

Singapore Management University

Institutional Knowledge at Singapore Management University

Research Collection School Of Information
Systems

School of Information Systems

7-2019

Practical and effective sandboxing for Linux containers

Zhiyuan WAN

David LO

Singapore Management University, davidlo@smu.edu.sg

Xin XIA

Liang CAI

Follow this and additional works at: https://ink.library.smu.edu.sg/sis_research



Part of the [Programming Languages and Compilers Commons](#), and the [Software Engineering Commons](#)

Citation


WAN, Zhiyuan; LO, David; XIA, Xin; and CAI, Liang. Practical and effective sandboxing for Linux containers. (2019). *Empirical Software Engineering*. 24, 4034-4070. Research Collection School Of Information Systems.

Available at: https://ink.library.smu.edu.sg/sis_research/4502

This Journal Article is brought to you for free and open access by the School of Information Systems at Institutional Knowledge at Singapore Management University. It has been accepted for inclusion in Research Collection School Of Information Systems by an authorized administrator of Institutional Knowledge at Singapore Management University. For more information, please email library@smu.edu.sg.



Practical and effective sandboxing for Linux containers

Zhiyuan Wan^{1,2,3} · David Lo⁴ · Xin Xia⁵  · Liang Cai^{1,3}

Published online: 04 July 2019

© Springer Science+Business Media, LLC, part of Springer Nature 2019

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 the system call interface. In this paper, we present an approach that mines sandboxes and enables fine-grained sandbox enforcement for containers. We first explore the behavior of a container by running test cases and monitor the accessed system calls including types and arguments during testing. We then characterize the types and arguments of system call invocations and translate them into sandbox rules for 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 a sandbox for each of the containers. The estimation of system call coverage of sandbox mining ranges from 96.4% to 99.8% across the containers under the limiting assumptions that the test cases are complete and only static system/application paths are used. The enforcement of mined sandboxes incurs low performance overhead. The mined sandboxes effectively reduce the attack surface of containers and can prevent the containers from security breaches in reality.

Keywords Container · System call · Sandbox · Testing · Monitoring · Cloud computing · Docker · Seccomp

Communicated by: Antonia Bertolino

✉ Xin Xia
xin.xia@monash.edu

Zhiyuan Wan
wanzhiyuan@zju.edu.cn

David Lo
davidlo@smu.edu.sg

Liang Cai
leoncai@zju.edu.cn

¹ College of Computer Science and Technology, Zhejiang University, Hangzhou, China

² Department of Computer Science, University of British Columbia, Vancouver, Canada

³ Alibaba-Zhejiang University Joint Institute of Frontier Technologies, Hangzhou, China

⁴ School of Information Systems, Singapore Management University, Singapore, Singapore

⁵ Faculty of Information Technology, Monash University, Melbourne, Australia

1 Introduction

Platform-as-a-Service (PaaS) cloud is a fast-growing segment of the cloud market, projected to reach \$7.5 billion by 2020 (GlobalIndustryAnalystsInc 2015). 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. The operating system level virtualization gives users the appearance of having their own operating system with near-native performance and no additional virtualization overhead. Container technologies, such as Docker (Merkel 2014), enable easy packaging and rapid deployment of applications.

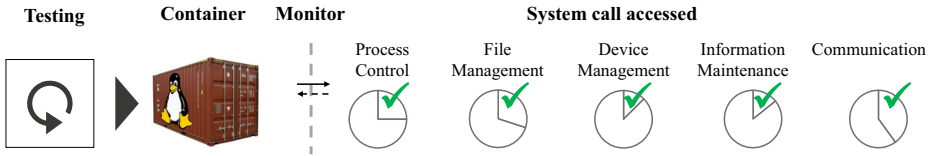
A number of security mechanisms have been proposed or adopted to enhance container security, e.g., *CGroup* (Menage 2004), *Seccomp* (Corbet 2009), *Capabilities* (Hallyn and Morgan 2008), *AppArmor* (Cowan 2007) and *SELinux* (McCarty 2005). Related works leverage these security mechanisms and propose an extension to enhance container security. For example, Mattetti et al. (2015) propose a LiCSHield framework which traces the operations of a container, and constructs an AppArmor profile for the container.

The primary source of security problems in containers is system calls that are not namespace-aware (Felter et al. 2015). Non-namespace-aware system call interface facilitates the adversary to compromise applications running in containers and further 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 (CVE-2016-0728 2016).

How can cloud providers protect clouds from exploitable containers? One straightforward way is to place each of the containers 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 (Garfinkel et al. 2003). Sandboxing techniques based on system call interposition have been developed in the past (Goldberg et al. 1996; Provos 2003; Acharya and Raje 2000; Fraser et al. 1999; Ko et al. 2000; Kim and Zeldovich 2013). Most of them focus on implementing sandboxing techniques and ensuring secure system call interposition. However, generating accurate sandbox policies for a program is always challenging (Provos 2003). We are inspired by a recent work *BOXMATE* (Jamrozik et al. 2016), 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 intend to port the idea of *sandbox mining* in *BOXMATE* to be able to confine Linux containers.

A container comprises multiple processes of different functionalities that access distinct system calls. Thus different containers may present different behaviors on a system call level. A common sandbox for all the containers is too coarse. In this paper, we present an approach to *automatically mine sandbox rules* and enable fine-grained sandbox policy enforcement for a given container. The approach is composed of two phases shown in Fig. 1:

1. Sandbox mining



2. Sandbox enforcing

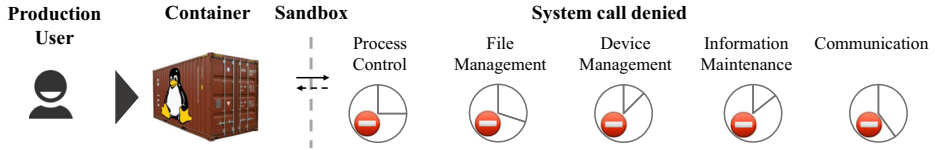


Fig. 1 Our approach in a nutshell. Mining phase monitors accessed system calls when testing. These system calls make up a *sandbox* for the container, which later prohibits access to system calls not accessed during testing

- **Sandbox mining.** In the first phase, we mine sandbox rules for a container. Specifically, we use automatic testing to explore behaviors of a container, monitor all accesses to system calls, and capture types and arguments of the system calls.
- **Sandbox enforcing.** In the second phase, we assume system call behavior that does not appear during the mining phase should not appear in production either. Consequently, during sandbox enforcing, if the container requires access to system calls in an unexpected way, the sandbox will prohibit the access.

To the best of our knowledge, our approach is the first technique that leverages automatic testing to mine 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:

- **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 the Docker containers and check the system call convergence. It takes less than two minutes for the set of accessed system calls to saturate for the selected static test cases. Also, we compare our mined sandboxes with the default sandbox

provided by Docker. The default sandbox allows more than 300 system calls (DockerDocs 2017) 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. How sufficient does sandbox mining cover system call behaviors?

We estimate the system call coverage for sandbox mining by using 10-fold cross-validation. If a system call S is not accessed during the mining phase, later non-malicious access to S would trigger a false alarm. We further run use cases that cover the basic functionality of containers to check whether the enforcing mined sandboxes would trigger alarms. The result shows that the estimation of system call coverage for sandbox mining ranges from 96.4% to 99.8% across the container, and the use cases end with no false alarms. A limiting assumption is that the use cases only tested static system/application paths and included test cases during sandbox mining.

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 a low end-to-end performance overhead. Our mined sandboxes also provide a slightly lower performance overhead than that of the default sandbox.

RQ4. Can the mined sandboxes effectively protect an exploitable application running in a container?

We analyze how our mined sandboxes can protect an exploitable application by reducing the attack surface. In addition, we conduct a case study by considering a security vulnerability in reality (CVE-2013-2028 in *Nginx* 1.3.9-1.4.0). We attempt to understand if enforcing mined sandboxes could prevent exploits of the vulnerability. The result shows that our mined sandboxes can effectively protect an exploitable application running in a container, and prevent security breaches in reality. A threat to validity is that the automatic test cases for the *Nginx* container only achieve code coverage of 13.7%, so there might be a significant number of false alarms in practice.

This paper extends our preliminary work which appears as a research paper of ICST 2017 (Wan et al. 2017). In particular, we extend our preliminary work in several directions: (1) In addition to system call types, we characterize the arguments of system calls and translate the characteristics into fined-grained sandbox rules; (2) To enable fine-grained sandbox enforcement, we leverage *seccomp-BPF* to intercept system calls and *ptrace* interface to examine the arguments of system call invocations; (3) We have repeated experiments for our mined fine-grained sandboxes to answer three research questions in our ICST 2017 paper; (4) We further address RQ4 to evaluate the effectiveness of our mined sandboxes to protect exploitable containers.

The remainder of this paper is organized as follows. After discussing background and related work in Section 2, Section 3 specifies the threat model and motivation of our work. Sections 4 and 5 detail two phases of our approach. We evaluate our approach in Section 6 and discuss threats to validity and limitations in Section 7. Finally, Section 8 closes with conclusion and future work.

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

```
{
  "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": []
    },
    ...
  ]
}
```

2 Background and Related Work

2.1 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 (Garfinkel et al. 2003).

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 (Garfinkel et al. 2003). Garfinkel studies the common mistakes and pitfalls, and uses the system call interposition technique to enforce security policies in the Ostia tool (Garfinkel et al. 2004). System call interposition tools, such as Janus (Goldberg et al. 1996; Wagner 1999), Systrace (Provos 2003), and ETrace (Jain and Sekar 2000), can enforce fine-grained policies at granularity of the operating system's system call infrastructure. System call interposition is also used for sandboxing (Goldberg et al. 1996; Provos 2003; Acharya and Raje 2000; Fraser et al. 1999; Ko et al. 2000; Kim and Zeldovich 2013) and intrusion detection (Hofmeyr et al. 1998; Forrest et al. 1996; Wagner and Dean 2001; Bhatkar et al. 2006; Kiriansky et al. 2002; Warrender et al. 1999; Somayaji and Forrest 2000; Sekar et al. 2001; Mutz et al. 2006).

Seccomp-BPF framework (Corbet 2012) is a system call interposition implementation for Linux Kernel introduced in Linux 3.5. It is an extension to *Seccomp* (Corbet 2009), 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 (DockerDocs 2017). Figure 2 shows a snippet of *Seccomp profile* used by Docker, written in the JSON (JSON 2017) format.

By default, Docker disallows 44 system calls out of 300+ for all of the containers to provide wide application compatibility (DockerDocs 2017). However, the principle of least

privilege (Saltzer and Schroeder 1975) 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.

2.2 System Call Policy Generation

Generating an accurate system call policy for an existing program has always been challenging (Provos 2003). 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 the static analysis determines some behavior is impossible, the behavior can be safely excluded. Janus (Goldberg et al. 1996) recognizes a list of dangerous system calls statically. Wagner and Dean (2001) derive system call sequences from program source code.

The limitation of the static analysis is *over-approximation*. The analysis often assumes 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 benign program 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 (Forrest et al. 1996; Hofmeyr et al. 1998; Somayaji and Forrest 2000). Instead of analyzing system call sequences, some studies take into account the arguments of system calls. Sekar et al. (2001) uses finite state automata (FSA) techniques to capture temporal relationships among system calls (Mutz et al. 2006; Kruegel et al. 2003). Some studies keep track of data flow between system calls (Bhatkar et al. 2006; Fetzer and Süßkraut 2008). Other researchers also take advantage of machine learning techniques, such as Hidden Markov Models (HMM) (Warrender et al. 1999; Gao et al. 2006), Neural Networks (Endler 1998), and k-Nearest Neighbors (Liao and Vemuri 2002).

The fundamental limitation of the 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) (Jamrozik et al. 2016; Le et al. 2018; Bao et al. 2018). The dynamic analysis would profit from an abundance of test cases. A great amount of research effort has been put on automatic test case generation (Anand et al. 2013). As a result, a significant number of different techniques for test case generation have been advanced and investigated, e.g., symbolic execution and program structural coverage testing (Cadaru and Sen 2013; Wan and Zhou 2011), model-based test case generation (Utting and Legeard 2010), combinatorial testing (Nie and Leung 2011), adaptive random testing (Chen et al. 2010; Ciupa et al. 2008) and search-based testing

(Harman and McMinn 2010). Notably, a SQL test generator¹ could achieve much higher coverage in MySQL or PostgreSQL, whereas a test generator for Web pages (e.g., CrawlJax²) could do the same for Web servers.

Dynamic analysis requires sufficient “normal” executions to be trained with, and would profit from automatic test case generation.

2.3 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 (Jamrozik et al. 2016). 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* (Zeller 2015). In our case, system call interposition-based sandboxing can guarantee that anything not seen yet will not happen. Note that our approach does not aim to provide ideal sandboxing, i.e., no false positives or false negatives. To provide ideal sandboxing, testing must cover all and only legitimate executions; but as noted in Forrest et al. (1997), it is theoretically impossible to get perfect discrimination between legitimate and illegitimate activities. We attempted to propose a sandboxing approach with low rates of false positives and few false negatives. Nevertheless, the 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.

3 Threat Model and Motivation

Most applications that run in the containers, e.g., Web server, database systems, and customized applications, are too complicated to trust. Even with access to the source code of these applications, it is difficult to reason about their security. An exploitable container might be compromised by carefully craft inputs that exploit vulnerabilities, and 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 (CVE-2016-0728 2016); it can also acquire packet of another container via ARP spoofing (Whalen 2001). 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 the system call interface is the only gateway to make persistent changes to the underlying systems (Provos 2003). Nevertheless, the system call interface is dangerously wide; less-exercised system calls are a major source of kernel exploits.

¹<https://mattjibson.com/random-sql/>

²<https://github.com/crawljax/crawljax/>

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.

4 Sandbox Mining

4.1 Overview

During the mining phase, we automatically explored container behaviors, monitored its system call invocations, and characterized system call behavior for all seen system calls. This section illustrates three fundamental steps of our approach during the mining phase as shown in Fig. 3.

4.2 Enabling Tracing

The first step is to prepare the kernel to enable tracing. We used container-aware monitoring tool *sysdig* (DraisInc 2017) 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, the process that executes the system call, thread ID (which corresponds to the process ID for single-threaded processes), and list of system call arguments;
- an *exit* entry for a system call, with the properties mentioned above, except that replacing the list of arguments with the return value of the system call.

4.3 Automatic Testing

In this step, we selected a test suite that covers the functionality of a container. Then we ran the test suite on the targeted container. During testing, we automatically copied the tracing logs at constant time intervals. This allowed us to compare at what time the system call was accessed. Therefore, we can monitor the growth of the sandbox rules over time based on these snapshots.

4.4 Characterizing System Call Behavior

We characterized two types of system call behavior of a container: system call types and arguments. We first characterized the system call types of accessed system calls for each container. We then characterized the system call arguments of top 20 frequently accessed system calls for each container. Finally, we obtained models of system call name for all accessed system calls, as well as models of system call name and argument(s) for most frequently (top 20) accessed system calls. The details of how we characterize system call behavior are discussed below.

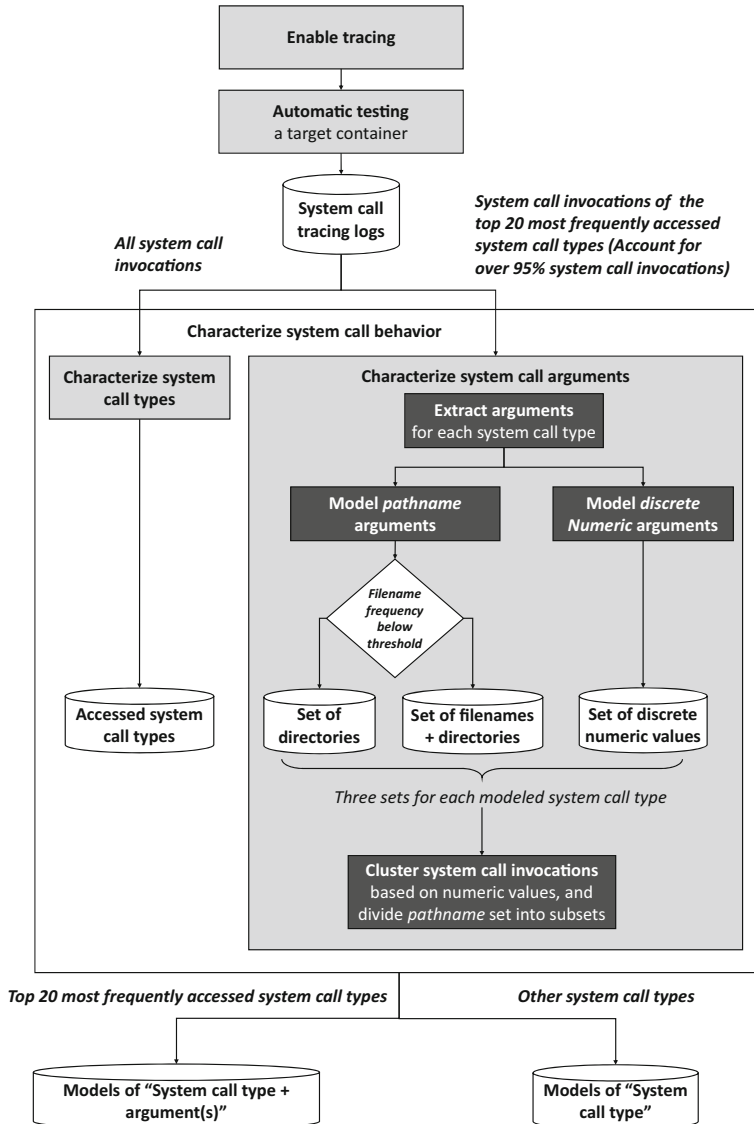


Fig. 3 Process to mine sandbox rules for a container

4.4.1 Characterizing System Call Types

We extracted the set of system call types accessed by a container from the tracing logs. As an example of how our approach characterizes system call types, let us consider the *hello-world* container (DockerHub 2017b). This container employs a Docker image which simply prints out a message and does not accept inputs. We discovered 24 system calls during testing. The Docker *init* process (OpenContainerInitiative 2017) and the *hello-world* container invoke the system calls as follows (Note that functions in [] are that first trigger the system calls):

```
[github.com/opencontainers/runc/libcontainer/utils/utils_unix.go:
  CloseExecFrom]
```

```
1 openat()
2 getdents64()
3 lstat()
4 close()
5 fcntl()
```

- Right after the *Seccomp profile* is applied, the Docker init process closes all unnecessary file descriptors that are accidentally inherited by accessing `openat()`, `getdents64()`, `lstat()`, `close()`, and `fcntl()`.

```
[github.com/opencontainers/runc/libcontainer/capabilities_linux.go:
  newCapWhitelist]
```

```
6 getpid()
7 capget()
```

- Then the Docker init process creates a whitelist of capabilities with the process information by accessing `getpid()` and `capget()`.

```
[github.com/opencontainers/runc/libcontainer/system/linux.go:
  SetKeepCaps]
```

```
8 prctl()
```

- The Docker init process preserves the existing capabilities by accessing `prctl()` before changing user of the process.

```
[github.com/opencontainers/runc/libcontainer/init_linux.go: setupUser]
```

```
9 getuid()
10 getgid()
11 read()
```

- The 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()`.

```
[github.com/opencontainers/runc/libcontainer/init_linux.go:
  fixStdioPermissions]
```

```
12 stat()
13 fstat()
14 fchown()
```

- The 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.

```
[github.com/opencontainers/runc/libcontainer/init_linux.go: setupUser]
```

```
15 setgroups()
```

```
[github.com/opencontainers/runc/libcontainer/system/syscall_linux_64.go:
  Setgid]
```

```
16 setgid()
```

```
[github.com/opencontainers/runc/libcontainer/system/syscall_linux_64.go:
  Setuid]
```

```
17 futex()
```

```
18 setuid()
```

- The Docker init process changes groups, group ID, and user ID for current process by accessing `setgroups()`, `setgid()`, `futex()` and `setuid()`.

```
[github.com/opencontainers/runc/libcontainer/capabilities_linux.go: drop
]
```

```
19 capset()
```

- The Docker `init` process drops all capabilities for current process except those specified in the whitelist by accessing `capset()`.

```
[github.com/opencontainers/runc/libcontainer/init_linux.go:
    finalizeNamespace]
20 chdir()
```

- The Docker `init` process changes current working directory to the one specified in the configuration file by accessing `chdir()`.

```
[github.com/opencontainers/runc/libcontainer/standard_init_linux.go:
    Init]
21 getppid()
```

- The 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.

```
[github.com/opencontainers/runc/libcontainer/system/linux.go: Execv]
22 execve()
```

- The final step of the Docker `init` process is accessing `execve()` to execute the initial command of the *hello-world* container.

```
[github.com/docker-library/hello-world/hello.c: _start()]
23 write()
24 exit()
```

- The initial command of the *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 expected to capture the set of system calls accessed only by the container. However, the captured set included some system calls that are accessed by the Docker `init` process. This is because applying sandbox rules is a privileged operation; the Docker `init` process should apply sandbox rules before dropping capabilities. We noticed that the Docker `init` process invokes 22 system calls to prepare runtime environment before the container starts. If the 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 the *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 would be more complicated if a container accepts inputs to determine its behavior.

4.4.2 Characterizing System Call Arguments

- **Extraction phase:** We extracted system call arguments of each container from the tracing logs. We found that the top 20 accessed system call types account for over 95% system call invocations for each container. To provide the reliability of characterization models, we only modeled the arguments of top 20 accessed system call types invoked by each container.
- **Modeling phase:** During the modeling phase, we create separate models for different types of system call arguments. According to previous study (Maggi et al. 2010), four types of arguments are passed to system call: pathnames and filenames, discrete numeric values, arguments passed to programs for execution, user and group identifiers (UIDs and GIDs). For each type of argument, we designed a representative model. In

Table 1 Association of models to syscall arguments

Syscall	Models used for the arguments
access	pathname → Path Name mode → Discrete Numeric
epoll_wait	maxevents → Discrete Numeric
exit	status → Discrete Numeric
fcntl	cmd → Discrete Numeric
futex	futex_op → Discrete Numeric
lstat	pathname → Path Name
mmap	prot, flags → Discrete Numeric
open	pathname → Path Name flags → Discrete Numeric
openat	pathname → Path Name flags → Discrete Numeric
poll	timeout → Discrete Numeric
recvfrom	len → Discrete Numeric
semop	nsops → Discrete Numeric
sendto	len → Discrete Numeric
shutdown	how → Discrete Numeric
socket	domain, type, protocol → Discrete Numeric
socketpair	domain, type, protocol → Discrete Numeric
stat	pathname → Path Name

Table 1, we summarize the association of the models with the arguments of each system call type we take into account.

Pathnames are frequently used in system calls. They are difficult to model properly because of their complex structure. Pathnames are comprised of directory names and file names. File names are usually too variable to allow a meaningful model to be always created. Thus we set up a system-wide threshold below which we believe the file names are not so regular to form a significant model. For the pathnames with a frequency below the threshold, we represented the pathnames using their directories to be a learned set. For those pathnames with a frequency above the threshold, we considered the file names along with the corresponding directory to be a learned set. During sandbox enforcing, the argument of pathname was compared against the two types of learned sets. Obviously, this solution is effective only if the argument values are limited in number, static and not deployment dependent (e.g., file system calls, SQL administrative commands, etc.). For containers that violate these requirements, e.g., an OS container, a Web server container with dynamically generated pages with PHP, or distributed system containers like Cassandra, our system may need to be trained in production as it might introduce false alerts in unknown numbers.

Discrete numeric values such as flags and opening modes are usually chosen from a limited set for a system call type. Therefore, we can store all the discrete numeric values of a system call type that appear during testing to be a finite set. During sandbox enforcing, the argument of the discrete numeric value is compared against the stored value list.

- **Clustering phase:** During the clustering phase, we built correlations among the models for different arguments of a system call type. We divided the invocations of a single system call into subsets. The invocations in a subset have arguments with higher similarity. We were interested in creating models on these subsets, and not on the general

system calls. This facilitated to capture the normality and deviation. For instance, the common top 20 accessed system call `open` of the eight containers in our experiment has two parameters `pathname` and `mode`. The parameter `flags` represents a set of flags indicating the type of open operation (e.g., `O_RDONLY` read-only, `O_CREAT` create if nonexisting, `O_RDWR` read-write). We first aggregated system call invocations of `open()` over the argument `flags` of discrete numeric values. We then built models over the argument `pathname` for each cluster with same `flags`. Through the clustering, we divided each “polyfunctional” system call into “subgroups” that are specific to a single functionality. Consider the system call `open()` in the *Nginx* container as an example. We divided the invocations into 5 subgroups over the *flags* including `O_APPEND | O_CREAT | O_WRONLY, O_RDONLY, O_TRUNC | O_CREAT | O_RDWR, O_RDONLY | O_CLOEXEC, and O_NONBLOCK | O_RDONLY`. The resulting model is shown in Fig. 4.

5 Sandbox Enforcing

5.1 Overview

The second phase of our approach is sandbox enforcing, which monitors and possibly prevents container behavior. We need a technique that conveniently allows the user to sandbox any container. To this end, we leveraged *Seccomp-BPF* (Corbet 2012) 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 (Vlasenko 2017) is generated and sent to the tracer if there exists one. Whenever the applied *BPF* program generates a *ptrace* event during the target container execution, the kernel stops the execution of the container and transfers control to our tracer. Our tracer intercepts the event and examines the target’s internal state of system call arguments via `ptrace()` interface. This section illustrates the two steps of our approach during the sandboxing phase.

5.2 Generate Sandbox Rules

This step translates the models of system call behavior discovered in mining phase into sandbox rules. We derived two types of system call models during sandbox mining as shown in Fig. 3, i.e., models of system call types, and models of system call types + arguments. We further divided system calls into three types based on their derived models:

- System calls *with models of string type arguments*;
- System calls *only with models of non-string type arguments*;
- System calls *only with models of system call types*.

We then generated sandbox rules for the three kinds of system call types by following the three sequential steps as follows:

System Calls with Models of String Type Arguments Translating models of string type arguments into sandbox rules is comprised of two steps:

Step 1: Generating rules in Seccomp profile. We use the `awk` tool to translate each system call that has models with string type arguments into a sandbox rule with action `SCMP_ACT_TRACE`. Specifically, we write a script which automatically generates a snippet in the JSON format for each system call. We take the the system call `open()` of the container *Nginx* as an example, whose model is shown in Fig. 4. The generated sandbox rule for `open()` in Seccomp profile is as follows:

```
{
    "name": "open",
    "action": "SCMP_ACT_TRACE",
    "args": []
}
```

By enforcing above sandbox rule, once the system call `open()` is accessed by the container during sandboxing, a `ptrace` event is generated and sent to the tracer of the container. The tracer further checks the arguments for each system call invocation.

Step 2: Implementing models for string type arguments. We wrote a Python program (388 lines) which translated the system call models of string type arguments into a module in C programming language. The module implements the argument checking process of distinct system calls for a particular container. For example, the argument checking snippet for system call `open()` in *Nginx* is as follows:

```
if (regp->orig_rax == __NR_open) {
    int len = read_arg_str(buf, MAX_PATH, pid, (char*)regp->rdi)
    ;
    char* arg0 = buf;
    int arg1 = regp->rsi;
    int allow = 0;
    switch (arg1) {
    case O_APPEND|O_CREAT|O_WRONLY:
        allow = compare_str_argument(0, arg0,
            open_allowed_flags_14);
        break;
    case O_RDONLY:
        allow = compare_str_argument(0, arg0,
            open_allowed_flags_1);
        break;
    case O_TRUNC|O_CREAT|O_RDWR:
        allow = compare_str_argument(0, arg0,
            open_allowed_flags_263);
        break;
    case O_RDONLY|O_CLOEXEC:
        allow = compare_str_argument(0, arg0,
            open_allowed_flags_4097);
        break;
    case O_NONBLOCK|O_RDONLY:
        allow = compare_str_argument(1, arg0,
            open_allowed_flags_65);
        break;
    }
    if (!allow) {
        fprintf(false_alarm, "NGINX > open(pathname=
    )
    }
    return allow;
}
```

To check the arguments for each system call invocation of `open()`, the tracer invokes the module that implements the argument models. The module then reads accessed pathname from memory by following the pointer specified by the system call argument `pathname`, and check if the argument `pathname` is allowed by the argument models. Each prohibited system call invocation will be recorded in a log file.

System Calls Only with Models of Non-String Type Arguments We wrote a Python script (107 lines) to translate the models of non-string type arguments for each system call into sandbox rules in Seccomp profile. Consider the system call `socketpair()` in *Nginx* as an example. The system call `socketpair()` has a model which has constraints on three non-string type arguments: `arg0: domain = 1, arg1: type = 1` and `arg2: protocol = 0`. We translate this model into a sandbox rule in Seccomp profile as follows:

```

{
  "names": [
    "socketpair"
  ],
  "action": "SCMP_ACT_ALLOW",
  "args": [
    {
      "index": 0,
      "value": 1,
      "valueTwo": 0,
      "op": "SCMP_CMP_EQ"
    },
    {
      "index": 1,
      "value": 1,
      "valueTwo": 0,
      "op": "SCMP_CMP_EQ"
    },
    {
      "index": 2,
      "value": 0,
      "valueTwo": 0,
      "op": "SCMP_CMP_EQ"
    }
  ]
}

```

By enforcing this sandbox rule, once the system call `socketpair()` is invoked with arguments that satisfy those constraints during sandboxing, the invocation will be permitted according to the specified action, i.e., `SCMP_ACT_ALLOW`.

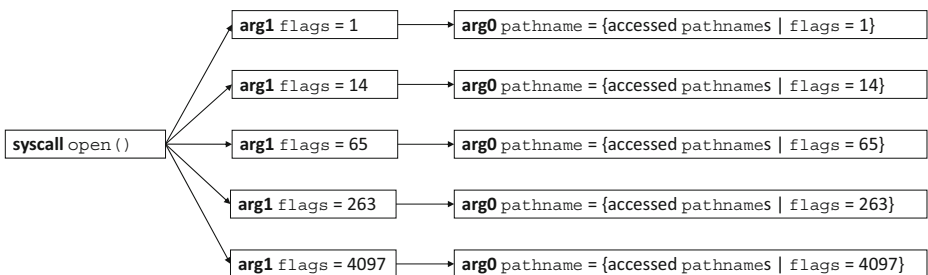


Fig. 4 Argument model of the system call `open()` for the container *Nginx*

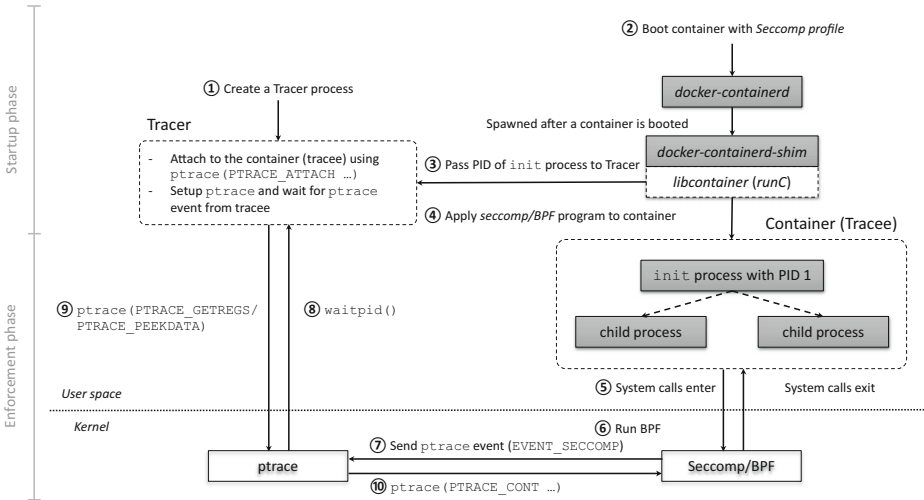


Fig. 5 Process to enforce sandbox rules for a container

System Calls Only with Models of System Call Types For those system calls only with models of system call types, we translated those system call types into sandbox rules using awk tool. For instance, write () is one of the discovered system call during sandbox mining for the hello-world container. We generated a sandbox rule with name write, action SCMP_ACT_ALLOW, and no constraint applied to the arguments (args) as below:

```

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

```

By enforcing this rule, once the system call write () is accessed during sandboxing, the invocation would be allowed according to the specified action, i.e., SCMP_ACT_ALLOW.

After translating all system call models for each container, the resulting Seccomp profile and parameter checking module constituted a sandbox for that container. We defined the default action of the sandbox as follows:

```

"defaultAction": "SCMP_ACT_ERRNO"

```

The default action indicates that the generated sandbox rules constitute a whitelist of system calls that are allowed by the sandbox. For the system call behavior that is not included in the whitelist, the sandbox will deny the behavior during sandboxing and make the system call invocation return an error number (SCMP_ACT_ERRNO). In particular, the system call invocation fails and its function will not be executed; the container will receive an error number for this system call invocation.

5.3 Enforcing Sandbox Rules

Figure 5 illustrates the process that we incorporate seccomp/BPF and ptrace to enforce generated sandbox rules. The process includes two phases, the startup phase and the enforcement phase.

At the startup phase, we first created a Tracer process (1), which executes with the privileges of an isolated process. The Tracer process builds a named pipe for receiving

tracee's PID. Next, we started the target container with corresponding *Seccomp profile* using `docker run --security-opt seccomp (2)`. The *docker-containerd* process then spawns a *docker-containerd-shim* process that issues command to a container runtime (*runC*). Before namespacing the PID of target container's *init* process, *runC* sends the PID to the *Tracer* (3) through the established pipe. The *Tracer* process receives PID of the container and attaches to the target process by calling `ptrace (PTTRACE_ATTACH . . .)`. Then the *Tracer* invokes `waitpid()` to wait for `ptrace` event generated by tracee. Lastly, *runC* loads the *seccomp/BPF* program specified in the *Seccomp profile* into kernel (4) and calls `execve()` to run the initial command of the target container. At this point, the target container starts execution.

At the enforcement phase, the *seccomp/BPF* program runs and decides whether to intercept or not system call invocations (5). A sandbox rule with action `SCMP_ACT_ALLOW` will allow the system call invocations that satisfy the constraints specified by the rule without intercepting them (6). A sandbox rule with action `SCMP_ACT_TRACE` will generate a `ptrace` event if the system call name matches (7). The `ptrace` event (`EVENT_SECCOMP`) is sent to the *Tracer* waiting for a `ptrace` event (8). Then the *Tracer* queries the states of the tracee via the `ptrace` interface, e.g., `ptrace(PTTRACE_GETREGS` and `ptrace(PTTRACE_PEEKDATA)` (9). After examining the system call arguments, the *Tracer* continues the tracee by invoking `ptrace` with `PTTRACE_CONT` (10).

6 Experiments

6.1 Overview

In this section, we evaluated our approach on eight containers. The eight containers are among the most popular application containers in Docker Hub (2017a) and have a large number of downloads. The details of them are shown in Table 2. The eight application containers can be used in PaaS, and provide domain-specific functions. We deliberately eliminated all OS containers (e.g., *Ubuntu* container) which provide basic functions, and can potentially access all system calls. We also eliminated containers for distributed applications (e.g., *Cassandra*) or containers for dynamic file systems/path (e.g., *PHP*) that are outside the ability of our approach (see Section 7 for the threat to validity). Note that *Python* as a programming language provides a wide range of functionality, and a *Python* container can potentially access all system calls. Mining a sandbox for the *Python* container will be useless because the mined sandbox will be too coarse. Thus we set up a Web framework *Django* (DjangoSoftwareFoundation 2015) on top of the *Python* container. This makes the *Python* container have specific functionality.

We would like to answer three research questions as follows:

RQ1. How efficiently can our approach mine sandboxes?

We evaluated 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 (2017a) and have a large number of downloads. The details of them are shown in Table 2. 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 a *Python* container can potentially access all system

Table 2 Experiment subjects

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

Open https://hub.docker.com/_/<identifier> for details

calls. Mining sandbox for the *Python* container will be useless because the mined sandbox will be too coarse. Thus we set up a Web framework *Django* (DjangoSoftwareFoundation 2015) on top of the *Python* container. This makes the *Python* container have specific functionality. In addition, we compared the mined sandboxes with the default one provided by Docker to see if the attack surface is reduced.

RQ2. How sufficient does sandbox mining cover system call behaviors?

Any non-malicious system call behavior not explored during testing implies a false alarm during production. We evaluated the risk of false alarms: how likely is it that sandbox mining misses system call behavior, and how frequently will containers encounter false alarms. We estimated the system call coverage for sandbox mining by using 10-fold cross-validation. In addition, we checked 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 measured the end-to-end performance of containers – *transactions per second*. We compared the end-to-end performance of a container running in four environments: 1) natively without sandbox, 2) with syscall “type” sandbox mined by our approach, 3) with syscall “type+argument” sandbox mined by our approach, and 4) with default Docker sandbox.

RQ4. Can the mined sandboxes effectively protect an exploitable application running in a container?

We analyzed how our mined sandboxes can protect an exploitable application by reducing the attack surface. We further conducted a case study by considering a security vulnerability in reality (CVE-2013-2028 in *Nginx* 1.3.9-1.4.0). While running a *Nginx* container with syscall “type+argument” sandbox mined by our approach, we exploited the security vulnerability and attempted to attack the container.

6.2 Setup

The containers in the experiments ran on a 64-bit Ubuntu 16.04 operating system inside VirtualBox 5.2.0 (4GB base memory, two processors). The physical machine is with an Intel Core i5-6300 processor and 8GB memory.

6.2.1 Sandbox Mining: Automatic Testing

We describe the test suites that we run for automatic testing during sandbox mining in the experiment as follows. The automatic testing generates the “training set” for sandbox mining. Note that sandbox mining is conducting during the pre-production phase in practice.

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 (Mosberger and Jin 1998) also from that the 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` (redislabs 2017) 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` (Mongodb 2017) 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* (Kopytov 2017) 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 *sptest* 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* (PostgreSQL 2017) 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*.

6.2.2 Statistics

During sandbox mining, the eight containers executed approximately 5,340,000 system calls. The number of system call execution of the eight containers is shown in Fig. 6. 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 considerable performance penalty. To achieve low performance penalty, we only traced and analyzed system calls in sandbox mining phase. A decomposition of the most frequent system calls of each container is shown in Fig. 7. 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()*.

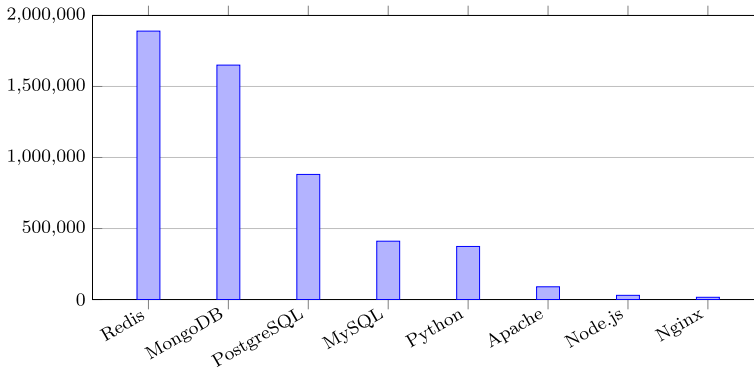


Fig. 6 Number of system call execution of the containers

6.3 RQ1: Sandbox Mining Efficiency

Figure 8 shows the sandbox rule saturation charts for the eight containers. For sandbox rules of system call type, we can see that six charts “flatten” before one minute mark, and the remaining two before two minutes. For sandbox rules of both system call type and argument, we can observe that five charts “flatten” before one minute mark, two charts before two minutes (*redis* and *postgres*), and the remaining one before three minutes (*node*).

For sandbox rules of system call type, 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. For sandbox roles of system call type and argument, our approach has discovered 90, 91, 121, 122, 115, 79, 89, 83 sandbox rules respectively, which reflected the significant argument models of system calls. The attack surface is further reduced by restricting the arguments of system call invocations.

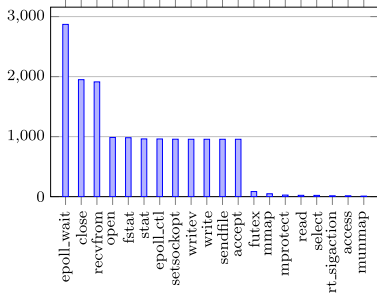
During the warm-up phase, the number of system calls accessed by each of the containers grew rapidly. After the warm-up phase, for all of the Web servers except *Apache*, the simple HTTP request caused a further increase and the number of system calls converges; for *Apache* container, *httperf* caused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

The answer of **RQ1** is: our approach can mine the saturated sandbox rules within three minutes. The mined sandboxes reduce the attack surface.

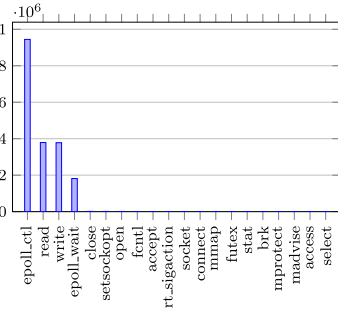
Sandbox mining quickly saturates accessed system calls for the selected static test cases.

6.4 RQ2: System Call Coverage

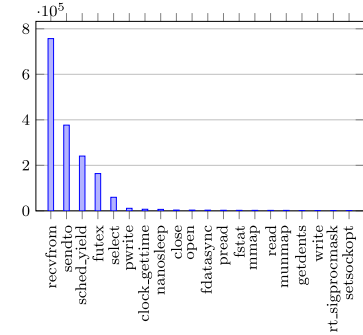
To estimate the system call coverage of sandbox mining, we follow the steps as below:



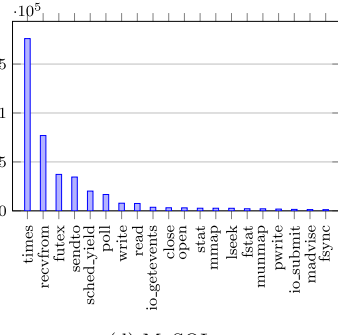
(a) Nginx



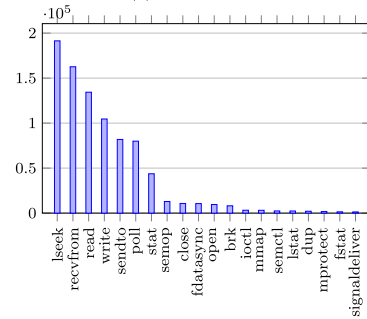
(b) Redis



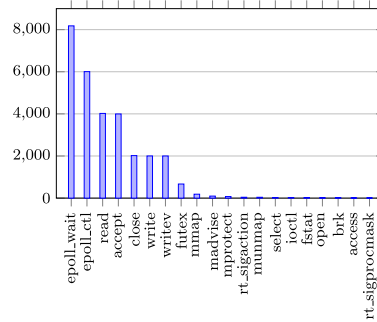
(c) MongoDB



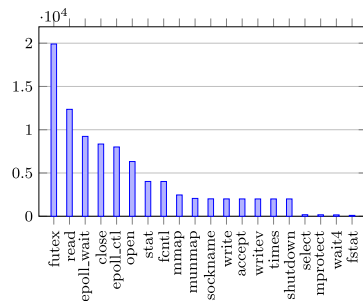
(d) MySQL



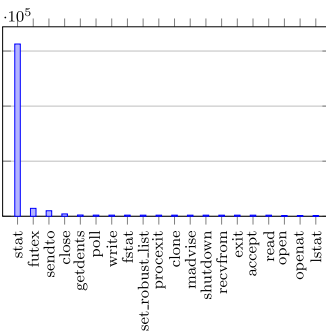
(e) PostgreSQL



(f) Node.js



(g) Apache



(h) Python Django

Fig. 7 Histogram of system call frequency for each of the containers

Table 3 Estimation of system call behavior coverage

	Min	Max	Median	Mean
Nginx	93.0%	100.0%	98.2%	97.5%
Redis	93.9%	100.0%	98.5%	98.6%
MongoDB	95.2%	100.0%	100.0%	99.0%
MySQL	98.9%	100.0%	98.9%	99.3%
PostgreSQL	98.9%	100.0%	100.0%	99.8%
Node.js	88.7%	100.0%	98.1%	96.4%
Apache	96.7%	100.0%	98.4%	98.2%
Python Django	96.8%	100.0%	100.0%	99.0%

1. Randomly split the tracing log for each container into two, i.e., a training set and a testing set, by using the 10-fold cross validation (we use `KFold()` function in *Scikit-learn*);
2. Mine sandboxes on the training set;
3. Compare the list of allowed system calls on each training set with the list of system calls on complete tracing log.

We repeat the above steps 10 times and present the statistics of system call coverage for each container in Table 3. The average coverage rates range from 96.4% to 99.8% across the containers in our experiment.

To further investigate if most important functionality of a container was found during sandbox mining, we read the documentation of the containers and prepare 30 *use cases* which reflect containers' typical usages. Table 4 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 the 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* (Grubb 2017) 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.

The "Message # in *auditd*" column in Table 4 summarizes the number of messages logged by *auditd*. When we enforced sandboxes of system call types, no message was logged by *auditd* for the 30 use cases. The number of false alarm is zero. When enforcing sandboxes of system call types and arguments, one message was logged by *auditd* for the second use case of the *Nginx* container - accessing the non-existent page `hello.html` was denied by our sandbox. Accessing non-existent pages were not "normal" behaviors. Thus we did not consider the one message in *auditd* as a false alarm.

The set of use cases we have prepared for assessing the risk of false alarms (Table 4) 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 usages may yield different results.

The answer of **RQ2** is the estimation of system call coverage for sandbox mining range from 96.4% to 99.8%. We did not find any impact from the mined sandboxes on the basic functionalities of the containers. As we noted, this might not be true for containers which

Table 4 Use cases

Container	Use case	Function	Message # in <i>auditd</i> (type / type+arg)
Nginx	Access static page	Access default page <code>index.html</code> , <code>50x.html</code>	0/0
	Access non-existent page	Access non-existent page <code>hello.html</code>	0/0
Redis	SET command	Connect to Redis server, set key to hold the string value	0/1
	GET command	Connect to Redis server, get the value of key	0/0
	INCR command	Connect to Redis server, increment the number stored at key by one	0/0
	LPUSH command	Connect to Redis server, insert all the specified values at the head of the list stored at key.	0/0
	LPOP command	Connect to Redis server, remove and returns the first element of the list stored at key	0/0
	SADD command	Connect to Redis server, add the specified members to the set stored at key	0/0
	SPOP command	Connect to Redis server, remove and return one or more random elements from the set value store at key	0/0
MongoDB	LRANGE command	Connect to Redis server, return the specified elements of the list stored at key	0/0
	MSET command	Connect to Redis server, replace multiple existing values with new values	0/0
	insert	Connect to mongod, use database test, insert record <code>{image:"redis",count:"1"}</code> into collection <code>falsealarm</code> , exit	0/0
MySQL	save	Connect to mongod, use database test, update record in collection <code>falsealarm</code> , exit	0/0
	find	Connect to mongod, use database test, list all records in collection <code>falsealarm</code> , exit	0/0
	CREATE DATABASE	Connect to MySQL server, create database test, list all databases, exit	0/0
	CREATE TABLE	Connect to MySQL server, use database test, create table <code>falsealarm</code> , insert record, exit	0/0
	INSERT	Connect to MySQL server, use database test, insert record into table <code>falsealarm</code> , exit	0/0
	UPDATE	Connect to MySQL server, use database test, update record, exit	0/0
	SELECT	Connect to MySQL server, use database test, list all records, exit	0/0
PostgreSQL	CREATE DATABASE	Connect to PostgreSQL server, create database test, list all databases, exit	0/0
	CREATE TABLE	Connect to PostgreSQL server, connect to database test, create table <code>falsealarm</code> , exit	0/0

Table 4 (continued)

Container	Use case	Function	Message # in <i>auditd</i> (type / type+arg)
	INSERT	Connect to PostgreSQL server, connect to database test, insert record into table FalseAlarm, exit	0/0
	UPDATE	Connect to PostgreSQL server, connect to database test, update record in table FalseAlarm, exit	0/0
	SELECT	Connect to PostgreSQL server, connect to database test, list all records in table FalseAlarm, exit	0/0
Node.js	Access existent URI	Access /	0/0
	Access non-existent URI	Access non-existent URI /hello	0/0
Apache	Access static page	Access default page index.html	0/0
	Access non-existent page	Access non-existent page hello.html	0/0
Python Django	Access existent URI	Access /	0/0
	Access non-existent URI	Access non-existent URI /hello	0/0

auditd logs a message when a system call invocation is denied by the sandbox

require access to dynamic paths or deployment of specific functionalities. For example, in the case of database containers, we did not include administrative operations in the test cases. In those cases, our approach may generate an unknown number of false alarms.

The estimation of system call coverage for sandbox mining range from 96.4% to 99.8%. The mined sandboxes require no further adjustment on use cases of basic functionalities for the executions included in the selected static test cases.

6.5 RQ3: Performance Overhead

To analyze the performance overhead of sandbox enforcing, we ran the eight containers in three environments: 1) natively without sandbox as a baseline, 2) with syscall “type” sandbox mined by our approach, 3) with syscall “type+argument” sandbox mined by our approach, and 4) with default Docker sandbox.

We measured the throughput of each container as an end-to-end performance metric. To minimize the impact of the network, we ran each of the containers using host networking via `docker run --net=host`. We repeated each experiment 10 times with a less than 5% standard deviation.

For the *Redis*, *MongoDB*, *PostgreSQL* and *MySQL* containers, we evaluated the *transactions per second* (TPS) of each container by running the aforementioned tools in Section 6.3. The percentage reduction of TPS per container for *Redis*, *MongoDB*, *PostgreSQL* and *MySQL* is presented in Fig. 9. We noticed that enforcing mined sandboxes incurred a small TPS reduction (0.6% - 2.14% for syscall “type” sandboxes, 1.22% - 3.76% for syscall “type+argument” sandboxes) for the four containers. Syscall “type” sandboxes produced 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 sandboxes, and thus the corresponding *BFP program* needs more computation during sandboxing. The TPS reduction of syscall “type+argument” sandboxes is close to that of the default sandbox.

For the Web server containers, we evaluated the throughput, i.e., *responses per second*, of each container by running *httperf* tool. To measure the response rate of each container, we increased the number of requests per second that were sent to the container. The result is shown in Fig. 10. Web server containers running with sandboxes except for *Nginx* achieved performance very similar to that of the containers running without sandboxes. We can see that the achieved throughput increased linearly with offered load until the container starts to become saturated. The saturation points of *Nginx*, *Node.js*, *Apache* and *Python Django* are around 11,000, 7,000, 4,000 and 300 requests per second respectively. After the offered load increased beyond that point, the response rate of the container started to fall off slightly.

For the *Nginx* container, enforcing syscall “type+argument” sandbox incurred a significant reduction of throughput (around 27%). Whenever the applied *BPF program* generated a *ptrace* event during a target container’s execution, the kernel stopped the execution of the target process and transferred control to our *Tracer*. The *Tracer* could then examine the string arguments of the target’s system call invocations by using *ptrace* interface. However, using *ptrace* interface imposes high runtime overhead on the target due to two context switches, from target to the *Tracer* and back (Guo and Engler 2011). During our performance evaluation, the *Nginx* container extremely frequently accessed the system call `open()` to open a Web page. This caused frequent invocations to the *ptrace* interface, and further resulted in a significant reduction of throughput.

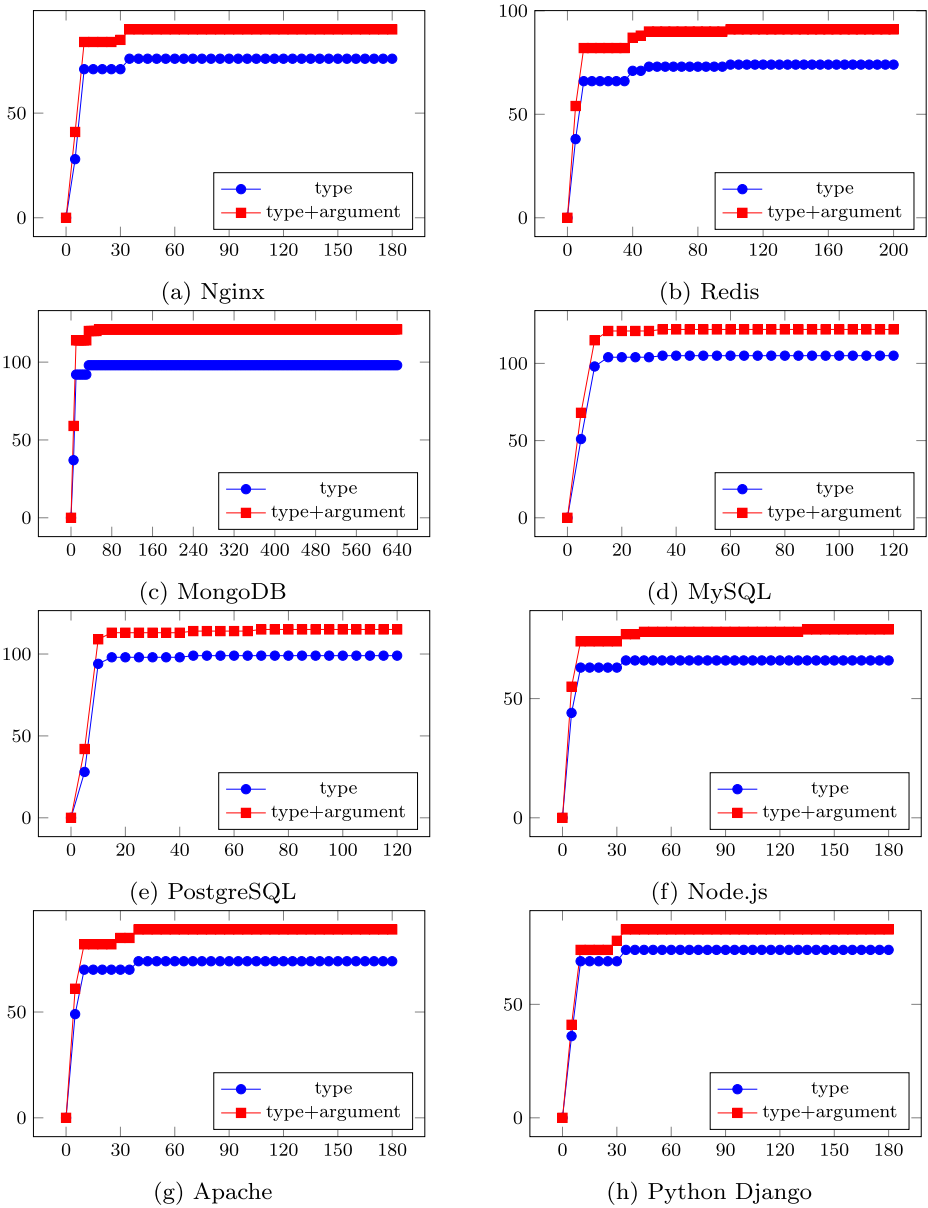


Fig. 8 Per-container sandbox rule saturation for containers in Table 2. y axis is number of sandbox rules, x axis is seconds spent

The answer of **RQ3** is: enforcing sandboxes 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.

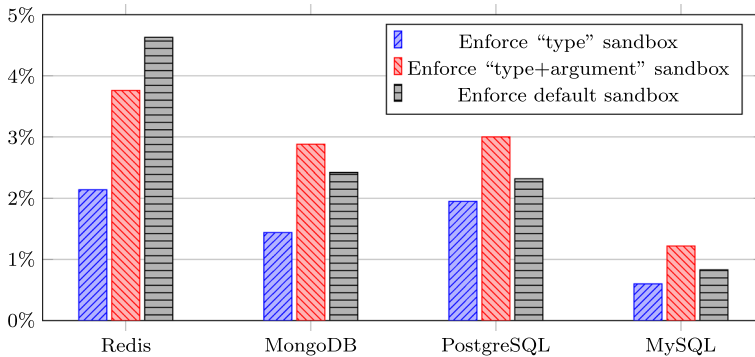


Fig. 9 Percentage reduction of transactions per second (TPS) due to sandboxing

For Web server containers, we evaluated the throughput, i.e., *responses per second*, of each container by running *httperf* tool. To measure the response rate of each container, we increased the number of requests per second that were sent to the container. The result is shown in Fig. 10. Web server containers running with sandboxes except for *Nginx* achieved the 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* were around 11,000, 7,000, 4,000 and 300 requests per second respectively. After the offered load increased beyond that point, the response rate of the container started to fall off slightly.

For the *Nginx* container, enforcing syscall “type+argument” sandbox incurred a significant reduction of throughput (around 27%). Whenever the applied *BPF* program generated a *ptrace* event during a target container’s execution, the kernel stopped the execution of the target process, and transferred control to our *Tracer*. The *Tracer* could then examine the string arguments of the target’s system call invocations by using *ptrace* interface. However, using *ptrace* interface imposed high runtime overhead on the target due to two context switches, from target to the *Tracer* and back (Guo and Engler 2011). During our performance evaluation, the *Nginx* container extremely frequently accessed the system call `open()` to open a Web page. This caused frequent invocations to the *ptrace* interface, and further resulted in a significant reduction of throughput.

The answer of **RQ3** is: enforcing sandboxes 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.

6.6 RQ4: Security Analysis

Since containers share the same non-namespace-aware system call interface, it is critical to constrain the available system calls for each container to reduce the attack surface. For the containers we tested on Linux kernel 4.4.0, the number of available system calls during sandbox enforcing could be reduced from 373 to 66-105. In addition, the mined sandboxes with constraints on system call arguments further reduce the attack surface.

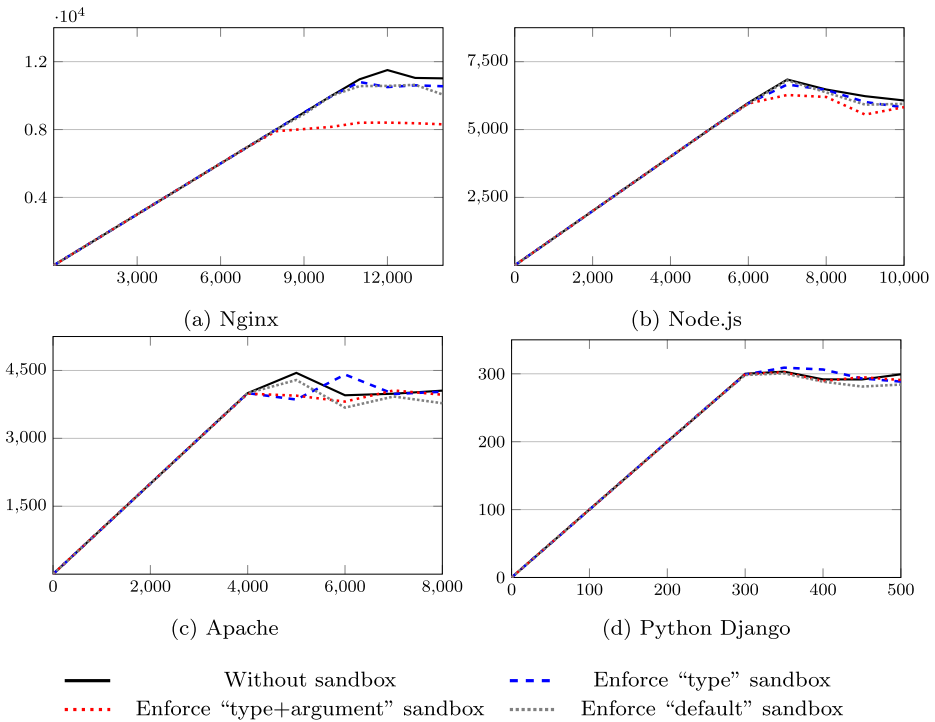


Fig. 10 Comparison of per-container reply rate for *Nginx*, *Node.js*, *Apache*, and *Python Django* that run without sandbox, with mined “type” sandbox, with “type+argument” sandbox and with default sandbox. y axis is response rate (responses per second), x axis is request rate (requests per second)

Through reducing available system call types and arguments, we can effectively reduce the attack surface of the host OS and lower the risk that an exploitable application escapes from the container and gains control of the host OS.

On the one hand, some vulnerable system calls could be prohibited and prevented from being exploited by attackers. For instance, among the 297 prohibited system calls by our mined sandboxes for the container *Nginx*, we found some vulnerable system calls with CVE security level MEDIUM or above, e.g., `sigaltstack()`,³ `setsid()`,⁴ and `setsockopt()`.⁵

On the other hand, some high privileged system calls could be prohibited and prevented from being misused by attackers to launch attacks after exploited, e.g., `chmod()`, `fchmod()` and `mknodat()`.

Preventing Security Breach in Reality We further provided an in-depth analysis of our mined sandboxes by looking at CVE-2013-2028, a memory corruption vulnerability in *Nginx* 1.3.9-1.4.0. We attempted to attack a running container of *Nginx* by exploiting the vulnerability. Since there existed no available Docker image for *Nginx* 1.3.9 or 1.4.0,

³<https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-2847>

⁴<https://www.cvedetails.com/cve/CVE-2002-1644>

⁵<https://www.cvedetails.com/cve/CVE-2017-6074>

we built a corresponding Docker image by using `docker build`. We first built binary from source code of *Nginx* 1.4.0.⁶ Then we identified the runtime dependencies by using *dockerize*⁷ and prepared the *Dockerfile*. Finally, we ran `docker build` to package the dependencies and make a Docker image.

CVE-2013-2028 reports a signedness bug in the component that handles chunked Transfer encoding. The bug can be exploited by overflowing the stack (MacManus et al. 2014) or corrupting header data (Le 2014). We now discuss the bug in CVE-2013-2028 in more detail as shown in Fig. 11. Attackers can have full control over `content_length_n` at line 8. Note that the variable `content_length_n` is a signed integer. The macro `ngx_min` at line 7 processes two signed integers and returns the less one. Therefore, once attackers feed *Nginx* a negative integer, `ngx_min` will always return the negative integer. The negative integer will then be converted to an unsigned integer and assigned to `size` at line 7. At line 11, the code invokes the function pointer `recv` to populate the array `buffer` at line 4 with the attacker-controlled variable `size`. Note that the length of `buffer` is smaller than the variable `size`. The array will overflow, which could further lead to code injection or code reuse attacks.

We leveraged the vulnerability by sending a POST request to the target container with keyword `chunked` in the Transfer-Encoding header. The request contained a chunked data block with a negative integer as its size. After receiving the request, the worker process of the *Nginx* container repeatedly read data of size defined by the crafted great integer. Consequently, the *Nginx* container refused to process subsequent requests. This indicated that the attack successfully exploited vulnerability. We then ran a *Nginx* container with our mined syscall “type+argument” sandbox and attacked the container using the same exploit. The attack failed this time because our mined sandbox prohibited the worker process from invoking `recvfrom()` system call when handling the crafted request. The specific sandbox rule that denied the invocation of `recvfrom()` system call with a great integer as argument 2 `len` is as follows:

```
{
  "name": [
    "recvfrom"
  ],
  "action": "SCMP_ACT_ALLOW",
  "args": [
    {
      "index": 2,
      "value": 1024,
      "valueTwo": 0,
      "op": "SCMP_CMP_EQ"
    }
  ]
}
```

The sandbox rule prevented `recvfrom()` system call invocations from receiving messages with length that are greater than 1024 through a socket. This greatly reduces the attack surface of the *Nginx* container. Notice that our mined sandbox of system call types cannot prevent the *Nginx* containers from the exploits because `recvfrom()` system call could be invoked in benign behavior.

⁶<http://nginx.org/download/>

⁷<https://github.com/jwilder/dockerize>

```

1 #define ngx_min(val1, val2) ((val1 > val2) ? (val2) : (val1))
2 #define NGX_HTTP_DISCARD_BUFFER_SIZE 4096
3 ...
4 u_char buffer[NGX_HTTP_DISCARD_BUFFER_SIZE];
5 ...
6 /* content_length_n is of type off_t, a signed integer type */
7 size = (size_t) ngx_min(
8 r->headers_in.content_length_n, /* attacker-controlled */
9 NGX_HTTP_DISCARD_BUFFER_SIZE);
10 n = r->connection->recv(r->connection, buffer, size);

```

Fig. 11 A memory corruption vulnerability in *Nginx* 1.3.9-1.4.0 (CVE-2013-2028)

The answer of **RQ4** is: our mined sandboxes effectively reduce the attack surface of target containers, and indeed prevent exploitation of CVE-2013-2028 in *Nginx* 1.3.9-1.4.0. A limitation is that the test cases of the *Nginx* container only cover 13.7% of the codebase. Thus, there might be potential false alarms for legitimate execution that are not captured by our experiment.

Our mined sandboxes reduce the attack surface of target containers, and can prevent containers from security breaches in reality. This might happen at the price of false alarms for executions not covered by the test cases.

7 Discussions and Threats

Granularity of Sandbox Rules A general dilemma exists in choosing an adequate granularity for sandbox rules. Coarse-grained sandbox rules may be too inaccurate to correctly separate attacks from legitimate use. However, as more fine-grained sandbox rules would operate, two problems occur. First, more test cases would be required that cover the behavioral diversity of the program. With the low code coverage of automatic testing (e.g., 13.7% for the *Nginx* container), it does not help much that all system calls would be covered (e.g., `write()`). This is because there would be plenty of code yet uncovered whose results eventually end up in the output (e.g., `write()`).

Second, the more fine-grained the sandbox rules are, the higher the burden becomes for any operator who would like to check the sandbox rules against expected behavior.

Given that the mined rules cannot rule out misclassification, the effort of manual adjustment can still occur. The effort of manual adjustment can still occur. The refinement of sandbox rules typically involves analyzing audit logs to identify misclassification. To reduce the manual effort required to refine sandbox rules, future studies could propose approaches and tools for automatic analysis and refinement of sandbox rules.

Defense in Depth Our approach aims at reducing the attack surface due to non-namespaces system calls. However, the system calls that are allowed by our mined sandboxes could be vulnerable. In that case, our approach may fail to prevent attackers from exploiting those vulnerable system calls. To further protect the containers against the exploitation of those vulnerable system calls, we could combine our approach with other Linux security mechanisms. For instance, we could combine our approach and the Linux Capabilities mechanism (Hallyn and Morgan 2008) to block the exploitation of vulnerability CVE-2016-9793 in system call `setsockopt()`. Specifically, `CAP_NET_ADMIN` capability is required to exploit the vulnerability; If the vulnerable system call `setsockopt()` is allowed by our

sandbox rules, we can still prevent this vulnerability by removing the `CAP_NET_ADMIN` capability from the container.

System Call Completeness In our experiment, we trace the system calls of target containers during automatic testing using application build-in benchmarks and HTTP workload generation tools. We further use the tool *gcov* to evaluate the code coverage of our test suites during automatic testing. We notice that the code coverage is relatively low. For instance, the code coverage of the automatic testing for the *Nginx* container is 13.7%. To facilitate the application of our approach in practice, container developers could combine our approach with the testing process of the application development. Since container developers might also be the application developers, they would have a deeper understanding of the typical and exceptional usage of the application. As suggested by Bacis et al. (2015), the container developers could then publish their mined sandboxes with the images. Thus, the burden of completeness would be moved from the container users to the container developers.

One alternative to dynamic analysis is to statistically determine the set of system calls that can be invoked by a container. However, as discussed in Zeng et al.'s work (Zeng et al. 2014), it is typically difficult to identify system call invocation rules in terms of types, sequences, and arguments, even for program developers. This is because system calls are generally not invoked directly but through library APIs. Furthermore, a number of theoretical and practical barriers remain for static analysis-based approaches (Wan et al. 2014; Wan and Zhou 2015). We use the tool *cflow* to analyze the system calls in the source code of the application. For instance, we discover a list of 64 system calls in the source code of the application part for the container *Nginx*. We further compare the list with our mined sandbox for *Nginx* and find that only 34 system calls are overlapped. This indicates that 42 system calls might be invoked through library APIs and 30 system calls are not covered during our automatic testing. To improve the code coverage of automatic testing, container developers could combine our approach with the testing process of the application development.

Risky System Calls Some system calls are riskier than the other. The ability to execute programs (`exec()`) is riskier than the ability to access a file (`access()`) or check a semaphore (`semop()`). We notice that some risky system calls (e.g., `execve()`) are only accessed by the Docker `init` process for initialization before the target containers start running. We can provide two mined sandboxes, one for the initialization phase and the other for the running phase. This also helps to further reduce the attack surface. In addition, during selecting system calls for argument modeling, we plan to provide multiple strategies in our future work, e.g., focusing on more risky system calls.

Diversity of the Container Evaluation Although our experimental results demonstrate the feasibility of sandbox mining for containers, our current evaluation only focuses on two most popular categories of Application containers, i.e., database systems and Web servers, which count for half of all deployed containers. The diversity of containers brings challenges to sandbox mining. First, for the containers that include dynamically generated scripts (e.g., PHP), a variety of pathname for file access exist. An iterative method could be adopted to update models of string type arguments through a longer sandbox mining phase and by using test cases that are more consistent with usage in production. Second, for the OS containers (e.g., BusyBox), they may intend to invoke arbitrary system calls. Sandboxing based on system call interposition is not a suitable solution in this case. We could leverage other Linux security mechanisms to protect those containers. Third, for the containers of distributed systems (e.g., Cassandra), different nodes in the cluster may present different system call

behavior. Thus, a distinct sandbox may be required for each node in the distributed systems; we may have to mine multiple sandboxes for each node in the distributed systems. 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.

False Positives and False Negatives 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 positives.** 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 the mining phase, and thus not added to sandbox rules to be allowed. The number of false alarms can be reduced by better testing.
- **False negatives.** 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 negatives 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 combining our approaches with other security mechanisms.
 - **False negatives seen during sandbox mining.** The container may be initially malicious. We risk mining the malicious behaviors of the container during the mining phase. Thus malicious system calls would be included in the sandbox rules. This issue can be addressed by identifying malicious behaviors during the mining phase.

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.

8 Conclusion and Future Work

In this paper, we present an approach to mine sandboxes for Linux containers. During sandbox mining, the approach first explores the behaviors of a container by automatically running test suites and monitors the system call invocations of the container. The approach then characterizes the system call names and arguments and translates the models of 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 for the selected static test cases. For containers which require access to dynamic file paths, have a deployment of dependent features, or have largely incomplete test cases, our approach may generate an unknown number of false alerts. In our experiment, automatic testing sufficiently covers container behaviors, and sandbox enforcement incurs low overhead.

Future work could be mining more fine-grained sandbox policy, taking into account temporal features of system calls, internal states of a container, or data flow from and to sensitive resources. The more fine-grained sandbox may lead to more false positives and increase performance overhead. It requires to search for sweet spots that both minimize false positives and performance overhead. Future work could also leverage modern test case generation techniques to systematically explore container behaviors. This may help to cover more normal behaviors of a container. Also, for now, we enforce one system call policy on a whole container. However, a container may comprise multiple processes which have distinct behaviors. To further reduce the attack surface, future work could enforce a distinct policy for each process which corresponds to the behavior of that process.

Acknowledgements This research was partially supported by the National Key Research and Development Program of China (2018YFB1003904), NSFC Program (No. 61602403), Project of Science and Technology Research and Development Program of China Railway Corporation (P2018X002), and the Fundamental Research Funds for the Central Universities.

References

- Acharya A, Raje M (2000) Mapbox: using parameterized behavior classes to confine untrusted applications. In: Proceedings of the 9th conference on USENIX security symposium. USENIX Association
- Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, Mcminn P, Bertolino A et al (2013) An orchestrated survey of methodologies for automated software test case generation. *J Syst Softw* 86(8):1978–2001
- Bacis E, Mutti S, Capelli S, Paraboschi S (2015) Dockerpolicymodules: mandatory access control for docker containers. In: 2015 IEEE Conference on communications and network security (CNS). IEEE, pp 749–750
- Bao L, Le TDB, Lo D (2018) Mining sandboxes: are we there yet? In 2018 IEEE 25th International conference on software analysis, evolution and reengineering (SANER). IEEE, p 445–455
- Bhatkar S, Chaturvedi A, Sekar R (2006) Dataflow anomaly detection. In: 2006 IEEE Symposium on security and privacy (S&P'06). IEEE, pp 15–pp
- Cadar C, Sen K (2013) Symbolic execution for software testing: three decades later. *Commun ACM* 56(2):82–90
- Chen TY, Kuo FC, Merkel RG, Tse T (2010) Adaptive random testing: the art of test case diversity. *J Syst Softw* 83(1):60–66
- Ciupa I, Leitner A, Oriol M, Meyer B (2008) Artoo: adaptive random testing for object-oriented software. In: Proceedings of the 30th international conference on software engineering. ACM, pp 71–80
- Corbet J (2009) Seccomp and sandboxing. <https://lwn.net/Articles/332974>, [Online; Accessed 2017-11-28]
- Corbet J (2012) Yet another new approach to seccomp. <http://lwn.net/Articles/475043>, [Online; Accessed 2017-11-28]
- Cowan C (2007) Apparmor linux application security CVE-2016-0728 (2016) CVE-2016-0728. <http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=2016-0728>, [Online; Accessed 2017-11-28]
- DjangoSoftwareFoundation (2015) Django: a high-level Python Web framework. <https://www.djangoproject.com>, [Online; Accessed 2017-11-28]
- DockerDocs (2017) Seccomp security profiles for Docker. <https://docs.docker.com/engine/security/seccomp>, [Online; Accessed 2017-11-28]
- DockerHub (2017a) Docker Hub. <https://hub.docker.com/explore>, [Online; Accessed 2017-11-28]
- DockerHub (2017b) Hello-world container. https://hub.docker.com/_/hello-world, [Online; Accessed 2017-11-28]
- DraisInc (2017) Sysdig. <http://www.sysdig.org>, [Online; Accessed 2017-11-28]
- Endler D (1998) Intrusion detection. Applying machine learning to solaris audit data. In: Computer Security Applications Conference, 1998. Proceedings. 14th Annual. IEEE, pp 268–279
- Felter W, Ferreira A, Rajamony R, Rubio J (2015) An updated performance comparison of virtual machines and linux containers. In: 2015 IEEE International symposium on performance analysis of systems and software (ISPASS). IEEE, pp 171–172

- Fetzer C, Stübkrant M (2008) Switchblade: enforcing dynamic personalized system call models. *ACM SIGOPS Oper Syst Rev* 42(4):273–286
- Forrest S, Hofmeyr SA, Somayaji A, Longstaff TA (1996) A sense of self for unix processes. In: 1996 IEEE Symposium On Security And Privacy, 1996. Proceedings. IEEE, pp 120–128
- Forrest S, Hofmeyr SA, Somayaji A (1997) Computer immunology. *Commun ACM* 40(10):88–97
- Fraser T, Badger L, Feldman M (1999) Hardening cots software with generic software wrappers. In: Proceedings of the 1999 IEEE Symposium on Security and Privacy, 1999. IEEE, pp 2–16
- Gao D, Reiter MK, Song D (2006) Behavioral distance measurement using hidden markov models. In: International workshop on recent advances in intrusion detection. Springer, pp 19–40
- Garfinkel T et al (2003) Traps and pitfalls: practical problems in system call interposition based security tools. In: NDSS, vol 3, pp 163–176
- Garfinkel T, Pfaff B, Rosenblum M et al (2004) Ostia: a delegating architecture for secure system call interposition. In: NDSS
- GlobalIndustryAnalystsInc (2015) Platform as a Service PaaS Market Trends. <http://www.strategyr.com/MarketResearch/Platform.as.a.Service.PaaS.Market.Trends.asp>, [Online; Accessed 2017-11-28]
- Goldberg I, Wagner D, Thomas R, Brewer EA et al (1996) A secure environment for untrusted helper applications: confining the wily hacker. In: USENIX security symposium
- Grubb S (2017) auditd. <http://linux.die.net/man/8/auditd>, [Online; Accessed 2017-11-28]
- Guo PJ, Engler DR (2011) Cde: using system call interposition to automatically create portable software packages. In: USENIX Annual technical conference, p 21
- Hallyn SE, Morgan AG (2008) Linux capabilities: making them work. In: Linux symposium, vol 8
- Harman M, McMinn P (2010) A theoretical and empirical study of search-based testing: local, global, and hybrid search. *IEEE Trans Softw Eng* 36(2):226–247
- Hofmeyr SA, Forrest S, Somayaji A (1998) Intrusion detection using sequences of system calls. *J Comput Secur* 6(3):151–180
- Jain K, Sekar R (2000) User-level infrastructure for system call interposition: a platform for intrusion detection and confinement. In: NDSS
- Jamrozik K, von Styp-Rekowsky P, Zeller A (2016) Mining sandboxes. In: Proceedings of the 38th international conference on software engineering. ACM, pp 37–48
- JSON (2017) Introducing JSON. <http://www.json.org>, [Online; Accessed 2017-11-28]
- Kim T, Zeldovich N (2013) Practical and effective sandboxing for non-root users. In: USENIX Annual technical conference (USENIX ATC 13), pp 139–144
- Kiriansky V, Bruening D, Amarasinghe SP et al (2002) Secure execution via program shepherding. In: USENIX Security symposium, vol 92, p 84
- Ko C, Fraser T, Badger L, Kilpatrick D (2000) Detecting and countering system intrusions using software wrappers. In: USENIX security symposium, pp 1157–1168
- Kopytov A (2017) SysBench. <https://github.com/akopytov/sysbench>, [Online; Accessed 2017-11-28]
- Kruegel C, Mutz D, Valeur F, Vigna G (2003) On the detection of anomalous system call arguments. In: European symposium on research in computer security. Springer, pp 326–343
- Le L (2014) Exploiting nginx chunked overflow bug, the undisclosed attack vector. http://ropshell.com/slides/Nginx_chunked_overflow_the_undisclosed_attack_vector.pdf, [Online; Accessed 2017-11-28]
- Le TB, Bao L, Lo D, Gao D, Li L (2018) Towards mining comprehensive android sandboxes. In: 2018 23rd International conference on engineering of complex computer systems (ICECCS), pp 51–60 <https://doi.org/10.1109/ICECCS2018.2018.00014>
- Liao Y, Vemuri VR (2002) Use of k-nearest neighbor classifier for intrusion detection. *Comput Secur* 21(5):439–448
- MacManus G, Hal, Saelo (2014) Nginx HTTP Server 1.3.9-1.4.0 chunked encoding stack buffer overflow. https://www.rapid7.com/db/modules/exploit/linux/http/nginx_chunked_size, [Online; Accessed 2017-11-28]
- Maggi F, Matteucci M, Zanero S (2010) Detecting intrusions through system call sequence and argument analysis. *IEEE Trans Depend Secur Comput* 7(4):381–395
- Mattetti M, Shulman-Peleg A, Allouche Y, Corradi A, Dolev S, Foschini L (2015) Securing the infrastructure and the workloads of linux containers. In: 2015 IEEE Conference on communications and network security (CNS). IEEE, pp 559–567
- McCarty B (2005) Selinux: Nsa's open source security enhanced linux, vol 238. O'Reilly
- Menage P (2004) CGroups. <https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt>, [Online; Accessed 2017-11-28]
- Merkel D (2014) Docker: lightweight linux containers for consistent development and deployment. *Linux J* 2014(239):2
- Mongodb (2017) Mongo-perf. <https://github.com/mongodb/mongo-perf>, [Online; Accessed 2017-11-28]

- Mosberger D, Jin T (1998) httpf: a tool for measuring web server performance. *ACM SIGMETRICS Perform Eval Rev* 26(3):31–37
- Mutz D, Valeur F, Vigna G, Kruegel C (2006) Anomalous system call detection. *ACM Trans Inf Syst Secur (TISSEC)* 9(1):61–93
- Nie C, Leung H (2011) A survey of combinatorial testing. *ACM Comput Surv (CSUR)* 43(2):11
- OpenContainerInitiative (2017) runc libcontainer version 0.1.1. https://github.com/opencontainers/runc/blob/v0.1.1/libcontainer/standard_init_linux.go, [Online; Accessed 2017-11-28]
- PostgreSQL (2017) pgbench. <https://www.postgresql.org/docs/9.3/static/pgbench.html>, [Online; Accessed 2017-11-28]
- Provos N (2003) Improving host security with system call policies. In: *Unix security*
- redislabs (2017) How fast is Redis? <http://redis.io/topics/benchmarks>, [Online; Accessed 2017-11-28]
- Saltzer JH, Schroeder MD (1975) The protection of information in computer systems. *Proc IEEE* 63(9):1278–1308
- Sekar R, Bendre M, Dhurjati D, Bollineni P (2001) A fast automaton-based method for detecting anomalous program behaviors. In: *2001 IEEE Symposium on security and privacy, 2001. S&P 2001. Proceedings. IEEE*, pp 144–155
- Somayaji A, Forrest S (2000) Automated response using system-call delay. In: *Unix security symposium*, pp 185–197
- Utting M, Legeard B (2010) *Practical model-based testing: a tools approach*. Elsevier
- Vlasenko D (2017) Ptrace documentation. <https://lwn.net/Articles/446593>, [Online; Accessed 2017-11-28]
- Wagner DA (1999) Janus: an approach for confinement of untrusted applications PhD thesis. Department of Electrical Engineering and Computer Sciences, University of California at Berkeley
- Wagner D, Dean R (2001) Intrusion detection via static analysis. In: *2001 IEEE Symposium on Security and Privacy 2001. S&P Proceedings. IEEE*, pp 156–168
- Wan Z, Zhou B (2011) Effective code coverage in compositional systematic dynamic testing. In: *2011 6th IEEE Joint international information technology and artificial intelligence conference, vol 1. IEEE*, pp 173–176
- Wan Z, Zhou B (2015) Points-to analysis for partial call graph construction. *J Zhejiang Univ (Engineering Science Edition)* 49(6):1031–1040
- Wan Z, Zhou B, Wang Y, Shen Y (2014) Efficient points-to analysis for partial call graph construction. In: *International conference on software engineering and knowledge engineering*, pp 416–421
- Wan Z, Lo D, Xia X, Cai L, Li S (2017) Mining sandboxes for linux containers. In: *2017 IEEE International conference on software testing, verification and validation (ICST). IEEE*, pp 92–102
- Warrender C, Forrest S, Pearlmutter B (1999) Detecting intrusions using system calls: alternative data models. In: *1999 IEEE Symposium on proceedings of the security and, privacy. IEEE*, pp 133–145
- Whalen S (2001) An introduction to arp spoofing
- Zeller A (2015) Test complement exclusion: Guarantees from dynamic analysis. In: *Proceedings of the 2015 IEEE 23rd international conference on program comprehension. IEEE Press*, pp 1–2
- Zeng Q, Xin Z, Wu D, Liu P, Mao B (2014) Tailored application-specific system call tables. Tech rep., Technical report, Pennsylvania State University

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.



Zhiyuan Wan is a post-doctoral research fellow in the Software Practices Lab at the University of British Columbia, Canada, and College of Computer Science and Technology, Zhejiang University, China. She was a research scientist in the School of Information Systems, Singapore Management University, Singapore in 2018. She received her Ph.D. and Bachelor degrees from the College of Computer Science and Technology, Zhejiang University, China. Her current research focuses on empirical studies to better understand how software practitioners work, how software is developed, and how software technologies evolve. More information at: <https://zhiyuan-wan.github.io/>.



David Lo received his PhD degree from the School of Computing, National University of Singapore in 2008. He is currently an Associate Professor in the School of Information Systems, Singapore Management University. He has more than 10 years of experience in software engineering and data mining research and has more than 200 publications in these areas. He received the Lee Foundation and Lee Kong Chian Fellow for Research Excellence from the Singapore Management University in 2009 and 2018, and a number of international research and service awards including multiple ACM distinguished paper awards for his work on software analytics. He has served as general and program co-chair of several prestigious international conferences (e.g., IEEE/ACM International Conference on Automated Software Engineering), and editorial board member of a number of high-quality journals (e.g., Empirical Software Engineering).



Xin Xia is a lecturer at the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a post-doctoral research fellow in the software practices lab at the University of British Columbia in Canada, and a research assistant professor at Zhejiang University in China. Xin received both of his Ph.D and bachelor degrees in computer science and software engineering from Zhejiang University in 2014 and 2009, respectively. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. More information at: <https://xin-xia.github.io/>.



Liang Cai is an associate professor of the College of Computer Science at Zhejiang University. He received the BS and PhD degrees from the College of Computer Science in Zhejiang University. He serves as the Vice Dean of College of Software Technology in Zhejiang University, Executive Deputy Director of Zhejiang University Blockchain Research Center. His research interests include blockchain, cloud computing and software engineering.