Evaluating Container Debloaters

Muhammad Hassan*[1], Talha Tahir*[1], Muhammad Farrukh[1], Abdullah Naveed[1], Anas Naeem[1], Fareed Zaffar[1], Fahad Shaon[2], Ashish Gehani[3], Sazzadur Rahaman[4]
[1] Lahore University of Management Sciences, [2] Google, [3] SRI International, [4] University of Arizona
{23100199, 23100293, 23100240, 23100239, 23100290, fareed.zaffar}@lums.edu.pk,
fshaon@google.com, ashish.gehani@sri.com, sazz@cs.arizona.edu

***Abstract*—DOCKER containers have been widely used by organizations because they are lightweight and single hardware can run multiple instances of a container. However, this ease of virtualization comes with weaker isolation as compared to virtual machines. A compromised container can allow the attacker to escape to the host and gain privileged access. Several approaches have been developed to reduce the attack surface of containers either through the reduction of system calls or through slimming container images. Unfortunately, measuring the performance of container debloaters is challenging as there exists no platform for this purpose. This paper aims to address this gap, by building a unified platform to benchmark them.**

Currently, our benchmark includes 7 workload applications, and 3 container debloaters, i.e., SPEAKER, CONFINE (syscalls reduction tools), and SLIMTOOLKIT (image size reduction tool). We added several evaluation metrics in the framework, which include category-based system call reduction, CVEs mitigated, size reduction, and execution correctness.

Our evaluation revealed interesting insights into the existing techniques. Both the system call reduction tools were able to produce correct debloated containers as compared to SLIMTOOLKIT (tool to reduce image size) which worked well too by reducing almost 79 percent of the size of the image but it failed to produce correct results on 2 out of 7 applications.

I. INTRODUCTION

DOCKER containers are an essential part of the rapid growth of cloud computing because they allow the developers to easily build distributed applications and manage them through orchestrators, such as KUBERNETES [1]. Containers are typically built in the form of layers where every layer contains some dependency of the software that needs to be run in the container. However, it may contain several functionalities and files that are not needed to run a given application [2]. These irrelevant components not only cause performance issues but also become a security risk [3], [4], [5].

There has been extensive research to automatically debloat containers [6], [7], [8], [9], [10]. The main goal of all the automated container debloaters is to remove as much unnecessary code as possible while attempting to preserve the intended functionality. These tools adopt various methods to achieve the goal, such as the use of static, dynamic, or hybrid analysis to restrict unnecessary system calls or to remove unused layers of dependencies from the container images. To guide future research in this area, it is critical to evaluate and compare the performance of these container debloaters from different analysis paradigms. However, the diversity in their methodology and functioning makes setting these tools for unified analysis challenging. For example, container debloaters like CONFINE [7] and SPEAKER [8] produce a black list and white list of system calls respectively. This suggests that a unified benchmark platform that can handle such diversities is necessary. Due to the lack of such a unified platform, efforts to compare container debloaters extensively, are nonexistent. This paper takes the first step toward addressing this gap. We propose, DEBLOATBENCH$_C$, a unified framework to evaluate a diverse set of container debloaters that are capable of handling the diversity of design and execution environments for container debloating. Our contributions can be summarized as follows:

- We develop a new easy-to-extend benchmarking framework named DEBLOATBENCH$_C$ to evaluate container debloating techniques. In the current version, we integrated three different tools (i.e., CONFINE [7], SPEAKER [8] and SLIMTOOLKIT [6] previously known as "DockerSlim").
- We perform a holistic comparative analysis of three container debloaters under various metrics. Our evaluation shows that all debloated containers produced by SPEAKER and CONFINE were correct. SPEAKER Significantly outperformed others in system call reduction. SLIMTOOLKIT was able to reduce almost 79 percent of the sizes of DOCKER images, however, falls short of retaining correctness for 2 out 7 workload applications from our benchmark.

II. CONTAINER DEBLOATING METHODS

A range of debloating research aims to increase the performance and security of applications. This is undertaken at different abstraction levels, such as removing unused shared libraries [11], [12], customizing the operating system kernel [13], [14], or limiting the set of system calls supported [6], [15], [16], [17]. Although the techniques differ, they use similar high-level approaches at a fundamental level – that is, variants of static, dynamic, or hybrid analysis. DEBLOATBENCH$_C$ aims to unravel the strengths and weaknesses of each method as it is used in a particular context.

A. Static Analysis

Static analysis techniques analyze code without executing it. Unlike dynamic analysis, this class of techniques is generally more conservative. In most cases, it provides complete coverage of all possible control flow paths. However, static

*These authors contributed equally to this work

analysis is dependent on the language of the target program as it uses the implementation of the target program to attain visibility into it.

CONFINE [7] is a container debloater that uses static analysis to block excess system calls through `Seccomp` profiles. *`Seccomp` [18] is a feature provided within the Linux kernel that can be utilized to limit the available system calls from a given container.* CONFINE utilizes `Seccomp` to reduce the attack surface of the Linux kernel by exposing only required system calls to the containerized applications. It accepts as input a container application image and generates a customized, restrictive `Seccomp` profile. During the initialization phase, CONFINE runs the container to identify the sets of binary executables within the container. For example, when we run CONFINE on MONGO, it will record every executable launched within the configurable time period (default value is 30 seconds). Some of those binaries are **numactl**, **grep**, **id**, **libs.out** and **libc.so.6**. Next, it statically derives the system calls used by these programs. The main insight is, that user programs usually interact with the operating system using system calls facilitated by libraries like **glibc** or **musl-libc**. CONFINE extracts the call graph from these libraries to identify system calls.

Although during the initialization phase, CONFINE runs the container to find executables running within it, we mark CONFINE as a static analysis tool, since, the core engine is built with static analysis. It is worth noting that CONFINE also allows manual configurations which can be used to indicate programs executing inside a container. Should this be the case, the dynamic analysis part is skipped.

B. Dynamic Analysis

The other broad class of approaches used for debloating incorporates dynamic analysis. The properties of an application are captured by executing the program and monitoring it. The advantage of this class is that the source code of an application is not needed to conduct the analysis. Instead, the application's behavior is monitored while running it. The original program source (from which the executable is derived) does not need to be inspected. A major drawback of such techniques is that programs must be iteratively invoked with all possible combinations of arguments and inputs to obtain comprehensive coverage.

To better understand this approach, we studied SPEAKER, which uses dynamic analysis to obtain the list of all the system calls used by the container application. It then blocks the remaining system calls (deeming them unnecessary) when the DOCKER container for the application is run. The approach divides the container life cycle into two phases – specifically, the booting and running phases. The booting phase is responsible for setting the container environment and initializing the service within. It is expected to be completed in minutes. In the longer-running phase, the container service begins to accept requests and sends back responses. Dissimilar sets of system calls are invoked in the two stages. The difference is due to the divergence in the type of activity performed in each. SPEAKER reduces the attack surface of a container by first distilling sets of calls for each stage. Then the set of calls permitted is restricted to those needed for the corresponding execution phase. This is done by recording all the system calls invoked during the setup phase and then creating a new kernel module which gets uploaded to the file system during a particular phase. If the workload fails to invoke a system call associated with a given feature during the setup phase, during the operation phase, any test case requiring that system call will fail. This means the tool is as reliable as the tracer it uses in the setup phase.

C. Hybrid Analysis

The third debloating approach we studied employs the hybrid strategy of using both static and dynamic analysis. This is a powerful technique for program analysis. It can be employed for removing unused code from software. The strategy combines the strengths of both static and dynamic analysis to achieve better results than either technique alone. By combining approaches, a hybrid strategy can detect dead code that static analysis may not be able to reason about (such as loops) and that dynamic analysis may not otherwise reach (if the path condition is rare). A disadvantage of the hybrid approach is that it can be computationally expensive. It may therefore require significant resources to utilize adequately.

SLIMTOOLKIT [6] employs a hybrid of analysis techniques on DOCKER images. It collapses layers to generate a smaller container, aiming to enhance its security by eliminating irrelevant files to reduce the attack surface. To run SLIMTOOLKIT, a DOCKER image is required as input. The output is a debloated DOCKER image – that is, a "slimed" version of the input. SLIMTOOLKIT employs two methods of training. The default method is HTTP probing, where SLIMTOOLKIT pings all exposed HTTP ports to trace program behavior. HTTP probing doesn't require any manual intervention and is entirely automated. However, for applications without any exposed ports, it requires providing training cases manually. SLIMTOOLKIT would execute these training cases and then analyze the behavior of the applications.

The primary goal of SLIMTOOLKIT is to optimize DOCKER images by reducing their deployed size. DOCKER images may be large if they contain multiple layers with redundant or unnecessary data. This leads to longer download and deployment times, as well as higher storage requirements. To address this issue, SLIMTOOLKIT was developed to inspect the metadata and data of containers. This static analysis is used to build an artifact graph of the necessary components. The graph is then used to determine which files, libraries, executables, and dependencies are required for the container to function correctly. The resulting image includes these components, while redundant or unnecessary data is removed. The tool provides support for the reduction of attack surfaces by leveraging two pre-existing systems – `Seccomp` and `AppArmor`. Like, `Seccomp` [18], `AppArmor` [19] is also a Linux kernel module. It provides name-based mandatory access controls

to restrict individual programs to a set of listed files and permissions.

III. Research Questions

A. Security Hardening

The main goal for container debloaters is to reduce attack surfaces at the container level [15], [7]. Given an application deployment context, container debloaters *i)* block system calls, *ii)* reduce system image size, or *iii)* both. As a side effect – it might also enhance system security by mitigating CVEs. To measure the effectiveness of container debloaters in terms of security hardening, we pose the following research questions. **RQ1 (Section IV-A):** *How effective are container debloaters at reducing the number of system calls? How do they perform in reducing different classes of system calls?* **RQ2 (Section IV-B):** *How do they perform in terms of avoiding the number of CVEs?*

B. Correctness and Reduction

A debloating method is useful only if the debloated images are correct – means they retain all of their intended functionality. However, retaining correct behavior in the debloated version is a non-trivial goal for automated debloaters [20]. For example, container debloaters achieve size reduction by removing certain system files or imposing restrictions on specific system calls. As a side effect, it may also result in limited functionality. Thus, evaluating correctness becomes crucial to determine if the reduction in system calls or size comes at the expense of critical functionality, which motivated us to study the following research questions. **RQ3 (Section IV-C):** *How do the container debloaters impact the correctness of target containerized applications?* **RQ4 (Section IV-D):** *How do they perform in terms of reducing container image size?*

IV. Measurement Methodology

In this section, we define our measurement methodology and metrics to answer the research questions posed in Section III.

A. RQ1: System Call Reduction

A system call typically serves a specific purpose, however, some of the system calls have overlapping functionalities. For example, any of the following system calls can be used to get file status: `int` **`stat`**`(const char *file_name, struct stat *buf), int` **`fstat`**`(int filedes, struct stat *buf), int` **`lstat`**`(const char *file_name, struct stat *buf).` Similarly, any of the three following syscalls can be used for receiving messages from a socket: `int` **`recv`**`(int s, void *buf, int len, unsigned int flags), int` **`recvfrom`**`(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen), int` **`recvmsg`**`(int s, struct msghdr *msg, unsigned int flags).` Removing a subset of system calls from such groups will not effectively block the corresponding functionality. Thus, we evaluate the system call reduction in terms of system call classes.

By following the method discussed in [21], we categorize 335 system calls into eight classes: File Access (total 66 system calls), File Descriptor (47), Time Related (23), Process Control (80), Shared Memory (7), Message Queues (10), Network (21), and System-Wide (81). We determine the percentage reduction in system call categories by comparing the total number of system calls allowed by the system before and after the debloating methods were applied. This analysis sheds light on how well the tools work to minimize redundant system calls, thereby improving system performance and security.

B. RQ2: CVEs Mitigated

In addition to the reduction in system calls, we also measure the number of Common Vulnerabilities and Exposures (CVEs) that can be avoided after container debloating. The main insight is if a function containing a CVE x, can only be accessed via a system call y, then we can say that x can be blocked by blocking the system call y. If a debloating method blocks the execution of a system call, then all the CVEs associated with the system call are essentially inaccessible by the adversary. Thus, by associating the reduced system calls with corresponding CVEs, we quantify the reduction in CVEs achieved by each of the container debloaters for different workload applications. To associate CVEs with system calls, we use the method developed in CONFINE [7]. We first crawl the CVE website [22] and create a mapping of each CVE to its respective functions. To create this mapping, we analyze the Git history to find patches related to specific CVEs. After that, we identify the modified file and function in each patch. We exclude CVEs for which the above method fails to a corresponding function. After that, we build the Linux kernel call graph using KIRIN [23] and analyze the parts that are exclusively accessed by a given system call.

We use `Seccomp` profiles to find out which system calls are blocked. There are two types of `Seccomp` profiles `SCMP_ACT_ALLOW`, `SCMP_ACT_ERRNO` and as the name suggests, the first one gives allowed system calls list and the later one gives blocked system calls list. To ensure consistency, we added a plugin to convert `SCMP_ACT_ALLOW` to `SCMP_ACT_ERRNO`, so that the rest of the part of the analysis does not get affected by it.

C. RQ3: Tool Correctness

Our evaluation focuses on assessing whether a workload application can successfully perform its intended functions after debloating the container it runs on. We used two different test suites to check correctness. Firstly, we created a test suite to systematically test the *functional correctness* of all the retained features. These test cases (84 in total) enabled us to determine whether the debloating process preserves the intended functionality or if it inadvertently crashes the application. Secondly, to evaluate the *performance correctness* of a debloated workload application, we utilize existing benchmarks from the application domain. We define performance correctness as the changes in the overhead after debloating. For example, to evaluate the performance of database applications

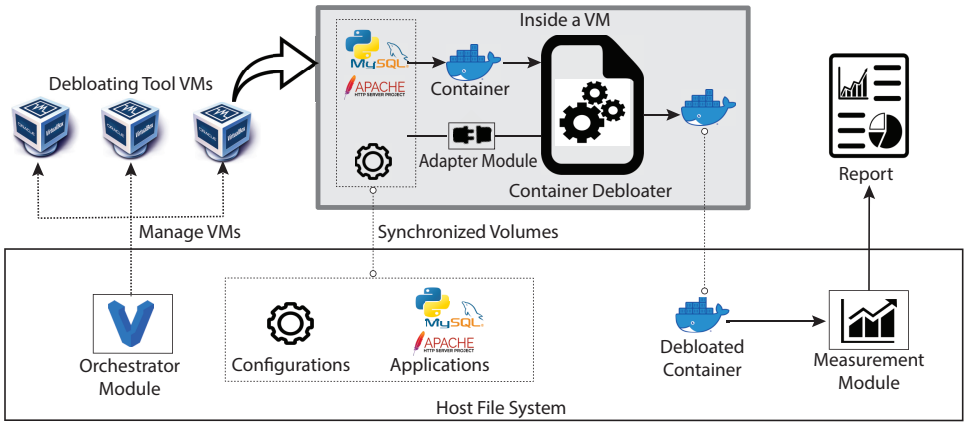


Fig. 1. Overview of our DEBLOATBENCH$_C$ framework.

(i.e., MYSQL, REDIS, MONGO), we leveraged the TPC [24], Redis-Benchmark [25], and Mongo-perf [26], respectively, for network applications, we used ApacheBench [27], for PYTHON, we used *pytest* benchmark [28] and for NODEJS, we used ApacheBench [27] and *Benchmark.JS* [29]. This is because these benchmarks provide standardized and widely recognized test cases for measuring performance, ensuring a comprehensive evaluation of the debloated images.

D. RQ4: Size Reduction

Size reduction is an important metric in the context of DOCKER images, as smaller images consume less space and have faster deployment times. Additionally, smaller images contribute to the overall efficiency of the system. To calculate the percentage reduction, we compare the original image size available on DOCKERHUB with the size of the reduced image created by the tools. However, we acknowledge that not all container debloating tools necessarily reduce the size of containers, and as a result, some tools may not meet this metric. Nevertheless, this metric provides users with insights into whether a tool will offer any size reduction, enabling them to choose the most suitable tool for their specific use cases.

V. MEASUREMENT FRAMEWORK

In this work, our main objective is to systematically analyze and evaluate docker container debloating tools. We aim to generalize the debloating process for different tools as follows - 1) pull the container image, 2) configure the debloating tools, 3) run the tool, and 4) measure performance. For a specific container image, steps 2 and 3 differ for different tools. In contrast, we also aim to retain the same measurement interface for all the tools to uniformly compare them. In addition, our goal is to provide isolation among different tools to avoid one tool interfering with another.

Ali *et al.* [30] designed an extensible benchmarking framework to evaluate application debloaters. Similarly, we also follow the *SOLID principles* to design our framework [31]. Specifically, we prioritize, *i)* the *Single-responsibility principle*, where each component or module only implements a single functionality; *ii) Open–closed principle*, where anyone can add a custom implementation to a defined specification; and *iii)* the *Interface segregation principle*, where core algorithms can utilize specific implementation without a side effect. The overall design of the framework is shown in Figure 1.

We split the DEBLOATBENCH$_C$ (as shown in Figure 1) into three core modules – *1)* the *orchestrator* module that creates isolated environments for the tools and bootstraps these environments with proper packages and configurations; *2)* the *adapter* module that encapsulates the low-level details of a debloating tool execution to generate debloated containers; and *3)* the *measurement* module that verifies and collect measurement results from the output containers.

In our implementation, we utilized virtual machine-based isolation. Some debloating tools require updating the operating system kernel, e.g., loading new kernel modules or creating custom kernels, which is easier in a VM than in a container. We use VAGRANT [32] for creating isolated VMs for the tools. VAGRANT also offers a declarative language to define and automate the lifecycle management of a given VM.

Workflow: In a typical workflow user executes the orchestrator module to start a debloating execution with a debloating job description, which contains the name of the container to debloat, the name of the target debloater(s), debloater-specific configurations, and the name of the validation and measurement module to use. Then in the orchestrator module, we initialize isolated VMs for all the target debloaters, provision the VMs with necessary packages for the debloater, and pull the containers from a container registry. Specifically, we used DOCKERHUB [33], since it is the most popular and default docker container registry. Also, this is configurable in our system so that users can utilize private registry and debloat private containers. Next, in the execution flow, we execute the adapter module to debloat the container and save the debloated containers. Then we run the verification and measurement modules for the program, which generates the report.

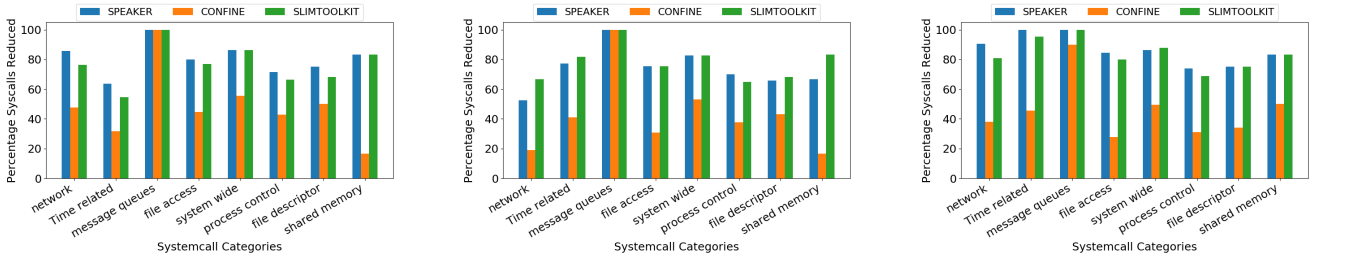
A. Orchestrator Module

The orchestrator module acts as the central coordinator and orchestrates the execution of various tasks within the

Applications	Selected Arguments/functionalities	Test Cases (84)	Benchmark
Mongo	show use drop insert complexDB complexQuery flushing	7	mongo-perf
MySQL	show create insert update delete join groupby drop	8	TPC-C
Redis	set get ping lpush rpush lpop rpop mset lrange100 flushDB hset hget mset mget ltrim scan hscan publish	25	Redis-bench
Nginx	-t -c -p -s	4	Apcahe-bench
Httpd	-v -h -t -k -V -X -L -l -D -M	17	Apache-bench
NodeJS	-c -p -e sync read async read write http	7	Benchmark.js
Python	-h -version -I -u args save -m -W -q output-redirection timeout	16	pytest

TABLE I

TARGET APPLICATIONS AND THE ARGUMENTS USED FOR EVALUATION. THE TEST CASES COLUMN INDICATES THE NUMBER OF TEST CASES WE DESIGNED TO TEST THE FUNCTIONAL CORRECTNESS. THE "BENCHMARK" COLUMN INDICATES THE BENCHMARKS USED TO MEASURE THE PERFORMANCE CORRECTNESS OF A GIVEN WORKLOAD APPLICATION.



(a) Network Applications (NGINX and HTTPD) (b) Database Applications (MYSQL, REDIS and MONGO) (c) Language Applications (PYTHON and NODEJS)

Fig. 2. System calls Restriction per category for different applications categories

framework. It also serves as the single user-facing entity in DEBLOATBENCH$_C$ for managing all the integrated container debloating tools. Some of these tools concentrate on reducing the size of the container image, while others try to reduce the allowed system calls. The orchestration module hides the underlying complexities of running these tools and produces appropriate outputs based on their features.

Here we also optimize based on debloater types. For example, we generate a list of permissible system calls for tools aimed at decreasing system calls, which ensures that only essential system calls are permitted. In contrast, for tools designed to shrink the size of the DOCKER image, we provide both the freshly generated, size-optimized image and a list of allowed system calls. This strategy allows us to handle diverse types of tools seamlessly.

B. Adapter Module

We implement a plugin-based adapter module to integrate with different debloating tools. For each tool, we implement a plugin containing each tool's low-level commands. These plugins also contain tool-specific optimizations. Currently, DEBLOATBENCH$_C$ incorporates three tools, namely SPEAKER, SLIMTOOLKIT, and CONFINE as their implementation artifact is open-sourced, enabling us to study. Here, CONFINE, SPEAKER and SLIMTOOLKIT represent static, dynamic, and hybrid analysis-based debloaters, respectively.

C. Measurement Module

In the measurement module, we parse and analyze the output of the preceding steps. We also standardize the data collection procedure for all the debloating tools. We evaluate the performance of the tools using four metrics: i) the reduction in the number of system calls, ii) the number of CVEs mitigated, iii) the reduction in container size, and iv) the correctness of the generated `seccomp` profiles and slim containers. It is worth noting that we used CVEs as a metric for security evaluation because they are typically closely tied to the programs' functionalities. Note that, we do not assign any weight to the system calls based on the number of invocations. Instead, we use the unique list of system calls, regardless of their frequency of invocation. We then utilize these blocked system calls to identify mitigated CVEs (as discussed in Section IV-B).

D. Target Program Suite

In order to produce results that can be compared meaningfully, we select 7 widely used applications, each with a diverse range of deployment contexts. We categorize these applications into three categories: *network*, *database*, and *language interpreter* applications. We select heavy-loaded server-side applications so that we can observe the server load reduction using the debloating techniques. It is a widespread and popular use of debloating. Following are the seven applications that are selected:

- MYSQL (Relational Database)
- REDIS (Key-value Database)
- MONGO (Database)
- NGINX (Network)
- HTTPD (Network)
- PYTHON (Language)

- NODEJS (Langauge)

Table I shows the set of all the arguments we used to check the functional correctness and the corresponding number of test cases (84 in total) we used (Details in Section IV-C). It also shows the benchmarks that we leveraged to measure the performance correctness.

E. Extending the Framework

1) Adding New Debloating Tool: One needs to implement a new plugin for the adapter module to add a new debloating tool to the framework. This contains various lifecycle-hook methods and classes.

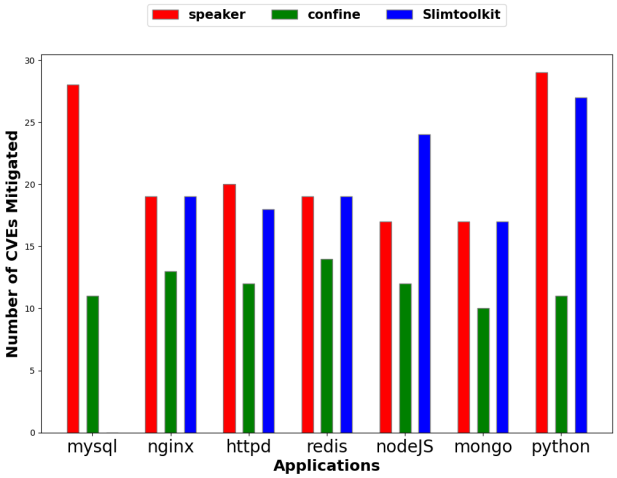
2) Supporting new target program: The framework needs program-specific verification and test cases in the measurement module to add a new target program. One can also utilize the program domain-specific benchmarking tools as part of the test cases. Next, we present our evaluation results of these container debloaters.

VI. EVALUATION RESULTS

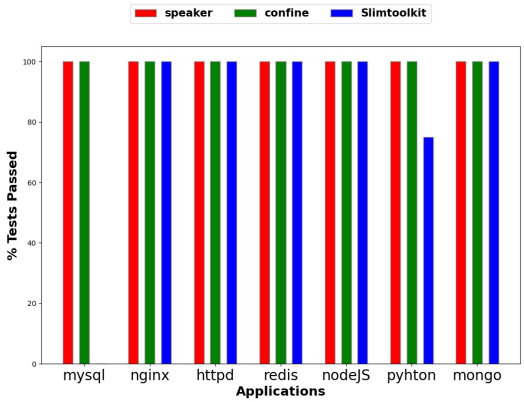
Here, we present our evaluation results of 3 container debloaters on the 7 workload application that we integrated into our DEBLOATBENCH$_C$ framework. SPEAKER and SLIMTOOLKIT run on kernel 4.15.0-99, which has 323 system calls, and CONFINE runs on kernel 5.15.0-83, which has 335 system calls (12 more). As updating kernels might incur non-trivial efforts with unintended consequences, we kept the version unchanged and reported the normalized results. All these debloater have dynamic components to set up (or train) the debloating process. CONFINE only required running the workload application thus do not require any test cases during the setup phase. For SLIMTOOLKIT, we utilized network probing for four applications and relied on train cases for the other three (Python, Node, and MySQL), as these applications are not network-facing. For these three applications, we created a set of 12 test cases to train SLIMTOOLKIT. For SPEAKER, we created a set of 50 test cases to run the workload applications during the setup phase. These test cases are designed to trigger various arguments of the workload applications.

A. RQ1: System Calls Reduction

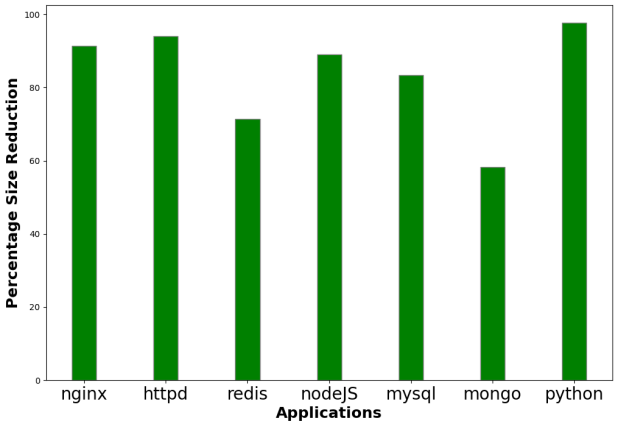
In Figure 2, we show how each debloater handles different system call categories across various application types. SPEAKER consistently performed better across all the system call and application type categories. SLIMTOOLKIT and SPEAKER demonstrate an aggressive approach in restricting network-related system calls for Language interpreters while adopting a more lenient approach for network and database applications. We also found that SLIMTOOLKIT, CONFINE, and SPEAKER effectively eliminate all Message Queues system calls. Furthermore, we observe that for at least one application (i.e., PYTHON, NGINX, and MYSQL) from each category Shared Memory system calls are completely eliminated. For the rest, only the **madvise** system call is retained, which serves to advise the kernel on memory usage. CONFINE *was consistently performing worse due to the conservative nature of the analysis.*



(a) CVEs removed by debloaters for each target application



(b) The average percentage of tests passed for each target application



(c) The reduction in image size by SLIMTOOLKIT (Other two tools do not support code reduction, thus size reduction does not apply for them)

Fig. 3. Comparative effectiveness of different tools on different applications.

B. RQ2: CVEs Mitigated

We show the results of our CVE analysis in Figure 3(a). It shows that both the SLIMTOOLKIT and SPEAKER have comparable results with just one exception: SLIMTOOLKIT failed to debloat the MYSQL image (thus the column is empty). Because of using dynamic analysis, SPEAKER consistently performed better. As by nature, CONFINE's static analysis is conservative in blocking system calls, it consistently performed worst in this metric.

C. RQ3: Tool Correctness

As discussed in Section IV-C, we leveraged a curated set of 84 test cases to measure the functional correctness of the applications after container debloating. In Figure 3(b), we see that the debloated containers generated by CONFINE and SPEAKER passed all the test cases and SLIMTOOLKIT failed correctness cases for MYSQL (all) and PYTHON (25%). SLIMTOOLKIT debloats images by deleting binaries and blocking system calls that were found irrelevant by its tracer. Inaccuracies in the tracer caused the deletion of necessary binaries in the debloated containers of MYSQL and PYTHON. *This unveils a major shortcoming in the test cases-based hybrid container debloating paradigms.* The performance correctness evaluation provides similar results. The containers debloated by CONFINE and SPEAKER pass all test cases of the aforementioned benchmarking tools without any performance overhead. SLIMTOOLKIT, however, fails for TPC-C. The debloated container for MYSQL generated by SLIMTOOLKIT crashes and as a result, fails on both functional and performance correctness. It is worth noting that PYTHON passes all the test cases from *pytest* benchmark with no performance penalty, however, in our functional correctness test cases, it fails for the `-u, -m and timeout` flags. The reason is *pytest* does not thoroughly check the flags and only uses small code snippets to test PYTHON to measure the performance and latency.

D. RQ4: Size Reduction

Among the tools examined, only SLIMTOOLKIT is capable of reducing the size of images. Figure 3(c) illustrates the size reduction of DOCKER images achieved by SLIMTOOLKIT. As evident from the graph, SLIMTOOLKIT consistently achieves a significant reduction in size, with an average reduction of 79%. This remarkable reduction is accomplished through the removal of binaries considered unnecessary by SLIMTOOLKIT.

E. Case Studies

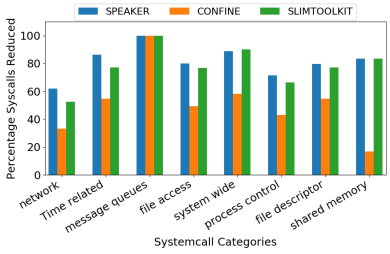
In this section, we provide an in-depth analysis of the container debloaters. We describe individual case studies with applications from 3 different categories (networks, language runtime, and database). To provide a meaningful comparison, we considered an application that worked for all tools from each category. For these case studies, we also manually analyzed the system calls that were removed or retained to generate new insights.

*1) **Network Application (`httpd`):*** Figure 4(a) shows that overall SPEAKER achieves better performance than the other tools, in terms of blocking system calls across all the categories. Figure 4(b) illustrates that 63 system calls are deemed necessary for the `httpd` application by all the container debloaters. Since both SLIMTOOLKIT and SPEAKER retain 100% correctness and share these 63 system calls – it indicates more opportunities for reduction. A close inspection shows that no Message Queue system calls are retained and only the `madvise` system call from the shared memory category remains. However, upon analyzing this system call, we see that it is merely an advisory call to the Linux system and could potentially be removed. Surprisingly, the tools unanimously fail to eliminate it, despite its non-essential nature.

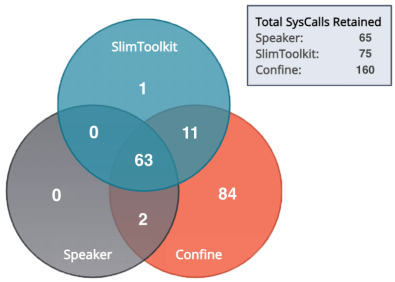
Notably, as outlined by Figure 4(c), out of the 66 file access system calls, only 13 common system calls are retained by the tools, and merely 9 out of 47 file descriptor system calls are preserved. Additionally, a majority of the removed system calls fall under the system-wide category, with only 8 calls being retained out of 81. Furthermore, the process-related system calls are heavily restricted, with a mere 20 out of 80 calls being retained. Considering that HTTPD is an Apache server primarily utilized for network applications, it is interesting to observe that out of the 21 network system calls, only 8 essential calls are retained. These crucial network system calls, namely `setsockopt`, `listen`, `socket`, `bind`, `getsockname`, `recvmsg`, `sendto`, and `connect`, play a vital role in the functioning of HTTPD – removing any of these system calls would likely result in HTTPD crashing or failing the correctness test cases.

Next, we explore the system calls that could potentially be removed but were retained. This analysis can provide some new insights into designing new debloaters. For SLIMTOOLKIT and SPEAKER, we note that they mitigate all the possible CVEs without compromising the functionality of the HTTPD. However, CONFINE leaves three distinct CVEs associated with three system calls: `semctl`, `shmctl`, and `clock_nanosleep`. We found that these system calls are from the 84 exclusive system calls retained by CONFINE. This means these system calls are not likely needed for correctness, and thus can potentially be eliminated.

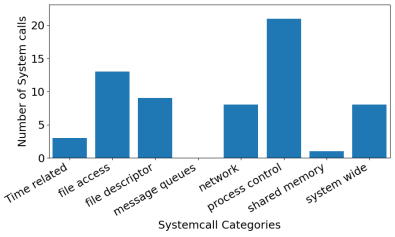
*2) **Database Application (`redis`):*** Now, let us delve into REDIS, which is a key-value database. Figure 5(a) shows SPEAKER outperforms the other tools for REDIS too. Our findings (Figure 5(b)) reveal that for REDIS, all the debloaters retain a set of 57 common system calls, while SPEAKER and SLIMTOOLKIT share another 4 in their retention sets. This says that at maximum 57 system calls are necessary for REDIS to function correctly – showing opportunities for more reduction. Upon further examination of the system calls, we discovered that similar to HTTPD, the Message Queue category is completely eliminated, and only the `madvise` system call from the shared memory category is retained. Furthermore, as outlined in Figure 5(c), out of the 66 file access system calls, only 11 common system calls are retained by the tools, and merely 4 out of 21 network system calls are preserved.



(a) System calls restricted per category.

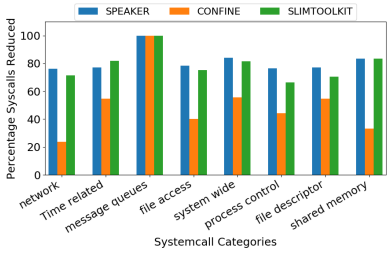


(b) System calls retained by each tool.

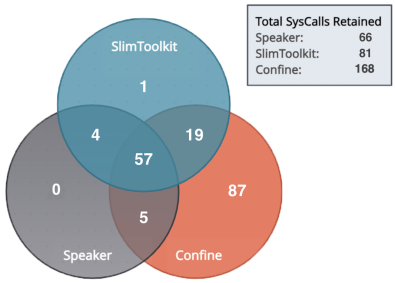


(c) Distribution of the common 63 system calls.

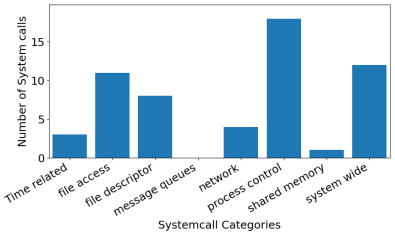
Fig. 4. Case Studies: Network Application (`httpd`)



(a) System calls restricted per category.



(b) System calls retained by each tool.



(c) Distribution of the common 57 system calls.

Fig. 5. Case Studies: Database Application (`redis`)

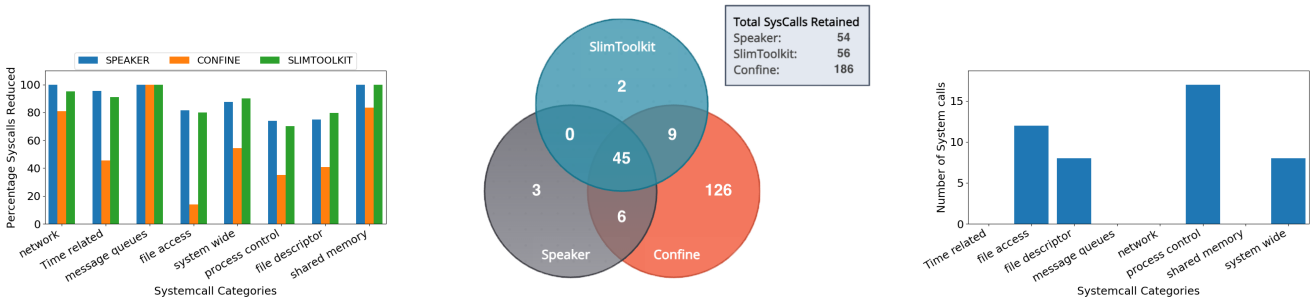
Additionally, a majority of the removed system calls fall under the system-wide category, similar to HTTPD, with only 12 calls being retained out of 81. Moreover, the process-related system calls are heavily restricted, with only 18 out of 80 calls being retained. Considering that REDIS is a data store, it primarily utilizes file descriptor system calls for reading and writing data. It is interesting to note that out of 47 file descriptor system calls, only 8 are retained. These crucial system calls, namely `munmap`, `close`, `read`, `ioctl`, `fcntl`, `dup2`, `pipe2`, and `write` are likely to play a vital role in the functioning of REDIS.

Figure 5(b) shows that for SLIMTOOLKIT, 24 system calls could have potentially been removed, 9 for SPEAKER, and at least 111 for CONFINE. We further observe that for SLIMTOOLKIT, out of these 24 system calls, only `mprotect` is associated with a denial of service vulnerability. For CONFINE, we observe that four system calls - `recvfrom`, `mount`, `clock_nanosleep`, and `mprotect` - are associated with four CVEs, in the categories of privilege escalation, denial of service, buffer overflow, and bypassing a restriction. For SPEAKER, out of the 9 system calls, none are associated with any CVEs.

*3) **Language Interpreter Application** (PYTHON):* Lastly, we will discuss PYTHON as part of our language runtime case study. As illustrated in Figure 6(a), SPEAKER outperforms other tools for PYTHON, too. It also shows that time-related, network, message queue, and shared memory system calls are not necessary for its proper functioning. Figure 6(b) shows that there are 45 system calls that are common across all the tools – showing room for improvement for each of them.

Delving deeper into the system calls, Figure 6(c) reveals that only file access, file descriptor, process control, and system-wide calls are required. As outlined in Figure 6(c), out of the 66 file access system calls, only 12 system calls are retained by all the tools, and merely 8 out of 47 file descriptor system calls are preserved. The majority of the removed system calls fall under the network category, with all the system calls in that category being removed, while the system-wide category comes in second with only 8 calls retained out of 81. Furthermore, the process-related system calls are also heavily restricted, with a mere 17 out of 80 calls being retained.

Next, we explore the system calls that could potentially have been removed, but were not, by the tools and discuss the associated CVEs. Figure 6(b) illustrates that for SLIMTOOLKIT, 11 system calls could have potentially been removed, 9 for SPEAKER, and 141 for CONFINE. We further observe that for SPEAKER, out of these 9 system calls, only `epoll_ctl` is associated with a denial of service CVE. For SLIMTOOLKIT, only two of these system calls, namely `kill` and `rt_sigreturn` are associated with three CVEs. For CONFINE, we observe that 11 out of 141 system calls are associated with 6 CVEs.



(a) System calls restricted per category. (b) System calls retained by each tool. (c) Distribution of common 45 system calls.

Fig. 6. Case Studies: Language Interpreter Application (PYTHON)

VII. DISCUSSIONS

A. Comments on existing container debloaters

On one hand, CONFINE uses static analysis, therefore, the programming language of the containerized application affects the analysis method. On the other hand, because of being conservative, although it achieves 100% correctness, it performs worst in terms of security hardening. Our case study analysis shows that a significant portion of the system calls retained by CONFINE can indeed be blocked. SPEAKER performed best in terms of security hardening, reduction, and correctness. The performance of SLIMTOOLKIT on security hardening, and code reduction shows the promise of using hybrid analysis methods, however, its low performance in retaining the correct behavior seems counterintuitive. Our case study-based analysis of common and exclusive system calls sheds light on how all these container debloaters can further be improved.

B. Threat to Validity

Neither the list of container debloaters nor the list of workload applications is exhaustive. This affects the generalizability of our findings. The number of container debloaters we studied was hindered by a practical constraint – availability. While in theory, it is possible to implement them if the artifacts are not available, however, in reality, it requires non-trivial efforts. We integrated 7 workload applications into our framework. Since, adding a new application requires a nontrivial effort, to maximize the generalizability we chose applications from multiple popular categories. Additionally, our correctness evaluation relied on handcrafted test cases. Although these test cases cover a wide diversity of functionalities, this only provides a lower bound for correctness.

VIII. RELATED WORK

Other Container Debloaters: Container debloating techniques focus on reducing the size and improving the efficiency of container images. Xu *et al.* [34] present practical static and dynamic tools for identifying inefficient container usage in Java programs. CIMPLIFIER [15] is designed to address the challenges associated with application containers, such as those provided by DOCKER. While containers offer agile software deployment, they often suffer from bloated sizes and compromised security due to the inclusion of unnecessary components and multiple applications within a single container. CIMPLIFIER offers algorithms that, when applied to a container with user-defined constraints, partition it into simpler containers. These containers are isolated from each other, communicate only as necessary, and contain the minimal resources required for their functionality. PROF-GEN [35] automatically generates a restrictive system call policy for containers and addresses the security threat of container escaping in cloud-native computing. PROF-GEN utilizes static binary analysis and dynamic analysis to determine the minimum required system calls without prior knowledge. Since, a runnable artifact of these tools are not available, we do not include it.

Other System Call Analysis and Access Control: System call analysis and access control techniques to focus on container security. Jang *et al.* [36] address the critical concern of security in container adoption by proposing a method to quantify container system call exposure. They combine the analysis of a large number of exploit codes with comprehensive experiments to uncover the syscall pass-through behavior of container runtimes. Their technique ranks system calls by their risk weights using information retrieval techniques. Win *et al.* [37] focuses on addressing the security concerns in container-based virtualization by proposing an access control solution that protects guest containers from a compromised host. The solution combines system call interception and the AppArmor mandatory access control (MAC) approach to prevent the host from accessing guest containers and their data.

Bloat Analysis and Performance Evaluation: Bloat analysis and vulnerability assessment techniques are used to identify the quantity, source, performance impact, and vulnerabilities associated with bloat in containers. Bhattacharya *et al.* [38] address the issue of excessive temporary object generation, also known as *object churn* in Java programs. While we don't analyze dynamic behavior of applications after debloating, it will be interesting to investigate in the future. Brown *et al.* [39] challenge the prevailing idea that reducing the number of code reuse gadgets through software debloating improves security. They demonstrate the flaws in using gadget count reduction as a metric and show that high reduction rates may not limit an

attacker's ability to construct an exploit, thus prescribed the use of constructing gadget classes. Since, code reuse gadgets are more interesting in the context of application debloating, we do not include it in our framework.

In summary, there has been growing interest in container debloating techniques to improve the security and performance of container images. However, there is a lack of standardized benchmarks for evaluating these techniques. To the best of our knowledge, no previous work has focused on developing a standardized benchmark for evaluating container debloating techniques. Our work aims to address this gap by proposing a benchmark suite for container debloating tools that can be used to compare different debloating approaches and their effectiveness in a fair and consistent manner.

IX. Conclusion

We presented DebloatBench$_C$, an extensible and sustainable benchmarking framework for rigorous evaluation of container debloaters. Our analysis showed that Speaker, which uses dynamic analysis, reduces more system calls as compared to Confine which uses static analysis. For SlimToolkit, which uses a hybrid approach, we observed that it reduces image size up to 79% and performs best in system calls reduction but it failed to produce correct results for MySQL and Httpd. Our analysis of these tools can open up new paths for future explorations. For example, our case study-based analysis of common and exclusive system calls sheds light on how all these container debloaters can further be improved.

Acknowledgements

This material is based upon work supported by the National Science Foundation (NSF) under Grant ACI-1440800 and the Office of Naval Research (ONR) under Contracts N68335-17-C-0558 and N00014-18-1-2660. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF or ONR. We thank our shepherd Prof. Shiyi Wei and the reviewers for their insightful and constructive feedback.

References

[1] "Kubernetes production-grade container orchestration." https://kubernetes.io.

[2] M. T. Chung, N. Quang-Hung, M.-T. Nguyen, and N. Thoai, "Using docker in high performance computing applications," in *2016 IEEE Sixth International Conference on Communications and Electronics (ICCE)*, pp. 52–57, IEEE, 2016.

[3] S. Bhattacharya, K. Rajamani, K. Gopinath, and M. Gupta, "The interplay of software bloat, hardware energy proportionality and system bottlenecks," in *Proceedings of the 4th Workshop on Power-Aware Computing and Systems*, pp. 1–5, 2011.

[4] A. Quach, R. Erinfolami, D. Demicco, and A. Prakash, "A multi-os cross-layer study of bloating in user programs, kernel and managed execution environments," in *Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation, FEAST@CCS 2017, Dallas, TX, USA*, pp. 65–70, 2017.

[5] G. Xu, N. Mitchell, M. Arnold, A. Rountev, and G. Sevitsky, "Software bloat analysis: Finding, removing, and preventing performance problems in modern large-scale object-oriented applications," in *In Proceedings of the FSE/SDP workshop on Future of software engineering research*, pp. 421–426, 2010.

[6] "Dockerslim." https://github.com/slimtoolkit/slim.

[7] S. Ghavamnia, T. Palit, A. Benameur, and M. Polychronakis, "Confine: Automated system call policy generation for container attack surface reduction," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2020.

[8] L. Lei, J. Sun, K. Sun, C. Shenefiel, R. Ma, Y. Wang, and Q. Li, "Speaker: Split-phase execution of application containers," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14*, pp. 230–251, Springer, 2017.

[9] V. Rastogi, D. Davidson, L. D. Carli, S. Jha, and P. Mcdaniel, "Cimplifier: automatically debloating containers," *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017.

[10] V. Rastogi, C. Niddodi, S. Mohan, and S. Jha, "New directions for container debloating," in *In Proceedings of the 2017 Workshop on Forming an Ecosystem Around Software Transformation (FEAST '17). Association for Computing Machinery, New York, NY, USA*, pp. 51–56, 2017.

[11] C. Mulliner and M. Neugschwandtner, "Breaking payloads with runtime code stripping and image freezing," *Black Hat USA*, 2015.

[12] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through api specialization," in *Proceedings of the 34th Annual Computer Security Applications Conference*, pp. 1–16, 2018.

[13] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack surface metrics and automated compile-time os kernel tailoring.," in *NDSS*, 2013.

[14] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "Face-change: Application-driven dynamic kernel view switching in a virtual machine," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 491–502, IEEE, 2014.

[15] V. Rastogi, D. Davidson, L. De Carli, S. Jha, and P. McDaniel, "Cimplifier: automatically debloating containers," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pp. 476–486, 2017.

[16] Q. Xin, Q. Zhang, and A. Orso, "Studying and understanding the tradeoffs between generality and reduction in software debloating," in *37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–13, 2022.

[17] Z. Wan, D. Lo, X. Xia, L. Cai, and S. Li, "Mining sandboxes for linux containers," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pp. 92–102, IEEE, 2017.

[18] "Seccomp bpf." https://www.kernel.org/doc/html/v4.16/userspace-api/seccomp_filter.html.

[19] "Apparmor." https://ubuntu.com/server/docs/security-apparmor/.

[20] Q. Xin, M. Kim, Q. Zhang, and A. Orso, "Subdomain-based generality-aware debloating," in *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pp. 224–236, IEEE, 2020.

[21] R. Sekar, "Classification and grouping of linux system calls." http://seclab.cs.sunysb.edu/sekar/papers/syscallclassif.htm.

[22] "Cve-2017-5123." https://www.cvedetails.com/cve/CVE-2017-5123/.

[23] T. Zhang, W. Shen, D. Lee, C. Jung, A. M. Azab, and R. Wang, "PeX: A permission check analysis framework for linux kernel," in *28th USENIX Security Symposium (USENIX Security 19)*, (Santa Clara, CA), pp. 1205–1220, USENIX Association, Aug. 2019.

[24] "Tpc-c." https://www.tpc.org/tpcc/.

[25] "Redis-benchmark." https://redis.io/topics/benchmarks.

[26] "Mongo-perf benchmark." https://github.com/mongodb/mongo-perf.

[27] "Apache benchmark." https://github.com/CloudFundoo/ApacheBench-ab.

[28] "pytest benchmark." https://github.com/pytest-dev/pytest.

[29] "nodejs benchmark." https://benchmarkjs.com/.

[30] M. Ali, M. Muzammil, F. Karim, A. Naeem, R. Haroon, M. Haris, H. Nadeem, W. Sabir, F. Shaon, F. Zaffar, V. Yegneswaran, A. Gehani, and S. Rahaman, "SoK: A Tale of Reduction, Security, and Correctness - Evaluating Program Debloating Paradigms and Their Compositions," in *ESORICS 2023 - 28th European Symposium on Research in Computer Security*, Lecture Notes in Computer Science, Springer, 2023.

[31] R. Martin, *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Robert C. Martin Series, Pearson Education, 2017.

[32] "Vagrant." https://www.vagrantup.com/.

[33] "Docker Hub." https://hub.docker.com/.

[34] G. Xu and A. Rountev, "Detecting inefficiently-used containers to avoid bloat," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 160–173, 2010.

[35] S. Kim, B. J. Kim, and D. H. Lee, "Prof-gen: Practical study on system call whitelist generation for container attack surface reduction," in *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pp. 278–287, IEEE, 2021.

[36] S. Jang, S. Song, B. Tak, S. Suneja, M. V. Le, C. Yue, and D. Williams, "Secquant: Quantifying container system call exposure," in *Computer Security–ESORICS 2022: 27th European Symposium on Research in Computer Security, Copenhagen, Denmark, September 26–30, 2022, Proceedings, Part II*, pp. 145–166, Springer, 2022.

[37] T. Y. Win, F. P. Tso, Q. Mair, and H. Tianfield, "Protect: Container process isolation using system call interception," in *2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC)*, pp. 191–196, IEEE, 2017.

[38] S. Bhattacharya, M. G. Nanda, K. Gopinath, and M. Gupta, "Reuse, recycle to de-bloat software," in *ECOOP 2011–Object-Oriented Programming: 25th European Conference, Lancaster, Uk, July 25-29, 2011 Proceedings 25*, pp. 408–432, Springer, 2011.

[39] M. D. Brown and S. Pande, "Is less really more? towards better metrics for measuring security improvements realized through software debloating.," in *CSET@ USENIX Security Symposium*, 2019.