Observability and Chaos Engineering on System Calls for Containerized Applications in Docker

Jesper Simonsson¹, Long Zhang¹, Brice Morin², Benoit Baudry¹, and Martin Monperrus¹

¹KTH Royal Institute of Technology, Sweden
²SINTEF, Norway

Abstract—In this paper, we present a novel fault injection system called CHAOS ORCA for system calls in containerized applications. CHAOS ORCA aims at evaluating a given application’s self-protection capability with respect to system call errors. The unique feature of CHAOS ORCA is that it conducts experiments under production-like workload without instrumenting the application. We exhaustively analyze all kinds of system calls and utilize different levels of monitoring techniques to reason about the behaviour under perturbation. We evaluate CHAOS ORCA on three real-world applications: a file transfer client, a reverse proxy server and a micro-service oriented web application. Our results show that it is promising to detect weaknesses of resilience mechanisms related to system calls issues.

Index Terms—fault injection, chaos engineering, system call, containers, observability

I. INTRODUCTION

Docker containers are increasingly getting adopted as an efficient mean to package and deploy applications [19]: over two million container images are available on DockerHub and popular images are downloaded millions of times. A typical system is composed of a number of Docker containers interacting with each others. Each container is an instance an image, which describes the software stack that is needed by the application, starting from the OS, for example debian or alpine, all the required dependencies, and so on. The compiled application and configuration files are copied in the image. The images used in production are typically optimized for performances and security, meaning that anything that is not specifically required by the application is stripped-down: in the Docker image there is no ssh access to log into the container, no remote debugger, and so on.

Docker is a key-enabling technology for microservices, an architectural style promoting simple, loosely-coupled, individually updateable services organized around business products [6]. Rather than imposing a generic framework that all microservices need to implement, a key principle of microservices is that different microservices share no code: each microservice is implemented using the technological stack that is best suited. This stack is then be specified as a portable Docker image. As a result, different microservices significantly differ one from another: they can be implemented in different programming languages, use different libraries, and so on. As Docker containers rely on the kernel of the host OS, the only thing they all have in common is the kernel. Thus, they all rely on system calls to delegate certain actions to the guest operating system (OS) kernel, such as networking or writing to a file.

Docker and microservices bring many benefits, in particular increasing the rate at which changes can be brought to production: for example, Netflix releases hundreds of changes to production every day. However, while the state of a monolithic system is rather binary, it either works according to the SLA or not, the state of a complex system built around microservices emerges from their interactions. In a monolithic system, a common framework or middleware would typically take care of providing reliable communications between components. This is no more the case with microservices, where each microservice needs to cope with the failures of other microservices and with the failures of the underlying stack. In this context, how to observe, analyze and improve the resilience of systems built over Docker and microservices?

This paper presents CHAOS ORCA, a tool to conduct chaos engineering experiments to assess the resilience of any Docker-based microservices. Following the principles of chaos engineering CHAOS ORCA supports the following.

First, CHAOS ORCA supports formalizing the steady state of the container, by automatically recording system metrics such as CPU and RAM consumption, network I/O and the system calls invoked by the container. In addition, CHAOS ORCA supports additional metrics provided by developers. Second, CHAOS ORCA supports triggering events than can impact the steady state of the container, by injecting perturbations into the system calls invoked by the containers. For a given system call invoked by the container, a perturbation can delay the actual execution of the system call or can force returning an error code. Third, CHAOS ORCA enables developers to specify experiments on the production system. By default, CHAOS ORCA perturbs the N most invoked system calls for a given container, and monitors it before, during and after the experiment. It then provides a report to the developers summarizing the experiment, for them to verify of falsify the resilience hypothesis for that container. Finally, CHAOS ORCA is built on isolation, it allows experiments to target specific containers, while the other containers running aside are not be impacted by the injected perturbations.

A key feature of CHAOS ORCA is that it is oblivious from the internals of the Docker containers involved in chaos

https://principlesofchaos.org/
experiment. In other words, no matter what software stack runs inside the container, CHAOSORCA will always be able to involve that container in a chaos experiment. Our empirical evaluation on T Torrent (written in Java), Nginx (written in C) and Bookinfo (written in Java, JavaScript, Python and Ruby) shows that CHAOSORCA 1) can observe Docker-based microservices of very different nature, 2) can identify issues related to the resilience of those microservices against perturbation in system calls, 3) can improve the observability of microservices and the relevance of chaos experiments and 4) it can be applied in production with low overhead, though not all kinds of overhead are equal.

To sum up our contributions are:

- A novel architecture for achieving observability and system call perturbation for containerized applications in Docker, able to analyze the resilience of Docker applications in a black-box manner.
- A prototype implementation based on the state-of-art Devops libraries used in industry, applicable to off-the-shelf Docker microservices. The prototype is called CHAOSORCA and it is publicly available at https://github.com/KTH/royal-chaos/tree/master/chaosorca.
- An evaluation of the approach on three case studies, representing typical polyglot microservices in Docker, namely T Torrent (written in Java), Nginx (written in C) and Bookinfo (written in Java, JavaScript, Python and Ruby).

The remainder of the paper is organized as follows. Section II gives a brief introduction about relevant concepts. Section III discusses the design of CHAOSORCA like the requirements, components and implementation. Section IV answers the 4 research questions by discussing about the evaluation experiments. Section V compares the related work in fault injection and observability. Section VI concludes the whole paper.

II. BACKGROUND

A. System Call

A system call is the fundamental interface between an application and the kernel. In the Linux operating system, there are more than 300 unique system calls and more than 100 different error codes that can be returned from system calls. System calls are primitives exposed by the OS kernel to deal with crucial resources like hardware devices, network or files in a systematic and safe manner. When an application requests the use of such a resource, it has to interact with the OS kernel through a system call. System calls are typically not invoked directly, but rather invoked through corresponding wrapper functions in the core library (in many cases the libc).

It is very common for applications to invoke system calls, even if developers are not aware of such invocations. Figure 1 is a flame graph that illustrates user events and kernel events when downloading a file with T Torrent. In the figure, the green (light) blocks stand for user events, caused by Java code. The red (dark) ones are events related to system calls inside the kernel. For T Torrent, those system calls mostly happen when it opens a file, uses network resources or writes downloaded blocks into a file.

B. Containerization

Containerization is a lightweight virtualization mechanism to support rapid shipment and deployment of applications. A containerized application relies on the kernel of the host OS. Sharing the kernel and avoiding the need for a guest OS typically yields better performance and a reduced footprint. One of the most popular implementations of containerization is Docker. Docker supports process-level isolation, with two concepts from the kernel that are used: namespaces and cgroups. Namespaces are used to isolate certain resources: a network stack, its own processes tree and other resources. For example, a container cannot see or interact with the processes running in another container. The cgroups are used for specifying what computer resources a container should be able to access, such as certain amount of RAM, CPU cores, file system access and similar.

Docker containers can share namespaces. For example, sharing process ID (PID) namespace allows one container to access other containers’ processes. Sharing an inter-process communication (IPC) namespace means that containers use the same network stacks and memory region. The feature of shareable namespaces in Docker is key for CHAOSORCA: it enables it to conduct experiments without modifying existing containers.

C. Observability

Observability is the ability to collect values about some parts of the internal state of a system, based on its external behaviour. Increasing a system’s observability is extremely valuable for debugging and reasoning about failures. Observability leverages three types of strategies: well designed event logs, sufficient monitoring metrics and tracing. An event log can be in different forms, plain text, structured or binary. Monitoring metrics usually combine information at different levels. For example, CPU and memory usage are useful for finding out problems like infinite loops or memory leaks. Also, the ratio of successful HTTP requests is helpful to describe whether the system is running under acceptable status. Tracing is a technique that combines distributed but related events.

![Figure 1: Flamegraph from T Torrent Execution. The green part shows application code, the red part shows OS code.](image-url)
together, such as a user request which spans multiple microservices. Tracing is hard as it requires each application in the request path to propagate tracing information [39].

D. Chaos Engineering

Chaos engineering is a technique that evaluates resilience in production environment. When an application is deployed into production, it faces uncertainties like unstable network, unavailable third-party services and unbalanced traffic. A well-designed application should be able to bear these unanticipated scenarios without any serious impact for the users. Tools for chaos engineering actively inject different kinds of failures into a production system and try to learn how the system behaves under perturbation. If the system behaves as expected, such experiments improve the confidence the developers have about the resilience of the system. Otherwise, the experiment points out weaknesses in the ability of the system to handle failures [23].

There are several principles in chaos engineering. The very first one is to build a series of hypotheses around the system’s steady state. The steady state is defined based on monitored metrics. A hypothesis describes how the steady state changes under a certain event which is triggered during chaos engineering experiments. Since these experiments are done in production environment, another principle is that the influence of the experiments on end users should be kept as small as possible.

III. Design of ChaosORCA

Following the principles of chaos engineering, ChaosORCA aims at improving the observability and automatically conducting chaos experiments in containerized applications. ChaosORCA supports monitoring, system call perturbation and chaos experiment orchestration. The core design principle is to gather sufficient insights about resilience, while causing minimal influence on the system under study. The requirements for designing ChaosORCA are described as follows:

Requirement 1. Black-box containers: ChaosORCA should consider containers as black-boxes. In a production system, the images specifying the containers are often optimized for performance and security, meaning that it is impractical, if even possible, to weave-in extra instrumentation in the image directly. All actions by ChaosORCA should thus be implemented in a way that does not require instrumenting the containers. In other terms, ChaosORCA should consider containers as pure, untouchable black-box entities. A positive implication of this requirement is that it reduces the effort of developers to the minimum for using ChaosORCA: an application container is deployed normally and ChaosORCA is attached to it when developers need to run chaos experiments.

Requirement 2. Monitoring: As mentioned in Section II, observability is a foundation to analyze how the system behavior differs during a chaos experiment. Given Requirement 1, the internals of a container cannot be directly observed. ChaosORCA must monitor the surroundings of the containers and gather information to assess resilience weaknesses. This includes monitoring the impact that the containers have on the physical resources (RAM, CPU), on the I/O (network activity), and how they use system calls on the host kernel.

Requirement 3. System call perturbation: ChaosORCA is designed for analyzing resilience with respect to system calls. Thus, ChaosORCA should be able to perturb a specific type of system call. The form of perturbations also include 1) force a system call to directly return with an error code, 2) add a delay before a system call is actually completed, and 3) a combination of delay and error injection.

Requirement 4. Minimizing blast radius: In a production system, a server may host hundreds of containers, it is thus critical for ChaosORCA to be able to target specific containers without influencing the others, so as to limit the blast radius of chaos experiments. Within this specific container, ChaosORCA should be able to only perturb one single process as well.

A. Overview of ChaosORCA

To fulfill the requirements listed above, ChaosORCA is divided into three components: 1) monitor, 2) perturbator and 3) orchestrator. The interactions among them are described in Figure 2. The monitor component is responsible for capturing the system behavior at runtime. The perturbator injects different kinds of failures with respect to system calls. The orchestrator controls the monitor and perturbator components to conduct chaos experiments and generates reports.

B. The ChaosORCA Monitor Component

Observability is essential to control and analyze chaos experiments. The monitor component implements requirement 2. In order to connect low-level system call perturbations with application-level behavior, the monitor component provides different levels of observability, respectively 1) container level, 2) system call level and 3) application level.

At the container level, the monitor records CPU time, memory usage, network traffic as follows. The monitoring component relies on shareable namespaces offered by Docker, it runs as a process which shares a specific namespace with the target container under study. This type of design fulfills requirement 1: the monitor provides strong observability with no modification or intrusion into the target container image.

For the system call level, the monitor component records the invoked system calls information including type, arguments and result. Regarding application level monitoring, the monitor cares about whether the application fulfills a user request, where the definition of fulfillment varies from one application to another. For example, for a web server, the monitor observes the HTTP response codes and the latency in handling HTTP requests.

C. The ChaosORCA Perturbator Component

The perturbator component is designed to inject system call failures at runtime. It addresses both requirement 3 and 1.
system call perturbation is defined as a tuple \((s, e, d)\) where \(s\) represents the type of a system call, \(e\) means the error code of this system call to be injected, and \(d\) denotes the delay before this system call is actually invoked. In CHAOSORCA, the perturbator works as follows, it intercepts an existing system call that is invoked by the application, then ask for commands from the orchestrator (cf. Section III-D) about whether injecting a specific failure in this system call. Per requirement \(5\), the perturbator supports failures that could be either an error code, or a certain delay, or a combination of both.

Let us consider the example of the TTorrent file download procedure shown in Figure. When TTorrent needs to save a piece of downloaded file on disk, the logic in method `PieceStorageImpl/savePiece` is executed. This leads the JVM (i) to invoke a `pwrite` system call, and (ii) to write bytes of data into a file through this system call. Our perturbator can target such types of system calls, it can intercept this `pwrite` system call and inject a failure in TTorrent.

For the perturbator component, it is also important to minimize the blast radius, according to requirement \(3\). Thanks to the shareable namespaces feature of Docker, introduced in Section II-B, it can perturb a specific running process in the target container, without influencing any other processes or containers.

**D. The CHAOSORCA Orchestrator Component**

The orchestrator component acts as an interface between CHAOSORCA and the fault injection strategies defined by the developers. It provides a command line interface for developers who want to apply CHAOSORCA. Based on the configuration, the orchestrator controls the monitor and perturbator components to explore a set of system call perturbations in the target application. Such a procedure is described in Algorithm 1. Line 1-2 are executed to gather the application’s normal behaviour. The triple for-loops construct all possible system call failures and gather the perturbed behaviour for a comparison.

There are three phases for a fault injection experiment: 1) before perturbation, 2) during perturbation and 3) after perturbation as follows. First of all, before perturbation, the orchestrator attaches the monitor, the perturbator and itself to the target application. It feeds the application with a piece of workload in order to gather information about the normal behaviour of this application. After that, the orchestrator activates the perturbator to inject a specific system call failure defined in Section III-C this is the perturbation phase. Finally, the perturbator is turned off and the after-perturbation phase starts, the orchestrator keeps pulling monitoring information and computes the metric differences. With those three phases, CHAOSORCA enables itself to detect whether a system call failure has an instantaneous influence (in phase during-perturbation) or it has a long lasting influence even after the perturbation (in phase during-after).

Regarding the behaviour analysis in the during- and after-phases in our experiments, CHAOSORCA considers a mix of generic and domain specific dimensions. The generic dimension is 1) whether the application continues to run without a crash and 2) how the metrics under monitoring vary compared to nominal behaviour. CHAOSORCA also considers domain specific dimensions. For example, for web applications, CHAOSORCA compares whether the HTTP response code is 200. A procedure of comparing an application’s perturbed behaviour with normal behaviour is called “Diff” in Algorithm 1.

**E. The CHAOSORCA Visualization Component**

By default all metrics monitored by CHAOSORCA are saved into a time-series based database. The developers can query this data as a table and investigate them after fault injection experiments. CHAOSORCA also has a separate component that visualizes all the monitored metrics as line charts.

For the convenience of resilience analysis, the following features are supported by the visualization component: 1) auto-
Algorithm 1 Fault Injection Experiments for Containers

Input:
- An application Docker container \( A \);
- A repeatable workload for this application \( W \);
- A set of system call types \( T \);
- A set of system call error codes \( E \);
- A set of delay values \( D \);

Output:
- \( R \) a map of behaviour difference per perturbation;

1: Attach monitor, feed \( A \) with \( W \)
2: normal\_behaviour ← monitored metrics;
3: for each syscall type \( t \in T \) do
4:   for each error code \( e \in E \cup \{ 0 \} \) do
5:     for each delay \( d \in D \cup \{ 0 \} \) do
6:       if \( e = 0 \) and \( d = 0 \) then
7:         Continue; // same to normal execution
8:       end if
9:       perturbation \( p \leftarrow < t, e, d > \);
10:      Feed \( A \) with \( W \) under \( p \)
11:     perturbed\_behaviour ← monitored metrics;
12:    diff ← Diff(normal\_behaviour, perturbed\_behaviour);
13:   \( R \leftarrow R \cup \{ < t, e, d > : diff \} \)
14:  end for
15: end for
16: end for
17: return \( R \);

Fig. 3: CHAOSORCA Visualization Component, with one monitored metric (green line), and one perturbation campaign (blue overlay)

The line chart describes how the CPU usage changes during fault injection experiments. A blue overlay indicates a period of system call perturbations, which also divides the figure into three phases: before, during, and after the perturbations. The green line explains that the CPU usage drops during the perturbations and catches up afterward. The CHAOSORCA visualization component draws one line chart for each of the monitored metrics similarly.

F. Implementation

The monitor component uses cAdvisor\(^2\) to gather containerspecific metrics like CPU usage, memory usage and network traffic metrics. For monitoring HTTP requests, the monitor combines PyShark\(^3\), a python library for TShark to get HTTP response code and latency. For collecting the system calls the monitor uses the open-source tool Bpftrace\(^4\) to observe the number and type of system calls. All the above monitoring information is pushed to a Prometheus time-series database\(^5\).

Then CHAOSORCA uses Grafana\(^6\) to visualize the data for developers. The perturbator component is mainly implemented with Strace\(^7\), which is a diagnostic, debugging and instructional userspace utility for Linux. The orchestrator component itself is written in Python. It provides a command line interface to developers to define system call perturbation arguments like type, error code and delay. CHAOSORCA is publicly-available at [https://github.com/KTH/royal-chaos/tree/master/chaosorca](https://github.com/KTH/royal-chaos/tree/master/chaosorca).

IV. Evaluation

This section discusses the evaluation of CHAOSORCA, which focuses on the following 4 research questions:

RQ 1. What system call perturbations are of interest for containerized applications?

RQ 2. What kind of resilience problems are identified with fault injection in containerized applications?

RQ 3. How is multi-layer observability helpful to analyze the results of chaos experiments?

RQ 4. What is the overhead of conducting chaos experiments using CHAOSORCA?

A. Subject Programs

We experiment CHAOSORCA on three different applications, in order to evaluate its ability at perturbing system calls and at detecting resilience weaknesses in containerized applications. We selected the applications according to the following criteria: 1) medium-sized, well implemented applications with source code available, 2) contain both client side and server side code 3) can be deployed as Docker containers and 4) it is possible to simulate production-like workloads in the laboratory.

![https://github.com/google/cadvisor](https://github.com/google/cadvisor)
[https://kiminewt.github.io/pyshark](https://kiminewt.github.io/pyshark)
[https://github.com/iovisor/bpftrace](https://github.com/iovisor/bpftrace)
[https://prometheus.io/](https://prometheus.io/)
[https://grafana.com/](https://grafana.com/)
[https://strace.io/](https://strace.io/)
The first application is TTorrent, a file downloading client that uses the BitTorrent protocol. The second application is Nginx, a popular and mature reverse proxy. The last one is Bookinfo, a micro-service based reference application which runs several containers written in different languages. The key descriptive metrics of those subjects are shown in Table I (e.g. the number of containers, programming languages, lines of code).

B. Experimental Protocol

Protocol for RQ1: In order to select the most interesting system calls for chaos experiments, we consider two dimensions: 1) the prevalence of a system call during execution 2) the functionality of the system call under study. The first dimension is investigated by monitoring applications. The second one needs more domain knowledge. For example, an application which frequently writes files invokes write a lot. So write takes a large ratio in all happened system calls. Before invoking write, another system call open needs to be invoked. It opens the target file and returns the file descriptor. open system call may not be invoked lots of times, but it is still interesting since write, read system calls rely on it.

Every application runs for 1 minute under a simulated workload, and we perform 3 rounds of executions in total for a statistical reason. In this experiment, CHAOSORCA only monitors the applications and does not inject any failure. The workloads are as follows: TTorrent is executed to download an Ubuntu installation file from the internet, the file is 1.86GB in size. For Nginx, three types of valid GET requests including /, /abc and /abc?cache=bust are sent to the container. For Bookinfo, a collection of valid GET and POST requests are used to trigger the following features: the login logic; accessing the main page; accessing a product page and logging out. For all three applications, CHAOSORCA calculates the rate of different types of system calls.

Protocol for RQ2: For each system call identified as relevant in RQ1, three different perturbation strategies are evaluated according to Algorithm 1: 1) injecting a system call error code without any delay 2) injecting a certain delay after which the system call succeeds and 3) injecting both an error code and a delay. We run chaos experiments with the 6 error codes and 2 different delay values (1s and 5s). The workload for these applications are the same with RQ1. Both generic

and domain specific dimensions described in Section III-D are used for behaviour comparison.

Protocol for RQ3: The experiment protocol is the same as for RQ2. However, for this research question, we focus on monitoring and data visualization. Consequently, RQ3 is discussed qualitatively. For example, by removing part of observability provided by CHAOSORCA, such as closing HTTP response code monitoring, we analyze whether it is still possible to correctly detect resilience problems.

Protocol for RQ4: A study of the overhead of CHAOSORCA is important to assess its applicability. To answer RQ4, we respectively evaluate the overhead caused by the monitor component and the perturbator component. For the network overhead, Nginx is used as a target application. Then, the tool Siege is used to simulate a number of concurrent users sending requests for 3 minutes. The number of users is respectively 5, 10, 25, 200, 1000, 2000 during the experiment.

For system call monitoring and perturbation overhead, in order to stimulate a high amount of system calls, dd command is used as a target application. dd is a Linux command which mainly coverts and copies files. It invokes a huge amount of read and write system calls. The command to test dd is `dd if=/dev/zero of=/dev/null bs=1 count=500000k`, which copies zero to the void for 500 million times.

C. RQ1: What system call perturbations are of interest for containerized applications?

The statistics of the invoked system calls are listed in Table II. Each line describes one of the system call invoked during the execution. Each column shows the number and percentage of system call invocations in one application. A blank cell means that a system call is never invoked by the corresponding application under the given workload. Some system calls are rarely used by the application, which are noted as < 0.01. For example, the first row of the table tells that TTorrent invokes the accept system call 1780 times every second, which takes about 9.92% of the total. While Nginx never invokes the accept system call, Bookinfo invokes it 293.45 times per second which takes 3.90% of the total.

From Table II we could see that the types of most used system calls differ in applications. For example, TTorrent invokes read and futex most frequently. This actually makes sense because TTorrent takes advantages of multi-threading to download files. System calls like read, futex are keys to support TTorrent’s functionality. Thus these most used system calls are of interest for evaluating TTorrent. Bookinfo invokes the most types of system calls during an experiment, because it is micro-service based and has a more complex business logic. For HTTP based applications including Nginx and Bookinfo, the system calls used in common are clone, getpid, getsockoptname, newfstat, newstat, open, recvfrom, setsockopt and write.

References:

11Siege (https://github.com/JoeDog/siege) is an http load tester which generates requests at a fast rate.
TABLE II: The Statistics of System Calls per Second for Each Application under The Given Workload without Any Perturbations

| System Call   | T Torrent | Nginx | Bookinfo |
|---------------|-----------|-------|----------|
| accept        | 1780 (9.92%) | 2.21 (1.17%) | 293.45 (3.90%) |
| bind          | 166.57 (9.17%) | 237.72 (3.09%) | 280.37 (3.73%) |
| close         | 92 (0.51%) | 163.03 (2.17%) | 19.24 (0.16%) |
| connect       | 3240 (18.06%) | 174.36 (2.32%) | 175.95 (2.34%) |
| dup           | 64.31 (0.36%) | 174.36 (2.32%) | 174.36 (2.32%) |
| dup2          | 62.4 (0.35%) | 161.89 (2.15%) | 161.89 (2.15%) |
| fdctl         | 3240 (18.06%) | 854.96 (11.36%) | 854.96 (11.36%) |
| getpud        | 160.52 (8.84%) | 159.6 (2.12%) | 159.6 (2.12%) |
| getsockname   | 48.77 (2.69%) | 174.84 (2.32%) | 174.84 (2.32%) |
| ioctl         | 167.27 (2.22%) | 189.22 (2.51%) | 189.22 (2.51%) |
| list          | 247.43 (3.29%) | 247.43 (3.29%) | 247.43 (3.29%) |
| listen        | < 0.01 | < 0.01 | < 0.01 |
| madvise       | 79.28 (0.44%) | 176.86 (2.35%) | 176.86 (2.35%) |
| mmap          | 191.09 (1.07%) | 157.48 (0.21%) | 157.48 (0.21%) |
| mprotect      | 677 (3.77%) | 206.71 (2.74%) | 206.71 (2.74%) |
| mmcmp         | < 0.01 | < 0.01 | < 0.01 |
| newfstat      | 89.96 (4.95%) | 100.27 (17.27%) | 100.27 (17.27%) |
| newlstat      | < 0.01 | 89.96 (4.95%) | 100.27 (17.27%) |
| newstat       | 166.15 (9.15%) | 222.19 (2.95%) | 222.19 (2.95%) |
| poll          | 1410 (7.86%) | 181.12 (2.41%) | 181.12 (2.41%) |
| read          | 4350 (23.90%) | 158.48 (2.11%) | 158.48 (2.11%) |
| recvfrom      | 90.37 (4.98%) | 130.69 (1.74%) | 130.69 (1.74%) |
| recvmsg       | 388.57 (5.16%) | 174.7 (2.32%) | 174.7 (2.32%) |
| sendfile       | 264.61 (14.57%) | 161.78 (2.15%) | 161.78 (2.15%) |
| sendfile64    | 29.41 (1.62%) | 287.41 (3.82%) | 287.41 (3.82%) |
| signal        | 46.78 (0.26%) | 170.71 (2.27%) | 170.71 (2.27%) |
| sendmsg       | 3040 (16.95%) | 263.94 (3.51%) | 263.94 (3.51%) |
| sendmsg       | 3210 (11.76%) | 238.31 (3.17%) | 238.31 (3.17%) |
| sendmsg       | 87.04 (4.79%) | 284.63 (15.68%) | 284.63 (15.68%) |
| sendmsg       | 55.18 (0.31%) | 284.63 (15.68%) | 284.63 (15.68%) |

With the help of CHAOSorca, developers are able to focus more on the most critical system calls per application. This makes the following resilience evaluation more targeted and effective.

Answer to RQ1

Over our three case studies, we observe a total of N different types of system calls exercised by the applications. By analyzing the prevalence and the functionality of each of them, 9 types of system calls are identified as relevant for fault injection experiments in containerized applications, namely open, write, writev, read, readv, sendfile, sendfile64, poll and select.

D. RQ2: What kind of resilience problems are identified with fault injection in containerized applications?

According to the experiment protocol, we have conducted 9x(6 + 2 + 6x2) = 180 rounds of experiments. After finishing all the experiments, we qualitatively analyzed the influence caused by each perturbation. In the following, we discuss the most used system calls. The interested reader can access to the comprehensive experimental data at [https://github.com/KTH/royal-chaos/tree/master/chaosorca](https://github.com/KTH/royal-chaos/tree/master/chaosorca).

1) Evaluations on T Torrent: [Table III] shows a sample of experiment results in T Torrent. As it is a file downloading client, system calls like open, write, read are the most used ones. In this table, every row describes one perturbation and T Torrent’s corresponding behaviour. Each cell is the result of the comparison of the application’s behaviour under and after perturbation.

From [Table III] it can be seen that only adding a delay to open, write, read system calls have a critical influence on network usage. Especially row 21 and 22 show that the read system call perturbation causes the biggest dip in network IO. Recalling [Table II] this is consistent because the read system call is the most common one, it occurs 20 percent of the total number of calls.

Another insight from the table is that for the open system call perturbations, network IO dips initially during the perturbation and then catches up after. For write and read system call perturbations, the network traffic stops completely and the number of system calls drops to nearly zero. This indicates that T Torrent is able to bear system call delays but is more sensitive to system call errors.

2) Evaluations on Nginx: [Table IV] describes the most interesting findings from our experiments on Nginx. For Nginx, the perturbations that have an obvious effect are those on open, write, writev and sendfile. As an example, the first row records the Nginx behaviour under an open system call perturbation: adding a 1 second delay with no system call error code. However, the HTTP response code becomes 0 and the latency increases to more than 50 ms. The number of total system calls decreases during the perturbation compared to the normal execution. The network and CPU usage stay low while some spikes occur. The usage of memory increases during the experiment. This proves that Nginx is able to handle open system call delays and the usage of calculation resources stays acceptable.

Row 1 to row 20 shows that an open system call perturbation has different effects depending on the injected error code. An EACCESS error results in HTTP 403 Forbidden, an ENOENT error leads to 404 Not Found instead. An EPERM, EINVAL, or ENOSYS results in HTTP 500 Internal Server Error. This could be considered as a good practice because developers designed a more detailed reaction strategy with respect to different system call errors.

Another interesting insight to look at is the read system call perturbations (row 26 to row 30). When such a perturbation is injected, the application still functions normally as the HTTP response code is still 200. The trend of system calls and the network IO stay normal as well. According to these experiments, developers gain more confidence that Nginx is able to survive from read system call errors.
### Table III: Representative Sample of Chaos Experiments on TTorrent

| 0 | System Call | Error Code | Delay | Trend of Syscalls | Network IO | Cpu Usage | Memory Usage |
|---|--------------|------------|-------|-------------------|------------|-----------|--------------|
| 1 | open         | -          | 1s    | fewer, spike      | small dip  | increase  | normal       |
| 2 | open         | -          | 5s    | increase          | dip        | increase  | normal       |
| 3 | open         | EACCESS    |       | increase          | normal     | increase  | normal       |
| 4 | open         | EACCESS    | 1s    | increase          | small dip  | increase  | normal       |
| 5 | open         | EACCESS    | 5s    | increase          | dip        | increase  | normal       |
| 6 | write        | -          | 1s    | tiny              | tiny dip   | normal    |              |
| 7 | write        | -          | 5s    | small             | tiny dip   | normal    |              |
| 8 | write        | EACCESS    |       | tiny              | stops      | tiny, tiny| normal       |
| 9 | write        | EACCESS    | 1s    | tiny, tiny spike  | stops      | tiny, tiny| normal       |
| 10| write        | EACCESS    | 5s    | tiny, small spike | stops      | tiny, tiny| normal       |
| 11| write        | EPERM      | -     | tiny              | stops      | tiny, tiny| normal       |
| 12| write        | EPERM      | 1s    | tiny, small spike | stops      | tiny, tiny| normal       |
| 13| write        | EPERM      | 5s    | small, increase   | tiny, small| normal   |              |
| 14| write        | ENOENT     | -     | small, spike      | stops      | tiny, tiny| normal       |
| 15| write        | ENOENT     | 1s    | spike, tiny       | stops      | tiny, tiny| normal       |
| 16| write        | ENOENT     | 5s    | small, spike      | stops      | tiny, tiny| normal       |
| 17| write        | EIO        | -     | small, spike      | stops      | tiny, tiny| normal       |
| 18| write        | EIO        | 1s    | small, small spike| stops      | tiny, tiny| normal       |
| 19| write        | EIO        | 5s    | small, small spike| stops      | tiny, tiny| normal       |
| 20| read         | -          | 1s    | big dip           | big dip    | normal    |              |
| 21| read         | -          | 5s    | big dip           | big dip    | normal    |              |
| 22| read         | EACCESS    |       | tiny              | stops      | tiny, tiny| normal       |
| 23| read         | EACCESS    | 1s    | tiny              | stops      | tiny, tiny| normal       |
| 24| read         | EACCESS    | 5s    | small, small      | stops      | tiny, tiny| normal       |
| 25| read         | EPERM      | -     | tiny              | stops      | tiny, tiny| normal       |
| 26| read         | EPERM      | 1s    | tiny              | stops      | tiny, tiny| normal       |
| 27| read         | EPERM      | 5s    | tiny              | stops      | tiny, tiny| normal       |
| 28| read         | ENOENT     | -     | tiny              | stops      | tiny, tiny| normal       |
| 29| read         | ENOENT     | 1s    | tiny              | stops      | tiny, tiny| normal       |
| 30| read         | ENOENT     | 5s    | small             | stops      | tiny, tiny| normal       |

### Table IV: Representative Sample of Chaos Experiments on Nginx

| 0 | System Call | Error Code | Delay | HTTP Res. Latency | Trend of Syscalls | Network IO | Cpu Usage | Memory Usage |
|---|-------------|------------|-------|-------------------|-------------------|------------|-----------|--------------|
| 1 | open        | -          | 1s    | 0                 | up after 50ms     | fewer during| tiny, spike| tiny, spike  |
| 2 | open        | -          | 5s    | 403               | 0                 | up after 96ms| tiny, spike| tiny, spike  |
| 3 | open        | EACCESS    | 1s    | 403 (fewer)       | increasing 102ms  | fewer, spike| tiny, spike| tiny, spike  |
| 4 | open        | EACCESS    | 5s    | 403 (FEW)         | increasing 140ms  | fewer, spike| tiny, spike| tiny, spike  |
| 5 | open        | EPERM      | 500   | 0                 | fewer            | small      | smaller    | increasing   |
| 6 | open        | EPERM      | 1s    | 500 (fewer)       | increasing 57ms   | normal     | spike, tiny| spike, tiny  |
| 7 | open        | EPERM      | 5s    | 500 (FEW)         | increasing 175ms  | FEWER, spike| spike, tiny| spike, tiny  |
| 8 | open        | ENOENT     | 404   | 0                 | fewer            | small      | small      | increasing   |
| 9 | open        | ENOENT     | 1s    | 404 (fewer)       | increasing 150ms  | normal     | tiny, spike| tiny, spike  |
| 10| open        | ENOENT     | 5s    | 404 (FEW)         | increasing 97ms   | FEWER, spike| tiny, spike| tiny, spike  |
| 11| open        | EIO        | 500   | 0                 | fewer            | small      | smaller    | increasing   |
| 12| open        | EIO        | 1s    | 500 (fewer)       | increasing 96ms   | increase   | spike, tiny| spike, tiny  |
| 13| open        | EIO        | 5s    | 500 (FEW)         | increasing 172ms  | FEWER, spike| spike, tiny| spike, tiny  |
| 14| open        | EIO        | 5s    | 500 (FEW)         | increasing 149ms  | FEWER, spike| tiny, spike| tiny, spike  |
| 15| open        | EIO        | 5s    | 500 (FEW)         | increasing 159ms  | FEWER, spike| tiny, spike| tiny, spike  |
| 16| open        | EIO        | 5s    | 500 (FEW)         | increasing 70ms   | FEWER, spike| tiny, spike| tiny, spike  |
| 17| open        | EIO        | 5s    | 500 (FEW)         |              | FEWER, spike| tiny, spike| tiny, spike  |
| 18| open        | ENOSYS     | 500   | 0                 | normal           | small      | smaller    | increasing   |
| 19| open        | ENOSYS     | 1s    | 500 (fewer)       | increasing 149ms  | normal     | spike, tiny| spike, tiny  |
| 20| open        | ENOSYS     | 5s    | 500 (FEW)         | increasing 159ms  | FEWER, spike| tiny, spike| tiny, spike  |
| 21| write       | -          | 1s    | up after 75ms     | fewer            | spike, tiny| spike, tiny| spike, tiny  |
| 22| write       | -          | 5s    | up after 51ms     | fewer, spike     | tiny, tiny | spike, tiny| spike, tiny  |
| 23| write       | EACCESS    | 200   | 0                 | normal           | normal     | normal     | increasing   |
| 24| write       | EACCESS    | 5s    | 0                 | up after 107ms   | FEWER, spike| spike, tiny| spike, tiny  |
| 25| read         | -          | 1s    | 200               | normal           | normal     | normal     | increasing   |
| 26| read         | -          | 5s    | 200               | normal           | normal     | normal     | increasing   |
| 27| read         | EACCESS    | 200   | 0                 | normal           | normal     | normal     | increasing   |
| 28| read         | EACCESS    | 5s    | 200               | normal           | normal     | normal     | increasing   |
| 29| read         | EACCESS    | 5s    | 200               | normal           | normal     | normal     | increasing   |
| 30| read         | EACCESS    | 5s    | 200               | normal           | normal     | normal     | increasing   |
| 31| sendfile     | -          | 1s    | up after 35ms     | fewer            | spike, tiny| spike, tiny| spike, tiny  |
| 32| sendfile     | -          | 5s    | up after 172ms    | FEWER, spike     | tiny, spike| spike, tiny| spike, tiny  |
| 33| sendfile     | EACCESS    | 0     | 0                 | increase         | small      | normal     | increasing   |
| 34| sendfile     | EACCESS    | 1s    | up after 99ms     | normal           | spike, tiny| spike, tiny| spike, tiny  |
| 35| sendfile     | EPERM      | 0     | 0                 | normal           | small      | normal     | increasing   |
| 36| sendfile     | EPERM      | 1s    | up after 64ms     | fewer            | spike, tiny| spike, tiny| spike, tiny  |
TABLE V: Representative Sample of Chaos Experiments on Bookinfo

| System Call | Error Code | Delay | HTTP Res. | Latency | Trend of Syscalls | Network IO | Cpu Usage | Memory Usage |
|-------------|------------|-------|-----------|---------|-------------------|------------|-----------|-------------|
| open        | -          | 1s    | 200 / 302 | increase 5ms | normal | normal | increase | normal |
| open        | -          | 5s    | 200(lower) / 302 | up after 2ms | increase | normal | increase | increase |
| open        | EACCESS    | -     | 200(lower) / 302 | 0 | fewer, up | smaller | increase | normal |
| read        | -          | 1s    | 200(fewer) / 302 | increase 2ms | increase | smaller | increase | increase |
| read        | -          | 5s    | 200(FEW) / 302 | up after 4ms | increase, spike | small, spike | increase, spike | increase |
| read        | EACCESS    | -     | 0 | 0 | increase, spike | small, spike | smaller, spike | increase |
| read        | EACCESS    | 1s    | 0 | up after 1ms | normal, spike | small, spike | smaller, spike | increase |
| select      | -          | 1s    | 200(FEW) / 302 | up after 6ms | normal, spike | tiny, spike | tiny, spike | small, spike |
| select      | -          | 5s    | 0 | up after 12ms | - | - | - | - |
| select      | EACCESS    | -     | crash | - | - | - | - | - |
| select      | EACCESS    | 1s    | crash | - | - | - | - | - |
| select      | EACCESS    | 5s    | crash | - | - | - | - | - |
| select      | EPERM      | -     | crash | - | - | - | - | - |
| select      | EPERM      | 1s    | crash | - | - | - | - | - |
| select      | EPERM      | 5s    | crash | - | - | - | - | - |

3) Evaluations on Bookinfo: Table V gives a representative sample of our experimental results. For the Bookinfo application, the system call perturbations that have a significant effect are open, write, read, poll and select. Compared to Nginx, when CHAOSORCA injects delays into the open system call, the code of responses are divided into 200 and 302. Code 302 means a redirection to another url. Considering the Bookinfo containers as a black box, we assume that this is because the injected delay causes a timeout exception in Bookinfo’s components. In order to lead users to a retry logic, Bookinfo redirects the request to an initial page. For the select system call perturbations, Bookinfo directly crashes if an error code is injected. This indicates that the application is sensitive to select system call errors, and its self-protection strategies are not able to pull Bookinfo back to normal when such a perturbation happens.

Answer to RQ2

By performing fault injection in three different containerized applications, we show that two kinds of resilience problems are identified by CHAOSORCA: 1) the application is not resilient: if certain system calls fail, the whole application crashes and 2) the application is sensitive to system call errors: if certain system calls fail, the application consumes significantly more system resources.

E. RQ3: How is multi-layer observability helpful to analyze the results of chaos experiments?

In order to improve the observability of target applications, the monitoring component in CHAOSORCA records different levels of metrics: 1) system call level, 2) container level and 3) application level. In order to easily compare the changes under perturbation with a normal execution, all of these metrics are visualized by CHAOSORCA in a Grafana dashboard.

For example, Figure 4 is a metrics visualization of an experiment on Nginx. In this series of experiments, CHAOSORCA injects an EACCESS error every time when the open system call is invoked. In the figure, a blue line stands for a specific system call perturbation. From the figure it can be seen that CHAOSORCA monitors Nginx before any perturbations for 5 minutes. Then it keeps intercepting open system calls and injecting an EACCESS error in them. After the perturbator is turned off, CHAOSORCA monitors Nginx for another 5 minutes. Thanks to the monitor component, the 5 metrics collected by CHAOSORCA namely network IO, HTTP latency, CPU and memory usage, the amount of invoked system calls are observable by developers. The visualization also makes it easier for developers to analyze the three experiment phases defined in Section III-D: the trend of metric changes before, during and after perturbations.

On the other hand, if the observability of target application is not sufficient, it brings more obstacles to distinguish abnormal behaviour. For example, during an OPEN system call perturbation in Figure 4 the rate of HTTP requests stays the same. However, most of requests are responded with a 403 “Forbidden” code according to sub-figure (f). If CHAOSORCA does not bring observability to web application’s HTTP response code, developers may think that everything is fine since Nginx still handles a proper amount of requests. But actually most of the responses are incorrect. Such a perturbation does have critical influence on end-users.

Answer to RQ3

For a containerized applications, there are different locations in the stack to improve observability. CHAOSORCA shows the feasibility of multi-layer monitoring at the system call level, the container level and the application level. Put all together, it becomes possible to compare the application behaviour under perturbation against the normal behaviour in nominal mode.

F. RQ4: What is the overhead of conducting chaos experiments using CHAOSORCA?

1) The Overhead of the Monitoring Component: Figure 5 shows the overhead of monitor component when Nginx is the
target application. The first sub-figure records the amount of HTTP requests per second during the experiment. The second sub-figure records the rate of system calls. The last two sub-figures are about the CPU and memory usage of the monitor components. Regarding \texttt{dd} command, the CPU and memory usage metrics are the same as Nginx’s ones.

For network monitoring, with the increase of HTTP request rate, sub-figure (a) in Figure 5 shows that the HTTP request monitoring even fails to monitor every request when the workload becomes too high. Sub-figure (c) describes that the CPU usage of network monitoring increases to almost two times of the application’s itself. The memory usage cost by network monitor ranges from 50MB to 200MB which is shown in sub-figure (d).

For system call monitoring, the overhead is quite low and stable. Even under a system call heavy tool like \texttt{dd}, the CPU usage of the system call monitor module (chaosorca.sysm) is almost zero (sub-figure c). The memory usage stays at 125MB and never increases (sub-figure d).

Overall for monitor component, the overhead of network monitoring becomes significant when the rate of user requests are higher than \(50/s\) in the test environment. The overhead caused by system call monitoring always stays low. As the memory usage, it is stable and there is no memory leak during the experiment. The overhead of \texttt{CHAOSORCA} can be considered as acceptable for a normal amount of workload.
Fig. 7: The Rate of System Calls Invoked By *dd* During a Perturbation. Each Blue Line Is When A Perturbation Occurs.

2) The Overhead of Perturbator Component: When the perturbator injects system call failures, the key metrics are described in Figure 6 and Figure 7. For Nginx, it shows that the HTTP requests under a normal workload remains the same. For the system call heavy application *dd*, the perturbations have an significant impact on the number of system calls. Overall for the perturbator component, the usage of system resources remains small during experiments.

### Answer to RQ4

The overhead caused by CHAOSORCA depends on system call usage and also on the workload. For the monitor component, the network monitor overhead increases with the workload; and the system call monitor overhead stays low. As for perturbator component, our experiment shows that the overhead is acceptable. Overall, our experiments suggest that the runtime overhead of CHAOSORCA is acceptable for analyzing resilience.

### V. RELATED WORK

#### A. Fault Injection

Fault injection was originally studied at the hardware level, for example using radiations to force bit flips [28] or by interacting with pins to generate processor errors [35]. Later on, software-based approach were developed to emulate hardware errors [31] e.g. by injecting memory or network errors [29] or by perturbing the OS kernel [32] [33]. A number of tools now facilitates the injection of such kinds of errors, such as Gremlin [8], ChaosCat [5] or cpu-troll [16]. The network layer is another fault injection vector, which has been studied in depth. Quite a few open-source tools [30] [20] [18] [10] [17] and closed-source tools [8] [5] combine Iptables with the Traffic Control network emulation tool to inject different kinds of network failures, including latency and dropping a percentage of traffic.

More related to our work, a number of approaches propose to inject faults at the system call level, for example by introducing bit flip during the execution of system calls [21], by corrupting those system calls [38], or by overriding parts of the glibc library [34], which wraps system calls into a C API. Besides, approaches have been developed to better control and protect the access to those system calls [41]. Containerization, cgroups and namespaces, which we presented in Section II provide process-level isolation and allows to finely control which resources a given container can access.

Chaos Engineering is an emerging discipline, building on those fault injection tools, but applying them directly on the production systems. Netflix for example created a tool called Chaos Monkey [30] to randomly disable production servers and to observe how their platform would handle such failures. Chaos Monkey has inspired a number of similar approaches for different production environments, such as Kubernetes [11] [14] [4], Docker [20] [15], AWS [3] or private clouds [11] or even the JVM [42] by forcing exceptions.

CHAOSORCA is a novel Chaos Engineering approach, which proposes to perturb systems calls in a controlled manner. The blast radius of experiments conducted with CHAOSORCA is controlled by cgroups and namespaces, which makes it possible to use it in production: only targeted containers will be affected.

#### B. Observability

Monitoring is a well researched area. For example, PyMon is a lightweight monitoring solution to collect monitoring metrics from the open-source tool monit [27]; Dargos is a decentralized architecture for monitoring cloud environments with a low overhead [37].

Moving to container-specific approaches, cAdvisor [7] by Google enables the collection and export of container metrics. It provides metrics about CPU, memory, disk and network usage, which can be for example used to decide when to scale in and out. Node_exporter [12] is a tool able to export many different metrics about hardware and operating system, examples include, CPU, disk, network and memory statistics.

In Chaos Engineering, metrics are necessary to evaluate the steady state of the system (before perturbation) and the current state of the system during an experiment. Netflix and Google has created a tool called Kayenta [13], which does automated canary analysis by comparing the key metrics of the new deployment version to the old one. If the canary degrades metrics to much it can automatically abort the canary [2].

CHAOSORCA does not advocate for a specific monitoring approach provided that it can push its data into a Prometheus timeseries database. If not an adapter needs to be implemented. Out of the box, CHAOSORCA integrates with cAdvisor to resources (CPU, RAM, etc), Bpftrace to monitor the invocation of system calls and PyShark to monitor HTTP requests.

### VI. CONCLUSION

In this paper, we have presented CHAOSORCA, a novel Chaos Engineering approach to actively inject system call failures into a containerized application in order to evaluate its
resilience in face of such perturbations. Under the hood, CHAOSORCA leverages Linux cgroups and namespaces to precisely control the blast radius of chaos experiments. CHAOSORCA also integrates with different and complementary monitoring solutions to increase the observability of the system. CHAOSORCA introduces perturbations by introducing errors or delays in the execution of the system calls, as specified in a resilience experiment.

Our evaluation on three different real-world applications shows that system call level perturbation is promising to detect weaknesses in self-protection and resilience mechanisms embedded in those applications. Our evaluation also shows that CHAOSORCA improves the observability of a containerized application without modifying any source code or configuration in the target container. In the future, we plan to make CHAOSORCA automatic in the context of a container orchestration framework such as Docker Swarm or Kubernetes.

REFERENCES

[1] asobti/kube-monkey: An implementation of Netflix’s Chaos Monkey for Kubernetes clusters. https://github.com/asobti/kube-monkey (Accessed Online: 2019-02-28).
[2] Automated Canary Analysis at Netflix with Kayenta - Netflix TechBlog - Medium. https://medium.com/netflix-techblog/automated-canary-analysis-at-netflix-with-kayenta-3260be77c07c (Accessed Online: 2019-04-01).
[3] bbc/chaos-lambda: Randomly terminate ASG instances during business hours. https://github.com/bbc/chaos-lambda (Accessed Online: 2019-02-28).
[4] bloomberg/powerfulseal: A powerful testing tool for Kubernetes clusters. https://github.com/bloomberg/powerfulseal (Accessed Online: 2019-02-28).
[5] ChaosCat: Automating Fault Injection at PagerDuty. https://www.pagerduty.com/blog/chaoscat-automating-fault-injection/ (Accessed Online: 2019-02-28).
[6] Docker overview | Docker Documentation. https://docs.docker.com/engine/docker-overview/ (Accessed Online: 2019-02-25).
[7] google/cadvisor: Analyzes resource usage and performance characteristics of running containers. https://github.com/google/cadvisor (Accessed Online: 2019-03-04).
[8] Gremlin. https://gremlin.com/ (Accessed Online: 2019-01-24).
[9] Linux Man Pages - System Calls. http://man7.org/linux/man-pages/dir_section_2.html (Accessed Online: 2019-07-01).
[10] mefellows/muxy: Chaos engineering tool for simulating real-world distributed system failures. https://github.com/meffellows/muxy/ (Accessed Online: 2019-02-28).
[11] outbrain/GomJabbar: Chaos Monkey for your private cloud. https://github.com/outbrain/GomJabbar (Accessed Online: 2019-02-28).
[12] prometheus/node_exporter: Exporter for machine metrics. https://github.com/prometheus/node_exporter (Accessed Online: 2019-03-20).
[13] spinnaker/kayenta: Automated Canary Service. https://github.com/spinnaker/kayenta (Accessed Online: 2019-04-01).
[14] target/pod-reaper: Rule based pod killing kubernetes controller. https://github.com/target/pod-reaper (Accessed Online: 2019-02-28).
[15] titpetric/docker-chaos-monkey: A Chaos Monkey system for Docker Swarm. https://github.com/titpetric/docker-chaos-monkey (Accessed Online: 2019-02-28).
[16] TrollScripts/cpu-troll: Part of the larger OpenChaos Suite, dedicated to raising CPU latency by the requested percentage and timespan. https://github.com/TrollScripts/cpu-troll (Accessed Online: 2019-03-04).
[17] tylertreat/comcast: Simulating shitty network connections so you can build better systems. https://github.com/tylertreat/comcast (Accessed Online: 2019-02-28).
[18] worstcase/blockade: Docker-based utility for testing network failures and partitions in distributed applications. https://github.com/worstcase/blockade (Accessed Online: 2019-02-28).
[19] Docker: Automated and Consistent Software Deployments. https://www.infoq.com/news/2013/03/Docker (Accessed Online: 2019-04-02).
[20] Laci Ledenev, 2017. Chaos Testing for Docker Containers. URL: https://codefresh.io/docker-tutorial/chaos-testing_docker/.
[21] Amarnath, R., Bhat, S.N., Munk, P., Thaden, E., 2018. A Fault Injection Approach to Evaluate Soft-Error Dependability of System Calls, in: 2018 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), IEEE. pp. 71–76.
[22] Anderson, C., 2015. Docker. IEEE Software 32, 102–c3. doi:10.1109/MS.2015.62.
[23] Basiri, A., Behnam, N., De Rooij, R., Hochstein, L., Kosowski, L., Reynolds, J., Rosenthal, C., 2016. Chaos Engineering. IEEE Software 33, 35–41. doi:10.1109/MS.2016.60.
[24] Casalicchio, E., Perciballi, V., 2017a. Auto-Scaling of Containers: The Impact of Relative and Absolute Metrics, in: 2017 IEEE 2nd International Workshops on Foundations and Applications of Self Systems (FASW), IEEE. pp. 207–214.
[25] Casalicchio, E., Perciballi, V., 2017b. Measuring Docker Performance: What a Mess, in: Proceedings of the 8th ACM/SPEC on international conference on performance engineering companion, ACM. pp. 11–16.
[26] Gregg, B., 2016. The flame graph. Commun. ACM 59, 48–57. URL: http://doi.acm.org/10.1145/2909476 doi:10.1145/2909476.
[27] Grobmann, M., Klug, C., 2017. Monitoring Container...
Services at the Network Edge, in: 2017 29th International Teletraffic Congress (ITC 29), ITC Press. pp. 130–133.

[28] Gunneflo, U., Karlsson, J., Torin, J., 1989. Evaluation of error detection schemes using fault injection by heavy-ion radiation, in: 1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers, IEEE. pp. 340–347.

[29] Han, S., Rosenberg, H.A., Shin, K.G., 1993. DOCTOR: An integrated software fault injection environment. Citeseer.

[30] Izrailevsky, Y., Ariel, T. The Netflix Simian Army. https://medium.com/netflix-techblog/the-netflix-simian-army-16e57fbeb116 (Accessed Online: 2019-02-06).

[31] Kanawati, G.A., Kanawati, N.A., Abraham, J.A., 1992. FERRARI: A tool for the validation of system dependability properties, in: 1992 FTCS-The Twenty-Second International Symposium on Fault-Tolerant Computing, IEEE. pp. 336–344.

[32] Kao, W.I., Iyer, R.K., Tang, D., 1993. FINE: A fault injection and monitoring environment for tracing the UNIX system behavior under faults. IEEE Transactions on Software Engineering 19, 1105–1118.

[33] Lee, H., Song, Y., Shin, H., 2000. SFIDA: a software implemented fault injection tool for distributed dependable applications, in: Proceedings Fourth International Conference/Exhibition on High Performance Computing in the Asia-Pacific Region, IEEE. pp. 410–415 vol.1.

[34] Lee, H.C., Kim, C., Yi, J., 2011. Experimenting with system and Libc call interception attacks on ARM-based Linux kernel, in: Proceedings of the 2011 ACM Symposium on applied computing. ACM. pp. 631–632.

[35] Madeira, H., Rela, M., Moreira, F., Silva, J.G., 1994. RIFLE: A general purpose pin-level fault injector, in: European Dependable Computing Conference, Springer. pp. 197–216.

[36] Miles Russ, 2019. Chaos Engineering Observability. First edit ed., O'Reilly Media, Inc.

[37] Povedano-Molina, J., Lopez-Vega, J.M., Lopez-Soler, J.M., Corradi, A., Foschini, L., 2013. DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds. Future Generation Computer Systems 29, 2041–2056. URL: http://www.sciencedirect.com/science/article/pii/S0167739X13000824, doi https://doi.org/10.1016/j.future.2013.04.022

[38] Rodríguez, M., Salles, F., Fabre, J.C., Arlat, J., 1999. MAFALDA: Microkernel Assessment by Fault Injection and Design Aid, in: Hlavička, J., Maehle, E., Pataricza, A. (Eds.), Dependable Computing - EDCC-3, Springer Berlin Heidelberg, Berlin, Heidelberg. pp. 143–160.

[39] Sridharan, C., 2018. Distributed Systems Observability. First edit ed., O'Reilly Media, Inc.

[40] System call. syscalls(2) - Linux manual page. http://man7.org/linux/man-pages/man2/syscalls.2.html (Accessed Online: 2019-04-12).

[41] Win, T.Y., Tso, F.P., Mair, Q., Tianfield, H., 2017. PROTECT: Container Process Isolation Using System Call Interception, in: 2017 14th International Symposium on Pervasive Systems, Algorithms and Networks & 2017 11th International Conference on Frontier of Computer Science and Technology & 2017 Third International Symposium of Creative Computing (ISPAN-FCST-ISCC), IEEE. pp. 191–196.

[42] Zhang, L., Morin, B., Haller, P., Baudry, B., Monperrus, M., 2018. A Chaos Engineering System for Live Analysis and Falsification of Exception-handling in the JVM.