SPEAKER: Split-Phase Execution of Application Containers

Lingguang Lei[1,3], Jianhua Sun[2], Kun Sun[3], Chris Shenefiel[5], Rui Ma[1], Yuewu Wang[1], and Qi Li[4]

[1] Institute of Information Engineering, Chinese Academy of Sciences, Beijing, China
[2] College of William and Mary, Williamsburg, USA
[3] George Mason University, Fairfax, USA
[4] Tsinghua University, Beijing, China
[5] Cisco Systems, Inc., Raleigh, USA

Abstract. Linux containers have recently gained more popularity as an operating system level virtualization approach for running multiple isolated OS distros on a control host or deploying large scale microservice-based applications in the cloud environment. The wide adoption of containers as an application deployment platform also attracts attackers' attention. Since the system calls are the entry points for processes trapping into the kernel, Linux seccomp filter has been integrated into popular container management tools such as Docker to effectively constrain the system calls available to the container. However, Docker lacks a method to obtain and customize the set of necessary system calls for a given application. Moreover, we observe that a number of system calls are only used during the short-term booting phase and can be safely removed from the long-term running phase for a given application container. In this paper, we propose a container security mechanism called SPEAKER that can dramatically reduce the number of available system calls to a given application container by customizing and differentiating its necessary system calls at two different execution phases, namely, booting phase and running phase. For a given application container, we first separate its execution into booting phase and running phase and then trace the invoked system calls at these two phases, respectively. Second, we extend the Linux seccomp filter to dynamically update the available system calls when the application is running from the booting phase into the running phase. Our mechanism is non-intrusive to the application running in the container. We evaluate SPEAKER on the popular web server and data store containers from Docker hub, and the experimental results show that it can successfully reduce more than 50% and 35% system calls in the running phase for the data store containers and the web server containers, respectively, with negligible performance overhead.

Keywords: Container, System Call, Seccomp

1 Introduction

Linux containers have emerged as one popular operating system level virtualization approach, and state-of-the-art container managers such as Docker [2] and

Rocket [13] are enabling the wide adoption of Linux containers. Recently, major cloud providers are adding support for Docker containers on Linux VMs [19, 49, 8]. In general, there are two types of Linux containers, *OS container* and *application container*. OS containers are useful for running identical or different flavors of an OS distro. Basically, they are designed to run multiple processes and services. Container technologies, such as LXC [21], OpenVZ [48], Linux VServer [32], BSD Jails [25] and Solaris zones [40], are suitable for creating OS containers. In contrast, application containers are designed to package and run a single service. Platforms for creating and deploying application containers include Docker [2] and Rocket [13]. The idea behind application containers is to create different containers for each of the components in a specific application. This approach works especially well when it comes to deploy a distributed, multi-component system using the microservices architecture.

In traditional hypervisor-based virtualization technologies, virtual machines (VMs) are presented with a hardware abstraction layer created by the hypervisor. Direct communications between the processes in the VMs and the host hardware are mediated by the hypervisor. In comparison, Linux containers, as one OS level virtualization technology, relies on security primitives such as *namespace* and *cgroups* provided by the Linux kernel to achieve the isolation among containers. A container can be considered as a group of processes sharing a bunch of isolated but dedicated Linux kernel resources. Therefore, applications running in the containers can achieve near native performance. Essentially, all containers share the same host OS kernel. Unfortunately, this sharing also exposes the entire host kernel interface to malicious processes running in any one of the containers. It has been demonstrated that an unwary container process can manage to escape into the host kernel space [7].

A number of security mechanisms have been proposed or adopted to enhance the security of containers [20, 37, 44, 6, 14, 9, 36, 48]. Since the system calls are the entry points for processes in the container trapping into the kernel, seccomp [6] has been integrated into the popular container management tools such as Docker to effectively constrain the system calls available to the container. For instance, Docker provides a whitelist of available system calls. However, Docker lacks a method to obtain and customize the set of system calls in the seccomp profile for a given application. Instead, it only provides a coarse-grained setting recommendation for all application containers.

We observe that an application container usually requires different sets of available system calls at different phases during the lifetime of its execution. Since most of the application containers are used to run long-term services such as web servers and database servers, the lifetime of those containers can be generally divided into two phases, namely, the *booting phase* and the *running phase*. The booting phase is responsible for setting the container environment and initializing the service within a couple of minutes. In the long-term running phase, the container service begins to accept service requests and send back responses. Due to the different sets of functions demanded in these two phases, a number of system calls invoked in the booting phase may no longer be needed and

thus can be removed in the running phase. For instance, compared to the default 313 available system calls through the entire lifetime of a Docker container, our experiments show only 116 system calls are invoked in the booting phase and 58 system calls are needed in the running phase of the MySQL database server.

In this paper, we develop a split-phase container execution mechanism called SPEAKER to dramatically reduce the number of necessary system calls during the lifetime of an application container's execution. For a given application container, it first profiles the two sets of system calls required for the booting phase and the running phase, respectively. Based on the profiling results, it can constrain the available system calls accordingly. Both system call profiling and constraint setting run outside the container, so it requires no changes on the image of the container.

To profile the system calls required in either booting phase or running phase, we first need to find a time point to separate the two phases. We provide a polling-based method to identify an accurate phase splitting time point for a given container image. In addition, we implement a coarse-grained phase separation approach, which can find a generic separation time point for application containers running on a specific platform. After obtaining the phase splitting time point, we perform dynamic program analysis to record the system calls invoked during the container booting and running phases.

For the split-phase container execution, we statically configure the set of available system calls in the booting phase and then dynamically change the available system calls when the container switches from the booting phase into the running phase. In the Linux kernel, the available system call list can be represented as a seccomp filter of one process, which can be set by two system calls `prctl()` and `seccomp()`. However, they can only be called to install the seccomp filters onto the calling process, but we need to change the seccomp filters of the processes inside the container from another process outside the container. Otherwise, a malicious process with the root privilege inside the container may be able to disable the constraints on the system calls. Since all processes inside one container share the same seccomp filter, we can change the seccomp filter of one process to update the available system calls for the entire container. To modify a container's seccomp filter from outside, we need to fill the semantic gap to find and change the data structures of seccomp filter.

The most popular usage of Docker containers is to deploy web applications [11], so we apply our mechanism on two closely related categories of application containers from Docker hub, namely, *web server containers* and *data store containers*. We study the top four web server container images (i.e., nginx, Tomcat, httpd, and php) and the top four data store container images (i.e., MySQL, Redis, MongoDB, and Postgres). The experimental results show that SPEAKER can reduce more than 50% and 35% system calls in the running phase for the data store containers and the web server containers, respectively. The number of system calls for web server containers may vary when deploying different web applications; however, they share most of system calls since the primary functions of web servers such as processing HTTP requests and web pages are the

same. Actually, for all website applications tested in our experiments, about 80% system calls will be invoked for just fetching one web page.

In summary, we make the following contributions.

- We develop a split-phase execution mechanism called SPEAKER to minimize the system call interface in one container at two different execution phases. It can successfully reduce the attack surface of containers by removing unnecessary system calls that may be misused by malicious processes in the container.
- We develop an out-of-the-box method to profile the necessary system calls for a given application container in the booting phase and the running phase, respectively.
- We develop a new method to dynamically change the sets of available systems calls by filling the semantic gap on the data structure of seccomp filter. SPEAKER does not require any modifications to the existing container management software or the application images.
- We implement SPEAKER as a tool set and evaluate its effectiveness on the popular container images downloaded from Docker hub. The experiments show that SPEAKER can effectively reduce the number of available system calls for both data store containers and web server containers with negligible performance overhead.

2 Background

In this section, we provide some background on namespace and seccomp mechanisms, which are highly related to our system design and implementation.

2.1 Linux Namespace

Namespace is one core mechanism that allows isolation of an application's view of system resources within a container. Linux kernel utilizes six types of namespaces: `pid`, `user`, `uts`, `mnt`, `net`, and `ipc`. Specifically, `pid` namespace isolates the process ID number space, which means each container may have a process whose PID is 1 and processes in different `pid` namespace can have the same PID value. In addition, all processes inside a container have a mapping PID on the Linux Kernel host outside the container. For instance, the process with PID 1 in one container could be the process on the host with PID 1001. In this paper, we use this mapping to help profile container's system calls from outside.

2.2 Seccomp

Secure computing (seccomp) is a sandboxing tool in the Linux kernel to restrict a process from making certain system calls. Since the system calls provide entry points for the processes in one container into the host kernel, a malicious app may misuse system calls to disable all the security measures and escape out of the container [52]. Seccomp can be used to reduce the number of entry points

into the kernel space, thereby reducing the kernel attack surface. Since Docker version 1.11.0, a `--security-opt seccomp` option is supported to set a seccomp profile when the container is launched. It allows the user to set the list of system calls available to be called inside the container. Currently the default seccomp profile by Docker has 313 available system calls [5].

Seccomp has three working modes: *seccomp-disabled*, *seccomp-strict*, and *seccomp-filter*. The seccomp-filter mode allows a process to specify a filter for the incoming system calls. Linux kernel provides two system calls, `prctl()` and `seccomp()`, to set the seccomp filter mode. However, they can only be used to change the seccomp filter mode of the calling thread/process and cannot set the seccomp filter mode of other processes.

3 Design and Implementation

Figure 1 shows the architecture of SPEAKER, which consists of two major modules, *the Tracing Module* and *the Slimming Module*, working in five sequential steps. For a given application container, the tracing module is responsible for profiling the available system calls in the booting phase and the running phase, respectively. The tracing module shares the system call lists with the slimming module, which is responsible for constraining the available system calls when the container boots up and runs. Both modules run outside of application containers as root-privileged processes in the host OS. SPEAKER is non-intrusive, so it does not require any modification to the applications or the container deployment tool.

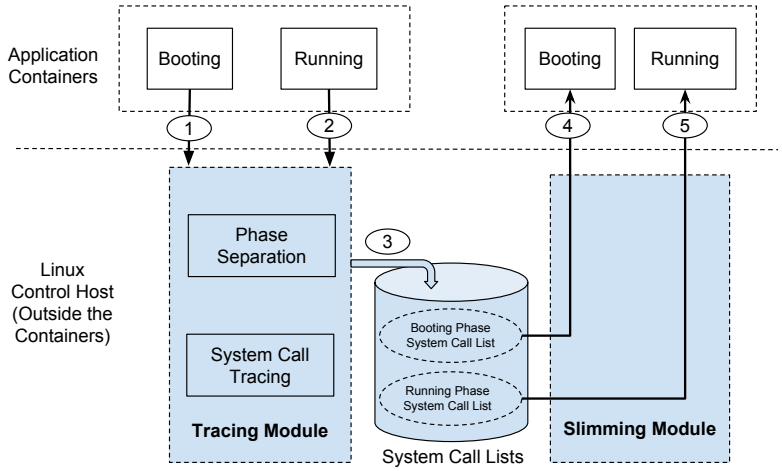

Fig. 1. SPEAKER Architecture

3.1 Tracing Module

This module is to generate system call sets for the booting phase and the running phase, respectively. It is transparent to the applications inside the container and consists of two components, *phase separation* and *system call tracing*.

Phase Separation The phase separation is in charge of separating the execution of the application containers into two phases, namely, the booting phase and the running phase. Though the booting phase is short, it may require a number of extra system calls to setup the execution environments, and those system calls are no longer necessary in the running phase. Moreover, the running phase may require some extra system calls to support the service's functions. Thus, it is important to find the running point that separates these two phases in order to profile their system calls. For instance, in the booting phase of the Apache web server, the container and the web server are booted and all modules needed for the service execution, such as `mod_php` and `mod_perl`, are loaded. In the running phase, the Apache web server accepts and handles the requests and generates the responses.

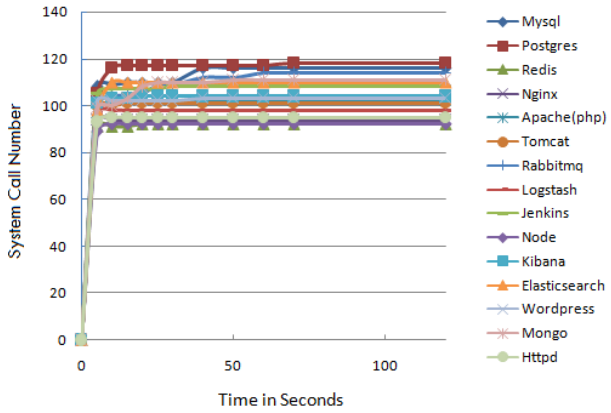

Fig. 2. Number of System Calls Invoked over Container Execution Time.

We can achieve a reliable phase separation through a polling-based method, which can find the splitting time point by continuously checking the status changes of the running service. Once the booting up finishes, the service enters the `running` status. Most current Linux distributions provide a `service` utility to uniformly manage various services, such as apache, mysql, nginx etc. Therefore, our polling-based method can find the split-phase time point by checking the service status through running the `service` command with `status` option. This method works well when the service creates its own `/etc/init.d` script.

We also develop a coarse-grained phase separation approach, which is generic and service independent. This method is based on two observations. First, the

container and service booting can finish quickly in tens of seconds. Second, the number of invoked system calls keeps increasing during the booting phase and becomes stable after the booting process ends and the container enters an idle running state. We verify both observations using the 15 most popular application container images in the Docker hub. Figure 2 shows that the numbers of system calls increase quickly in the first 10 seconds after the container starts to boot for all the 15 containers, and the number of system calls becomes stable after 70 seconds when all 15 containers enter the idle running state. Therefore, we can choose a rough time point (e.g., 100 seconds) for all containers. This time point may be different for other services on different hardware platform; however it should not be larger than a couple of minutes. Though it may include some extra system calls invoked in the service idle state into the whitelist of booting phase, since the booting phase is short, the chance for those extra system calls being misused by attackers is minor.

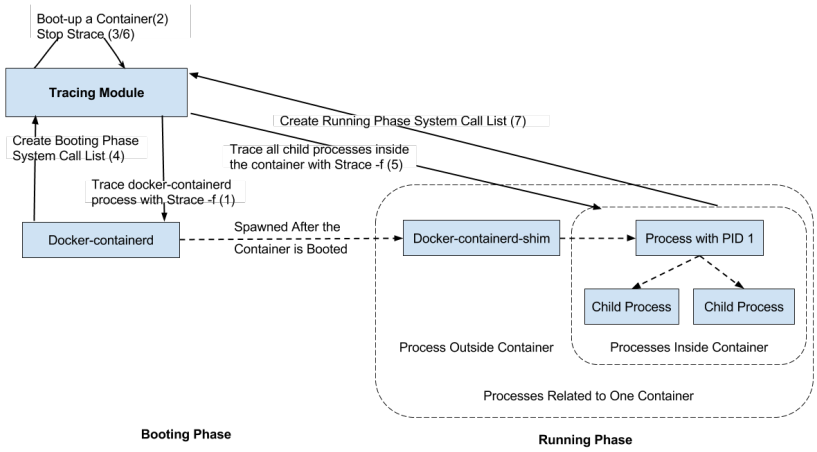

Fig. 3. Workflow of the Tracing Module.

System Call Tracing The system call tracing component is responsible for tracing the execution of the container as well as the hosted application to obtain the necessary system calls used in each phase. In most cases, multiple processes may be running inside a container even if only one service is hosted. Therefore, the system call tracing needs to ensure that all the processes inside the container are correctly identified to adequately collect the invoked system calls.

System call tracing can be done either using a static analyzer to extract all the system calls used from a container image or using a dynamic analyzer to collect the system calls invoked during the container booting and running stages. We choose to use the Linux *strace* tool to dynamically trace the necessary system calls for a given application container. We solve the challenge of tracing container processes from outside the container through utilizing the process mapping between the container and the host OS.

As shown in Figure 3, the tracing process consists of 7 steps, where steps 1 to 4 target at creating the booting phase system call list and steps 5 to 7 are to generate the running phase system call list. To guarantee the completeness of the tracing results, we enable the `-f` option of *strace* to trace the children of the processes currently being traced.

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 different attributes such as pid, uid, and gid when viewed from the host OS. For example, each container contains a process with PID 1, but the same process may have a PID larger than 1000 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 a different PID.

The arrowed dash lines in Figure 3 demonstrate the parent-child relationship among the container processes from the host's point of view. We can see that each container contains a *docker-containerd-shim* process, which is the parent of all the remaining container processes. All the *docker-containerd-shim* processes are spawned from the process *docker-containerd*. Therefore, we can obtain the booting phase system call list through tracing the process *docker-containerd*. Similarly, the running phase system call list can be obtained through tracing all processes inside the container when the booting is finished, which refer to the child processes of the container's *docker-containerd-shim* process.

3.2 Slimming Module

The slimming module is responsible for monitoring the execution of the container and dynamically changing the available system call list for all processes inside one container during the different execution phases. It restricts the container to use only system calls in the booting phase system call list during the booting phase. During the running phase, it only allows system calls in the running phase system call list, which may add new system calls and remove old system calls in the booting phase system call list. Note we cannot rely on one process inside the container to implement the slimming module, otherwise, a malicious process with the root privilege in the container may also be able to manipulate the slimming module to disable the constraints on the system calls. To solve this problem, we put the slimming module out of the target container.

The slimming module is implemented as a user space program on the host OS with root privilege. Since it is based on the seccomp mechanism in Linux kernel, we first introduce some detailed internal design of the seccomp mechanism and then present our implementation details.

A seccomp filter records an available system call list. In Linux, it is possible for a process to be attached with multiple seccomp filters, and all seccomp filters are organized in an 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. Each `seccomp_filter` structure has a `prog` pointer pointed to the `bpf_prog` structure. Each instruction is a 4-tuple structure that includes the actual filter code, the jump offset when the filter codes returns `true`, the jump

offset when `false` is returned, and a generic value. Figure 4 shows an example of a process attached with two seccomp filters. The first filter restricts the process to use the system calls `read()`, `write()`, `rt_sigreturn()` and `exit()`, as shown in the right-bottom `bpf_prog` structure. The k values in the `bpf_prog` structure correspond to the system call numbers of these four system calls. The second seccomp filter in the left bottom of Figure 4 restricts the process to use only `read()` and `write()` system calls.

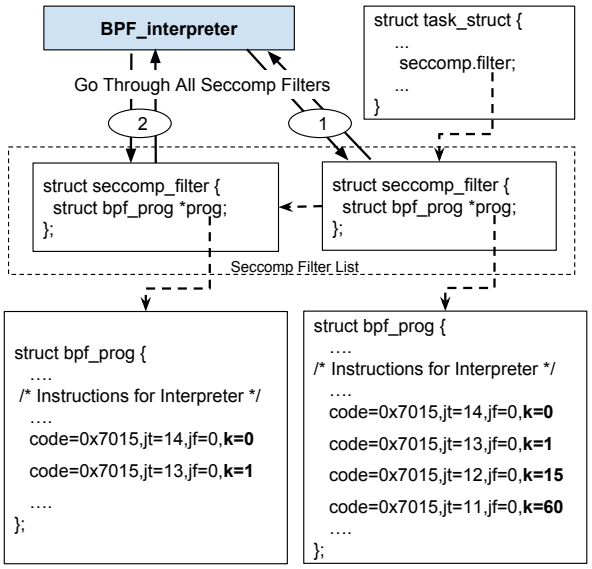

Fig. 4. Seccomp Data Structure in Linux Kernel

Seccomp Filters in Linux Kernel Seccomp mechanism restricts the set of system calls available to a process. The list of available system calls is represented as a *seccomp filter* data structure, as shown in Figure 4. Each process has a `task_struct` structure that contains an `seccomp` structure, which defines the seccomp state of the process. If the process is being protected by seccomp, the `filter` field of the `seccomp` structure points 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, it goes through all the seccomp filters attached to this process. As long as one seccomp filter does not include this system call, this process is not allowed to invoke this system call. Now let's see an example in Figure 4. When a `rt_sigreturn()` system call is invoked by the process, the BPF_interpreter first checks the seccomp filter on the right bottom and finds that the `rt_sigreturn()` system call is allowed. However, when it continues to check the second seccomp filter on the left-bottom, it finds that `rt_sigreturn()` system call is not allowed. After combining these two results, the BPF_interpreter denies the `rt_sigreturn()` system call.

Linux kernel provides two system calls `prctl()` and `seccomp()` to change the seccomp filters of one process. However, we cannot directly use them to dynamically change the seccomp filters of one container. First, they can only install the seccomp filters onto the calling process, but we need to change the seccomp filters of the processes inside the container from outside the container. Second, after one seccomp filter is installed, it cannot be removed or changed when the process is running. In other words, we can use these two system calls to add new seccomp filters but cannot remove any existing filters. Meanwhile, our experimental results show that some system calls used in the running phase are not necessary for the booting phase, and vice versa. Therefore, we choose to locate the memory address of the seccomp filters and directly modify their contents in the memory. It requires us to fill the semantic gaps on recovering the seccomp filter related data structures.

Workflow of Slimming Module The basic idea of the slimming module is to first construct a new `bpf_prog` struct in the memory based on the available system call list, and then redirect the `prog` pointer to it. The slimming module workflow consists of three steps, as shown in Figure 5. First, the container is booted with the system calls in the booting phase system call list. Second, we develop a *Seccomp Filter Constructor* to generate a `bpf_prog` struct that records the seccomp instructions according to the running phase system call list. The third and final step is changing the seccomp filters of all the processes inside a container to the newly crafted one.

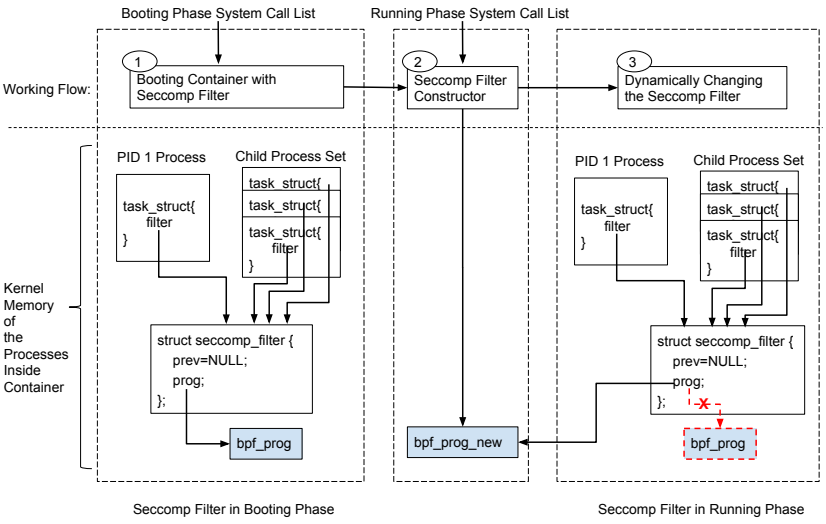

Fig. 5. Workflow of the Slimming Module.

Booting Container with Seccomp Filter As shown in Figure 3, before a container is created, a *docker-containerd-shim* process is spawned. It is the parent process of all the remaining processes inside the container. The seccomp filter for the booting phase can be enabled to protect the booting phase once the *docker-containerd-shim* process has been created. Alternatively, we can transform the booting phase system call list into a Docker seccomp profile and utilize the Docker *"–security-opt seccomp"* optional function to launch the container by the command *"docker run -id –security-opt seccomp:booting-phase-system-call-profile.json image_name"*. This approach is less secure than the first one, since the seccomp filter may not be enabled immediately after the container starts to boot. However, this time gap is small.

Seccomp Filter Constructor We develop a seccomp filter constructor to generate the kernel data structure of the seccomp filter for a given system call list. The constructor is composed of a user level process called *UIApp* and a Linux kernel module to perform a three-step transformation, as shown in Figure 6. First, the UIApp takes the system call list as the input and generates the corresponding bpf filter program. We use the libseccomp [12] library to convert one available system call list to the bpf filter instructions. Particularly, we use the `seccomp_rule_add()` method from the libseccomp library to add all available system calls and use the `seccomp_export_bpf()` method to export the resulting bpf filter program. Next, the bpf filter program generated in the user space is passed into the kernel module, which converts the program into a seccomp filter. Since some *code* of the bpf filter program is specific to seccomp filter, they are slightly different from those in the classic bpf filters. Finally, the `bpf_prog` structure is generated, that is the `bpf_prog_new` structure in Figure 5. For the sake of performance, in Linux kernel a bpf program with a new instruction set totally different from the ones in the seccomp filter program is accepted by the BPF_interpreter, which is included in `bpf_prog` structure.

Normally, we can use an internal kernel function `bpf_prog_create_from_user()` to do the last two steps of transformation. The function accepts two parameters, a pointer to the user space filter program and a function pointer to an internal kernel function `seccomp_check_filter()`. First, the user space filter buffer is copied into a kernel buffer, and then the passed-in `seccomp_check_filter()` function is called to transform the classic bpf filter program into a seccomp bpf filter program. Finally, the `bpf_prog` struct including a bpf program with a new instruction set is generated.

`bpf_prog_create_from_user()` is an internal non-exported kernel function. In addition, when we install the kernel module with the `system("insmod kernel-module")` function call in UIApp, the installed kernel module is not running in the same address space as the UIApp. Therefore, we could not pass user space filter buffer pointer created in step one directly into `bpf_prog_create_from_user()`. In our implementation, we directly pass the data content of the user space bpf filter into kernel module as the kernel module parameter, and create a copy one in the kernel. Then we change

`bpf_prog_create_from_user()` slightly to enable it accept the kernel buffer filter data, and incorporate it into our kernel module. In addition, the function `seccomp_check_filter()` is not exported too. Thus, we extract all related code and implement the function of `seccomp_check_filter()` as a kernel module.

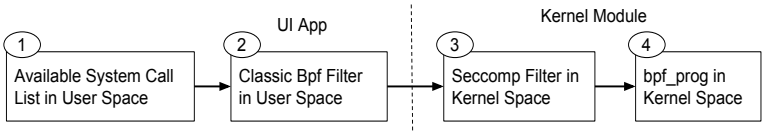

Fig. 6. Seccomp Filter Constructor.

Dynamically Changing the Seccomp Filter With the new `bpf_prog_new` structure, we can dynamically change the seccomp filters of all the container processes to enforce the available system calls in the running phase. When the container is successfully booted, usually more than one process will be running inside the container. For example, 6 processes will be created after the Apache service is launched in the container. Therefore, we need to change the seccomp filter for all processes. The inheritance attribute of the seccomp filter indicates that a forked child process will inherit the seccomp filter of its parent. As shown in Figure 5, when the service and the container is booted through command *"docker run -id –security-opt seccomp"*, the first process in the container has only one seccomp filter. All its child processes inherit this seccomp filter and contain the same pointer pointing to the `bpf_prog` structure. When we change the `bpf_prog` structure of one process in the container, the seccomp filters of all processes will be changed. In Figure 5, initially all processes possess the same `bpf_prog` struct in the booting phase. After changing the system call list, all the processes now share the new `bpf_prog_new` structure, which records the running phase system call list. In brief, the kernel module in Figure 6 first locates the `task_struct` of the first process inside the container through its PID, which is the process with PID 1 in Figure 3. Next, it finds the pointer of `bpf_prog` through the `task_struct`, as shown in Figure 5 and then modifies the pointer to point to the new constructed `bpf_prog_new` structure.

4 Experimental Results

We apply the SPEAKER toolkit on the popular web server containers and data store containers downloaded from Docker hub. We measure its effectiveness on reducing the number of available system calls from the default setting. We observe minimal performance overhead triggered by our system. Our experiments are conducted on the machine with a 4-core 1.6 GHz Intel i5 CPU and 8GB of RAM. We use the Ubuntu 15.10 Desktop Edition with Docker 1.11.0 installed.

4.1 System Call Reduction

We evaluate both web server containers and data store containers, which count around half of the deployed Docker application containers in real world.

Web Server Containers We study four popular web server images in Docker Hub, namely, *nginx*, *php*, *httpd*, and *tomcat*. Httpd is the Apache HTTP server and php is the Apache httpd server with PHP installed. Therefore, we only need to evaluate php, nginx, and tomcat. For all three web server containers, we directly pull the images from Docker hub and deploy a wiki software on it. For Tomcat a JSP-based open source wiki software JSPWIKI is deployed. Nginx and php are web servers with PHP installed, so we deploy the PHP-based open source wiki application DOKUWIKI. Since the available system calls in the running phase vary among different applications, we deploy a popular bulletin board system PHPBB3 on the Apache php web server. The installed software versions are Nginx web server 1.4.6 with PHP 5.5.34, Apache web server 2.4.10 with PHP 5.6.20, and Apache Tomcat 8.0.33 with JVM 1.7.0.

We use the HTTPERF[39] tool to access the web server continuously with increasing numbers of requests per second. We also scan the web servers using Skipfish in order to exercise as many code paths as possible. Skipfish is a tool conducting automated security check for the web applications by recursive crawl and dictionary-based probes, hence exercising many edge-cases. In addition, we manually access the websites to trigger all popular functions.

Table 1. System Call Reduction on Web Servers.

Server Name	Booting& Running	Booting Phase	Running Phase
Nginx	124	107 (86.3%)	79 (63.7%)
Tomcat	111	106 (95.5%)	46 (41.4%)
Php(Apache)DOKUWIKI	117	102 (87.2%)	70 (59.8%)
Php(Apache)PHPBB3	112	102 (91.1%)	67 (59.8%)

Table 1 shows the tracing results. We can see that the number of system calls invoked in the booting phase ranges from 90 to 120, but a large fraction of these system calls are not needed in the running phase. Moreover, accessing the web server through HTTPS does not incur substantially more system calls compared to the case of accessing through HTTP. By default, all the system calls needed in both the booting phase and the running phase must be enabled when the container enters the running phase. In contrast, SPEAKER can significantly reduce the system calls in the running phase by dynamically changing the seccomp filter. At the running phase, the system call reduction rates are 36.3%, 58.6%, 40.2%, and 40.2% for nginx, tomcat, php with DOKUWIKI, and php with PHPBB3, respectively.

Though the system call numbers may vary when deploying different web applications, the primary functions on processing HTTP requests and rendering

web pages are similar in web servers. For all four web applications tested, about 80% system calls are invoked when just fetching one page through WGET and HTTPERF[39]. Moreover, half of the remaining 20% are file-operation-related system calls such as `chmod()`, `ftruncate()`, `rename()`, `unlink()`, `sendfile()`, `dup()`, `pread64()` etc. All three PHP-enabled applications invoke 61 identical system calls, which occupy 77.21%, 87.14%, and 91.04% of the total running phase system calls for three web applications, respectively. It means the system calls invoked are more affected by the programming language used rather than the functions of the applications. The number of system calls does not correlate with the complexity of the web application, and we see that the seemingly more complicated bulletin board system invokes less system calls than DOKUWIKI.

Table 2. System Call Reduction on DB Servers.

Server Name	Booting & Running Phase	Booting Phase	Running Phase
Redis	102	92 (90.2%)	42 (41.2%)
MongoDB	116	110 (94.8%)	55 (47.4%)
MySQL	118	116 (98.3%)	58 (50.0%)
Postgres	118	116 (98.3%)	52 (44.1%)

Data Store Containers We evaluate four data store containers, namely, Redis server v3.2.0, MongoDB v3.2.6, MySQL v5.7.12, and Postgres v9.5.3, directly pulled from the Docker hub. The last three are traditional database platforms, while Redis is an in-memory data structure store that can serve as the database, cache and message broker. Since the operations on the data store containers are more deterministic when comparing to the web application containers. we generate their workloads in three aspects. First, we exercise the commands manually according to the official manual references of each data store [15, 45, 38, 4]. Second, we utilize the load testing tools to test the case when accessing these platforms concurrently. The load testing and benchmarking tool HammerDB is used to exercise on Postgres, MySQL, and Redis. For MongoDB, we use Cloud System Benchmark provided by Yahoo. Third, we use penetration testing tools to enumerate all the data in the data store platforms, (e.g., databases and tables in MySQL). As our purpose is to simulate normal user behavior, other penetration tests such as attacks exploiting are not triggered. SQLMap is used on sql-based databases including MySQL and Postgres, while NoSQL-Exploitation-Framework and NoSQLMap are used for Redis and MongoDB, respectively.

Table 2 shows the experimental results for the data store containers. We observe that the number of system calls necessary for the container booting ranges from 90 to 120, and more than half of the system calls invoked in the booting phase are not needed in the running phase. The system call reduction achieved by SPEAKER is above 50% for all data store containers.

4.2 Performance Overhead

We evaluate the performance impacts of SPEAKER on the running containers. In general, our system introduces negligibly small overhead on both system resource and application performance. Since all the processes in one container share the same set of seccomp filter, 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. Since the seccomp filter is enabled by default for Linux application containers, the changes of the filtering rules can barely have impacts on application performance. For a list with 116 system calls, it only takes around 41 milliseconds to change the seccomp filter, and it only occurs once. For the Apache web server container, we measure the throughput using HTTPERF [39]. For the MySQL data store container, we use benchmark tools coming with the MySQL to measure the response delay. Figure 7 shows that the performance differences on request throughput are minor when we enable either Docker default seccomp profile or SPEAKER.

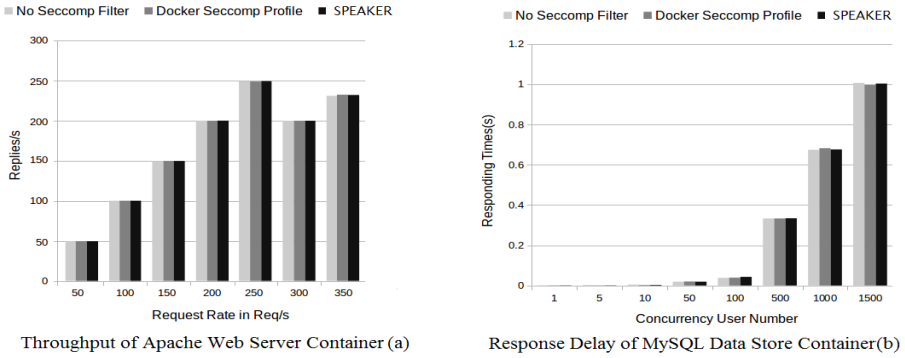

Fig. 7. Performance Overhead.

We implement the tracing module as a bash script with 570 SLOC. The slimming module is written in C language, consisting of a user level code with 2256 SLOC for converting the system call lists into Bpf filter format and a kernel module with 1088 SLOC for dynamically updating the seccomp filter.

5 Security Analysis

Since containers share the same kernel, it is critical to constrain the available system calls for each container to reduce the attack surface. Our security analysis mainly targets at the system calls being removed in our experiments, so those system calls not invoked by the containers (e.g., `seteuid()` and `setegid()`) are out of the scope. For the containers we tested in kernel 4.2, the number of available system calls during both booting and running phases could be reduced

from 370 to 111-124. By separating running phase from the booting phase our method can further reduce the number of available system calls. In the long-term running phase, we reduce the number of system calls to 46-79 out of 111-124.

By shrinking the number of available system calls, we can efficiently reduce the attack surface of the host OS and lower the risk that a malicious process may escape from the container and gain control of the host OS. On the one hand, we can remove the system calls which are vulnerable due to lack of sanity checking and thus may be misused by attackers. For instance, among the 47 system calls removed from the running phase of the Apache web server with DOKUWIKI, we found 14 vulnerable system calls with CVE security level MEDIUM or above, including `getrlimit()`, `listen()`, `sigaltstack()`, `socketpair()`, `prctl()`, `setsid()`, `setsockopt()`, `uname()` etc. Similarly, 23 vulnerable system calls can be prevented after we remove 60 unnecessary system calls in the running phase of the MySQL server, including `bind()`, `brk()`, `chdir()`, `epoll_ctl()`, `execve()`, `io_setup()`, `socketpair()`, `setsid()`, `listen()`, `setgroups()`, `setgid()`, `uname()`, `setuid()`, etc. Among those removed vulnerable system calls, 6 and 12 system calls may be exploited to achieve privilege escalation for Apache and MySQL containers, respectively. On the other hand, since attackers may misuse some high-privileged system calls (e.g., `fchown()`, `fchmodat()`, `mknodat()`) to launch attacks, we can reduce the attack surface by removing those high-privileged system calls. For instance, among the removed system calls, around 25% may be called to gain full control of the system [10, 35].

Particularly, `setuid()` is a dangerous system call frequently used in shellcode, it is eliminated from all data store containers we tested. And we remove the `exec` family of system calls such as `execve()` from the running of MySQL server. `execve()` system call is usually used to run a binary executable after creating a new process, and it may be misused by attackers to create a shell.By default, Docker seccomp profile has all the `exec` system calls enabled. After eliminating `execve()` from the system call interface, we can at least increase the difficulty on exploiting *shellshock* vulnerability. As another example, when the `setsockopt()` system call is eliminated from the Apache server with DOKUWIKI, we remove potential vulnerabilities that can be exploited by `setsockopt()` for launching heap memory corruption and denial of service attack against the host.

Our mechanism targets at reducing the attack surface; however, since the available system call list may still contain vulnerable system calls, we cannot prevent attackers from exploiting those remaining system calls. For the Apache web server container with Dokuwiki, 22 out of the total 70 available system calls may expose vulnerabilities. Similarly, for the MySQL server container, 16 out of the 58 available system calls may expose some vulnerabilities. To further constrain those vulnerable system calls, we can combine SPEAKER with careful sanity checking on the parameters of system calls [28, 34, 23] and strict resource access control [17, 16, 26, 53, 41]. For example, we can combine our SPEAKER system and the Linux capability mechanism [20] to block the exploitation of vulnerability CVE-2016-9793 [1], which requires both CAP_NET_ADMIN capa-

bility and `setsockopt()` system call. When `setsockopt()` cannot be removed from some containers, we still can prevent this vulnerability by removing the CAP_NET_ADMIN capability from the vulnerable processes in the container.

Since we cannot change the container processes' seccomp filter inside the container, we rely on a kernel module to dynamically change the seccomp filter from outside of an container. The kernel module does not export any APIs, and it can be removed from the system once the container enters the running phase and the seccomp filter has been changed. Therefore, it can hardly be misused by malicious processes in the container. In addition, it is difficult, if not impossible, for the attackers in the container to modify the seccomp filter through direct memory overwriting. First, it is difficult for the user processes in the containers to understand and locate the seccomp filter related structures in the kernel space. Second, even if the programs in the user space could resolve the semantic gap and locate the seccomp filter related structures, they cannot write into the kernel space memory.

6 Limitations and Discussion

System Call Tracing Completeness We trace system calls involved in the container boot-up phase and application running phase using both automatic workload generation tools and manual operations. However, this tracing process may still be incomplete. 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 may exist and may not be accounted in our tracing process. For example, the Apache web server can exhibit abnormal running patterns when experiencing extremely large amount of request traffic. As SPEAKER is a tool transparent to the applications running inside the container, we can combine the system call tracing phase with the `testing` process [24] of the application development to better trace the system call profile of a specific application. In addition, the static analysis approach (e.g., [54, 50]) can be integrated to further improve the profiling accuracy. We consider them as our future work to enhance our toolkit. Third, the existence of dynamically generated scripts such as PHP script can make the tracing more complicated, as these scripts can arbitrarily invoke certain system calls. A reiterative method can be adopted to add new available systems calls during a longer trace and verification stage.

System Call Control Granularity We adopt a simple container-specific whitelist model to reduce the unnecessary system calls, which is easy for deployment with its near zero overhead, comparing to the sophisticated and costly FSA or PDA [47, 50, 18] models based system call intrusion detection system. As other stateless model [54], SPEAKER also suffers a risk of system call misuse attacks, such as mimicry attack [51], which transforms a malicious attack sequence to a seemingly valid sequence by inserting `no-op` system calls. However, Kruegel et al. [27] and Zeng et al. [54]'s work illustrates that a stateful FSA or PDA model is actualy not much better than the stateless whitelist model, as it

is easy and can be automated for a mimicry attack to evade FSA models. To achieve a more fine-grained protection of the system calls, we could integrate the system call interposition based approach such as MBox [26] or the argument constraint solutions [28, 34, 23]. We leave it as our future work.

Deployment Extensibility We divide one container's lifetime into the booting phase and the running phase; however, the same paradigm can be extended to achieve multiple phase execution. Moreover, for those application containers deployed using tools other than Docker, our approach works well as long as the container execution can be dissected into separated phases. SPEAKER can be smoothly integrated into the container management services, such as Amazon EC2 container service (Amazon ECS) [8], Docker Datacenter [3], and OpenShift Enterprise [22] to prevent malicious applications in one container from escaping into the hosting virtual machine (VM).

Phase Separation Efficiency We provide a polling-based method and coarse-grained timing-based method for phase separation. However, the timing-based solution might cause some extra system calls invoked in the service idle state being added into the whitelist of booting phase. While the polling-based method will subject to the implementation constraints of the services, e.g. it will not work well for the services without their own `/etc/init.d` scripts. We can adopt a binary-based execution partition scheme [31] to achieve a fine-grained phase separation. It is based on one 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 initialization phase (booting phase) from the running phase. We leave it as one of our future work.

Diversity of the Container Evaluation Our current evaluation focuses on two most popular categories of Application containers, i.e. web server container and data store container, which count for half of all real deployed containers. A more variety of application containers could boost the impact of the proposed approach. We will leave it as our future work to perform evaluation on other types of containers, such as RabbitMQ, Redis, Node.js, Logstash etc.

7 Related Work

Linux Kernel Security Primitives Linux kernel provides other primitives which can be utilized to enhance container security. *CGroups* 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, by default only 14 out of the 38 capabilities are

allowed [20]. Contrary to MAC, for DAC the access is determined by the owner of the object or resource in question.

Container Security Reshetova et al. [46] gives a comparative study of several OS-level virtualization systems and identify the gaps in current security solutions based on a spectrum of attack models. [14] analyzes the security level of Docker containers and how Docker interact with the security features of Linux kernel. However, they only talk about how container isolation is achieved through namespaces and cgroups. An extension to the *Dockerfile* is proposed in [9] to ship a specific SELinux policy for processes running in a Docker image, which incurs great burden for container image maintainers to build up a dedicated policy module. In contrast, we have designed a practical, non-intrusive, and systematic framework to enhance the security of application containers. [36] proposes an approach that combines customized AppArmor/SELinux rules based on container operation tracing with host-based intrusion detection. However, we focus on proactively eliminating the unnecessary system calls to reduce the potential vulnerabilities exploitable by malicious containers.

Application Sandboxing System call interposition based application sandboxing [16] 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 to create a tailored seccomp policy instead of intercepting every system call. MBOX [26] is a lightweight sandboxing mechanism that interposes on a sandboxed program's system calls to layer a sandbox filesystem on the host filesystem. Similarly, we also utilize seccomp/BFP as a means for interposing system calls invoked by processes in the containers. The Capsicum sandboxing framework [53] isolates processes from global kernel resources by disabling system calls which address resources via global namespaces. Their approach is different from ours as we utilize the seccomp mechanism. *Systrace* [41] is a solution that confines multiple applications running according to accurate policies. We have adopted its tracing capability to generate audit logs for execution of container processes. Moreover, some other system call monitoring based anomaly detection and prevention solutions have been proposed [33, 47, 50, 18] and may be integrated to further enhance the container security.

Attack Surface Reduction Seccomp [6] allows a process to restrict a 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 to determine the policy itself. In addition, we split the container execution into two different phases and dynamically change the seccomp filters from outside the containter to further reduce the attack surface. Cimplifier [43] and Docker-slim [42] 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 runnable system call list during various phases of container booting and running. A different approach for system hardening is trimming [29, 30], 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 container host.

8 Conclusions

In this paper, we design and develop a system call reduction mechanism called SPEAKER to reduce the attack surface of Linux application containers. It works by first tracing the available system calls necessary for the booting phase and the running phase of the application containers, and then dynamically changing the seccomp filter to udpate the available system calls for each phase. SPEAKER runs outside the container and is completely non-intrusive to the application containers. Our evaluation results show that SPEAKER can significantly reduce the system call interface and incurs almost no performance overhead.

Acknowledgments We would like to thank our shepherd Andrea Lanzi and our anonymous reviewers for their valuable comments and suggestions. We would also like to thank Xianchen Meng, Chong Guan, Yue Li, and Shengye Wan for their feedback and advice. This work is partially supported by U.S. ONR grants N00014-16-1-3216 and N00014-16-1-3214, the National Basic Research Program of China under GA No. 2013CB338001 (973 Program), the National Key Research Development Program of China under GA No. 2016YFB0800102, and a Cisco award.

References

1. CVE-2016-9793 Detail. https://nvd.nist.gov/vuln/detail/CVE-2016-9793
2. Docker. https://www.docker.com/
3. Docker Datacenter. https://www.docker.com/products/docker-datacenter
4. PostgreSQL 9.5.3. http://www.postgresql.org/docs/current/static/sql-commands.html
5. Seccomp security profiles for Docker. https://github.com/docker/docker/blob/master/docs/security/seccomp.md
6. SECure COMPuting with filters. https://www.kernel.org/doc/Documentation/prctl/seccomp_filter.txt
7. Vulnerability summary for cve-2014-9357. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-9357
8. AWS: Amazon EC2 Container Service. https://aws.amazon.com/ecs/
9. Bacis, E., Mutti, S., Capelli, S., Paraboschi, S.: DockerPolicyModules: mandatory access control for docker containers. In: Communications and Network Security (CNS), 2015 IEEE Conference on. pp. 749–750. IEEE (2015)
10. Bernaschi, M., Gabrielli, E., Mancini, L.V.: Enhancements to the linux kernel for blocking buffer overflow based attacks. In: Annual Linux Showcase & Conference (2000)
11. Boettiger, C.: An introduction to docker for reproducible research. ACM SIGOPS Operating Systems Review 49(1), 71–79 (2015)

12. Bruno, L.: Libseccomp: An enhanced seccomp (mode 2) helper library. https://github.com/seccomp/libseccomp
13. Bruno, L.: rkt - app container runtime. https://github.com/coreos/rkt
14. Bui, T.: Analysis of docker security. arXiv preprint arXiv:1501.02967 (2015)
15. Corporation, O.: Mysql 5.7 reference manual. http://dev.mysql.com/doc/refman/5.7/en/tutorial.html
16. Garfinkel, T., Pfaff, B., Rosenblum, M., et al.: Ostia: A delegating architecture for secure system call interposition. In: NDSS (2004)
17. Garfinkel, T., et al.: Traps and pitfalls: Practical problems in system call interposition based security tools. In: NDSS. vol. 3, pp. 163–176 (2003)
18. Giffin, J.T., Jha, S., Miller, B.P.: Detecting manipulated remote call streams. In: USENIX Security Symposium. pp. 61–79 (2002)
19. Google: Container engine on Google cloud platform. https://cloud.google.com/container-engine/
20. Hallyn, S.E., Morgan, A.G.: Linux capabilities: Making them work. In: Linux Symposium. vol. 8 (2008)
21. Helsley, M.: LXC: Linux container tools. IBM devloperWorks Technical Library (2009)
22. Inc., R.H.: Red Hat OpenShift Container Platform. https://www.openshift.com/enterprise/trial.html
23. Jachner, J., Agarwal, V.K.: Data flow anomaly detection. IEEE transactions on software engineering (4), 432–437 (1984)
24. Jacobson, I., Booch, G., Rumbaugh, J., Rumbaugh, J., Booch, G.: The unified software development process, vol. 1. Addison-wesley Reading (1999)
25. Kamp, P.H., Watson, R.N.: Jails: Confining the omnipotent root. In: the 2nd International SANE Conference. vol. 43, p. 116 (2000)
26. Kim, T., Zeldovich, N.: Practical and effective sandboxing for non-root users. In: USENIX Annual Technical Conference. pp. 139–144 (2013)
27. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: Proceedings of the 14th conference on USENIX Security Symposium-Volume 14. pp. 11–11. USENIX Association (2005)
28. Kruegel, C., Mutz, D., Valeur, F., Vigna, G.: On the detection of anomalous system call arguments. In: European Symposium on Research in Computer Security. pp. 326–343. Springer (2003)
29. Kurmus, A., Sorniotti, A., Kapitza, R.: Attack surface reduction for commodity os kernels: trimmed garden plants may attract less bugs. In: Proceedings of the Fourth European Workshop on System Security. p. 6. ACM (2011)
30. Kurmus, A., Tartler, R., Dorneanu, D., Heinloth, B., Rothberg, V., Ruprecht, A., Schröder-Preikschat, W., Lohmann, D., Kapitza, R.: Attack surface metrics and automated compile-time os kernel tailoring. In: NDSS (2013)
31. Lee, K.H., Zhang, X., Xu, D.: High accuracy attack provenance via binary-based execution partition. In: NDSS (2013)
32. des Ligneris, B.: Virtualization of linux based computers: the linux-vserver project. In: HPCS'05. pp. 340–346. IEEE (2005)
33. Linn, C., Rajagopalan, M., Baker, S., Collberg, C.S., Debray, S.K., Hartman, J.H.: Protecting against unexpected system calls. In: Usenix Security (2005)
34. Maggi, F., Matteucci, M., Zanero, S.: Detecting intrusions through system call sequence and argument analysis. IEEE Transactions on Dependable and Secure Computing 7(4), 381–395 (2010)

35. Martignoni, L., Christodorescu, M., Jha, S.: Omniunpack: Fast, generic, and safe unpacking of malware. In: Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual. pp. 431–441. IEEE (2007)
36. Mattetti, M., Shulman-Peleg, A., Allouche, Y., Corradi, A., Dolev, S., Foschini, L.: Securing the infrastructure and the workloads of linux containers. In: Communications and Network Security (CNS), 2015 IEEE Conference on (2015)
37. Menage, P., Jackson, P., Lameter, C.: Cgroups. https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt
38. MongoDB, I.: Mongodb manual reference. https://docs.mongodb.com/manual/reference/command/
39. Mosberger, D., Jin, T.: Httperf: A tool for measuring web server performance. ACM SIGMETRICS Performance Evaluation Review 26(3), 31–37 (1998)
40. Price, D., Tucker, A.: Solaris zones: Operating system support for consolidating commercial workloads. In: Proceedings of the 18th USENIX Conference on System Administration. LISA (2004)
41. Provos, N.: Improving host security with system call policies. In: Usenix Security. vol. 3, p. 19 (2003)
42. Quest, K.C.: docker-slim: Lean and mean docker containers. https://github.com/docker-slim/docker-slim
43. Rastogi, V., Davidson, D., De Carli, L., Jha, S., McDaniel, P.: Towards least privilege containers with cimplifier. arXiv preprint arXiv:1602.08410 (2016)
44. RedHat: Docker selinux security policy. https://access.redhat.com/documentation/en/red-hat-enterprise-linux-atomic-host/7/container-security-guide/chapter-6-docker-selinux-security-policy
45. Redislabs: Redis commands reference. http://redis.io/commands
46. Reshetova, E., Karhunen, J., Nyman, T., Asokan, N.: Security of os-level virtualization technologies. In: Secure IT Systems, pp. 77–93. Springer (2014)
47. Sekar, R., Bendre, M., Dhurjati, D., Bollineni, P.: A fast automaton-based method for detecting anomalous program behaviors. In: Proceedings of IEEE Security and Privacy. pp. 144–155 (2001)
48. Soltesz, S., Pötzl, H., Fiuczynski, M.E., Bavier, A., Peterson, L.: Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. In: ACM SIGOPS Operating Systems Review. vol. 41, pp. 275–287. ACM (2007)
49. van Surksum, K.: Microsoft announces support for docker container virtualization for next version of windows server (2014)
50. Wagner, D., Dean, R.: Intrusion detection via static analysis. In: Security and Privacy, Proceedings. 2001 IEEE Symposium on (2001)
51. Wagner, D., Soto, P.: Mimicry attacks on host-based intrusion detection systems. In: Proceedings of the 9th ACM Conference on Computer and Communications Security. pp. 255–264. ACM (2002)
52. Walsh, D.J.: Docker security in the future. https://opensource.com/business/15/3/docker-security-future
53. Watson, R.N., Anderson, J., Laurie, B., Kennaway, K.: Capsicum: Practical capabilities for unix. In: USENIX Security Symposium. vol. 46, p. 2 (2010)
54. Zeng, Q., Xin, Z., Wu, D., Liu, P., Mao, B.: Tailored Application-specific System Call Tables. Tech. rep., Pennsylvania State University (2014)