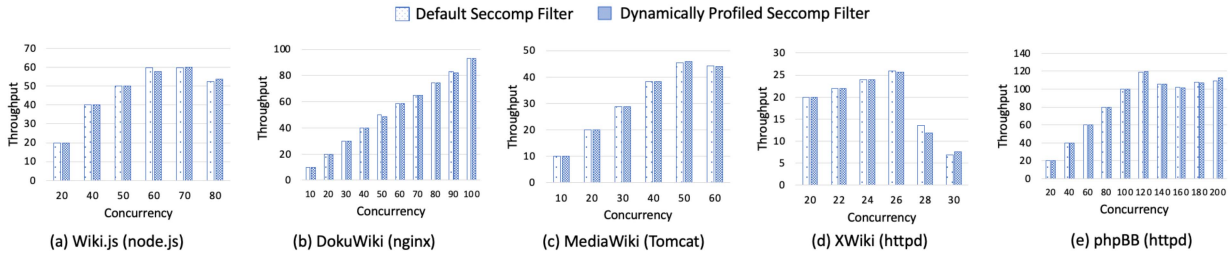


Fig. 9. Performance overhead for the Web Server containers.

testing tools. Specifically, HammerDB is used to test Postgres, MySQL, and Redis, while YCSB is for Redis and MongoDB. Note that we test Redis with both HammerDB and YCSB to evaluate the impact of using various testing tools on the overall outcomes. We set the concurrency intensity as 1, 2, 4, 8, 12, 16, 20, and 24 (i.e., the number of concurrently connected clients) for HammerDB and YCSB.

Furthermore, as illustrated in Fig. 9, we evaluate the throughput of web server containers by using httperf [35] to generate load for Wiki.js, DokuWiki, XWiki, and MediaWiki, which are the node.js, nginx, tomcat, and httpd web servers with wiki software installed, respectively. Additionally, since the available system calls in the running phase might vary among different applications, we also perform evaluations on a httpd web server with a popular bulletin board system with PHPBB3 deployed. Particularly, we record the replies obtained per second under different request rates simulated through httperf. We utilize 10,000 as an increasing rate of requests per second to issue the connections and continue the test after the server achieves the saturation point.

As shown in Figs. 8 and 9, comparison of the throughput results indicates negligible differences (i.e., less than 5% and 10% in average) between the dynamically updated and default seccomp filters for both web server and database containers.

4) Vulnerability Reduction: To investigate further, we explore specific use cases of reduced attack surface when blocking unnecessary system calls for database and web server applications during their execution phases. We focus on the running phase, where the most important operations will take place during the execution of a container. For this purpose, we consider system calls with MEDIUM or higher severity levels on all UNIX-like operating systems (e.g., Linux) to be CVE entries. We also identify high-privileged system calls (e.g., *fchown()*, *fchmodat()*, *mknodat()*, etc.), which can be misused by the attackers to gain full control over the underlying system [44], [45]. In total, we identified 113 possibly vulnerable system calls from the default seccomp filter, including 92 CVE entry-related and 21 high-privileged system calls.

TABLE VII
THE NUMBER OF REQUIRED/REMOVED VULNERABLE SYSCALLS RELATED TO CVES/HIGH-PRIVILEGED FUNCTIONALITIES

	Image Name	Required System Calls			Removed System Calls		
		Total	CVE	H.P.	Total	CVE	H.P.
Database	MySQL	47	17	1	301	75	20
	Postgres	53	22	3	295	70	18
	Mongo	48	21	2	300	71	19
	Redis	27	12	1	321	80	20
Web Server	Wiki.js	70	27	4	278	65	17
	DokuWiki	110	43	13	238	49	8
	XWiki	88	32	7	260	60	14
	MediaWiki	82	33	5	266	59	16
	phpBB	91	35	5	257	57	16

H.P. = high-privileged

As shown in Table VII, our phase-specific dynamic profiling can significantly reduce the threats associated with abusing/misusing these two types of system calls on the analyzed containers. For instance, we removed 301 system calls from the default filter during the running phase for MySQL. This resulted in reducing about 82% (75 out of 92) of the CVE entry-related system calls and about 95% (20 out of 21) of the high-privileged ones. Similarly for Redis, we were able to remove about 87% (80 out of 92) CVE entry-related and 95% (20 out of 21) high-privileged system calls. In general, as summarized in Table VII, we can significantly reduce the attack surface on database containers by removing more than 76% CVE entry-related system calls and 86% high-privileged system calls. On the other hand, the reduction rates for web server containers are between 53%-71% and 38%-81% for the CVE entry and high-privileged system calls (Table VII), respectively.

V. Discussion

Previous work discussed the need for software debloating by analyzing the implications of removing unnecessary functionalities from programs/systems on the overall performance and security of the target systems [46], [47], [48], [49]. In line with that, given the fact that containerized applications are widely deployed within cloud-based environments, several studies highlight the need for building more effective tools/techniques for analyzing the widely deployed containers and securing the cloud operations by eliminating access to unnecessary system calls by containers [10], [11], [12], [13]. By doing so, one can lower the risk of escaping malicious processes from the container, which can gain control over the host OS. Furthermore, one can also minimize the threats associated with misusing/abusing vulnerable system calls due to a lack of sanity checking.

Hybrid Approach. In this paper, we build upon previous work and propose a multi-level hybrid approach for analyzing 193 commonly used Docker container images from 5 different categories (e.g., database). First, we tried to address the limitations of previous static analysis techniques (e.g., Confine [11] and Ghavamnia et al. [13]) and propose a scalable tool/system that can be automated for analyzing containers while significantly reducing the set of required system calls for the operation of the core applications within containers. Specifically, our static profiling, which relies on static analysis of the layered container image binary files, addressed the following main issues as compared to Confine: (i) successfully extracted system calls from all analyzed container images, while Confine failed to analyze about 22% of the examined containers; (ii) resulted in identifying a smaller number of required system calls for the majority (94%) of the analyzed containers; and (iii) produced a more accurate and complete system call whitelist for OS-related containers. As compared to the work presented by Ghavamnia et al. [13]: (i) we automatically determine the transition point and extend another phase (i.e., `shutdown`), to guarantee the normal running of the whole life cycle of a container; and (ii) our method can be more scalable as it does not require manual analysis and human-labor to execute the containers before performing the static analysis/profiling.

As a result, we extend the previously introduced dynamic analysis technique (i.e., SPEAKER [10]) to implement more fine-grained system call profiling for the three-phase container execution life-cycle (i.e., booting, running, and shutdown) to supplement our static profiling results. Indeed, our in-depth experimental analysis results with select database and web server containers demonstrate the effectiveness and efficiency of our dynamic profiling approach towards disabling unused system calls while significantly reducing the attack surface when running Docker containers. Finally, given the fact that dynamic profiling may not completely cover all execution paths/branches, our hybrid approach provides a practical means for running containers by relying on the static profiling results as a superset to guarantee secure execution of containers when encountering exceptions and corner cases in real-life.

Security Analysis. Our multi-level approach relies on static analysis of layered container image binary files to extract an initial list of necessary system calls for executing the core application. In fact, such whitelists can significantly limit the number of available system calls, thus taking a first step towards reducing the overall attack surface. Additionally, since the booting and running phases are typically completed in less than tens of seconds, our security analysis mainly focused on the system calls that are allowed in the long-term running phase, which are most critical to the security of a container. Indeed, our analysis results shed light on a relatively large number of removed system calls that are related to vulnerable CVE entries and/or high-privileged functionalities (Table VII).

Despite that, it is worth noting that some of the whitelisted system calls might also be associated with vulnerabilities and security issues. For instance, the database containers might be exposed to vulnerabilities due to a range of 12–22 CVE entry-related system calls and 1–3 high-privileged system calls in their whitelisted profiles. Similarly, for web server containers, we found between 27–43 CVE entry-related system calls and 4–13 high-privileged system calls that might be misused. To further constrain those vulnerable system calls, we can combine our approach with careful sanity checking on the parameters of system calls [50], [51], [52] and strict resource access control [53], [54], [55], [56], [57]. For instance, exploiting the CVE-2016-9793 vulnerability [58] requires both CAP_NET_ADMIN capability and *setsockopt()* system call. Therefore, we can leverage the Linux capability mechanism [59] to block such exploitation by removing the CAP_NET_ADMIN, even when the *setsockopt()* cannot be removed from some containers.

Deployment Usability. Our proposed hybrid approach relies on the static analysis of target binaries extracted from image containers, which can be done effectively and automatically across containers from various categories. Additionally, while our dynamic analysis divides the Docker container's lifetime into the booting, running, and shutdown phases, the same paradigm can be adopted to analyze application containers that are deployed using tools other than Docker. This is subject to the ability to dissect the container execution into separate phases. Moreover, our approach can be smoothly integrated into container management services such as Amazon EC2 container service [60], Docker Datacenter [61], and OpenShift Enterprise [62], to name a few. The ultimate goal is to prevent malicious applications in one container from escaping into the hosting virtual machine.

VI. Limitations and Future Work

Evaluating the Static/Dynamic Profiling Results. As described in Section IV-A2, validating/evaluating the identified system calls using our static profiling is difficult due to the lack of ground truth and benchmarking tools/data for dynamically analyzing all containers. Moreover, due to the well-recognized limitations of the dynamic analysis techniques, we cannot cover all program-related branches using existing benchmarking and fuzzing techniques, which often results in an incomplete list of system calls. Despite such limitations, our exhaustive analysis of database and web server containers using various benchmarking tools confirms the static analysis results as a valid superset, which can be safely used to execute containers.

System Call Tracing Completeness. The dynamic profiling was done by executing the containers once to collect all the necessary system calls during the booting, running, and shutdown phases. However, this one-time execution and monitoring process may still produce an incomplete list of system calls due to several reasons. First, different versions of the services on different platforms may use various kinds of system calls, which greatly increases the complexity of the tracing procedure. Second, some corner cases might not be accounted for during the specific execution of a given container. For example, the Apache web server can exhibit abnormal running patterns when experiencing an extremely large amount of request traffic. Third, the existence of dynamically generated scripts such as PHP scripts can make the tracing more complicated, as these scripts can arbitrarily invoke certain system calls. To address these limitations, an iterative execution and monitoring approach can be adopted to combine system calls into an aggregated whitelist during several runs. Further, to avoid unanticipated container termination due to invoking a blocked system call, we can leverage the SCMP_ACT_LOG action instead of SCMP_ACT_KILL to log invoked system calls during consecutive runs and add the final superset to the whitelist. Moreover, as our analysis is transparent to the applications running inside the container, we can combine the system call tracing phase with the testing processof the application development to better trace the system call profile of a specific application. Finally, as we discussed in this work, the static analysis results from our method or other tailored static analyzers [63], [64] can be integrated to further improve the profiling accuracy by validating and/or expanding the dynamically profiled system call whitelists.

System Call Control Granularity. We adopt a container-dependent whitelist model to reduce unnecessary system calls, which is more efficient for deployment with minimal overhead as compared to sophisticated and costly Finite State Automata (FSA) or Pushdown Automata (PDA) models for system call intrusion detection system [64], [65], [66]. As a stateless model [63], our approach also suffers from the risk of system call misuse attacks, such as mimicry attack [67], which transform a malicious attack sequence into a seemingly valid sequence by inserting *no-op* system calls. However, previous work by Kruegel et al. [68] and Zeng et al. [63] illustrate that a stateful FSA or PDA model is actually not much better than the stateless whitelist model, as a mimicry attack can be easily automated to evade such models. To achieve more fine-grained protection of the system calls, we could integrate the system call interposition based approach such as MBox [55] or argument constraint solutions [50], [51], [52]. We leave it in our future work.

Phase Separation Accuracy. We provide multiple methods for phase separation. Among them, methods to identify the start of the running phase, including fine-grained behavior-based and coarse-grained timing-based ones, may cause some extra system calls invoked in the service idle state and be added to the whitelist of the booting phase. Although the booting phase is short and the chances of launching attacks during it are very low, we can adopt a binary-based execution partition scheme [69] to achieve more accurate results if necessary. This approach is based on the observation that most long-running applications include an initialization phase followed by certain event-handling loops for processing inputs/requests. Thus, by locating those loops, we can separate the booting phase from the running phase. We leave this as one of our future works.

Calling Graph Accuracy. We currently do not consider functions invoked through function pointers. Our approach is based on identifying invoked libc functions directly through the binary code or indirectly through invoked functions from other libraries. In general, function pointers are commonly used to call functions at runtime, which makes it extremely difficult to identify using static analysis techniques. Nevertheless, if a function explicitly calls another function using a pointer, the callee function can be identified by the disassembler (e.g., objdump). Given that, we can construct the calling graph to identify "indirect function calls", as described in our approach. Indeed, our analysis demonstrates the effectiveness of our approach towards constructing the mapping between the invoked libc functions and the underlying system calls. Thus, guiding our static profiling approach and resulting in identifying the required system calls by each container. Finally, while analyzing function pointers is a challenging task, it often requires analyzing the source code to see which functions are assigned to pointers to begin with. We will consider such analysis in the future.

VII. Related Work

Linux Kernel Security Primitives. The Linux kernel provides other primitives that can be utilized to enhance container security. Linux cgroups (control groups) can be used for fine-grained limitation and prioritization of resources. By setting up a quota of the maximum resources available to a container, cgroups can be utilized to mitigate the resource exhaustion attacks initiated from inside a compromised container. Mandatory Access Control (MAC) refers to the access control enforced at the kernel level based on a predefined set of rules. By default, Docker on Ubuntu uses Apparmor, and Docker on Redhat uses SELinux. Discretionary access control (DAC) mechanisms can be used to protect kernel resources from malicious containers. DAC mainly involves capabilities and file mode access control. In the Docker container, only 14 out of 38 capabilities are allowed by default [59]. Contrary to MAC, the access in DAC is determined by the owner of the object or resource in question.

Container Security. Reshetova et al. [70] give a comparative study of multiple OS-level virtualization systems and identify the gaps in current security solutions based on a spectrum of attack models. Bui et al. [71] analyze the security level of Docker containers and how Docker interacts with the security features of the Linux kernel, while focusing on how container isolation is achieved through namespaces and cgroups. An extension to the Dockerfile is proposed [72] to ship a specific SELinux policy for processes running in a Docker image, which incurs a great burden for container image maintainers to build up a dedicated policy module. In contrast, our method is more practical, non-intrusive, and systematic to enhance the security of application containers. Mattetti et al. [73] propose an approach that combines customized AppArmor/SELinux rules based on

container operation tracing with host-based intrusion detection. Ours, on the other hand, focuses on proactively eliminating unnecessary system calls to reduce the potential vulnerabilities exploitable by malicious containers.

Application Sandboxing. System call interposition based application sandboxing [54] regulates and monitors application behavior by intercepting each system call according to a predefined policy profile. Our system traces the execution of the running application container to create a tailored seccomp policy instead of intercepting every system call. MBOX [55] is a lightweight sandboxing mechanism that interposes on a sandboxed program's system calls to layer a sandbox file system on the host file system. Similarly, we also utilize seccomp and bpf to interpose system calls invoked by processes in the containers. The Capsicum sandboxing framework [56] isolates processes from global kernel resources by disabling system calls that address resources via global namespaces. Their approach is different from ours, as we utilize the seccomp mechanism. Systrace [57] is a solution that confines multiple applications to running according to accurate policies. We have adopted its tracing capability to generate audit logs for the execution of container processes. Moreover, some other system call monitoring based anomaly detection and prevention solutions have been proposed [64], [65], [66], [74], which can be leveraged to further enhance container security.

Attack Surface Reduction. Seccomp [9] allows a process to restrict the set of system calls it can execute. Although our system is built on seccomp, we aim to adjust the set of available system calls for the entire application container instead of letting the process determine the policy itself. In addition, we split the container execution into three different phases and dynamically change the seccomp filters from outside the container to further reduce the attack surface. Cimplifier [75] and Docker-slim [76] use dynamic or static analysis to identify a minimal set of resources for running a specific application, thereby greatly reducing the size of application container images. However, our approach can dynamically adjust the available system call list during various phases of container booting, running, and shutdown. A different approach for system hardening is trimming [77], [78], which effectively reduces the attack surface by removing or preventing the execution of unused kernel code sections. Specifically, they are able to remove unnecessary features through automated compile-time kernel configuration tailoring. This approach can serve as a complement to our system to guarantee the general security of the host.

Furthermore, some works focus on constraining the available system calls through static analysis. Confine [11] launches a container, monitors the running binaries in this container within a configurable time period, and then analyzes the required system calls for all these binaries statically. Ghavamnia et al. [13] proposed a work that limits the set of system calls available to an application depending on its phase of execution (i.e., initialization and serving phases). To minimize the set of system calls, the approach relies on sophisticated points-to analysis to generate a call graph of reachable functions and system calls, however, the expensive points-to analysis is quite time-consuming and the manual transition point identification limits its scalability. Sysfilter [79] uses the Egalito framework to statically extract the system calls from the binary, and it relies heavily on the position-independent code and the presence of *stack unwinding information* (`.eh_frame` section) in the target applications. However, some compiled binaries are not position-independent, such as the git server, which impacts the scalability of this method. Chestnut [80], on the other hand, relies on angr to statically create a CFG for binary analysis; however, when the calling relation is very complex, angr may introduce path explosion, causing the analyzing workflow to be hampered. What's more, when backtracking the syscall invocation in the binary, Chestnut ignores some complicated cases, for example, the `xor` operation to set `eax` register to `0`. These works, however, do not apply to hierarchical structures, such as container images, and were unable to obtain the system calls associated with the running environment. These static analysis schemes could be used as a complementary method to help profile a more complete system call list.

VIII. Conclusion

In this paper, we design and develop a hybrid system call reduction mechanism that relies on static and dynamic profiling techniques to reduce the number of required system calls during various phases of the application container execution life-cycle. Our analysis results with 193 Docker containers from 5 different categories demonstrate a significant reduction in the number of required system calls for the execution of various containers. Indeed, such system call reduction can be useful towards software debloating and, thus, reducing the overall attack surface by removing vulnerable system calls. Additionally, we leverage empirical analysis to demonstrate the effectiveness of our approach in comparison to state-of-the-art system call reduction techniques. Finally, while we present a practical approach that can be automated to analyze various container images, such analysis can be done efficiently and with minimal overhead.

Acknowledgement

We would like to thank our anonymous reviewers for their valuable comments and suggestions.

References

[1] sysdig, "Sysdig 2021 container security and usage report," 2021. [Online]. Available: https://dig.sysdig.com/c/pf-2021-container-security-and-usage-report?x=u_WFRi
[2] CVE-2022–0185, 2022. [Online]. Available: https://www.crowdstrike.com/blog/cve-2022--0185-kubernetes-container-escape-using-linux-kernel-exploit/
[3] CVE-2021–31440, 2021. [Online]. Available: https://www.tigera.io/blog/cve-2021--31440-kubernetes-container-escape-using-ebpf/
[4] Overview of Linux namespaces, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/namespaces.7.html
[5] Cgroups - Linux container groups, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/cgroups.7.html
[6] Capability, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html
[7] AppArmor, 2023. [Online]. Available: https://ubuntu.com/server/docs/security-apparmor
[8] SELinux Project, 2023. [Online]. Available: https://github.com/SELinuxProject

[9] Seccomp, 2023. [Online]. Available: https://man7.org/linux/man-pages/man2/seccomp.2.html
[10] L. Lei et al., "SPEAKER: Split-phase execution of application containers," in *Proc. Int. Conf. Detection Intrusions Malware Vulnerability Assessment*, 2017, pp. 230–251.
[11] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *Proc. 23rd Int. Symp. Res. Attacks Intrusions Defenses*, 2020, pp. 443–458.
[12] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for Linux containers," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation*, 2017, pp. 92–102.
[13] S. Ghavamnia, T. Palit, S. Mishra, and M. Polychronakis, "Temporal system call specialization for attack surface reduction," in *Proc. 29th USENIX Secur. Symp.*, 2020, Art. no. 99.
[14] Most popular images on the Docker Hub. [Online]. Available: https://hub.docker.com/search?q=&type=image
[15] The GNU C Library. [Online]. Available: https://www.gnu.org/software/libc/
[16] D. J. Walsh, "Docker security in the future," [Online]. Available: https://opensource.com/business/15/3/docker-security-future
[17] Seccomp security profiles for Docker. [Online]. Available: https://docs.docker.com/engine/security/seccomp/
[18] Dockerfile. [Online]. Available: https://docs.docker.com/engine/reference/builder/
[19] Floyd–Warshall algorithm. [Online]. Available: https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm
[20] Egypt. [Online]. Available: https://www.gson.org/egypt/egypt.html
[21] MySQL 5.7 reference manual. [Online]. Available: http://dev.mysql.com/doc/refman/5.7/en/tutorial.html
[22] PostgreSQL 9.5.3. [Online]. Available: http://www.postgresql.org/docs/current/static/sql-commands.html
[23] MongoDB manual reference. [Online]. Available: https://docs.mongodb.com/manual/reference/command/
[24] Redis commands reference. [Online]. Available: http://redis.io/commands
[25] Wiki.js 2.0. [Online]. Available: https://docs.requarks.io/install/docker
[26] DokuWiki. [Online]. Available: https://github.com/crazy-max/docker-dokuwiki
[27] XWiki 11.10.12. [Online]. Available: https://hub.docker.com/_/xwiki
[28] MediaWiki 1.35.0. [Online]. Available: https://hub.docker.com/_/mediawiki
[29] HammerDB. [Online]. Available: https://www.hammerdb.com
[30] YCSB. [Online]. Available: https://github.com/brianfrankcooper/YCSB
[31] SQLmap. [Online]. Available: https://github.com/sqlmapproject/sqlmap
[32] F. Alexander, "NoSQL Exploitation Framework," [Online]. Available: https://github.com/torque59/Nosql-Exploitation-Framework
[33] NoSQLMap. [Online]. Available: https://github.com/codingo/NoSQLMap
[34] Bitnami/phpBB. [Online]. Available: https://hub.docker.com/r/bitnami/phpbb/
[35] Httperf. [Online]. Available: https://github.com/httperf/httperf
[36] Skipfish. [Online]. Available: https://linux.die.net/man/1/skipfish
[37] System auditing. [Online]. Available: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/security_guide/chap-system_auditing
[38] An introduction to KProbes. [Online]. Available: https://lwn.net/Articles/132196/
[39] P. Panchamukhi, "Kernel debugging with Kprobes," [Online]. Available: https://www.ibm.com/developerworks/library/l-kprobes/index.html
[40] libseccomp. [Online]. Available: https://github.com/seccomp/libseccomp
[41] Fuzzing. [Online]. Available: https://en.wikipedia.org/wiki/Fuzzing
[42] Confine source code. [Online]. Available: https://github.com/shamedgh/confine
[43] Common vulnerabilities and exposure. [Online]. Available: https://cve.mitre.org/
[44] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Enhancements to the Linux kernel for blocking buffer overflow based attacks," in *Proc. 4th Annu. Linux Showcase Conf.*, 2000.
[45] L. Martignoni, M. Christodorescu, and S. Jha, "OmniUnpack: Fast, generic, and safe unpacking of malware," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf.*, 2007, pp. 431–441.
[46] A. Quach, A. Prakash, and L. Yan, "Debloating software through {Piece-Wise} compilation and loading," in *Proc. 27th USENIX Secur. Symp.*, 2018, pp. 869–886.
[47] I. Agadakos, D. Jin, D. Williams-King, V. P. Kemerlis, and G. Portokalidis, "Nibbler: Debloating binary shared libraries," in *Proc. 35th Annu. Comput. Secur. Appl. Conf.*, 2019, pp. 70–83.
[48] B. A. Azad, P. Laperdrix, and N. Nikiforakis, "Less is more: Quantifying the security benefits of debloating web applications," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1697–1714.
[49] C. Qian, H. Hu, M. Alharthi, P. H. Chung, T. Kim, and W. Lee, "RAZOR: A framework for post-deployment software debloating," in *Proc. 28th USENIX Secur. Symp.*, 2019, pp. 1733–1750.
[50] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *Proc. Eur. Symp. Res. Comput. Secur.*, Springer, 2003, pp. 326–343.
[51] F. Maggi, M. Matteucci, and S. Zanero, "Detecting intrusions through system call sequence and argument analysis," *IEEE Trans. Dependable Secure Comput.*, vol. 7, no. 4, pp. 381–395, Fourth Quarter 2010.
[52] J. Jachner and V. K. Agarwal, "Data flow anomaly detection," *IEEE Trans. Softw. Eng.*, vol. SE-10, no. 4, pp. 432–437, Jul. 1984.
[53] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2003.
[54] T. Garfinkel et al., "Ostia: A delegating architecture for secure system call interposition," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2004.
[55] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Proc. USENIX Annu. Tech. Conf.*, 2013, pp. 139–144.
[56] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *Proc. USENIX Secur. Symp.*, 2010.
[57] N. Provos, "Improving host security with system call policies," in *Proc. USENIX Secur. Symp.*, 2003, pp. 257–272.
[58] CVE-2016–9793, 2016. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2016--9793
[59] S. E. Hallyn and A. G. Morgan, "Linux capabilities: Making them work," in *Proc. Linux Symp.*, 2008, p. 163.
[60] Amazon EC2 Container Service, 2023. [Online]. Available: https://aws.amazon.com/ecs/
[61] Docker Datacenter, 2023. [Online]. Available: https://www.docker.com/products/docker-datacenter
[62] Red Hat OpenShift Container Platform, 2023. [Online]. Available: https://www.openshift.com/enterprise/trial.html
[63] Q. Zeng, Z. Xin, D. Wu, P. Liu, and B. Mao, "Tailored application-specific system call tables," Pennsylvania State University, Tech. Rep., 2014.
[64] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proc. IEEE Symp. Secur. Privacy*, 2000, pp. 156–168.
[65] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proc. IEEE Symp. Secur. Privacy*, 2001, pp. 144–155.
[66] J. T. Giffin, S. Jha, and B. P. Miller, "Detecting manipulated remote call streams," in *Proc. USENIX Secur. Symp.*, 2002, pp. 61–79.
[67] D. Wagner and P. Soto, "Mimicry attacks on host-based intrusion detection systems," in *Proc. 9th ACM Conf. Comput. Commun. Secur.*, 2002, pp. 255–264.
[68] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna, "Automating mimicry attacks using static binary analysis," in *Proc. USENIX Secur. Symp.*, 2005, pp. 11–11.
[69] K. H. Lee, X. Zhang, and D. Xu, "High accuracy attack provenance via binary-based execution partition," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2013.
[70] E. Reshetova, J. Karhunen, T. Nyman, and N. Asokan, "Security of OS-level virtualization technologies," in *Proc. Nordic Conf. Secure IT Syst.*, Springer, 2014, pp. 77–93.
[71] T. Bui, "Analysis of docker security," 2015, *arXiv:1501.02967*.
[72] E. Bacis, S. Mutti, S. Capelli, and S. Paraboschi, "DockerPolicyModules: Mandatory access control for docker containers," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2015, pp. 749–750.
[73] M. Mattetti, A. Shulman-Peleg, Y. Allouche, A. Corradi, S. Dolev, and L. Foschini, "Securing the infrastructure and the workloads of Linux containers," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2015, pp. 559–567.
[74] C. Linn, M. Rajagopalan, S. Baker, C. S. Collberg, S. K. Debray, and J. H. Hartman, "Protecting against unexpected system calls," in *Proc. USENIX Secur. Symp.*, 2005, pp. 239–254.
[75] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Towards least privilege containers with cimplifier," 2016, *arXiv:1602.08410*.
[76] K. Quest, "Docker-Slim: Lean and mean docker containers," 2018. [Online]. Available: https://github.com/slimtoolkit/slim
[77] A. Kurmus, A. Sorniotti, and R. Kapitza, "Attack surface reduction for commodity OS kernels: Trimmed garden plants may attract less bugs," in *Proc. 4th Eur. Workshop Syst. Secur.*, 2011, Art. no. 6.
[78] A. Kurmus et al., "Attack surface metrics and automated compile-time OS kernel tailoring," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2013.

[79] N. DeMarinis, K. Williams-King, D. Jin, R. Fonseca, and V. P. Kemerlis, "sysfilter: Automated system call filtering for commodity software," in *Proc. 23rd Int. Symp. Res. Attacks Intrusions Defenses*, 2020, pp. 459–474.
[80] C. Canella, M. Werner, D. Gruss, and M. Schwarz, "Automating seccomp filter generation for Linux applications," in *Proc. Cloud Comput. Secur. Workshop*, 2021, pp. 139–151.

Yunlong Xing received the BSc degree in information security and the MSc degree in cyberspace security from Wuhan University, Wuhan, China, in 2018 and 2021. He is currently working toward the PhD degree with the Department of Information Sciences and Technology, George Mason University, supervised by Dr. Kun Sun. His research interests mainly lie in container security, system security, and software security.

Xinda Wang received the BEng degree in information security from the Harbin Institute of Technology, Harbin, China, in 2017. She is currently working toward the PhD degree with the Department of Information Sciences and Technology, George Mason University, supervised by Dr. Kun Sun. Her research interests mainly lie in software security, including vulnerability detection, software patch analysis, and container security.

Sadegh Torabi received the MSc degree from the University of British Columbia, in 2016, and the PhD degree from Concordia University. He is currently an assistant professor with George Mason University and a fellow researcher with the Center for Secure Information Systems. His research interests include Internet measurements, network/systems security, and operational cyber security of IoT and Cyber-Physical Systems.

Zeyu Zhang received the BEng degree in computer science from the Beijing University of Posts and Telecommunications, in 2018, and the MEng degree in computer science and technology from Tsinghua University, in 2021, supervised by Dr. Qi Li. He was a visiting scholar with the CSIS, George Mason University, from 2019 to 2020, advised by Dr. Kun Sun. His research interests lie in system security and trusted computing.

Lingguang Lei received the PhD degree from the University of Chinese Academy of Sciences, in 2013. She is an associate research fellow with the Institute of Information Engineering, CAS. She worked as a visiting scholar with the College of William and Mary (2015-2016) and George Mason University (2016-2017). Her research focuses on information security, including mobile security, container security, network security, and information system evaluation.

Kun Sun received the PhD degree from the Department of Computer Science, North Carolina State University, in 2006. Now, he is a full professor with the Department of Information Sciences and Technology, George Mason University. He is also the director of Sun Security Laboratory. Before joining GMU, he was an assistant professor with the College of William and Mary. He has more than 15 years working experience in both industry and academia, publishing more than 80 conference and journal papers. His current research focuses on trustworthy computing environment, moving target defense, software security, password management, and software defined networking.