10th IEEE International Conference on Software Testing, Verification and Validation

# Mining Sandboxes for Linux Containers

Zhiyuan Wan*, David Lo†, Xin Xia*‡, Liang Cai*, and Shanping Li*
*College of Computer Science and Technology, Zhejiang University, China
†School of Information Systems, Singapore Management University, Singapore
{wanzhiyuan, xxkidd, leoncai, shan}@zju.edu.cn, davidlo@smu.edu.sg

***Abstract*—A container is a group of processes isolated from other groups via distinct kernel namespaces and resource allocation quota. Attacks against containers often leverage kernel exploits through system call interface. In this paper, we present an approach that mines sandboxes for containers. We first explore the behaviors of a container by leveraging automatic testing, and extract the set of system calls accessed during testing. The set of system calls then results as a sandbox of the container. The mined sandbox restricts the container's access to system calls which are not seen during testing and thus reduces the attack surface. In the experiment, our approach requires less than eleven minutes to mine sandbox for each of the containers. The enforcement of mined sandboxes does not impact the regular functionality of a container and incurs low performance overhead.**

## I. Introduction

Platform-as-a-Service (PaaS) cloud is a fast-growing segment of cloud market, being projected to reach $7.5 billion by 2020 [1]. A PaaS cloud permits tenants to deploy applications in the form of application executables or interpreted source code (e.g. PHP, Ruby, Node.js, Java). The deployed applications execute in a provider-managed host OS, which is shared with applications of other tenants. Thus a PaaS cloud often leverages OS-based techniques, such as Linux containers, to isolate applications and tenants.

Containers provide a lightweight operating system level virtualization, which groups resources like processes, files and devices into isolated namespaces. This gives users the appearance of having their own operating system with near native performance and no additional virtualization overhead. Container technologies, such as Docker [2], enable an easy packaging and rapid deployment of applications. However, most containers that run in the cloud are too complicated to trust. The primary source of security problems in containers is system calls that are not namespace-aware [3]. Non-namespace-aware system call interface facilitates the adversary to compromise untrusted containers to exploit kernel vulnerabilities to elevate privileges, bypass access control policy enforcement, and escape isolation mechanisms. For instance, a compromised container can exploit a bug in the underlying kernel that allows privilege escalation and arbitrary code execution on the host [4].

How can cloud providers protect the clouds from untrusted containers? One straightforward way is to place the container in a sandbox to restrain its access to system calls. By restricting system calls, we could also limit the impact that an adversary can make if a container is compromised. System call interposition is a powerful approach to restrict the power of a program by intercepting its system calls [5]. Sandboxing techniques based on system call interposition have been developed in the past [6], [7], [8], [9], [10], [11]. Most of them focus on implementing sandboxing techniques and ensuring secure system call interposition. However, generating accurate sandbox policies for a program are always challenging [7]. We are inspired by a recent work *BOXMATE* [12], which learns and enforces sandbox policies for Android applications. *BOXMATE* first explores Android application behavior and extracts the set of resources accessed during testing. This set is then used as a sandbox, which blocks access to resources not used during testing. We would like to port the idea of *sandbox mining* in *BOXMATE* to be able to confine Linux containers.

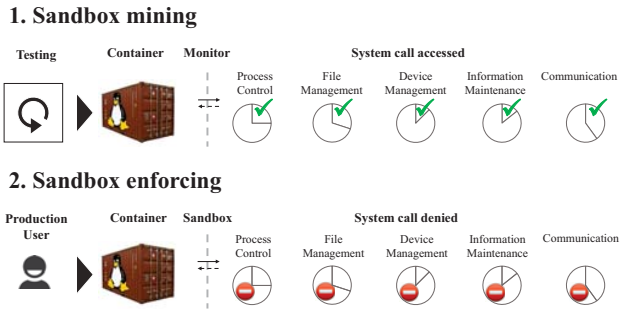


Fig. 1: Our approach in a nutshell. Mining phase monitors accessed system calls when testing. These system calls make up a *sandbox* for the container, which later prohibits access to system calls not accessed during testing.

A container comprises multiple processes of different functionalities that access distinct system calls. Different containers may access distinct sets of system calls. Therefore, a common sandbox for all the containers is too coarse. In this paper, we present an approach to *automatically extract sandbox rules* for a given container. The approach is composed of two phases shown in Fig. 1:

- **Sandbox mining.** In the first phase, we mine the rules that will make the sandbox. We use automatic testing to explore container behaviors, and monitor all accesses to system calls.
- **Sandbox enforcing.** In the second phase, we assume that system calls which are not accessed during the mining phase should not be accessed in production either.

‡Corresponding author

978-1-5090-6031-3/17 $31.00 © 2017 IEEE
DOI 10.1109/ICST.2017.16

Consequently, if the container (unexpectedly) requires access to a new system call, the sandbox will prohibit access.

To the best of our knowledge, our approach is the first technique to leverage automatic testing to extract sandbox rules for Linux containers. While our approach is applicable to any Linux container management service, we selected Docker as a concrete example because of its popularity. Our approach has a number of compelling features:

- **No training in production.** In contrast to anomaly detection systems, our approach does not require training process in production. The "normal" system call access would already be explored during the mining phase.
- **Reducing attack surface.** The mined sandbox detects system calls that cannot be seen during the mining phase, which reduces the attack surface by confining the adversary and limiting the damage he/she could cause.
- **Guarantees from sandboxing.** Our approach runs test suites to explore "normal" container behaviors. The testing may be incomplete, and other (in particular malicious) behaviors are still possible. However, the testing covers a safe subset of all possible container behaviors. Sandboxing is then used to guarantee that no unknown system calls aside from those used in the testing phase are permitted.

We evaluate our approach by applying it to eight Docker containers and focus on three research questions:

**RQ1. How efficiently can our approach mine sandboxes?**

We automatically run test suites on Docker containers, and check the system call convergence. It takes less than two minutes for the set of accessed system calls to saturate. In addition, we compare our mined sandboxes with the default sandbox provided by Docker. The default sandbox allows more than 300 system calls [13] and is thus too coarse. On the contrary, our mined sandboxes allow 66 - 105 system calls for eight containers in the experiment, which significantly reduce the attack surface.

**RQ2. Can automatic testing sufficiently cover behaviors?**

If a system call $S$ is not accessed during the mining phase, later non-malicious access to $S$ would trigger a false alarm. We run use cases that cover core functionality of containers to check whether the enforcing mined sandboxes would trigger alarms. The result shows that all the use cases end with no false alarms.

**RQ3. What is the performance overhead of sandbox enforcement?**

We evaluate the performance overhead of enforcing mined sandboxes on a set of containers. The result shows that sandbox enforcement incurs very low end-to-end performance overhead (0.6% - 2.14%). Our mined sandboxes also provide a slightly lower performance overhead than that of the default sandbox.

The reminder of this paper is organized as follows. After discussing background and related work in Section II, Section III specifies the threat model and motivation of our work.

```
{
        "defaultAction": "SCMP_ACT_ERRNO",
        "architectures": [
                "SCMP_ARCH_X86_64",
                "SCMP_ARCH_X86",
                "SCMP_ARCH_X32"
        ],
        "syscalls": [
                {
                        "name": "accept",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                {
                        "name": "accept4",
                        "action": "SCMP_ACT_ALLOW",
                        "args": []
                },
                ...
        ]
}
```

Fig. 2: A snippet of Docker *Seccomp profile*, expressed in JavaScript Object Notation (JSON).

Section IV and V detail two phases of our approach. We evaluate our approach in Section VI and discuss threats to validity and limitations in Section VII. Finally, Section VIII closes with conclusion and future work.

## II. Background and Related Work

### A. System Call Interposition

System calls allow virtually all of a program's interactions with the network, filesystem, and other sensitive system resources. System call interposition is a powerful approach to restrict the power of a program [5].

There exists a significant body of related work in the domain of system call interposition. Implementing system call interposition tools securely can be quite subtle [5]. Garfinkel studies the common mistakes and pitfalls, and uses the system call interposition technique to enforce security policies in the Ostia tool [14]. System call interposition tools, such as Janus [6], [15], Systrace [7], and ETrace [16], can enforce fine-grained policies at granularity of the operating system's system call infrastructure. System call interposition is also used for sandboxing [6], [7], [8], [9], [10], [11] and intrusion detection [17], [18], [19], [20], [21], [22], [23], [24], [25].

*Seccomp-BPF* framework [26] is a system call interposition implementation for Linux Kernel introduced in Linux 3.5. It is an extension to *Seccomp* [27], which is a mechanism to isolate a third-party application by disallowing all system calls except for reading and writing of already-opened files. *Seccomp-BPF* generalizes *Seccomp* by accepting *Berkeley Packet Filter* (BPF) programs to filter system calls and their arguments. For example, the *BPF* program can decide whether a program can invoke the `reboot()` system call.

In Docker, the host can assign a *Seccomp BPF* program for a container. Docker uses a *Seccomp profile* to capture a *BPF* program for readability [13]. Fig. 2 shows a snippet of *Seccomp profile* used by Docker, written in the JSON [28] format.

By default, Docker disallows 44 system calls out of 300+ for all of the containers to provide wide application compatibility [13]. However, the principle of least privilege [29] requires that a program must only access the information and resources necessary to complete its operation. In our experiment, we notice that top-downloaded Docker containers access less than 34% of the system calls which are whitelisted in the default *Seccomp profile*.

*Containers are granted more privileges than they require.*

### B. System Call Policy Generation

Generating an accurate system call policy for an existing program has always been challenging [7]. It is difficult and impossible to generate an accurate policy without knowing all possible behaviors of a program. The question "what does a program do?" is the general problem of *program analysis*. Program analysis falls into two categories: *static* analysis and *dynamic* analysis.

Static analysis checks the code without actually executing programs. It sets an upper bound to what a program can do. If static analysis determines some behavior is impossible, the behavior can be safely excluded. Janus [6] recognizes a list of dangerous system calls statically. Wagner and Dean [19] derive system call sequences from program source code.

The limitation of static analysis is *over-approximation*. The analysis often assume that more behaviors are possible than actually would be. Static analysis is also undecidable in all generality due to the halting problem.

*Static analysis produces over-approximation.*

Dynamic analysis analyzes actual executions of a running program. It sets a lower bound of a program's behaviors. Any (benign) behavior seen in past executions should be allowed in the future as well. Given a set of executions, one can learn program benign behaviors to infer system call policies. There is a rich set of articles about system call policy generation through dynamic analysis. Some studies look at a sequence of system calls to detect deviations to normal behaviors [18], [17], [23]. Instead of analyzing system call sequences, some studies take into account the arguments of system calls. [24] uses finite state automata (FSA) techniques to capture temporal relationships among system calls [25], [30]. Some studies keep track of data flow between system calls [20], [31]. Other researchers also take advantage of machine learning techniques, such as Hidden Markov Models (HMM) [22], [32], Neural Networks [33], and k-Nearest Neighbors [34].

The fundamental limitation of dynamic analysis is *incompleteness*. If some behavior has not been observed so far, there is no guarantee that it may not occur in the future. Given the high cost of false alarms, a sufficient set of executions must be available to cover all of the normal behaviors. The set of executions can either derive from testing, or from production (a training phase is required) [12].

*Dynamic analysis requires sufficient "normal" executions to be trained with.*

### C. Consequences

Sandboxing, program analysis and testing are mature technologies. However, each of them has limitations: sandboxing needs policy, dynamic analysis needs executions, and testing cannot guarantee the absence of malicious behavior [12]. Nonetheless, Zeller et al. argue that combining the three not only mitigates the limitations, but also *turns the incompleteness of dynamic analysis into a guarantee* [35]. In our case, system call interposition-based sandboxing can guarantee that anything not seen yet will not happen.

## III. Threat Model and Motivation

Most containers that run in the cloud, e.g., Web server, database systems and customized applications, are too complicated to trust. Even with access to the source code, it is difficult to reason about the security of a container. An untrusted container might be compromised by carefully craft inputs because of exploitable vulnerabilities. A compromised container can further do harm in many ways. For instance, a compromised container can exploit a bug in the underlying kernel that allows privilege escalation and arbitrary code execution on the host [4]; it can also acquire packet of another container via ARP spoofing [36]. We assume the existence of vulnerabilities to the adversary that he/she can use to gain unauthorized access to the underlying operating system and further compromise other containers in the cloud.

We observe that system call interface is the only gateway to make persistent changes to the underlying systems [7]. Nevertheless, system call interface is dangerously wide; less-exercised system calls are a major source of kernel exploits. To limit the impact an adversary can make, it is straightforward to sandbox a container and restrict the system calls it is permitted to access. We notice that the default sandbox provided by Docker disallows only 44 system calls – the default sandbox is too coarse. Containers are granted more privileges than they require. To follow the principle of least privilege, our approach automatically mines sandbox rules for containers during testing; and later enforces the policy by restricting system call invocations through sandboxing.

## IV. Sandbox Mining

### A. Overview

During the mining phase, we automatically explore container behaviors, and monitor its system calls. This section illustrates the fundamental steps of our approach during the mining phase.

*1)* ***Enable tracing****:* The first step is to prepare the kernel to enable tracing. We use container-aware monitoring tool *sysdig* [37] to record system calls that are accessed by a container at run time. The monitoring tool *sysdig* logs:

- an *enter* entry for a system call, including timestamp, process that executes the system call, thread ID (which corresponds to the process ID for single-threaded processes), and list of system call arguments;

```
[github.com/opencontainers/runc/libcontainer/utils/
    utils_unix.go: CloseExecFrom]
1 openat()
2 getdents64()
3 lstat()
4 close()
5 fcntl()
[github.com/opencontainers/runc/libcontainer/
    capabilities_linux.go: newCapWhitelist]
6 getpid()
7 capget()
[github.com/opencontainers/runc/libcontainer/system/
    linux.go: SetKeepCaps]
8 prctl()
[github.com/opencontainers/runc/libcontainer/
    init_linux.go: setupUser]
9 getuid()
10 getgid()
11 read()
[github.com/opencontainers/runc/libcontainer/
    init_linux.go: fixStdioPermissions]
12 stat()
13 fstat()
14 fchown()
[github.com/opencontainers/runc/libcontainer/
    init_linux.go: setupUser]
15 setgroups()
[github.com/opencontainers/runc/libcontainer/system/
    syscall_linux_64.go: Segid]
16 setgid()
[github.com/opencontainers/runc/libcontainer/system/
    syscall_linux_64.go: Seuid]
17 futex()
18 setuid()
[github.com/opencontainers/runc/libcontainer/
    capabilities_linux.go: drop]
19 capset()
[github.com/opencontainers/runc/libcontainer/
    init_linux.go: finalizeNamespace]
20 chdir()
[github.com/opencontainers/runc/libcontainer/
    standard_init_linux.go: Init]
21 getppid()
[github.com/opencontainers/runc/libcontainer/system/
    linux.go: Execv]
22 execve()
[github.com/docker-library/hello-world/hello.c:
    _start()]
23 write()
24 exit()
```

Fig. 3: 24 system calls accessed by *hello-world* container discovered by our approach, and functions (in `[]`) that first trigger them.

- an *exit* entry for a system call, with the properties mentioned above, except that replacing the list of arguments with return value of the system call.

*2) **Automatic testing**:* In this step, we select a test suite that covers functionality of a container. Then we run the test suite on the targeted container. During testing, we automatically copy the tracing logs at constant time intervals. This allows us to compare at what time system call was accessed. Therefore, we can monitor the growth of the sandbox rules overtime based on these snapshots.

*3) **Extract system calls**:* A script extracts the set of system calls accessed by a container from the tracing logs.

### B. Case Study

As an example of how our approach explores container behaviors, let us consider *hello-world* container [38]. This container employs a Docker image which simply prints out a message and does not accept inputs. We discover 24 system calls during testing. The actual system calls are listed in Fig. 3. Docker `init` process [39] and *hello-world* container invoke the system calls as follows:

- **SYSCALL 1** Right after the *Seccomp profile* is applied, Docker `init` process closes all unnecessary file descriptors that are accidentally inherited by accessing `openat()` , `getdents64()`, `lstat()`, `close()`, and `fcntl()`.
- **SYSCALL 6** Then Docker `init` process creates a whitelist of capabilities with the process information by accessing `getpid()` and `capget()`.
- **SYSCALL 8** Docker `init` process preserves the existing capabilities by accessing `prctl()` before changing user of the process.
- **SYSCALL 9** Docker `init` process obtains the user ID and group ID by accessing `getuid()` and `getgid()`; Later it reads the groups and password information from configuration file by accessing `read()`.
- **SYSCALL 12** Docker `init` process fixes the permissions of standard I/O file descriptors by accessing `stat()`, `fstat()`, and `fchown()`. Since these file descriptors are created outside of the container, their ownership should be fixed and match the one inside the container.
- **SYSCALL 15** Docker `init` process changes groups, group ID, and user ID for current process by accessing `setgroups()`, `setgid()`, `futex()` and `setuid()`.
- **SYSCALL 19** Docker `init` process drops all capabilities for current process except those specified in the whitelist by accessing `capset()`.
- **SYSCALL 20** Docker `init` process changes current working directory to the one specified in the configuration file by accessing `chdir()`.
- **SYSCALL 21** Docker `init` process then compares the parent process with the one from the start by accessing `getppid()` to make sure that the parent process is still alive.
- **SYSCALL 22** The final step of Docker `init` process is accessing `execve()` to execute the initial command of *hello-world* container.
- **SYSCALL 23** The initial command of *hello-world* container executes `hello` program. The `hello` program writes a message to standard output (file descriptor 1) by accessing `write()` and finally exits by accessing `exit()`.

Ideally, we expect to capture the set of system calls accessed only by the container. However, the captured set include some system calls that are accessed by Docker `init` process. This is because applying sandbox rules is a privileged operation;

Docker `init` process should apply sandbox rules before dropping capabilities. We notice that Docker `init` process invokes 22 system calls to prepare runtime environment before the container starts. If Docker `init` process accesses fewer system calls before the container starts, our mined sandboxes could be more fine-grained.

The system calls characterize the resources that *hello-world* container accesses in our run. Since the container does not accept any inputs, we find the 24 system calls are an exhausted list. The testing will be more complicated if a container accepts inputs to determine its behavior.

## V. Sandbox Enforcing

### A. Overview

The second phase of our approach is sandbox enforcing, which monitors and possibly prevents container behavior. We need a technique that conveniently allows user to sandbox any container. To this end, we leverage *Seccomp-BPF* [26] for sandbox policy enforcement. Docker uses operating system virtualization techniques, such as *namespaces*, for container-based privilege separation. *Seccomp-BPF* further establishes a restricted environment for containers, where more fine-grained security policy enforcement takes place. During sandbox enforcement, the applied *BPF* program checks whether an accessed system call is allowed by corresponding sandbox rules. If not, the system call will return an error number; or the process which invokes that system call will be killed; or a *ptrace* event [40] is generated and sent to the tracer if there exists one. This section illustrates the two steps of our approach during sandboxing phase.

*1)* ***Generate sandbox rules****:* This step translates the set of system calls discovered in mining phase into sandbox rules using *awk* tool. For instance, `write()` is one of the discovered system calls during sandbox mining for *hello-world* container. It will be translated to a sandbox rule with name `write`, action `SCMP_ACT_ALLOW`, and no constraint applied to the arguments (`args`) as follows:

```
{
    "name": "write",
        "action": "SCMP_ACT_ALLOW",
        "args": []
}
```

When the system call `write()` is accessed during sandboxing, it will be permitted according to the specified action, i.e., `SCMP_ACT_ALLOW`. After translating each system call entry into a sandbox rule, these rules constitute a whitelist of system calls that are allowed by the sandbox. We define the default action of the sandbox as follows:

```
"defaultAction": "SCMP_ACT_ERRNO"
```

During sandboxing, when a container accesses a system call which is not included in the whitelist, the sandbox will deny the system call and make the system call return an error number (`SCMP_ACT_ERRNO`).

*2)* ***Enforce sandbox rules****:* The resulting *Seccomp profile* now contains all sandbox rules that allow the system calls observed in the mining phase. We then start the container with the *Seccomp profile* to enforce mined sandbox rules using `docker run --security-opt seccomp`.

### B. Case Study

As an example of how our approach operates, consider *hello-world* container again. The default Docker sandbox allows more than 300 system calls, which is a considerable attack surface. In that default setting, a compromised *hello-world* container could simply mount a directory that contains a carefully crafted program. The program could open a socket by accessing system call `socket()`, which is an abnormal behavior. By enforcing our mined sandbox, *hello-world* container is not allowed to access `socket()`. Thus we prevent the container from opening a socket and doing further harm.

## VI. Experiments

### A. Overview

In this section, we evaluate our approach to answer three research questions as follows:

**RQ1. How efficiently can our approach mine sandboxes?**

We evaluate how fast the sets of system calls are saturated for eight containers. Notice that the eight containers are the most popular containers in Docker Hub [41] and have a large number of downloads. The details of them are shown in TABLE I. The eight containers can be used in PaaS, and provide domain-specific functions rather than basic functions provided by OS containers (e.g. *Ubuntu* container). Note that *python* as a programming language provides a wide range of functionality, and *python* container can potentially access all system calls. Mining sandbox for *python* container will be useless because the mined sandbox will be too coarse. Thus we setup Web framework *Django* [42] on top of *python* container. This makes *python* container have specific functionality. In addition, we compare the mined sandboxes with the default one provided by Docker to see if the attack surface is reduced.

**RQ2. Can automatic testing sufficiently cover behaviors?**

Any non-malicious system call behavior not explored during testing implies a false alarm during production. We evaluate the risk of false alarms: how likely is it that sandbox mining misses functionality, and how frequently will containers encounter false alarms. We check the mined sandboxes of the eight containers against the use cases. We carefully read the documentation of the containers to make sure the use cases reflect the containers' typical usage.

**RQ3. What is the performance overhead of sandbox enforcement?**

As a security mechanism, the performance overhead of sandbox enforcement should be small. Instead of CPU time, we measure the end-to-end performance of containers – *transactions per second*. We compare the end-to-end performance of a container running in three environments: 1) natively without sandbox, 2) with a sandbox mined by our approach, and 3) with default Docker sandbox.

TABLE I: Experiment subjects. Open https://hub.docker.com/_/<identifier> for details.

| Name | Version | Description | Stars | Pulls | Identifier (links to Web page) |
|---|---|---|---|---|---|
| Nginx | 1.11.1 | Web server | 3.8K | 10M+ | nginx |
| Redis | 3.2.3 | key-value database | 2.5K | 10M+ | redis |
| MongoDB | 3.2.8 | document-oriented database | 2.2K | 10M+ | mongo |
| MySQL | 5.7.13 | relational database | 2.9K | 10M+ | mysql |
| PostgreSQL | 9.5.4 | object-relational database | 2.5K | 10M+ | postgres |
| Node.js | 6.3.1 | Web server | 2.6K | 10M+ | node |
| Apache | 2.4.23 | Web server | 606 | 10M+ | httpd |
| Python | 3.5.2 | programming language | 1.1K | 5M+ | python |

The containers in the experiments run on a 64-bit Ubuntu 16.04 operating system inside VirtualBox 5.0.24 (4GB base memory, 2 processors). The physical machine is with an Intel Core i5-6300 processor and 8GB memory.

*1) Testing:* We describe the test suites we run in the experiment as follows:

**Web server (Nginx, Apache, Node.js, and Python Django).** After executing `docker run`, each container experiences a warm-up phase which lasts for 30 seconds. After the warm-up phase, the Web server gets ready to serve requests. We remotely start with a simple HTTP request using *wget* tool from another virtual machine. The request fetches a file from the server right after the warm-up phase. It is followed by a number of runs of *httperf* tool [43] also from that virtual machine. *httperf* continuously accesses the static pages hosted by the container. The workload starts from 5 requests per second, increases the number of requests by 5 for every run, and ends at 50 requests per second.

**Redis.** The warm-up phase of *Redis* container lasts for 30 seconds. After the warm-up phase, we locally connect to the *Redis* container via `docker exec`. Then we run the built-in benchmark test *redis-benchmark* [44] with the default configuration, i.e., 50 parallel connections, totally 100,000 requests, 2 bytes of SET/GET value, and no pipeline. The test cases cover the commands as follows:

- **PING**: checks the bandwidth and latency.
- **MSET**: replaces multiple existing values with new values.
- **SET**: sets a key to hold the string value.
- **GET**: gets the value of some key.
- **INCR**: increments the number stored at some key by one.
- **LPUSH**: inserts all the specified values at the head of the list.
- **LPOP**: removes and returns the first element of the list.
- **SADD**: adds the specified members to the set stored at some key.
- **SPOP**: removes and returns one or more random elements from the set value.
- **LRANGE**: returns the specified elements of the list.

**MongoDB.** The warm-up phase of *MongoDB* container lasts for 30 seconds. After the warm-up phase, we run *mongo-perf* [45] tool to connect to *MongoDB* container remotely from another virtual machine. *mongo-perf* measures the throughput of *MongoDB* server. We run each of the test cases in *mongo-perf* with tag *core*, on 1 thread, and for 10 seconds. The detail of test cases is described as follows:

- **insert document**: inserts documents only with object ID into collections.
- **update document**: randomly selects a document using object ID and increments one of its integer field.
- **query document**: queries for a random document in the collections based on an indexed integer field.
- **remove document**: removes a random document using object ID from the collections.
- **text query**: runs case-insensitive single-word text query against the collections.
- **geo query**: runs *nearSphere* query with *geoJSON* format and two-dimensional sphere index.

**MySQL.** The warm-up phase of *MySQL* container lasts for 30 seconds. After the warm-up phase, we create a database, and use *sysbench* [46] tool to connect to *MySQL* container. We then run the *OLTP* database test cases in *sysbench* with maximum request number of 800, on 8 threads for 60 seconds. The test cases include the following functionalities:

- **create database**: creates a database *test*.
- **create table**: creates a table *sbtest* in the database.
- **insert record**: inserts 1,000,000 records into the table.
- **update record**: updates records on indexed and non-indexed columns.
- **select record**: selects records with a record ID and a range for record ID.
- **delete records**: deletes records with a record ID.

**PostgreSQL.** The warm-up phase of *PostgreSQL* container lasts for 30 seconds. After the warm-up phase, we connect to *PostgreSQL* container using *pgbench* [47] tool. We first run *pgbench* initialization mode to prepare the data for testing. The initialization is followed by two 60-second runs of read/write test cases with queries. The test cases cover the functionalities as follows:

- **create database**: creates a database `pgbench`.
- **create table**: creates four tables in the database, namely `pgbench_branches`, `pgbench_tellers`, `pgbench_accounts`, and `pgbench_history`.
- **insert record**: inserts 15, 150 and 1,500,000 records into the aforementioned tables expect `pgbench_history` respectively.
- **update and select record**: executes *pgbench* built-in TPC-B-like transaction with prepared and ad-hoc queries: updating records in table `pgbench_branches`, `pgbench_tellers`,and `pgbench_accounts`, and then doing queries, finally inserting a record into table `pgbench_history`.

*2) Statistics:* During sandbox mining, the eight containers execute approximately 5,340,000 system calls. The number of

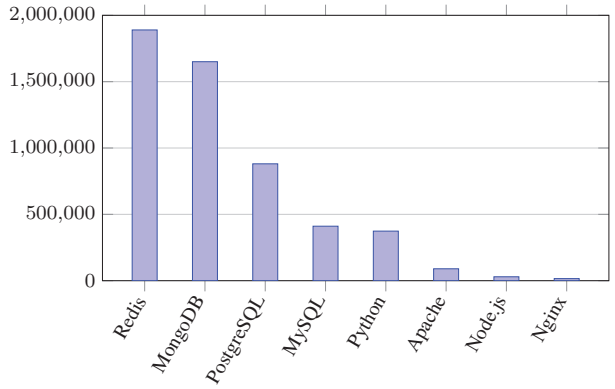


Fig. 4: Number of system call execution of the containers.

system call execution of the eight containers is shown in Fig. 4. We can see that the number of system call execution goes to thousands or even millions. Thus tracing and analyzing system calls on a real-time environment will cause a considerate performance penalty. To achieve low performance penalty, we only trace and analyze system calls in sandbox mining phase. A decomposition of the most frequent system calls of each container is shown in Fig. 5. The system call with the highest frequency is `recvfrom()` which is used to receive a message from a socket. The corresponding system call `sendto()` which is used to send a message on a socket has high frequency as well. The system calls that monitor multiple file descriptors are also prominent, such as `epoll_ctl()` and `epoll_wait()`. System calls that access filesystem are also executed frequently, such as `read()` and `write()`.

### B. Growth of System Calls

Fig. 6 shows the system call saturation charts for the eight containers. We can see that six charts "flatten" before one minute mark, and the remaining two before two minutes. Our approach has discovered 76, 74, 98, 105, 99, 66, 73, and 74 system calls accessed by *Nginx*, *Redis*, *MongoDB*, *MySQL*, *PostgreSQL*, *Node.js*, *Apache*, and *Python Django* containers respectively. The number of accessed system calls is far less than 300+ of the default Docker sandbox. The attack surface is significantly reduced.

During the warm-up phase, the number of system calls accessed by each of the containers grows rapidly. After the warm-up phase, for all of the Web servers except *Apache*, the simple HTTP request causes a further increase and the number of system calls converges; for *Apache* container, *httperf* causes a small increase and the number of system calls shows no change later. For *Redis* container, connecting to the container via `docker exec` causes a first increase after the warm-up phase; and later *redis-benchmark* triggers a small increase. For *MongoDB*, *MySQL* and *PostgreSQL* containers, *mongo-perf*, *sysbench* and *pgbench* cause a small increase after the warm-up phase.

The answer of **RQ1** is: our approach can mine the saturated set of system calls within two minutes. The mined sandboxes reduce the attack surface.

*Sandbox mining quickly saturates accessed system calls.*

### C. False Alarm

*1) Use cases:* Our approach stops discovering new accessed system calls before the testing ends. However, does this mean that the most important functionality of a container is actually found? To answer this question, we carefully read the documentation of the containers and prepared *use cases* which reflect containers' typical usages. TABLE II provides a full list of the use cases. We implemented all of these use cases as automated `bash` test cases, allowing for easy assessment and replication.

After mining the sandbox for a given container, the central question for the evaluation is whether these use cases would be impacted by the sandbox, i.e., a benign system call would be denied during sandbox enforcing. To recognize the impact of sandbox, we set the default action of sandboxes to be `SCMP_ACT_KILL` in the experiment. When the mined sandbox denies a system call, the process which accesses the system call will be killed, and *auditd* [48] will log a message of type `SECCOMP` for the failed system call. Note that the default action of our mined sandboxes is `SCMP_ACT_ERRNO` in production.

*2) Results:* The "Messages in *auditd*" column in TABLE II summarizes the number of messages logged by *auditd*. We can see that no message is logged by *audid* for the 30 use cases. The number of false alarm is zero.

The answer of **RQ2** is: we did not find any impact from the mined sandboxes on the regular functionalities of the containers. Even automatic testing of a small workload is suitable to cover sufficient "normal" behaviors for the use cases in TABLE II.

*Mined sandboxes require no further adjustment on use cases.*

### D. Performance Evaluation

To analyze the performance characteristics of our approach, we run the eight containers in three environments: 1) natively without sandbox as a baseline, 2) with a sandbox mined by our approach, and 3) with the default Docker sandbox. We measure the throughput of each container as an end-to-end performance metric. To minimize the impact of network, we run each of the containers using host networking via `docker run --net=host`. We repeat each experiment 10 times with a less than 5% standard deviation.

For *Redis*, *MongoDB*, *PostgreSQL* and *MySQL* containers, we evaluate the *transactions per second* (TPS) of each container by running the aforementioned tools in Section VI-B. The percentage reduction of TPS per container for *Redis*, *MongoDB*, *PostgreSQL* and *MySQL* is presented in Fig. 7. We notice that enforcing mined sandboxes incurs a small TPS reduction (0.6% - 2.14%) for the four containers. Mined sandboxes produce a slightly smaller TPS reduction than that of the default sandbox (0.83% - 4.63%). The reason is that the default sandbox contains more rules than mined

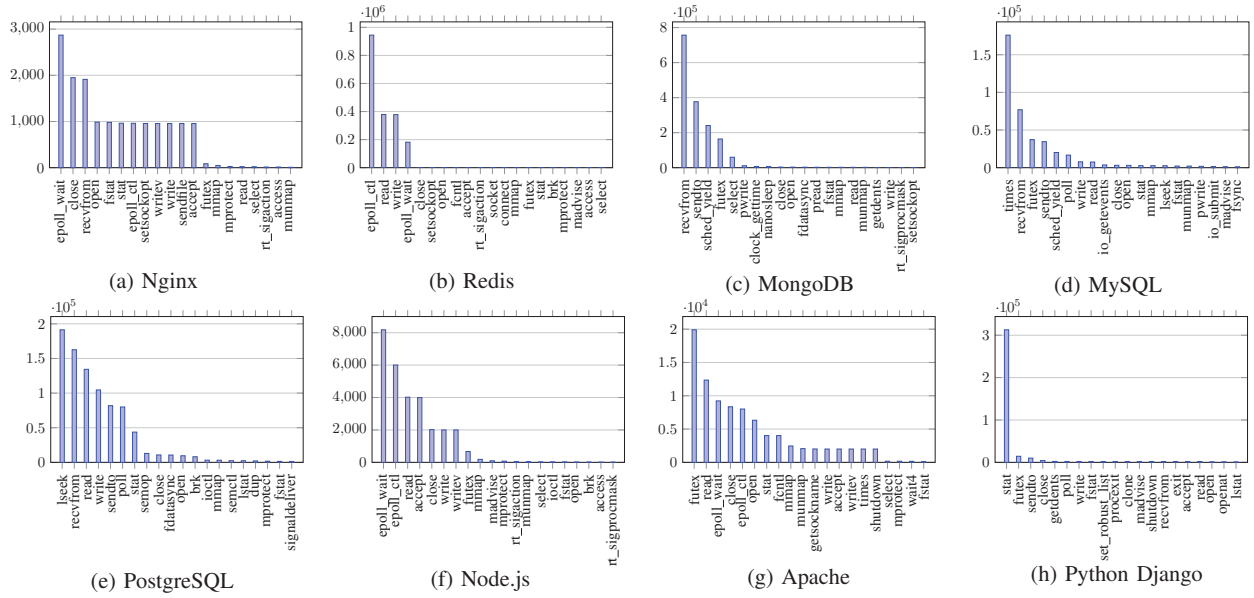


Fig. 5: Histogram of system call frequency for each of the containers.

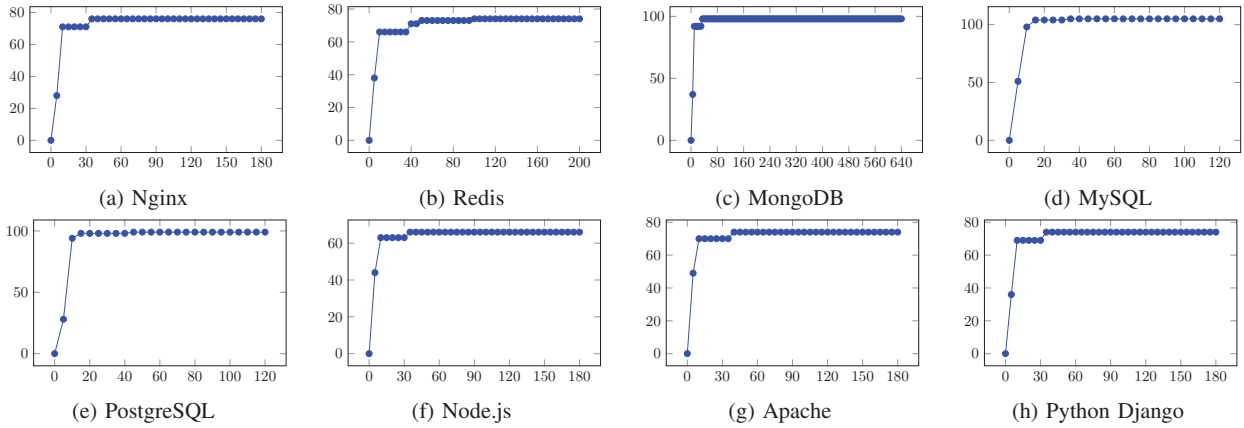


Fig. 6: Per-container system call saturation for the containers in TABLE I. y axis is the number of accessed system calls, x axis is seconds spent.

sandboxes, and thus the corresponding *BFP program* needs more computation during sandboxing.

For Web server containers, we evaluate the throughput, i.e., *responses per second*, of each container by running *httperf* tool. To measure the response rate of each container, we increase the number of requests per second that are sent to the container. The result is shown in Fig. 8. Web server containers running with sandboxes achieve a performance very similar to that of the containers running without sandboxes. We can see that the achieved throughput increases linearly with offered load until the container starts to become saturated. The saturation points of *Nginx*, *Node.js*, *Apache* and *Python Django* are around 7,000, 3,000, 2,500 and 300 requests per second respectively. After offered load is increased beyond that point, the response rate of the container starts to fall off slightly.

The answer of **RQ3** is: enforcing system call policies adds overhead to a container's end-to-end performance, but the overall increase is small.

*Sandboxes incur a small end-to-end performance overhead.*

## VII. Threats and Limitations

System call access is either benign or malicious. Our approach automatically decides on whether a system call accessed by a container should be allowed. As we do not assume a specification of what makes a benign or malicious system call access for a container, we face two risks:

- **False positive.** A *false positive* occurs when a benign system call is mistakenly prohibited by the sandbox, degrading a container's functionality. In our setting, a false alarm happens if some benign system call is not seen during mining phase, and thus not added to sandbox rules to be allowed. The number of false alarms can be reduced by better testing.

TABLE II: Use cases. *auditd* logs a message when a system call is denied by the sandbox.

| Container | Use Case | Functions | Messages in *auditd* |
|---|---|---|---|
| Nginx | Access static page | Access default page `index.html`, `50x.html` | - |
| | Access non-existent page | Access non-existent page `hello.html` | - |
| Redis | SET command | Connect to Redis server, set key to hold the string value | - |
| | GET command | Connect to Redis server, get the value of key | - |
| | INCR command | Connect to Redis server, increment the number stored at key by one | - |
| | LPUSH command | Connect to Redis server, insert all the specified values at the head of the list stored at key. | - |
| | LPOP command | Connect to Redis server, remove and returns the first element of the list stored at key | - |
| | SADD command | Connect to Redis server, add the specified members to the set stored at key | - |
| | SPOP command | Connect to Redis server, remove and return one or more random elements from the set value store at key | - |
| | LRANGE command | Connect to Redis server, return the specified elements of the list stored at key | - |
| | MSET command | Connect to Redis server, replace multiple existing values with new values | - |
| MongoDB | insert | Connect to mongod, use database `test`, insert record `{image:"redis",count:"1"}` into collection `falsealarm`, exit | - |
| | save | Connect to mongod, use database `test`, update record in collection `falsealarm`, exit | - |
| | find | Connect to mongod, use database `test`, list all records in collection `falsealarm`, exit | - |
| MySQL | CREATE DATABASE | Connect to MySQL server, create database `test`, list all databases, exit | - |
| | CREATE TABLE | Connect to MySQL server, use database `test`, create table `FalseAlarm`, insert record, exit | - |
| | INSERT | Connect to MySQL server, use database `test`, insert record into table `FalseAlarm`, exit | - |
| | UPDATE | Connect to MySQL server, use database `test`, update record, exit | - |
| | SELECT | Connect to MySQL server, use database `test`, list all records, exit | - |
| PostgreSQL | CREATE DATABASE | Connect to PostgreSQL server, create database `test`, list all databases, exit | - |
| | CREATE TABLE | Connect to PostgreSQL server, connect to database `test`, create table `FalseAlarm`, exit | - |
| | INSERT | Connect to PostgreSQL server, connect to database `test`, insert record into table `FalseAlarm`, exit | - |
| | UPDATE | Connect to PostgreSQL server, connect to database `test`, update record in table `FalseAlarm`, exit | - |
| | SELECT | Connect to PostgreSQL server, connect to database `test`, list all records in table `FalseAlarm`, exit | - |
| Node.js | Access existent URI | Access `/` | - |
| | Access non-existent URI | Access non-existent URI `/hello` | - |
| Apache | Access static page | Access default page `index.html` | - |
| | Access non-existent page | Access non-existent page `hello.html` | - |
| Python Django | Access existent URI | Access `/` | - |
| | Access non-existent URI | Access non-existent URI `/hello` | - |

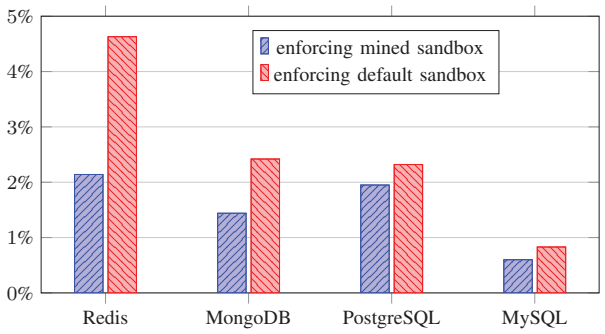


Fig. 7: Percentage reduction of transactions per second (TPS) due to sandboxing.

- **False negative.** A *false negative* occurs when a malicious system call is mistakenly allowed by the sandbox. In our setting, a false alarm can happen in two ways:
  - **False negative allowed during sandbox enforcing.** The inferred sandbox rules may be too coarse, and thus allow future malicious system calls. For instance, a container may access system calls `mmap()`, `mprotect()` and `munmap()` as benign behaviors. However, *code injection* attack could also invoke these system calls to change memory protection. This issue can be addressed by inferring more fine-grained sandbox rules.
  - **False negative seen during sandbox mining.** The container may be initially malicious. We risk to mine the malicious behaviors of the container during mining phase. Thus malicious system calls would be included in the sandbox rules. This issue can be addressed by identifying malicious behaviors during mining phase.

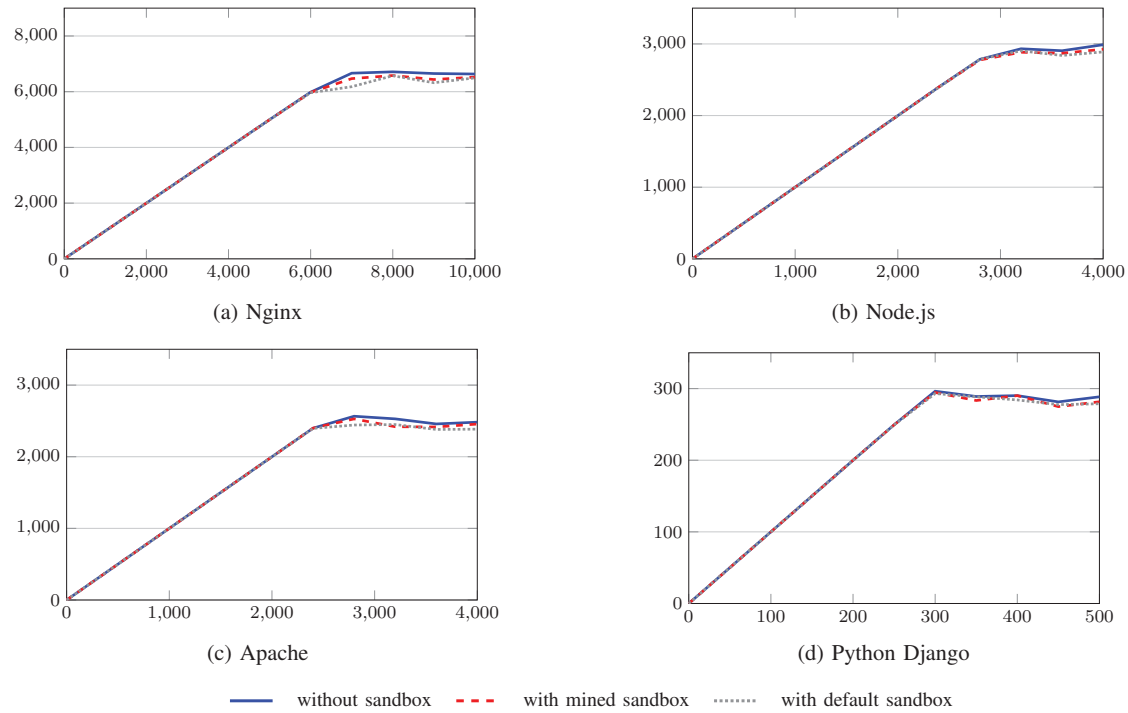


Fig. 8: Comparison of per-container reply rate for Nginx, Node.js, Apache, and Python Django without sandbox, with sandbox mined by our approach, and with default sandbox. y axis is response rate (responses per second), x axis is request rate (requests per second).

Although our experimental results demonstrate the feasibility of sandbox mining for containers, our sample of containers is small and the containers are database systems and Web servers. For other containers, we have to design different testing. In addition, some containers may comprise multiple processes which have distinct responsibilities, for instance, a Linux, Apache, MySQL and PHP (LAMP) stack in one container. This may increase attack surface, and lead to more false negatives.

The set of use cases we have prepared for assessing the risk of false alarms (TABLE II) does not and cannot cover the entire range of functionalities of the analyzed containers. Although we assume that the listed user cases represent the most important functionalities, other usage may yield different results.

Finally, in the absence of a specification, a mined policy cannot express whether a system call is benign or malicious. Although our approach cannot eliminate the risks of false positives and false negatives, we do reduce the attack surface by detecting and preventing unexpected behavior.

## VIII. Conclusion and Future Work

In this paper, we present an approach to mine sandboxes for Linux containers. The approach explores the behaviors of a container by automatically running test suites. From the execution trace, the approach extracts set of system calls accessed by the container during the mining phase, and translates the system calls into sandbox rules. During sandbox enforcement, the mined sandbox confines the container by restricting its access to system calls. Our evaluation shows that our approach can efficiently mine sandboxes for containers and substantially reduce the attack surface. In our experiment, automatic testing sufficiently covers container behaviors and sandbox enforcement incurs low overhead.

In the future, we would like to mine more fine-grained sandbox policy, taking into account the system call arguments, temporal features of system calls, internal states of a container, or data flow from and to sensitive resources. More Fine-grained sandbox may lead to more false positives and increase performance overhead. We have to search for sweet spots that both minimize false positives and performance overhead. Meanwhile, we have to avoid *Time-of-check-to-time-of-use* (TOCTTOU) problems when examining system call arguments. We also plan to leverage modern test case generation techniques to systematically explore container behaviors. This may help to cover more normal behaviors of a container. In addition, for now we enforce one system call policy on a whole container. Whereas a container may comprise multiple processes which have distinct behaviors. To further reduce the attack surface, We could enforce a distinct policy for each process which corresponds to the behavior of that process.

## Acknowledgment

We thank the anonymous reviewers for their insightful comments. This research is supported by NSFC Program (No. 61602403), and National Key Technology R&D Program of the Ministry of Science and Technology of China (No. 2015BAH17F01).

## REFERENCES


[1] G. I. A. Inc., "Platform as a Service PaaS Market Trends," http://www.strategyr.com/MarketResearch/Platform_as_a_Service_PaaS_Market_Trends.asp, 2015, [Online; accessed 2016-08-16].

[2] D. Merkel, "Docker: lightweight linux containers for consistent development and deployment," *Linux Journal*, vol. 2014, no. 239, p. 2, 2014.

[3] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS 2015)*. IEEE, 2015, pp. 171–172.

[4] "CVE-2016-0728," http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728, [Online; accessed 2016-08-16].

[5] T. Garfinkel *et al.*, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *Network and Distributed System Security Symposium (NDSS 2003)*, vol. 3, 2003, pp. 163–176.

[6] I. Goldberg, D. Wagner, R. Thomas, E. A. Brewer *et al.*, "A secure environment for untrusted helper applications: Confining the wily hacker," in *Proceedings of the Conference on USENIX Security Symposium*, 1996.

[7] N. Provos, "Improving host security with system call policies," in *Proceedings of the Conference on USENIX Security Symposium*, 2003.

[8] A. Acharya and M. Raje, "Mapbox: Using parameterized behavior classes to confine untrusted applications," in *Proceedings of the conference on USENIX Security Symposium*. USENIX Association, 2000.

[9] T. Fraser, L. Badger, and M. Feldman, "Hardening cots software with generic software wrappers," in *Proceedings 1999 IEEE Symposium on Security and Privacy (S&P 1999)*. IEEE, 1999, pp. 2–16.

[10] C. Ko, T. Fraser, L. Badger, and D. Kilpatrickv, "Detecting and countering system intrusions using software wrappers," in *Proceedings of the Conference on USENIX Security Symposium*, 2000, pp. 1157–1168.

[11] T. Kim and N. Zeldovich, "Practical and effective sandboxing for non-root users," in *Proceedings of the Conference on USENIX Annual Technical Conference (USENIX ATC 13)*, 2013, pp. 139–144.

[12] K. Jamrozik, P. von Styp-Rekowsky, and A. Zeller, "Mining sandboxes," in *Proceedings of the 38th International Conference on Software Engineering (ICSE 2016)*. ACM, 2016, pp. 37–48.

[13] "Seccomp security profiles for Docker," https://docs.docker.com/engine/security/seccomp, [Online; accessed 2016-08-16].

[14] T. Garfinkel, B. Pfaff, M. Rosenblum *et al.*, "Ostia: A delegating architecture for secure system call interposition," in *Network and Distributed System Security Symposium (NDSS 2004)*, 2004.

[15] D. A. Wagner, "Janus: an approach for confinement of untrusted applications," Ph.D. dissertation, Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, 1999.

[16] K. Jain and R. Sekar, "User-level infrastructure for system call interposition: A platform for intrusion detection and confinement," in *Network and Distributed System Security Symposium (NDSS 2000)*, 2000.

[17] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion detection using sequences of system calls," *Journal of computer security*, vol. 6, no. 3, pp. 151–180, 1998.

[18] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for unix processes," in *Proceedings 1996 IEEE Symposium on Security and Privacy (S&P 1996)*. IEEE, 1996, pp. 120–128.

[19] D. Wagner and R. Dean, "Intrusion detection via static analysis," in *Proceedings 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, 2001, pp. 156–168.

[20] S. Bhatkar, A. Chaturvedi, and R. Sekar, "Dataflow anomaly detection," in *Proceedings 2006 IEEE Symposium on Security and Privacy (S&P 2006)*. IEEE, 2006, pp. 15–pp.

[21] V. Kiriansky, D. Bruening, S. P. Amarasinghe *et al.*, "Secure execution via program shepherding," in *Proceedings of the Conference on USENIX Security Symposium*, vol. 92, 2002, p. 84.

[22] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting intrusions using system calls: Alternative data models," in *Proceedings 1999 IEEE Symposium on Security and Privacy (S&P 1999)*. IEEE, 1999, pp. 133–145.

[23] A. Somayaji and S. Forrest, "Automated response using system-call delay," in *Proceedings of the Conference on USENIX Security Symposium*, 2000, pp. 185–197.

[24] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A fast automaton-based method for detecting anomalous program behaviors," in *Proceedings 2001 IEEE Symposium on Security and Privacy (S&P 2001)*. IEEE, 2001, pp. 144–155.

[25] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Transactions on Information and System Security (TISSEC)*, vol. 9, no. 1, pp. 61–93, 2006.

[26] "Yet another new approach to seccomp," http://lwn.net/Articles/475043, [Online; accessed 2016-08-16].

[27] "Seccomp and sandboxing," http://lwn.net/Articles/475043, [Online; accessed 2016-08-16].

[28] "JSON," http://www.json.org, [Online; accessed 2016-08-16].

[29] J. H. Saltzer and M. D. Schroeder, "The protection of information in computer systems," *Proceedings of the IEEE*, vol. 63, no. 9, pp. 1278–1308, 1975.

[30] C. Kruegel, D. Mutz, F. Valeur, and G. Vigna, "On the detection of anomalous system call arguments," in *European Symposium on Research in Computer Security (ESORICS 2003)*. Springer, 2003, pp. 326–343.

[31] C. Fetzer and M. Süßkraut, "Switchblade: enforcing dynamic personalized system call models," in *ACM SIGOPS Operating Systems Review*, vol. 42, no. 4. ACM, 2008, pp. 273–286.

[32] D. Gao, M. K. Reiter, and D. Song, "Behavioral distance measurement using hidden markov models," in *International Workshop on Recent Advances in Intrusion Detection*. Springer, 2006, pp. 19–40.

[33] D. Endler, "Intrusion detection. applying machine learning to solaris audit data," in *Proceedings of the 14th Annual Computer Security Applications Conference (ACSAC 1998)*. IEEE, 1998, pp. 268–279.

[34] Y. Liao and V. R. Vemuri, "Use of k-nearest neighbor classifier for intrusion detection," *Computers & Security*, vol. 21, no. 5, pp. 439–448, 2002.

[35] A. Zeller, "Test complement exclusion: Guarantees from dynamic analysis," in *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension (ICPC 2015)*. IEEE Press, 2015, pp. 1–2.

[36] S. Whalen, "An introduction to arp spoofing," *Node99 [Online Document], April*, 2001.

[37] "sysdig," http://www.sysdig.org, [Online; accessed 2016-08-16].

[38] "hello-world," https://hub.docker.com/_/hello-world, [Online; accessed 2016-08-16].

[39] "runc libcontainer version 0.1.1," https://github.com/opencontainers/runc/blob/v0.1.1/libcontainer/standard_init_linux.go, [Online; accessed 2016-08-16].

[40] "Ptrace documentation," https://lwn.net/Articles/446593, [Online; accessed 2016-08-16].

[41] "Docker Hub," https://hub.docker.com/explore, [Online; accessed 2016-08-16].

[42] "Django: a high-level Python Web framework," https://www.djangoproject.com, [Online; accessed 2016-08-16].

[43] D. Mosberger and T. Jin, "httperf: a tool for measuring web server performance," *ACM SIGMETRICS Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[44] "How fast is Redis?" http://redis.io/topics/benchmarks, [Online; accessed 2016-08-16].

[45] "Mongo-perf," https://github.com/mongodb/mongo-perf, [Online; accessed 2016-08-16].

[46] "SysBench," https://github.com/akopytov/sysbench, [Online; accessed 2016-08-16].

[47] "pgbench," https://www.postgresql.org/docs/9.3/static/pgbench.html, [Online; accessed 2016-08-16].

[48] "auditd," http://linux.die.net/man/8/auditd, [Online; accessed 2016-08-16].