FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automatic Identification of Cross-Container Side-Channels

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Baumgartner, BSc
Matrikelnummer 01607460

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar R. Weippl
Mitwirkung: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Wien, 11. November 2019

Jakob Baumgartner   Edgar R. Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Automatic Identification of Cross-Container Side-Channels

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jakob Baumgartner, BSc
Registration Number 01607460

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr.techn. Edgar R. Weippl
Assistance: Univ.Lektor Dipl.-Ing. Dr.techn. Georg Merzdovnik, BSc

Vienna, 11th November, 2019

Jakob Baumgartner   Edgar R. Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.ac.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Jakob Baumgartner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 11. November 2019

Jakob Baumgartner

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Danksagung

Ich möchte mich bei Franziska Bollwein bedanken, dass sie mich immer motiviert hat, auch wenn ich dachte, ich hätte bessere Dinge zu tun. Durch ihre Ausdauer konnte ich laufend Fortschritte erzielen in den letzten Monaten. Ich möchte auch meinen Eltern danken, dass sie mir dieses Studium ermöglichten. Des Weiteren danke ich Benjamin Böck für das Korrekturlesen dieser Diplomarbeit.

Ein besonderer Dank geht an Georg Merzdovnik für die initiale Idee dieser Arbeit und für seine professionellen Ratschläge. Er nahm sich immer Zeit für mich, um den aktuellen Fortschritt und meine Fragen zu besprechen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I want to thank Franziska Bollwein for keeping me motivated when I thought I had better things to do. Thanks to her endurance I was able to continuously make progress in the last months. I also want to say thanks to my parents who made it possible for me to study. Furthermore, I want to thank Benjamin Böck for proofreading this thesis.

Special thanks go to Georg Merzdovnik for the initial idea of this work and for his professional advice. He always had time for me to discussion the current progress and my questions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

In den letzten Jahren stieg die Beliebtheit von Docker und anderen Container-Lösungen in vielen Unternehmen an. Da diese die Container-Software für immer mehr Applikationen verwenden, sind sie auch abhängig von der Sicherheit der eingesetzten Container-Lösung. Ziel jedes Containers ist die Bereitstellung eines eigenen isolierten Sub-Systems, indem eine Applikation, gebündelt mit ihren Abhängigkeiten, ausgeführt werden kann. Im Falle von Docker teilen sich alle Container den Betriebssystem-Kernel des Hosts. Es existieren dokumentierte Möglichkeiten, um die Kommunikation zwischen zwei Containern zu erlauben, wie zum Beispiel Netzwerkverbindungen und Shared Volumes. Allerdings könnte ein potentieller Side-Channel durch den Kernel existieren, wobei es in diesem Fall möglich sein könnte, dass ein Container die Existenz eines anderen Containers ermittelt, vertrauliche Daten ausliest oder das Verhalten eines anderen beeinflusst.

In dieser Arbeit wird ein neuartiges System vorgestellt, welches Interaktionen zwischen zwei Containern erkennen kann. Mithilfe eines System-Call-Fuzzers wird ein vollautomatisiertes System entwickelt, das zur Auffindung von Side-Channels in aktuellen Versionen von Linux und Docker verwendet wird. Dabei werden die Dynamic-Taint-Analysis-Funktionen von PANDA verwendet, einer quelloffenen Reverse-Engineering-Plattform. Das System wechselt dabei zwischen einer Aufzeichnung des Gastsystems mit zwei System-Call-Fuzzern in zwei verschiedenen Containern und der Analyse der Aufzeichnung ab. Sollten Findings während der Analyse auftreten, so wird die Aufzeichnung und die dazugehörige Log-Datei für eine spätere manuelle Analyse abgespeichert. Des Weiteren werden periodisch Statusinformationen mitgeloggt. Diese Logeinträge enthalten den aktuellen RAM-Bedarf des Systems, die Größe der Log-Datei und den Fortschritt der Analyse der Aufzeichnung.

Obwohl nach Wochen des Betriebs keine Schwachstellen gefunden wurden, wurde gezeigt, dass das System mit idealisierten Tests funktioniert. In diesen Tests werden die Container mit erweiterten (nicht default) Berechtigungen gestartet. Alle Komponenten wurden so konzipiert, dass sie unabhängig von der eingesetzten Container-Lösung und dem System-Call-Fuzzer sind. Mithilfe eines anderen Setups kann das System unverändert für den Einsatz mit anderen Container-Lösungen verwendet werden. Des Weiteren kann die entwickelte Software auch als Basis für andere Information-Leak-Analysen dienen, wie zum Beispiel für Forschungen zu Linux-Applikations-Sandboxen und LSM-Profilen.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

In recent years, Docker and other container engines became very popular in many companies. These companies are starting to rely on the security of the containerization software as they are used for more and more applications. Each container should run as its own isolated subsystem, where each application can be bundled with its dependencies. In case of Docker, all containers on a single host share the same operating system kernel. There are documented ways to allow communication between two containers, for example, network connections and shared volumes. However, there is a potential for side-channels over the kernel, where one container could discover the existence of another, read confidential data, or influence its behavior.

In this thesis, we develop a novel system that detects interactions between two containers. With the help of a system call fuzzer, we build a fully automated system to look for these kinds of side-channels in current versions of Linux and Docker. It uses the dynamic taint analysis capabilities of PANDA, an open source platform for reverse engineering. Our system alternately creates whole system recordings of two system call fuzzers running in two different containers and then analyzes them. If any findings occur, the current recording and the affiliated log file are stored for later manual analysis. Furthermore, status information of the running system is periodically logged. This includes the current RAM usage, log file size and analysis progress of a recording.

Although no vulnerabilities were found after weeks of running the system, we proved the functionality with idealized tests where the containers are started with additional (non-default) permissions. We designed the whole system independently from the container engine and system call fuzzer. Therefore, with a different setup it can be used to find communication between containers in other engine implementations. In addition, it can build the basis for finding information leaks in other types of software, like in research of Linux application sandboxes and LSM profiles.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung **xi**

Abstract **xiii**

Contents **xv**

1 Introduction **1**
1.1 Motivation 2
1.2 Problem Statement 2
1.3 Aim of the Work 3
1.4 Threat Model 3
1.5 Structure of the Work 4

2 State of the Art **5**
2.1 Cross-Container and Cross-VM Side-Channel Attacks 5
2.2 Taint Analysis 8
2.3 System Calls Fuzzers 10

3 Background **13**
3.1 Container Virtualization 13
3.2 Technologies of Linux Containers 17
3.3 Taint Analysis with PANDA 24
3.4 Linux System Call Calling Conventions 28

4 Methodology **29**
4.1 Tainting System Call Input Data 29
4.2 Detection of Cross-Container Tainted Data 30
4.3 Taint Labels 30
4.4 Automation Strategies 30

5 Implementation Details **33**
5.1 System Overview 33
5.2 PANDA Version 34
5.3 Retrieving the Currently Running Fuzzing Process 34

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 Tainting System Call Registers and Kernel Reads 35
5.5 Checking System Call Returns and Kernel Writes 36
5.6 Dereferenced Pointer Patch 37
5.7 Uniquification of Findings and System Call Skipping 38
5.8 Optional Debugging Outputs and Debugging Aid 38
5.9 Reported Finding Information 39
5.10 Automation of Runs 39

6 Evaluation and Results 43
6.1 Test Setup 43
6.2 Idealized Test Runs 44
6.3 Early Fuzzing Attempts and False Positives 49
6.4 Results 52

7 Discussion and Future Work 55

8 Conclusion 59

List of Figures 61

List of Algorithms 63

Listings 65

Abbreviations 67

Bibliography 69

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

Container technology plays an important role in modern *DevOps* collaborations. They bridge the gaps between developers and operators by bundling the software with the dependencies required for operation [1]. There exist several projects that provide software for deploying software in containers. With increasing investments in containerization software [2, 3] businesses start to rely on the security of those software solutions. Part of container security is the ability to isolate processes from another. There are several supported ways to allow communication between containers. In *Docker*[1], for example, communication channels can be established over a network connection[2] or shared volumes[3]. However, there is a potential of side-channel attacks where communication is possible although not allowed explicitly. With these attacks, it would be possible to discover containers on the same host, gather identification from neighboring containers or influence other containers.

One kind of side-channel could exist through the host operating system kernel because the host and the containers share the same kernel [4]. Currently, tools for automated identification of such side-channels through the kernel space do not exist. In this thesis, we present a novel tool that is capable of detecting side-channel between isolated containers. We show that the software we developed works as intended by creating tests with known channels. Furthermore, we automate the tool to find vulnerabilities in a current implementation of the Linux kernel and the Docker engine. Finally, we discuss the results of this fuzzing run and possible future work.

[1] `https://www.docker.com`
[2] `https://docs.docker.com/config/containers/container-networking/`
[3] `https://docs.docker.com/storage/volumes/`

1.1 Motivation

To our current knowledge, there has not been done any research that aims to perform automated tests that analyze the isolation of containers. Research with similar goals exist, for example, the cache mechanisms of modern CPUs can be used for an information leakage attack [5, 6] and other hardware bugs were discovered [7, 8, 9, 10]. The shared goal is the exfiltration or manipulation of data that is not accessible under normal conditions. However, the main difference is that the approach of this thesis is purely based on software, which is shared between isolated processes. Furthermore, our approach can be deployed as an automated process to potentially look for new vulnerabilities in newly developed software or in upcoming software releases. We propose that it is in general not possible to build an automated system that can find unknown hardware bugs.

In the field of malware analysis, taint tracking approaches exist. However, they do not deal with the isolation property of containers. Their focus is on user programs, which would not be adequate for this thesis because it aims to find side-channels over the kernel, which is not possible if the taint analysis does not include the operating system kernel.

This leads us to the development of a software that is capable to identify side-channels between containers. Once identified they can be manually examined and software vendors can develop a fix to mitigate the issue. This would improve the security for all users of the container software. Moreover, the security of other container engines could be improved by the same patches because they may rely on the same kernel technologies.

1.2 Problem Statement

As already mentioned, containers bundle programs with their dependencies. To achieve this, the containers must isolate certain resources from another, like the mounted file systems, processes IDs, hostname and users. In Linux, this isolation is done using kernel namespaces. In addition to namespaces the security is improved with technologies like capabilities, seccomp and LSM. These technologies are discussed in greater detail in Chapter 3. Every software solution for containerization can come with its own configuration of these technologies. Therefore, there exists the need to create an automated software that is capable of testing whole systems. As far as we are aware of, currently there is no software that can test a complete system for information leakage between two (or more) containers once they are running.

We define the following research questions: **Do side-channels between isolated containers over the kernel currently exist and how could a system look like that can automatically identify them?**

To answers these questions, we are going to develop an automated approach that can identify when one containerized application could influence the behavior of another container.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.3 Aim of the Work

The overall aim of this work is to improve the security of containers, especially the isolation property. This will be achieved by providing software that can automatically identify cross-container side-channels. The developed software should run independently of the used container engine and Linux distribution. Therefore, it will be possible to run the software for future versions of the Linux kernel and container software with minimal changes to the source code or the configuration. The syscall fuzzer, which is used to call as many different system calls as possible, can be replaced too. Any other software project that creates syscalls can be used.

The developed software should demonstrate that it is possible to identify the described side-channels in an automated fashion. This means that the software should be able to work with a setup provided by an operator and, if the described side-channels exist in the system under test, identify and report them as they are discovered. Therefore, the software must run stable if a valid configuration is provided. In this case, *configuration* does not only mean the software configuration but also the attributes of the host system, for example, enough memory must be provided.

The information outputted by the developed software will be sufficient to reconstruct the activities that caused this side-channel to occur. With this information, it should be feasible to find a possible solution to close this information leak.

1.4 Threat Model

There are different scenarios in which an adversary could benefit from the knowledge of the described side-channels. One scenario would be the discovery of neighboring containers, which would work with no direct write capabilities from the attacker's container. The knowledge about containers sharing the same physical machine can be used for further attacks, like *Denial of Service* attacks, where it would be important to find the physical target machine. In this scenario, an adversary could improve the effectiveness of such an attack if, for example, the victim's machines are deployed in a public IaaS cloud. The adversary could start virtual machines in the same public cloud and then check if they are located on the same physical host. If enough physical machines are discovered, such that the adversary thinks an attack would be possible, then a local (on the same physical machine) DoS attack could be started.

Another scenario exists when a potential attacker can influence the victim's container. In this case, attacks that are more intrusive may be possible. In a worst-case scenario, an attacker would have full control over the other container and thus gain the permissions of the victim container. The confidentiality, integrity and availability of the container's application would be affected significantly. Any other case where partial access to the victim's containers are possible are also conceivable.

The objective of this thesis does not include hardware bugs, for example, *Meltdown* [7] and *Spectre* [8]. This further excludes cache information leakage problems, like the

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

research of Zhang et al. [6] and *HomeAlone* [5], although many goals and the threat model are partially the same. Both try to retrieve or alter information normally not accessible due to security constraints. The focus of this works lies solely on the software isolation implemented in the Linux kernel and the profiles provided by the container engines.

1.5 Structure of the Work

The following chapters are structured as follows. In Chapter 2, State of the Art, we will go through similar research, which may share the same goals. Furthermore, we examine underlying research, which is necessary to complete this thesis, like syscall fuzzers and dynamic taint analysis. In the Background chapter, Chapter 3, we discuss information that is required to understand the technical details of container engines and the software developed in this thesis. Often these topics are not the result of research but from convention or chosen arbitrarily. Chapter 4 consists of a more theoretical view to explain how the aim of this works is resolved. It intentionally leaves out implementation details to focus on a proper approach to give answers to the research question. This is followed by the Implementation Details, Chapter 5, which builds upon the methodology and goes into details about the implementation. The results of multiple (test) runs are discussed in Chapter 6 and this knowledge is used to reject further false positive findings. Furthermore, we present the results of fuzzing runs in this chapter. In Chapter 7, Discussion and Future Work, we discuss the results found and give ideas about possible future work. The final chapter, Chapter 8, sums up the topic and the results.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

State of the Art

To the best of our knowledge, there has not been any academic research or software solution providing a mechanism for automated identification of cross-container side-channels through the operating system kernel. However, there are other types of side-channel attacks in the context of containers or virtual machines, which follow similar goals. We discuss them in Section 2.1. Furthermore, prior research in other areas resulted in software that can be used to solve the problem underlying this thesis.

2.1 Cross-Container and Cross-VM Side-Channel Attacks

Side-channels in the form of information leaks exist in previous versions of Docker. Hertz [11] discovered a leak in the *proc* filesystem. The file */proc/sched_debug* listed information about all processes and was therefore not PID namespaced. Thus, it is possible to exchange information between two containers by creating processes with specific names. A process in the other container could then read the file to receive the information stored in the process name. This issue was fixed in Docker in 2016[1].

The Docker engine before version 1.6.1 had issues with weak permissions for the dictionaries */proc/asound*, */proc/timer_stats*, */proc/latency_stats*, and */proc/fs*. The vulnerability CVE-2015-3630[2] allowed an attacker to gain sensitive information and to alter the host.

In older Docker versions, a breakout vulnerability, called *shocker*[3], was discovered. However, the issue is not unique to Docker and it should exist in every misconfigured container. The proof of concept code demonstrates the vulnerability by reading the host's file */etc/shadow*, which should not be accessible within the container. The main issue is the capability *CAP_DAC_READ_SEARCH* [12], which allows the usage of the syscall

[1] https://github.com/moby/moby/pull/21263
[2] https://www.cvedetails.com/cve/CVE-2015-3630/
[3] http://stealth.openwall.net/xSports/shocker.c

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

open_by_handle_at [13]. With this syscall it is possible to circumvent file permission checks and to read files that are opened by other processes. A detailed analysis of the exploit can be found in [14].

Other types of Docker vulnerabilities exist, where the exploit is possible due to a specially crafted container image or when the victim executes certain Docker commands[4]. As an example for the first case, there is CVE-2019-5736[5]. With this vulnerability an attacker could override the *runC*[6] executable if she previously had write access to an existing container (and the container can be attached with `docker exec`[7]) or the image is under the attacker's control. The vulnerability affects all container engines that use runC, thus, it is not a unique issue of Docker. Furthermore, the vulnerability CVE-2018-15664[8] is an example for an issue caused by an external command. The Docker command `cp`[9] was vulnerable to an attack, which allows attackers to read and write to files of the host system with root privileges. However, these issues are not in the scope of this thesis because they require external control of the container. The aim of this thesis, as already stated in Chapter 1, is to perform automated analysis in already running containers and focuses on the abilities an attacker has once she can execute code inside a container.

The research carried out by Zhang et al. [6] performed a cross-container side-channel attack on a cloud PaaS. They developed an attack based on CPU cache to successfully find neighboring tenants, gather sensitive user data and to break encryption. Their approach is based on an attack called *FLUSH+RELOAD* [15], which targets a last level cache. The basic technique is to flush out *chunks* (aligned memory regions of cacheline size) which page is shared with the victim. Then an attacker waits for a certain time while the victim may use a memory region with another CPU core. After this, the adversary performs the reload part by reusing the same chunk. If the reload was fast, then the victim accessed the suspected memory region, if not, the reload was slower and no access was performed.

Zhang et al. developed *HomeAlone* [5], which also performs cache based analysis to find physical neighboring virtual machines. It aims to create software that can be started on an unmodified hypervisor to verify that no other virtual machines are running on the same host. Without such an approach a tenant of a physically isolated machine has no way to make sure that service is indeed isolated as promised by the service provider. The idea is to minimize the usage of some cache regions. If unexpected activity would be measured, then there could be another unknown application on the same host. However, in practice there will always be some noise from scheduling operations and from the underlying hypervisor. Their approach is based on the cache attack *PRIME+PROBE* [16, 17], which starts by priming cache sets with a specified data. Enough pages must be allocated in

[4] `https://docs.docker.com/engine/reference/commandline/docker/`
[5] `https://www.cvedetails.com/cve/CVE-2019-5736/`
[6] `https://github.com/opencontainers/runc`
[7] `https://docs.docker.com/engine/reference/commandline/exec/`
[8] `https://www.cvedetails.com/cve/CVE-2018-15664/`
[9] `https://docs.docker.com/engine/reference/commandline/cp/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

order to prime any cache set. After a wait time the attacker checks the time required to access memory of a mapping to a certain cache set. If the access was slow, then a cache miss occurred and the victim accessed the memory mapping.

Branch Prediction Analysis [18, 19] enabled Acıiçmez et al. to extract sensitive information from parallel running processes by running a *carefully written* spy process. These attacks allow an unprivileged process to extract (partial) keys of an RSA implementation although other hardware security features are in place like memory protection, sandboxing or virtualization. They show that nearly all bits of a key can be extracted of a single RSA signing execution. This is different from classical timing attacks, which require many measurements to amplify small time differences depending on the key used.

Another class of attacks exist with an attack known as *Rowhammer* [20, 21]. It aims to influence neighboring memory cells that are not accessible under normal conditions. Gruss et al. [22] showed that this type of attack also works in scripting environments, particularly in websites with JavaScript. The reason for this problem is the high density of the current DRAM technology. The smaller the building blocks of the memory becomes the harder it is to make sure there is no electrical interaction between them. Rowhammer attacks share goals of this thesis, mainly the ability to influence other processes that should not be possible when they are isolated from another.

RAMBleed [23] extended the abilities of Rowhammer by creating a read side-channel. This shows that these kind of memory attacks is not only a threat to the integrity of the victim's system but also to the confidentiality. RAMBleed also works on system with error correcting memory, thus exposing a security threat even with server hardware, and without the need of huge pages. With their novel attack Kwong et al. managed to extract a RSA-2048 key from a SSH daemon running as the root user.

Lipp et al. [7] developed *Meltdown*, which uses side effects of the out-of-order execution present in modern CPUs. The attack allows adversaries to read arbitrary kernel space memory. Therefore, Meltdown impairs the security provided by address space isolation and paravirtualized environments. CPUs with out-of-order execution allow unprivileged processes to execute loads of privileged memory regions. Furthermore, some computations, like array accesses, can be performed based on the value of these temporary registers. If the instruction should not have been executed, the result from the memory lookup is discarded, which basically prevents security issues. However, a side-channel exists based on the behavior of the CPU cache. This enables an attacker to dump the full kernel memory with 3.2KB/s to 503KB/s depending on the hardware used.

Spectre [8] attacks are based on the same idea as Meltdown, where the speculative execution of the CPU performs possibly incorrect operations and reverts to a prior state in case the speculative path is not taken. The attacks use the branch prediction of a wide range of modern processes. Adversaries are capable to read memory and register content from other processes. Spectre consists in multiple variants, which differ in the way the speculative execution is achieved and the method for leaking the information. Kocher et al. propose that a long-term solution for these attacks needs fundamental changes of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

processor's instruction set architecture.

Van Bulck et al. [24] managed to extract keys from secure enclaves inside a CPU. *Foreshadow* is a Meltdown variant that targets Intel's SGX[10] technology. Intel's investigations on Foreshadow revealed the root cause and much broader issues, collectively named *Foreshadow-NG* [9]. Foreshadow-NG can be used by unprivileged applications to access kernel memory or by malicious virtual machines to read hypervisor data. The attacks allow adversaries to dump L1 data cache and to gather information from currently non-mapped physical memory. Therefore, previous mitigations for the initial Meltdown attack are necessary but not sufficient.

ZombieLoad [10] is another Meltdown-type attack and makes use of the fill-buffer logic. Data from unauthorized destinations can be stored in the fill buffer by faulting load instructions. Schwarz et al. discovered that this data leakage can occur across logical cores. They show the effectiveness of ZombieLoad by "*attack scenarios across CPU privilege rings, OS processes, virtual machines, and SGX enclaves*" [10]. Furthermore, they argue that the only workaround to fix ZombieLoad is to disable hyperthreading.

2.2 Taint Analysis

To find side-channels between containers a whole system taint analysis must be performed. This means that the taint engine includes analysis about the operating system kernel. There are some projects that offer a good starting point at first glance, but were in fact too limited. The tool *Bochspwn Reloaded* [25], for example, is based on the emulator *Bochs*[11] and it uses Bochs interfacing mechanisms for tainting. Moreover, the hypervisor system *Xenpwn* [26] is inspired by Bochspwn Reloaded and works by manipulating the *Extended Page Tables* [27] to handle exceptions when memory pages of interest are accessed. Both, Bochspwn Reloaded and Xenpwn, were created to find *Double Fetches*, a special form of *Time-of-Check-to-Time-of-Use* vulnerability [26].

Even though Bochspwn Reloaded describes itself as *taint tracking* (and Xenpwn has the same objective), we discovered that their approach is too limited for this thesis. They basically look for large memory operations and do not focus on taint analysis on a register level. Both systems work for whole system analysis but they implemented a simplified mechanism and for this thesis a more generalized one, which includes registers, is needed.

Taint analysis or other data-flow analysis that offer whole system analysis is also used in malware analysis. *K-Tracer* is "*an automated rootkit behavior analyzer for the Windows OS*" [28]. It is based on the emulator *QEMU*[12] and creates dynamic traces of execution paths for all kernel functionalities. The system recognizes events, which manipulate data

[10] https://www.intel.de/content/www/de/de/architecture-and-technology/software-guard-extensions.html
[11] http://bochs.sourceforge.net
[12] https://www.qemu.org

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

and access to sensitive information. It was tested against several rootkits, which showed that the system can gather information about malicious capabilities.

TaintQemu is also based on QEMU and analysis malware behavior with "*whole-system fine-grained taint analysis*" [29]. Virtual device inputs are tainted and their propagation is recorded as taint graphs. The *PolicyDB* stores policies, which a taint graph should satisfy. Violations are considered malicious behavior. Yin et al. also used QEMU to emulate the guest system. They use it instead of Bochs because Bochs' emulation is slower compared to QEMU. There is also *Panorama*, a whole system emulator, which performs "*fine-grained taint tracking*" [30]. Panorama involved the same researchers as TaintQemu and both projects share many properties. Panorama's approach marks sensitive data and tracks their propagation across the whole system. The taint propagation is performed at the hardware level and it is aware of operating system objects, for example, the current process. The result of the taint propagation and the knowledge about the current state of the operating system are *taint graphs*. Taint graphs include information about which processes accessed the tainted data, the flow of the data through the system and the destination of the data, for example, if it is written to a file or sent over the network. Furthermore, Yin et al. defined policies based on taint graphs, which specify certain types of behavior. The policies can be compared to taint graphs of unknown software to automatically detect behavior of certain categories. In the evaluation step, the system did not cause false negatives and it only reported a few false positives.

Chow et al. [31] used tainting at the hardware level to determine the lifetime of sensitive data, like passwords. They argue that by minimizing the duration a sensitive object is stored, the chance of data exposure is decreased. *TaintBochs* tracks the tainted data by running the complete software stack in a simulated environment. They implemented taint flags at processor states, device states and system memory. They showed that at the time many software projects did not take care of reducing the lifetime of sensitive data. Often buffers are deallocated without clearing them from sensitive information.

To our current knowledge the projects K-Tracer, TaintQemu, Panorama and TaintBochs are not open source, which could make the access to the source code and further development difficult. In case we could use any of those projects, we expect that it would not be possible to make the source code developed along this thesis publicly available. Therefore, we prefer an open source solution with an active community maintaining the source code.

PIRATE [32] is a system that implements information flow tracking based on the LLVM intermediate representation. It allows architecture independent analysis by translating all guest instructions to an intermediate representation. The result is that Whelan et al. only needed to write data tracking code for 29 instructions. PIRATE was then further developed into a full reverse engineering toolkit named *PANDA* [33]. It provides a rich plugin system and includes many useful plugins, like dynamic taint analysis and Operating System Introspection. Plugins can receive callbacks as well as provide callbacks to other plugins. We chose PANDA as a starting point for our implementation because

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

it is open source and comes with many features we need. More on the architecture of PANDA can be found in Section 3.3.

2.3 System Calls Fuzzers

Fuzzing or *Fuzz Testing* is a technique for software testing that tries to find software bugs by inputting (semi-)random data. It is therefore a blackbox-based technique. [34] Programmers have created fuzzers that aim to test syscalls since at least 1991. An early syscall fuzzer was *tsys*[13], which basically calls random syscalls with constant nonsense data. These first fuzzers found some obvious bugs. However, after a basic input validation was added, it becomes very hard for these early fuzzers to guess the input data just right so that the kernel does not reject the system call immediately. [35]

Trinity[14] is a Linux system call fuzzer that uses semi-intelligent arguments, so that most of the time the input validation will succeed and the actual implementation of a syscall will be tested. The fuzzer has knowledge about the datatype a certain syscall requires. For example, if a syscall needs a file descriptor, Trinity can provide a valid one. Likewise, for flags and ranges of values, if syscalls only accept these then trinity will provide such a valid value. In these cases, Trinity will sometimes throw in an invalid value at random such that more obscure bugs can be found. [35]

Furthermore, Trinity can apply sanitizing functions after the random data has been created, which ensure that the arguments are valid [36]. For example, Listing 2.1 shows the definition for fuzzing of syscall *rt_sigaction*. This section of code was taken from the Trinity source file *syscalls/sigaction.c*. The listing demonstrates that the sanitise function sometimes sets the argument 2 and 3 to *NULL* (pointer with value 0) and the argument 4 is always set to the size of the signal set, which is of the type `sigset_t`. In the struct `syscall_rt_sigaction`, the name is set to the syscall name, the number of arguments is defined, the sanitise function and the types of the arguments, with extra information like the upper and lower bound for ranges, is stored.

```
static void sanitise_rt_sigaction(struct syscallrecord *rec)
{
    if (RAND_BOOL())
        rec->a2 = 0;

    if (RAND_BOOL())
        rec->a3 = 0;

    rec->a4 = sizeof(sigset_t);
}
```

[13] https://groups.google.com/forum/?hl=en#!msg/alt.sources/V_B37EtnWKQ/NztsljVYV84J

[14] http://codemonkey.org.uk/projects/trinity/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
struct syscallentry syscall_rt_sigaction = {
    .name = "rt_sigaction",
    .num_args = 4,
    .sanitise = sanitise_rt_sigaction ,
    .arg1name = "sig",
    .arg1type = ARG_RANGE,
    .low1range = 0,
    .hi1range = _NSIG,
    .arg2name = "act",
    .arg2type = ARG_ADDRESS,
    .arg3name = "oact",
    .arg3type = ARG_ADDRESS,
    .arg4name = "sigsetsize",
};
```

Listing 2.1: Trinity fuzzer code for the syscall rt_sigaction

Trinity was chosen as a source of syscalls because it offers a higher success rate than the early syscall fuzzers due to the knowledge about system call argument types. There are more advanced fuzzers, like *syzkaller*[15] and *kAFL* [37], which have some notion of coverage. However, to keep the setup simple for this thesis, Trinity was chosen. For example, syzkaller requires that the kernel is compiled with special flags that enable the coverage analysis. This could potentially cause an increase of false positives for the purposes of this thesis. Please note, that the developed solution is independent of the used syscall fuzzer, so it is not hard to do the setup with a different fuzzer.

[15] https://github.com/google/syzkaller

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Background

The implementation of an automated process for identification of cross-container side-channels requires the knowledge about some key technologies and conventions. In this chapter, we summarize the necessary details to give background information for understanding of the implementation in Chapter 5.

3.1 Container Virtualization

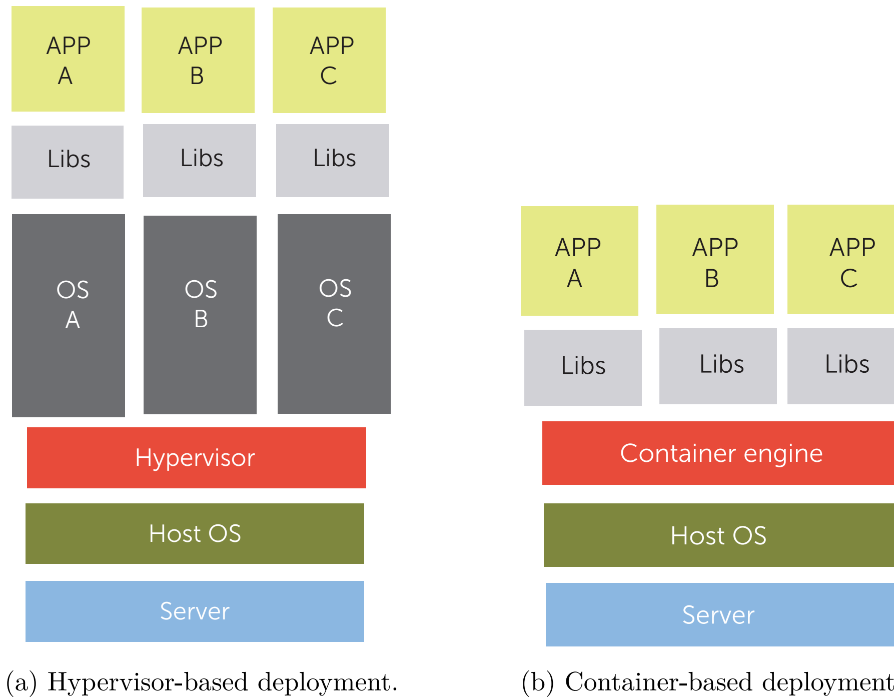
Container-based deployments provide a lightweight and standalone virtualization, which pack everything an application needs (e.g. libraries, code, tools and settings) into one package. They are portable, which means you can change the host environment without affecting the containerized application. [38, 39] Due to their small size it possible to run and store hundreds of containers on one physical host [39].

3.1.1 General Overview

With the raise in popularity of cloud computing systems, hypervisor-based virtualization became an often seen tool [39]. This form of virtualization allows you to run different operating systems on one physical host. In Figure 3.1 you can see an overview of the different virtualization approaches.

As you can see a hypervisor-based deployment enables users to run multiple operating systems on one host. This can be a requirement of a cloud customer, who is only familiar with certain operating systems, or a hosted application, which requires a certain operating system. Hypervisors come in two forms, type 1 hypervisors, where the hypervisor software runs directly on the physical hardware, and type 2, where a host operating system runs the hypervisor. [40] Figure 3.1a shows a type 2 hypervisor.

Container-based virtualization provides an isolated environment for processes. Containerized application share the same operating system kernel with other containers



(a) Hypervisor-based deployment. (b) Container-based deployment.

Figure 3.1: Differences between a hypervisor-based deployment and a container-based deployment. Figures taken from [39].

and the host system. Figure 3.1b shows that a container engine is used to administrate the containers. Containers have the benefit of being much smaller in size because they can share libraries. Depending on the application, they can also be started in a few milliseconds [39, 40].

From a developer's point of view, one container image is built for one application, which can be deployed on a system. There is no need to manage the host machine by hand or to worry about dependency conflicts of used software libraries. Containers can also improve the reproducibility of software builds (or even research results [41]) and the synchronization of dependency versions across different host machines. [40]

It is also possible to use both approaches in one system. In such a hybrid setup, users would run the host for the containers on top of a hypervisor and the containers in it. This would improve the isolation between the applications and have the benefit of easier management [40]. *Kata* containers[1] combine virtual machines and containers in a single runtime. They use lightweight virtual machines to provide a stronger isolation due to hardware virtualization.

[1]https://katacontainers.io

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.1.2 Architecture of Docker

For this thesis Docker was chosen as the container engine implementation. This is because Docker is currently very popular [42, 43]. However, many statements are true for some or all other container engines. The developed plugin of this thesis is independent of the software under test. Docker is not a monolithic peace of software. It builds upon other open source projects, where some are used by other container engines.

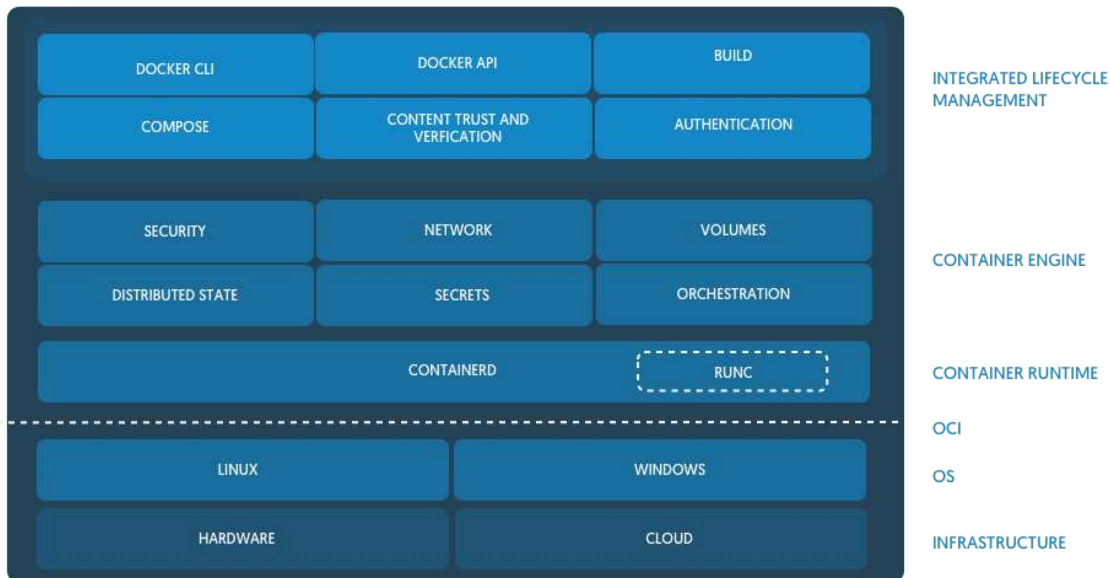


Figure 3.2: Overview of the Docker architecture. Figure taken from [44].

Figure 3.2 shows the architecture of containerized applications using Docker as a container engine. Docker provides a CLI and an API for managing containers. Docker containers require an image to start and these images are built upon other (base-)images. The images are created in a script format called *Dockerfile*, which consists of a series of commands to build an image. [39] The images can also be made publicly available using registries, like *Docker Hub*[2], which eases the deployment of a service due to documentation and comments from other users [40].

containerd[3] is the container runtime used by Docker [38]. It acts as an abstraction layer between the kernel level and higher level engines. It abstracts the use of syscalls like *clone* and *mount* with containers or snapshots. [44] containerd runs as a daemon and controls a complete container lifecycle. It also fully implements the Open Container Initiative (OCI) runtime specification.[4] The OCI runtime specification "*aims to specify the configuration, execution environment, and lifecycle of a container.*[5]".

[2] https://hub.docker.com
[3] https://containerd.io
[4] https://github.com/containerd/containerd
[5] https://github.com/opencontainers/runtime-spec/blob/master/spec.md

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Internally containerd uses runC to create the actual containers on Linux. runC is tightly coupled to the Open Container Initiative runtime specification and developed by the OCI itself.

3.1.3 Other Container Engines and Similar Software

As already mentioned in Section 3.1.2, we focus on Docker for this thesis. However, other runtime implementations and other software exist, which allow the containerization of processes. During our research, we discovered that many of these software products share the same security properties with Docker.

rkt[6] is a container engine, which is an alternative to all levels of the Docker architecture, namely Docker (CLI, API), containerd and runC [45]. The *Seccomp Isolators Guide* of rkt states that the default seccomp profile, when no profile is specified, is `@rkt/default-whitelist` and that rkt ships with the ability to use the default Docker profile [46]. Moreover, if we look in the implementation of the default rkt seccomp profile, the variable `RktDefaultSeccompWhitelist` is set to `DockerDefaultSeccompWhitelist`, thus the default rkt profile is the same as the default Docker profile[7]. The same is true for the default capabilities, where both share the same list of 14 capabilities[8].

Another project is *Apache Mesos*[9], which is a *distributed systems kernel*. It abstracts compute resources, which enables computer clusters to be elastic and fault-tolerant. Its default seccomp profile is very similar to Docker's[10]. The Mesos documentation about the seccomp isolator [48] states that the profile must be in the Docker format. Again, compared to the default Docker profile from September 2018, it is equal to the one of Docker, except it adds support for the syscall *pivot_root*.

This means that our choice of using Docker does not only cover the security of Docker containers. All software that uses the same profile or is based on such a container engine has (at least partially) the same isolation level as Docker.

For Linux systems software exists that allows developers to package applications. In case of the following examples, the software allows users to run the applications in a sandbox, which is implemented similar to containers. Therefore, they are also of interest for this thesis because they also aim to provide a secure application isolation. As an example, there is *Flatpak*[11], which makes use of a default seccomp profile for sandboxing.

---

[6] `https://coreos.com/rkt/`

[7] `https://github.com/rkt/rkt/blob/master/stage1/init/common/seccomp_wildcards.go#L412`

[8] Compare [47] with `https://github.com/moby/moby/blob/master/oci/defaults.go#L14-L30`.

[9] `http://mesos.apache.org`

[10] Compare `http://mesos.apache.org/documentation/latest/examples/seccomp_default.json` with `https://github.com/moby/moby/blob/ccd22ffcc8b564dfc21e7067b5248819d68c56c6/profiles/seccomp/default.json`.

[11] `https://flatpak.org`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

In the source code of Flatpak we found that it uses a blacklist approach, which is "*copied from linux-user-chroot*"[12]. Flatpak also uses Linux namespaces to provide a unique environment for each application. A complete list of the current default setup can be found under [49]. There is also *AppImage*[13], which uses *Firejail*[14] as an optional sandbox. Firejail uses a system of profiles that define the sandbox for a certain application. Over 400 profiles already exist for common software[15]. In the profile a list or pre-defined classes (or both) of syscalls can be found, which are filtered using seccomp. Furthermore, a list of Linux capabilities for the execution of the application can be defined.

3.2 Technologies of Linux Containers

Linux containers rely on several technologies built into the Linux kernel. These technologies provide isolation and security such that a process within a container cannot interfere with the host's and other containers' resources unless explicitly permitted.

3.2.1 Kernel Namespaces

One of the core concepts of containerized processes is *Kernel Namespaces*. They provide an abstraction to global resources such that processes within a namespace have their own instances of global resources. Changes to the resources only appear to the processes within the namespace and not to the other processes. [50] Linux Version 5.0 supports the following types of namespaces:

- Cgroup: A cgroup namespace isolates the view of the cgroups by having different cgroup root directories. The root directories act as base points for the relative paths found in */proc/[pid]/cgroup*. When a new cgroup namespace is created, the root directories are set to the caller's root directories. The paths found in */proc/[pid]/cgroup* are relative to the process's root directory, which can lead to paths starting with *../* if the read process's cgroup directory is outside the reading process's root directory. [51] We discuss more details of cgroups in Section 3.2.2.
- IPC: These namespaces isolate *System V IPC objects* and *POSIX message queues*. The objects within a namespace are not visible to processes outside the namespace and they are automatically destroyed when the last processes within the namespace exits. [50]
- Network: Each namespace has its own network devices, IP protocol stacks, IP routing tables, firewall rules, subdirectories in the *proc* and *sys* directories, sockets and *UNIX domain sockets*. Physical network interfaces can only be in one namespace.

[12] `https://github.com/flatpak/flatpak/blob/0bb64e9a8833e3158328c99391de2b9930cf6964/common/flatpak-run.c#L2427-L2428`

[13] `https://appimage.org`

[14] `https://firejail.wordpress.com`

[15] `https://firejail.wordpress.com/documentation-2/basic-usage/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

When a namespace gets destroyed, physical interfaces are assigned to the initial namespace and virtual interfaces will be destroyed. [52]

- Mount: Namespaces of this type isolate the list of mount points. When a new mount namespace is created, the mount point list is copied from the caller. Changes to the mounting points, by default, do not affect the mount points of other namespaces. [53]

- PID: Processes in different PID namespaces can have the same Process ID Number. This enables migrations of containers from one host to another without affecting the PID. The first PID in a new namespace is always 1, mimicking a new standalone system. This process acts as the *init* process and will be the parent for orphaned processes within the namespace. If this init process terminates, all processes within the namespace will receive the signal *SIGKILL*. PID namespaces are structured in a hierarchy, where each namespace has a parent namespace, except the initial namespace. Processes in the same PID namespace can see each other and they are also visible by their ancestor PID namespaces. Therefore, a process has a PID for each level in the hierarchy. A process can change its PID namespace to one of its child's but not to the PID namespace of its parent. [54]

- User: A user namespace isolates user IDs, group IDs, the root directory, keys and capabilities. The user ID and group ID of a process can be different whether it is viewed from within the namespace or not. Therefore, a (from the outside) unprivileged process can have a user ID of 0 when examined from inside. Inside the namespace the process has full privileges, however, it is unprivileged outside. Like PID namespaces, user namespaces can be nested. It is possible to gain capabilities by changing the namespace (if the process has the *CAP_SYS_ADMIN* capability). [55]

- UTS: Unix Time Sharing namespaces isolate the hostname and the Network Information System domain name [50].

The lifetime of a namespace ends when the last process terminates or leaves the namespace. However, there are exceptions to this rule, where namespaces exist without processes. For example, in case of nested namespaces, a child namespace can still have processes. A more extensive list of exceptions can be found at [50].

Namespaces can be controlled by several syscalls. For these, the calling process must have the *CAP_SYS_ADMIN* capability[16]. The system call *clone*[17] creates a new process (or thread). It has the parameter *flags* that influences the behavior of the syscall. The constants starting with *CLONE_NEW* create new namespaces for the newly created process. [56]

[16]Except for the creation of a user namespace [50].
[17]For more details on the *clone* syscall see [56].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

The syscall *setns*[18] can change the namespace of the calling process. The first parameter is a file descriptor referring to a */proc/[pid]/ns/* directory and the second parameter can specify the type of namespace (0 means any type). Some security restrictions apply to this system call, for example, it is not possible to reenter the namespace to gain dropped capabilities. [57]

With *unshare*[19] users can disassociate parts of the process from other processes when they have a shared execution context. For example, when forking a process, the new child process has the same mounting namespace. With *unshare* it is possible to unshare the mounting namespace by creating a copy of the mounting point list. Unshare also supports other *CLONE_NEW** constants from *clone*. [58]

Finally, the *ioctl*[20] system call reveals information about the relation of namespaces. The first parameter *fd* should refer to a file in */proc/[pid]/ns/* and the second is a constant, which name starts with *NS_*. Operations, like get the type of a namespace, get the user ID of process who created the user namespace and get the parent namespace (for hierarchical namespaces) are available. [59]

In the Linux kernel, there are two objects that are not namespaced, therefore they are shared between all containers (if accessible). The first one is time, which means that processes in different namespaces could have different clocks. There is no time namespace because there is probably no production use case. To change the time within a container, the container must be started the additional capability *CAP_SYS_TIME*. The second object is the kernel keyring. Currently, a user has access to its keys no matter in which namespace the process is running. There is already a patch to make keys user namespaced, however, it is only suitable for the orchestrated container system use case and not for other use cases[21]. In Docker (and in other container engines), the system calls to the kernel keyring is denied by the default seccomp filter, making it inaccessible from within a container. [60]

3.2.2 Control Groups

Control Groups, also known as cgroups, are a feature of the Linux kernel that allows to pose restrictions and to monitor some types of process resources. A *subsystem* or *controller* is a component that alters the behavior of a resource, for example, that the memory available to a cgroup is limited. The Control Groups are organized in a hierarchy in a pseudo-filesystem called *cgroupfs*. Subdirectories in this filesystem represent cgroups subgroups. The attributes, like limits and accounting information can be placed on any level of the hierarchy. Limits on a lower level cannot exceeded the limits on an upper level. [61]

[18]For more details on the *setns* syscall see [57].
[19]For more details on the *unshare* syscall see [58].
[20]For more details on the *ioctl* syscall in context of namespaces see [59].
[21]`https://patchwork.kernel.org/patch/9394983/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Currently, there exist two versions of Control Groups, version 1 and version 2. Both can be used simultaneously, however, a controller can only be mounted in one version's hierarchy. Version 2 is intended to replace version 1 eventually.

Details on which controllers exist and how to add or move processes to certain cgroups of version 1 and 2 can be found in [61].

Control Groups Version 1

In version 1 Control Groups it is possible to mount each controller to separate locations, thus, creating their own hierarchy. It is possible to mount the same set of controllers to multiple mounting points. In this case, they share the same view over the hierarchy. But it is not possible to mount different sets, which contain common controllers, to multiple locations. In many systems, especially when *systemd*[22] is used, the cgroupfs is automatically mounted under */sys/fs/cgroup*.

Version 1 also distinguishes between processes and threads, where it is possible to independently set the cgroup memberships of each thread of a process. This can cause problems or simply does not make sense in some cases, like memory limits because threads share the same memory space. In version 2 the distinction between processes and threads was removed.

Mounted cgroupfs filesystems can only be unmounted if they are not busy, which in this context means they have no child cgroups. To remove all child cgroups, all processes in the child cgroups must be moved to another cgroup before.

Control Groups Version 2

Control Groups of version 2 simplified the cgroup system by only having one hierarchy for all controllers. Different containers can be mounted in version 1 and version 2 hierarchies, but it is not possible that the same controller is mounted simultaneously in hierarchies of both versions.

As already mentioned in Section 3.2.2, tasks are no longer subject to cgroups. Another key difference to version 1 is the *no internal processes* rule. It requires that processes are only present in the hierarchy leaf nodes. The root cgroup is an exception of this rule. It is still possible to add processes at any level in the hierarchy by adding a leaf node at the desirable level and to put the processes in there. The rule makes relations between child cgroups explicit and eases the decision on how to partition resources between a cgroup and its child groups.

There are further minor differences between Control Groups of version 1 and 2, which can be found in [61].

[22] `https://wiki.freedesktop.org/www/Software/systemd/`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.2.3 Linux Kernel Capabilities

In the traditional UNIX world, a process is privileged if the effective user ID is 0, or unprivileged if the effective user ID is nonzero. When performing permission checks, the kernel skips all checks for privileged processes. However, to create a finer grained distinction, capabilities were introduced. [12]

In case of Docker, there exists a default whitelist of capabilities a containerized process has. In Docker, it is possible to start a process with more than default capabilities or less. [62] Here is the default list[23] in the current version of Docker[24]:

- CAP_CHOWN
- CAP_DAC_OVERRIDE
- CAP_FSETID
- CAP_FOWNER
- CAP_MKNOD
- CAP_NET_RAW
- CAP_SETGID
- CAP_SETUID
- CAP_SETFCAP
- CAP_SETPCAP
- CAP_NET_BIND_SERVICE
- CAP_SYS_CHROOT
- CAP_KILL
- CAP_AUDIT_WRITE

Capabilities can be set on threads and on executable files. On threads they affect the thread itself and the threads created by the thread. Capabilities on files define the capabilities a started program (*execve*) has. This can be thought as an extension of the SUID bit. Each thread has sets of capabilities, which define the behavior of this mechanism. The following sets exist: [12]

[23] Their exact meaning (and the meaning of all other capabilities) is described in [12].
[24] `https://github.com/moby/moby/blob/master/oci/defaults.go#L14-L30`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Permitted: It is a limiting superset for the capabilities in the effective set. If the current thread does not have the *CAP_SETPCAP* capability, it is also a limiting superset for the inheritable set. Once a capability is dropped from this set it cannot be acquired again, unless *execve* is called with a file that has the SUID bit or file capabilities set.

- Inheritable: This set is preserved when calling *execve*. However, this is generally not the case for programs running as a non-root user. The inheritable set is combined with the inheritable set of the file to form the permitted capabilities after the *execve*.

- Effective: The kernel uses this set to perform the permission checks.

- Bounding: This set can be used to limit the permission gained by calling *execve*.

- Ambient: The ambient set is preserved when *execve* is called in an unprivileged program. In this case, the ambient capabilities are added to the permitted set. The ambient set has the property that all capabilities in this set must be both in the permitted and inheritable set. This also means that a capability is removed from the ambient set if it is removed from the permitted and inheritable set. If a program changes the UID or GID due to a SUID or SGID bit or if any file capabilities are set, then the ambient capabilities will be cleared.

File capabilities are stored in the extended attribute *security.capability*. To be able to write to this attribute requires the capability *CAP_SETFCAP*. The file capabilities are used for determine the thread capability sets after *execve* is called. The following capability sets for files exist: [12]:

- Permitted: Unless in the bounding set, these capabilities are added in the permitted set of the thread.

- Inheritable: The intersection of this set with the inheritable set of the thread is added to the permitted set of the thread.

- Effective: This is only a single bit that determines if newly added permitted capabilities are also automatically added to the effective set.

To calculate the capabilities, *execve* performs the algorithm [12] as shown in Algorithm 3.1, where $P(\cdot)$ denotes the thread set before *execve* is called, $P'(\cdot)$ after *execve* is called and $F(\cdot)$ a file capability set.

For programs, which real or effective UID is zero, the file inheritable and permitted sets have all capabilities enabled. However, there is one exception to this rule, when a SUID to root program has file permissions set and is executed by a non-root user, then the file capabilities are taken (and not overridden to be all enabled). Thus, it is possible to create a SUID to root program with empty file capabilities sets that changes the effective

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Algorithm 3.1: Capability Algorithm for Threads

1 P'(ambient) ← (file is privileged) ? 0 : P(ambient)
2 P'(permitted) ← (P(inheritable) & F(inheritable)) | (F(permitted) & P(bounding)) | P'(ambient)
3 P'(effective) ← F(effective) ? P'(permitted) : P'(ambient)
4 P'(inheritable) ← P(inheritable)
5 P'(bounding) ← P(bounding)

user to root but does not enable more capabilities. For some of these rules there are several special cases and things to consider. For a detailed description, please have a look at [12].

The capabilities of a thread can be altered by the syscalls *capset* [63] and *prctl* [64], however, on systems with file permissions it is not possible to alter other processes because the intended way is to change the capabilities is with file permissions [63]. *prctl* only allows to drop capabilities from the bounding set and to alter the ambient set, whereas with *capset* all capabilities can be altered.

3.2.4 Secure Computing Mode

The secure computing state of a process comes in two variations. There is the strict mode that permits a program to only execute the system calls *read*, *write*, *_exit* and *sigreturn* [65]. This means that all necessary file descriptors and signal handlers must be set up before a process enters this mode.

The other variation is the filter mode, which allows a BPF filter to be installed for checking the syscalls. Child processes inherit the filter if creating child processes is permitted. The filter is preserved when *execve* is called. To use the filter mode the process must have the *CAP_SYS_ADMIN* capability or the *no_new_privs* bit must be set. If the bit is not set it must be set before adding the filter. [65] The *no_new_privs* bit prevents processes from ever gaining more privileges than they currently have, even if, for example, a program with the SUID bit is started [64]. Therefore, filters could be added, if still permitted, but never removed.

In case of Docker, a default *seccomp* profile[25] is applied to every new containerized process. The profile defines a default policy and a whitelist of syscalls. Each list of syscalls can be further narrowed down to specific arguments, architecture or set capabilities. The profile is then converted[26] using *libseccomp*[27] to a BPF filter.

The Berkeley Packet Filter (BPF) is a language for implementing packet filtering. It is now present in the Linux kernel and it is not limited to network filters. It can be used

[25] https://github.com/moby/moby/blob/master/profiles/seccomp/default.json
[26] https://github.com/opencontainers/runc/blob/2c632d1a2de0192c3f18a2542ccb6f30a8719b1f/libcontainer/seccomp/seccomp_linux.go
[27] https://github.com/seccomp/libseccomp

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

as an universal in-kernel virtual machine. BPF is utilized by, for example, seccomp and *bpftrace*[28]. [66]

3.2.5 Linux Security Modules

LSM is a framework that allows to build Mandatory Access Control (MAC) extensions into the kernel. The extensions are built into the kernel at compile-time and the default one can be selected at boot-time with a kernel command line argument. Some examples of these extensions include *SELinux*[29], *Smack*[30], *Tomoyo*[31], and *AppArmor*[32]. The default LSM is the Linux capabilities system and it can be used as a basis for building an extension upon. [67]

In Docker, it is possible to define an AppArmor policy, which is then applied for the starting container (if AppArmor is enabled). Otherwise, a default profile is applied. The profile must be loaded on the host before it can be used for containers. [68]

3.2.6 Other Security Aspects

There exist several other security mechanisms that can be enabled for a containerized process. Any kernel security improvement is also applicable for containers. For example, *PaX*[33] is a security patch for the kernel source code. If a host system runs with a *PaX* patched kernel, the security of the containers is also improved because they share the same kernel. [62]

3.3 Taint Analysis with PANDA

PANDA stands for Platform for Architecture-Neutral Dynamic Analysis and is an analysis engine based on QEMU [69, 33]. It can perform whole-system dynamic analysis and includes the ability to record and replay a certain execution. It features a plugin mechanism that simplifies the development of additional analysis modules.

PANDA has been chosen as a starting point of this thesis because it supports whole-system analysis. This means it can perform analysis for not only user programs but also kernel code. It includes plugins, which ease the development of a software like the one developed in this thesis. The plugins used are described in Section 3.3.2. The replay feature eases the development of plugins because the whole operating system is already started for each run and every execution of a recording is exactly the same. The PANDA projects provides rich documentation and includes examples, which greatly help to start with a project.

[28]https://github.com/iovisor/bpftrace
[29]https://selinuxproject.org
[30]http://schaufler-ca.com
[31]http://tomoyo.osdn.jp
[32]https://gitlab.com/apparmor/apparmor
[33]https://pax.grsecurity.net

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

3.3.1 PANDA Analysis Architecture

PANDA is based on QEMU, which supports multiple architectures by translating the native instructions to an intermediate language called *Tiny Code Generator*. However, more complex instructions are implemented in the C programming language [70]. These are called *helper functions*. For a more sophisticated analysis, a common intermediate language is needed. In [33] Dolan-Gavitt et al. used *clang*[34] to create LLVM[35] intermediate code from the helper functions. Furthermore, they used a module from the S2E[36] system to translate TCG code to the same intermediate language. PANDA then uses the LLVM intermediate language as a basis for the taint analysis.

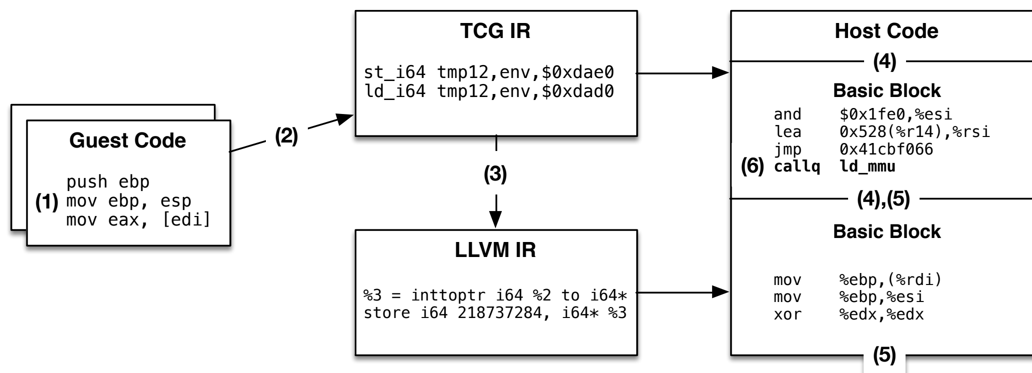


Figure 3.3: Overview of the PANDA instruction translation. Figure taken from [33].

The Figure 3.3 shows an overview of the described process including points where instrumentation can be performed. Plugins can register callbacks and these will be called when certain events happen. (1) refers to events when a guest instruction is executed. Callbacks are also called before and after the translation, which are shown in (2) and (3). (4) and (5) mark the events before and after a basic block is executed. Plugins can also register callbacks for memory accesses (6). [33] There are many more callbacks available, a current list can be found in the PANDA manual [69].

Plugins can further access functionality like reading and writing of memory, command line argument parsing and access to a precise program counter. The plugins themselves can provide callbacks. It is therefore possible to build more complex analysis programs on top of a set of already available plugins. This has already been done for some included plugins because it advocates the separation of concerns.

[34] https://clang.llvm.org
[35] https://llvm.org
[36] https://s2e.systems

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.3.2 PANDA Plugins

To develop the plugin for this thesis, included plugins from PANDA were used. Please note, that PANDA is currently in version 2 and the following plugins have been updated from version 1, hence the digit *2* at the end of the name of some plugins.

The osi and osi_linux Plugin

The *osi* plugin provides operating system introspection. This means that it can be used to get information stored in the current running system, like the current process's PID, PPID and name. It is able to fetch information about the current thread, kernel module and user libraries. The osi plugin itself is only a glue layer with common functionality between the plugin that uses it and the operating system specific implementation. The aim was to create a common interface, such that a plugin can be created that works with multiple operating systems. In case of Linux, the implementation is found in the *osi_linux* plugin. [71]

The osi_linux plugin provides the Linux specific implementation for the osi plugin. It needs a configuration file with a list of Linux kernel data structure offsets. With these offsets, it can traverse the data structures in the guest memory and find the requested information. The offsets change with different Linux kernel versions and compile flags, so to help with this a kernel module is provided. This kernel module must be compiled on the guest and then loaded to create the necessary configuration file. [72] Please note that this plugin had to be patched with changes from a fork so that it works with more current versions of the Linux kernels. We discuss the details of the version used for this thesis in Section 5.2.

The syscalls2 Plugin

The *syscalls2* plugin features callbacks for system calls on supported operating systems. Callbacks are available for the beginning (*enter*) and for the end (*return*) of a syscall. Callbacks can be registered for specific syscalls, for all syscalls and for unknown syscalls. The information a plugin gets when using the syscall2 plugin includes a *CPUState*, which is a representation of the current CPU state, the current program counter and the number of the syscall. Furthermore, newer callbacks exist that add additional information, like the syscall parameter's data types (for example, strings or pointers to structures) and size in bytes. [73]

The taint2 Plugin

With the *taint2* plugin it is possible to track the flow of data. Labels can be applied to memory addresses. It is then possible to follow the flow and to query addresses to get the applied labels. The word *address* refers to all types of memory that a label can be applied to. Labels can be applied to memory like, physical RAM, hard drives, IO buffers, port addresses, local values (similar to CPU registers but in the LLVM

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

intermediate representation), general purpose register and special addresses like floating point registers. [74]

taint2 supports label sets, which means one address can have multiple labels assigned. Labels can be applied additively, or not, which means that the current labels will be removed before the new one will be added. Depending on the operation and input parameter label sets, an output label set is created. For example, if some address a has the label set $\{1\}$ and a different address b has the label set $\{2\}$, then the resulting label set when performing an addition $(c = a + b)$ is $\{1, 2\}$. [74]

For every tainted address a computing number is stored. This number indicates how many operations were performed on the data. Furthermore, the control mask bits mark the bits of a byte that are (directly) under control of the initially tainted data, which we can think of as input from a potential attacker. Furthermore, the zero and one bit mask indicate bits of a byte that are always set to 0, respectively 1. Please note that these bytes are stored per address and not per label, which means they generally indicate the "best case" for a potential attacker.

The plugin can be configured with several options. By default, dereferenced pointers are also tainted. This means that indirect accessed memory is also tainted with labels of the source bytes and the pointers labels [74]. First all the labels of the pointer bytes are mixed and then the labels of each source byte are added to the destination byte. In the PANDA source code the following example can be found[37]: A pointer's bytes could have the label sets $\{1\}$, $\{2\}$, $\{3\}$ and $\{4\}$ and the source bytes could be labeled with $\{5\}$, $\{6\}$, $\{7\}$ and $\{8\}$. Then the first step would be to mix the pointer's labels, which results in $\{1, 2, 3, 4\}$. This new set is then added to each source label set and these are applied to the destination bytes. The result is that the destination bytes have the label sets $\{1, 2, 3, 4, 5\}$, $\{1, 2, 3, 4, 6\}$, $\{1, 2, 3, 4, 7\}$ and $\{1, 2, 3, 4, 8\}$.

Another option detaints addresses when the control mask bits are all zero, which means a potential attacker would have no influence on this data. In the default configuration this behavior is not enabled, which can cause false positive results. There is also a way to limit the taint compute number, which will remove the taint in complex operations.

The taint2 plugin also provides callbacks for other plugins. On every taint change the callback *on_taint_change* is called and on every branch, which depends on tainted data, *on_branch2* is called. In both cases, the called function will get the address of the taint change and the size in bytes. [74]

The taint mechanism works by inserting extra LLVM operations in the translated instructions before execution. These extra operations make sure that the label sets and extra information (control mask, zero and one mask) are correctly applied for each guest instruction. Furthermore, the taint2 callbacks will be called to notify other plugins of certain events.

[37] https://github.com/panda-re/panda/blob/f69e57d5830079c5de1d5eecbbc71396914e4520/panda/plugins/taint2/taint_ops.cpp#L297-L300

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3.4 Linux System Call Calling Conventions

System calls provide the fundamental interface between user application and the Linux kernel [75]. On the Intel i386 architecture syscalls are invoked with the instruction `int $0x80`. The *EAX* register sets the number of the requested system, in other words, it marks which syscall is executed. Furthermore, up to six arguments can be used on i386. These arguments are passed over in the registers *EBX*, *ECX*, *EDX*, *ESI*, *EDI* and *EBP*, where the register *EBX* is used for the first argument, *ECX* for the second and so on. The return value of the syscall is store in the *EAX* register. [76]

The arguments can directly represent the value used by the syscall, for example, in the system call *setpriority* [77] the third parameter is the new priority value. The value stored in the argument registers can also act as a pointer to more complex structures like strings (for example, the first argument of *sethostname* [78]) and structs (for example, the first argument of *uname* [79]). In case of more complex structures, the kernel is required to copy in and copy out user data. The implementations of syscalls are not allowed to access the user space directly so they have to use dedicated functions, like *copy_from_user()* and *copy_to_user()*, which are responsible for user space interactions. [80]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 4

Methodology

The stated problem will be solved by a proof of concept implementation, which can identify side-channels between containers. The basic idea is to label data that enters the kernel from a containerized process A with a label that marks the data as controllable by A. If for any reason, like a system call or a series of system calls, A marked bytes are accessible by a process B, which is executed in another container, then there exists a channel between A and B, where at least A can write some data and B can read it. The following sections present the ideas and design choices that aim to identify the side-channels over the kernel. Furthermore, we discuss the automation strategies to create a fully automated system that is capable of reporting findings.

4.1 Tainting System Call Input Data

There are two places where input labeling is performed. The first one happens when a syscall is called. The bytes of the system call argument registers, as described in Section 3.4, will be labeled additively with a label representing the current process. The word *additively* means that the labels already in place will not be removed or overridden by the new label. The second place where tainting must be performed is when the processor is in privileged mode (kernel code is executed) and the kernel reads data from user space. The read data will then be marked with a label of the current process. This case often occurs when the kernel copies data from the user space to the kernel space but also fits for syscalls, where the kernel copies data directly from one process to another without using the kernel space. Like the first case this happens additively.

Although the project PANDA taints dereferenced data, when it is marked by the pointer's and the source' labels (see Section 3.3.2), data located at an offset of a pointer will not be tainted automatically. This is caused by the calculation of the offset, which is basically an addition. The operation sets the control mask bits to 0. Moreover, this leads to the removal of all labels assigned to an address at the next instruction that operates on the

data. Therefore, the second case, where copied in user data will by marked, is essential. It also allows a better understanding of the syscall input data because every byte directly (argument registers) and indirectly (copied in from user space) can be logged.

4.2 Detection of Cross-Container Tainted Data

Detection of side-channels happens at three different places. Analog to the first labeling case, when a system call returns, the registers will be checked for labels of the other process. Again, all bytes must be checked. The second case exists when the kernel copies out data to the user space of another process. Once the data is written to the user space, the application can read the data. If memory is shared between two containers or pages are reused but not zeroed out, no write operation of this case will be detected. Furthermore, the second case does not include instances where the kernel runs as an unrelated process and writes to the user space of a container. Therefore, a third case checks data reads from the user space if the CPU is in user mode.

4.3 Taint Labels

The taint labels are constructed in such a way that they allow basic comprehensibility. For each of the two syscall input cases of Section 4.1, a distinct label format (bit mask) exists. Both formats store which process created them. A label created by the first case, where the registers are labeled directly when the syscall is called, includes information of which syscall number was called and which argument it represents.

In the other case, the last syscall number called is stored and a number that is increased on every creation of this type of label. Once the current number reaches a certain maximum it starts over from the initial value. It was added because there is not enough space to store a complete address and this way allows to store some information about the origin of the label. Please keep in mind, that this information is not unique because the number periodically starts over.

To reduce the log output, findings can be uniquified, so that one successful found side-channel label will only show once in the logs. Findings of the first type are considered equal if they have the same label. The second type findings are different. They are only the same if the label **and** the memory address, which was used in the write or read operation, are equal.

4.4 Automation Strategies

There exist two main operating modes for the developed plugin. It can process a previously created recording, which has the benefit of being repeatable, meaning multiple analysis runs always result in the same execution, but its length is always bounded because there cannot be a recording that has an infinite size. Furthermore, it is not interactive, which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

means it is not possible to change the execution during a replay and to see the effect such a modification would make. The second use of the plugin is to run it live during an emulation of a whole guest system. This dramatically slows down the execution of the guest, which could interfere with the guest processes, for example, timeouts are triggered because the emulation is too slow to fulfill a request. Otherwise this operating mode has the opposite properties of using a recording. It is not repeatable, it could run forever and it is interactive.

Due to these properties and the fact that RAM is always bounded we came up with two possible strategies to use the plugin to find side-channels. In the first case, the plugin could potentially run forever emulating the whole system while analyzing for side-channels. However, results cannot be examined closer, by, for example, enabling specific log messages, because in this operating mode repeatability is not available. Therefore, an additional fuzzing run is required that should be restricted to the syscalls occurring in the reported finding. This run could then be recorded for a detailed analysis. Additionally, snapshots could be created in periodic time intervals to be able to jump before the involved syscall was called. Please note, that this would still not be truly repeatable because the execution will most certainly be scheduled differently.

In the other strategy, a recording of arbitrary size would be made first. This recording would then be analyzed for side-channels. If no finding was reported, then the process could be repeated by continuing where the last recording stopped. Alternatively, we can reset the fuzzer to run with a new sequence of random syscalls. This is necessary because otherwise the system would create very similar recordings in each round. This strategy has the benefit that once a finding is reported a repeatable recording already exists. However, it chops up the execution into distinct runs, where one replay analysis can only detect labels inside one run. In practice this issue is not unique to this strategy because the first strategy also has a steadily increased RAM consumption. This is because each byte in the guest's memory could have up to 2^{32} different labels, which would be up to 2^{64} Bytes (16 Exbibyte) for a 32-bit guest system. Furthermore, the tainting plugin of PANDA taints hard disk space, which can be arbitrary large[1]. This means both strategies need a restart once a memory limit is reached. The difference is that in the first strategy the supervisor script can react to reaching the limit during the run and in the second case a useful guess of how long a recording should be must be made before the analysis.

[1]Simplification. There are limits due to, for example, a finite address space.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 5

Implementation Details

The following sections give an overview of the system and discuss details regarding the implementation of the developed source code. Chapter 4 acts as a guide for the implementation in this chapter. Furthermore, we discuss methods that help debugging and diagnose found results including the information already outputted by our system.

5.1 System Overview

Figure 5.1 illustrates the complete analysis system. It shows the two containers, called *Container 0* and *Container 1* in the figure, which contain the system call fuzzing processes. These processes are illustrated by their main loop, which generates random syscalls. The syscalls are then executed in the Linux kernel. The aim of this thesis is to find paths from one containerized process to the other, which is shown by the red dotted line. To create such a system, the whole operating system is emulated inside a QEMU/PANDA instance.

The PANDA plugin receives callbacks when certain guest system events occur. Depending on the callback, addresses are tainted or checked for labels from the other process. For both tasks, further information, like the process name, will be fetched from the PANDA core system or other plugins. Once a potential finding occurs, the system reports it and writes it out to a log file. It is also possible to enable addition logging for certain events.

A control script controls and monitors the described system. It starts a certain PANDA instance, checks the log output for certain entries, like the number of syscalls, and monitors the guest system process if it still exists.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek, WIEN Your knowledge hub

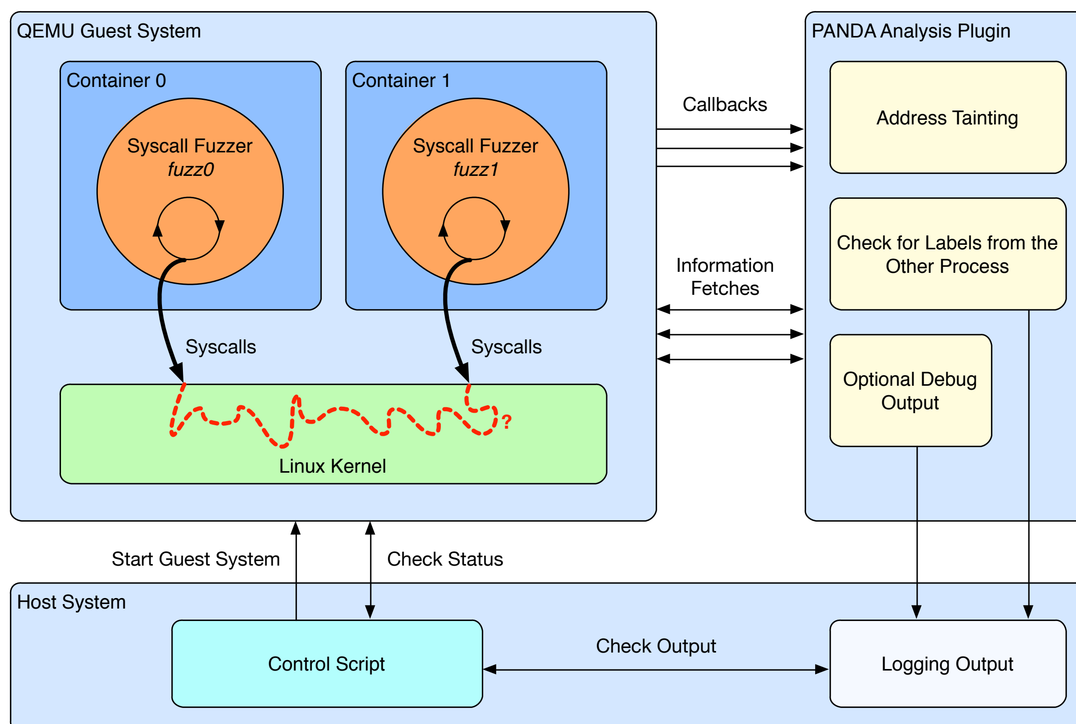


Figure 5.1: Overview of the analysis system.

5.2 PANDA Version

The PANDA project is used as a starting point for this thesis. We use the commit `f69e57d5830079c5de1d5eecbbc71396914e4520` because it already features much functionality needed for the implementation. At the time, this was the latest commit of PANDA. However, this version does not have a osi_linux plugin which supports Linux kernel versions equal or higher that 4.9[1]. Luckily a fork[2] from user *eshard* exists, which updates the structs and the kernel module for gathering the offsets to work with newer kernel versions. We merged the fork with the commit `f69e57d5830079c5de1d5eecbbc71396914e4520` to get a version that supports current kernels.

5.3 Retrieving the Currently Running Fuzzing Process

To differentiate the fuzzing processes from another we use the process name. The osi plugin provides the name of the current process. The Listing 5.1 shows the function that

[1]`https://github.com/panda-re/panda/issues/203`
[2]`https://github.com/eshard/panda/commit/2a8d097d101157bba2cae98806b54df4d587dcb8`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

gets a pointer to a *OsiProc* struct and returns the number of which fuzzing processes (*fuzz0* or *fuzz1*) the CPU currently handles. If none of the fuzzing processes are running, then the function returns -1.

```
#define PROCESS_PREFIX "fuzz"
#define PROCESS_PREFIX_LENGTH strlen(PROCESS_PREFIX)

inline int which_part(OsiProc *proc)
{
    if(
        strncmp(
            proc->name,
            PROCESS_PREFIX,
            PROCESS_PREFIX_LENGTH
        ) == 0
        && strlen(proc->name) >= PROCESS_PREFIX_LENGTH + 1)
        return proc->name[PROCESS_PREFIX_LENGTH] - 48;
    return -1;
}
```

Listing 5.1: Fetch information on which fuzzing process is currently running

The function takes the process name from the `proc` struct and checks if the first `PROCESS_PREFIX_LENGTH` characters equal the expected value `fuzz`. Furthermore, the length of the process name must be at least one more than the prefix. If so, then from the character after `fuzz`, 48 is subtracted, which converts the character from its ASCII representation to its integer value. This value is then returned.

5.4 Tainting System Call Registers and Kernel Reads

We use the callback `on_all_sys_enter2` to get a function called every time a system call is called. Registers, which are used as syscall arguments, are marked using the function `taint2_label_reg_additive()`. The function must be called for each byte where the *offset* parameter tells the tainting plugin which byte will be labeled. For each syscall all registers are labeled including *EAX*, which defines, which syscall is executed. A list of registers, which are used for storing syscall arguments on the i386 architecture, can be found in Section 3.4.

Analog to this, all bytes, which the kernel reads from user space, are labeled with `taint2_label_ram_additive()`. This is implemented with the PANDA callback `virt_mem_before_read`. However, PANDA's read callback only passes the source memory address but not the destination register [69]. The same is true for the write callback, where only the destination address is handed over. In a first approach, we just tainted the source address, which leads to false positives, for example, when memory mapped files are shared in a common Docker image. This behavior is also discussed in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Section 6.3. Therefore, we added a removal step for the just applied label in the taint change callback. This makes sure that the label will be applied to the destination of the read and that the labels of the source will be restored like they were before the read. In other words, a read operation does not apply labels to the source memory address, it only adds a label to the destination register.

To fetch whether the CPU currently is in privileged mode, we used the function `panda_in_kernel()`. The split between user space and kernel space in Linux is defined by the compile-time option *PAGE_OFFSET*, which is typically at `0xC0000000` on 32-bit systems[3]. This means the upper 1 GiB of virtual memory is used by the kernel, while the lower 3 GiB is user space. The function `taint2_label_ram_additive()` requires a physical memory address. It can be obtained with the function `panda_virt_to_phys()`.

A label in PANDA's tainting plugin has 32 bits. We use 1 bit to indicate which fuzzing process the label created. 15 bits indicate the number of the syscall argument register or, in the case of user space reads, a number that is increased on each created label. The number is set to an initial value greater than the maximum argument register number to differentiate between the two label types. If all number bits are ones (equals a value of $2^{15} - 1$), the number rolls over to the initial value. The remaining 16 bits are used to indicate the last entered syscall while creating the label.

5.5 Checking System Call Returns and Kernel Writes

Like in the labeling phase, the assigned function to the callback `on_all_sys_return2` checks if data labeled by the other fuzzing process is returned to the calling user process. We created the function `get_channel_set()`, which fetches all labels assigned to an address and filters out the labels created by the current process itself. The necessary memory to store these labels is allocated and extended as the set grows. The calling function is responsible for freeing the memory.

The function `get_channel_set()` is used to fetch the label set of every syscall argument register. In cases where a label does not originate from the current process, the system reports a finding. By the syscall calling convention, which is discussed in Section 3.4, only the register *EAX* returns data. However, we decided to check the same registers as in the input case because a user program could access them after the syscall return finishes.

In the check after the kernel writes to user space, the callback `virt_mem_after_write` did not show the correct label set. The reason is that the taint2 plugin updates only after the callback and we did not manage to delay our plugin to a point after the tainting plugin is executed. The documentation of PANDA says that "[...] *PANDA does not guarantee any fixed ordering for its callbacks across different plugins*"[69]. However, plugins can provide callbacks themselves, so we used the `on_taint_change` as an entry point for

[3] `https://elinux.org/images/b/b0/Introduction_to_Memory_Management_in_Linux.pdf`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

the detection function. The check is only executed if the CPU is in kernel mode and data is written to user space. As in the labeling phase, the split between kernel space and user space determines where the destination of the memory copy is. Like the first case, this function fetches the label set that does not include labels from the current process, which are then reported as a finding.

The third case, where the CPU executes a user program and reads memory, is similar to the previous case. However, the corresponding callback `virt_mem_after_read` can be used directly because the labels are already set by a prior writing instruction.

The log output of a finding includes the current fuzzing process, the register or memory location (virtual and physical) and the labeled data. Furthermore, the control mask, the compute number and the label are outputted. The label's information is also extracted, which includes the process that created the label, the syscall and the register or the number that identifies memory reads from user space by the kernel.

5.6 Dereferenced Pointer Patch

During the development, we discovered that pointer dereferencing tainting works in PANDA, however, the control mask is set to the control mask of the source. This leads to an issue when the source control mask is zero. In this case, the labels are correctly calculated as described in Section 3.3.2, however, when the next instruction is executed, all labels will be deleted because an attacker would not have any control over the result.

We propose that, if an attacker has full control over the pointer, the resulting (dereferenced) data should be under full control because she can point it to any data. Thus, we created a patch that checks if the pointer's control mask bits are all ones. If this is the case, it overrides the resulting control mask with all ones. Moreover, we increased the compute number by one to indicate that such an operation is performed during the analysis.

```
bool fully_controlled_ptr = true;
for (unsigned i = 0; i < ptr_size; i++) {
    TaintData td = shad_ptr->query_full(ptr + i);
    if (td.cb_mask != 0xff)
        fully_controlled_ptr = false;
}
[...]
if (fully_controlled_ptr)
    dest_td.cb_mask = 0xff, dest_td.tcn++;
else
    dest_td.cb_mask = byte_td.cb_mask;
```

Listing 5.2: Parts of the Dereferenced Pointer Patch

The Listing 5.2 shows the code created for the patch. The first part shows a loop that checks that every byte of the pointer is under full control. The second part overrides the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

control mask and increases the compute number in case that the pointer is under full control by an attacker.

This patch is not applicable in general. It does only make sense if the attacker can run arbitrary code on the victim machine, such that she knows about the location of user data in the memory. If the attack would be purely remote and without a prior information leak, then an attacker would have no knowledge about the current location of any data structure. We assume that this is the reason why PANDA behaves like it does.

5.7 Uniquification of Findings and System Call Skipping

We use the C++ class `std::unordered_set` to uniquify the findings. In case of register syscall findings, the label can be directly used as an `uint32_t` value. However, in the other cases, which also include memory addresses, as explained in Section 4.3, we use a pair of `uint32_t` values. The values are the label and the addresses, which form a unique finding when combined. To create a pair, a custom operator was added to the source code. The operator shifts the first part of a pair 32 bits to the left and then adds the second part by a bitwise OR.

In practice, especially when debugging a recorded finding, it can be very helpful to skip the first n syscalls and only then enable the tainting engine. The number of analyzed syscalls can be cut down, which reduces the findings complexity and shortens the time it takes for the complete analysis. This feature was implemented by counting to a defined number. When it is reached, the tainting engine is initialized and tainting is done. After this the number is no longer increased on every syscall because it is no longer needed.

5.8 Optional Debugging Outputs and Debugging Aid

In the final source code, some sections are intentionally commented out. These allow debugging of the source code and of findings. There is not a unique way to debug recordings, therefore we let all code we used in for future use. One example of such a code section is in the taint change callback. We used it for investigating a potential finding. The source code logs every taint change, which includes changes in registers, LLVM local values and physical memory.

It is also possible to adapt existing code for outputting kernel reads/writes from/to the user/kernel space. The plugin can print out extra information, when, for example, a write is performed to an examined address. This is especially useful when working with recordings because the addresses do not change during different replays of the same recording.

PANDA also provides a checkpointing plugin which creates memory checkpoints during a replay. With this it should be possible to reverse debug the execution of a recording. In the case of this thesis, this feature should be very helpful when looking for the source of a container side-channel. In contrast to this is the "normal" forward debugging, where

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the replay must be restarted every time the debugger should stop the execution before the current state is reached.

5.9 Reported Finding Information

Listing 5.3 shows the information when a finding is reported. This is a false positive finding of the type described in Section 6.3.

```
FOUND CHANNEL! MW RAM at 0xb6f890fe (0x0e025101) is tainted with
 other process. This is fuzz1 @0xc13faceb.
MASK: 0xff
COMPUTE NUMBER: 0
LABEL: 0x0006007d  which: 0  register: ESI  syscall:  125
DATA: 0x48 H
```

Listing 5.3: Reported information of a finding

The output shows that a new side-channel has been found and it also prints the type. In this case, the type is a memory write operation from the kernel space to the user space containing labeled data from the other fuzzing process. The operation is a four byte write starting at the virtual address `0xb6f890fe` and the byte of the physical destination address `0x0e025101` is tainted by *fuzz0*. The instruction pointer is currently at `0xc13faceb`, which is in the kernel space. If this finding would occur at the return of a syscall, instead of a memory write, the plugin would also print out the name of the current syscall.

Furthermore, the plugin outputs the control mask and the computing number, which means that the byte is under the full control of process *fuzz0* and it has not been subject to any computation. The label itself and the stored information are also printed out. Here, the label was created by *fuzz0* and the byte originates from the *ESI* register of the system call with the number 125. The operation wrote the byte `0x48`, which is the letter *H* in ASCII.

More examples of the output information can be found in Section 6.2.4. The demo replay discussed there includes all three detection types.

5.10 Automation of Runs

We made use of the QEMU snapshot feature, which allows us to save the current state of a running virtual machine. Later we can load the snapshot and the virtual machine continues to run from where the snapshot was taken. This made the startup process much faster because, for example, we could run the guest system without any instrumentation, stop it when both containerized syscall fuzzers finished their initialization and then take a snapshot. One system image can hold multiple snapshots and the commands can be

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

entered via the QEMU Monitor[4]. We use the snapshot feature in the automation script for a faster startup.

Furthermore, we created a *bash*[5] script, which automates certain runs. The script can be adapted for the current special goal, for example, if only some syscalls should be fuzzed by selecting a certain snapshot. We ended up with a script that can record a certain number of bytes, which is followed by analyzing the recording. This corresponds to the strategy described in Section 4.4.

The development of the script is straightforward. It consists of a loop where a recording is performed, which is then analyzed. During the second phase the recording emulator instance is stopped and the log output is checked for reported findings. If the analysis is finished the process starts all over again. During the execution, the script needs access to the Monitor of QEMU. It is possible to access the Monitor over the network. In this case, the option `-monitor tcp:0.0.0.0:5555,server,nowait;` was used to make the recording emulator Monitor accessible. The script accesses the Monitor by piping the commands to *nc*[81]. This is used for sending the commands for stopping and continuing the emulator, starting and ending a recording, loading a snapshot and making screenshots of the emulator.

Initially, the script would just stop and continue the recording emulator to create recordings. However, we observed that in some cases the syscall fuzzers stopped calling random syscalls. The exact reason is unknown but it was probably caused by not enough free memory available. Our system continued to recognize syscalls from the fuzzing processes. However, they were always from a small set of syscall. We concluded, that the fuzzer was not working properly. To mitigate this problem, we created a snapshot that starts the fuzzers after a delay and we checked that at every start a new seed is generated[6]. The first 120 seconds are skipped before a new recording, so that the syscall fuzzers have time for initialization. This should create a setup, where the guest system continuously creates new random syscall sequences without the possibility that the guest system gets stuck indefinitely in any way.

There are multiple checks to see if the analysis is still running or anything has been found. The script checks that the output log file does not become too large and that the analysis process still exists. It could not exist, for example, if the replay is finished or the execution crashed (which should not happen). After an analysis run finishes the loop, it starts from the beginning by resetting the recording emulator instance to the target snapshot to create a new recording. If the run has reported any found side-channels, then a new logging output file is used and the replay is saved under a different name. This makes it possible to run the process continuously and being able to examine findings at a later point.

[4] `https://en.wikibooks.org/wiki/QEMU/Monitor`

[5] `https://www.gnu.org/software/bash/`

[6] We set the system clock to the value of the hardware clock. This should influence the random number generator of Linux, although we did not find any sources for this claim.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Furthermore, the total number of analyzed syscalls is calculated by summing up the number of system call of each run. The script writes out this number after each analysis run. Periodically information about the free and used memory and the size of the logging output file is printed out.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Evaluation and Results

To evaluate the system, we created a test setup that runs idealized test scenarios. This makes sure that the system is working as intended and proves that we have built a working system that satisfies the aim of this thesis. Furthermore, we discuss false positive results and, if applicable, how we stopped the reporting of those. Finally, we present the results of the analysis runs and some additional facts about their execution.

6.1 Test Setup

All initial tests are run on server hardware equipped with a dual socket Intel[1] X5675 and 12 GiB of RAM. The tests are run with *Alpine*[2] Linux version 3.9 as a guest operating system. Alpine Linux was also used as the container image. The version of Alpine features the Linux kernel version 4.19.26-0. To make the Operating System Introspection work after reboots it is required to disable Kernel Address Space Layout Randomization at boot time. This can be done by using the kernel boot parameter *nokaslr*[3].

Alpine Linux was chosen as guest operating system because it has low system requirements, it is modern (in contrast to other small size distributions, like *Damn Small Linux*[4]) and secure[5]. The low system requirements, especially the required RAM, are necessary for the taint analysis because it requires at least 16-times the memory of the guest system. In the following tests, the guest operating system with Alpine Linux and two containers running syscall fuzzers could run on 256 MiB of RAM.

[1] https://www.intel.com

[2] https://alpinelinux.org

[3] https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/virtualization_security_guide/sect-virtualization_security_guide-guest_security-kaslr

[4] http://www.damnsmalllinux.org

[5] https://alpinelinux.org/about/

TU Bibliothek, WIEN, Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Alpine Package Keeper, the package manager of Alpine, also features a x86 (32-bit) compiled version of Docker[6]. Such binaries are not available in all Linux distributions, for example, the Docker CE setup for Ubuntu[7] requires a 64-bit operating system. The version of Docker used for the tests is 18.09.1-ce. The version of the syscall fuzzer *Trinity* used is 1.9.

There are configurations in container engines that could increase the security of containerized applications, like the usage of more restricted capabilities and seccomp profiles, or running under non-root user. However, to have the highest chance of finding such a side-channel and to be in the situation of a *default user*, we decided to perform the tests under default settings and without the usage of any LSM.

6.2 Idealized Test Runs

To verify that the developed plugin works as intended, we created idealized test scenarios, where the system should identify side-channels. However, these are not vulnerabilities because they require special permissions or a special setup to work. The tests were recorded using PANDA's record and replay functionality. This also eases debugging of the developed code because replays do not change between startups, which makes finding programming bugs much easier.

In every test, there are two programs. The first program, referred as *fuzz0* in the source code, executes a syscall that can be seen as a writing action (it sets some value) and the second program (*fuzz1*) reads the value directly or fetches data that contains the value. Performing tests with this construction shows that the implementation of the plugin works and that it is able identify the described side-channels.

6.2.1 Test Run setpriority and getpriority

The first test is designed to set the priority of the *init* process and then read the new value. The test is initially performed without containerization. The init process has always a PID of 1 [82] and the test is executed as user *root*.

The program *fuzz0*, which is listed in Listing 6.1, executes the system call *setpriority* [77] with arguments, such that it sets the priority (in this case also known as the *nice* value) to an arbitrary value. This syscall is used because it can set the *nice* value of another process, whereas the system call *nice* can only set the value of its own current thread [83]. In the test, we changed the *nice* value by 7, however, the exact value is not important. After *fuzz0* exits, *fuzz1* is started, which uses *getpriority* [77] to retrieve the value that was just set by *fuzz0*. Furthermore, it prints out the fetched value. The source code of *fuzz1* can be found in Listing 6.2.

```
int main(void) {
```

[6] https://pkgs.alpinelinux.org/package/edge/community/x86/docker
[7] https://docs.docker.com/install/linux/docker-ce/ubuntu/

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
    syscall(SYS_setpriority, PRIO_PROCESS, 1, 7);
    return 0;
}
```

Listing 6.1: Idealized priority test fuzz0 (include preprocessor directives omitted)

```
int main(void) {
    int n = syscall(SYS_getpriority, PRIO_PROCESS, 1);
    printf("%d\n", n);
    return 0;
}
```

Listing 6.2: Idealized priority test fuzz1 (include preprocessor directives omitted)

After recording the sequence of *fuzz0* and *fuzz1* we analyzed it with the developed plugin, which successfully identified the priority argument as a side-channel and logged it as a channel from *fuzz0* to *fuzz1*. The resulted output will be discussed in Section 6.2.4.

6.2.2 Test Run sethostname and uname

To test that the pointer dereferencing part and the copy in and copy out to and from the kernel space works we created a second test. It works by setting the hostname and reading it out. Again, this test is initially performed with the root user and without containerization.

fuzz0 calls the syscall *sethostname* [78] with an arbitrary string, which sets the hostname to the string. The source code of *fuzz0* is listed in Listing 6.3 and *fuzz1* can be found in Listing 6.4. Then the program *fuzz1* uses the syscall *uname* [79] (in this case the library function) to fetch the new hostname. This system call returns the information in a struct of type *struct utsname*, which contains additional information, like the operating system name and the operating system release. On Linux, the member *nodename* in a *struct utsname* is equal to the hostname set by *sethostname* [79].

```
int main(void) {
    syscall(SYS_sethostname, "demo1337", 8);
    return 0;
}
```

Listing 6.3: Idealized hostname test fuzz0 (include preprocessor directives omitted)

```
int main(void) {
    struct utsname s;
    uname(&s);
    printf("%s\n", s.nodename);
    return 0;
}
```

Listing 6.4: Idealized hostname test fuzz1 (include preprocessor directives omitted)

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The test successfully identified the data copied in by the kernel, which is then stored in kernel space and finally copied out from kernel space to user space. The output generated by our plugin can be found in Section 6.2.4.

6.2.3 Test Run in Docker Containers

An additional test was done by running the tests of Section 6.2.1 and 6.2.2 in two containers. We placed the *fuzz0* programs in one container and the *fuzz1* programs in the other one. The aim of this test is to show that the unaltered plugin still works if the two related processes are in two different containers. The design of the test requires access to a common process. This can be done be setting the PID namespace of the containers to the host PID namespace by using the command line option `--pid=host`[8]. Moreover, to access the hostname of the host the UTS namespace is set to the host's namespace by using the option `--uts=host`[9]. Furthermore, we started the containers in privileged mode[10] to make sure that they can set and read the priority of the init process and alter the hostname.

The test yielded the same result as the test without containerization except for the startup time in one particular setup. If the programs are started using the Docker command *exec*[11] then the time till *fuzz0* and *fuzz1* are started, is considerable increased. However, if the programs are started interactively by a shell running inside the container, then there is no significant difference in the time it takes to analyze a recording. The reason of the difference is therefore the time the *exec* command itself takes, although, it is not analyzed like the processes *fuzz0* and *fuzz1*.

6.2.4 Demo Replay

For demonstrational purposes a replay was recorded, which combines both tests described in Section 6.2.3 in one analysis run. The file can also be found in the project's repository[12], so that it can be used for showing that the developed system works and as an introduction if someone wants to do further research. The demo uses a Docker image, which contains a working PANDA system including our plugin. To build the image we provided the corresponding Dockerfile. The result of running the demo with an instance of the Docker image is shown in Listing 6.5 and 6.6. The Listings are truncated and split up for better readability, normally they would be in a single output file with additional information.

```
syscall  name: fuzz0  no: 97  which: 0    EAX: 0x00000061
EBX: 0x00000000  ECX: 0x00000001  EDX: 0x00000007
ESI: 0xbffffe00  EDI: 0xb7ffcf7c  EBP: 0xbffffd7c
```

[8] `https://docs.docker.com/engine/reference/run/#pid-settings---pid`

[9] `https://docs.docker.com/engine/reference/run/#uts-settings---uts`

[10] `https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities`

[11] `https://docs.docker.com/engine/reference/commandline/exec/`

[12] `https://gitlab.sba-research.org/theses/container-side-channel-identification`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
[...]
syscall  name: fuzz0  no: 74  which: 0   EAX: 0x0000004a
EBX: 0x00402000  ECX: 0x00000008  EDX: 0x00401224
ESI: 0xbffffdf0  EDI: 0xb7ffcf7c  EBP: 0xbffffd6c
[...]
before user read, add label: 0x0040004a at 0x00402000
(pa: 0x0bb2f000) size 1
read user 0x00402000  which: 0  data: 0x64
before user read, add label: 0x0041004a at 0x00402001
(pa: 0x0bb2f001) size 1
read user 0x00402001  which: 0  data: 0x65
before user read, add label: 0x0042004a at 0x00402002
(pa: 0x0bb2f002) size 1
read user 0x00402002  which: 0  data: 0x6d
before user read, add label: 0x0043004a at 0x00402003
(pa: 0x0bb2f003) size 1
read user 0x00402003  which: 0  data: 0x6f
before user read, add label: 0x0044004a at 0x00402004
(pa: 0x0bb2f004) size 4
before user read, add label: 0x0044004a at 0x00402004
(pa: 0x0bb2f005) size 4
before user read, add label: 0x0044004a at 0x00402004
(pa: 0x0bb2f006) size 4
before user read, add label: 0x0044004a at 0x00402004
(pa: 0x0bb2f007) size 4
read user 0x00402004  which: 0  data: 0x31333337
```

Listing 6.5: Output of demo replay (only the labeling is listed)

In the demo both *fuzz0* processes run after another and then the *fuzz1* processes are started. Listing 6.5 shows the labeling of the priority value and the new hostname. The first entry is the *setpriority* syscall from the process *fuzz0*, which has the number 97[13]. The *ECX* register corresponds to the PID of the init process, which is 1, and *EDX* is the relative value of the new process priority (*nice* value) [77, 83].

The second syscall is *sethostname*, which is also executed by *fuzz0* and sets the hostname to the value stored at the value of *EBX*. In this case, the register acts as a pointer to the new hostname string. Furthermore, you can see that the string is then copied in by the kernel, so that it is stored in the kernel memory space. The copy routine first separately copies the first 4 bytes of the string and then copies the last 4 bytes at once. For each new read operation, a new label is created and assigned to the data. As you may saw from the data of the string, the new hostname is "demo1337".

[13] As a source of i386 system calls, `https://github.com/torvalds/linux/blob/master/arch/x86/entry/syscalls/syscall_32.tbl` and `https://syscalls32.paolostivanin.com` can be used.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
FOUND CHANNEL! Register 0 is tainted with other process.
  This is fuzz1 sys_getpriority @0xb7f6fc0c.
MASK: 0xffffffffffffffff
COMPUTE NUMBER: 2
LABEL: 0x00020061  which: 0  register: EDX  syscall:  97
DATA: 0x0000000d

FOUND CHANNEL! MR RAM at 0xbffffd90 (0x05ad1d90) is tainted
  with other process. This is fuzz1 @0xb7f8df3d.
MASK: 0xff
COMPUTE NUMBER: 2
LABEL: 0x00020061  which: 0  register: EDX  syscall:  97
DATA: 0x0d
 [...]
FOUND CHANNEL! MW RAM at 0xbffffc84 (0x096f6c87) is tainted
  with other process. This is fuzz1 @0xc13fad28.
MASK: 0xff
COMPUTE NUMBER: 1
LABEL: 0x0003004a  which: 0  register: EBX  syscall:  74
DATA: 0x64 d

FOUND CHANNEL! MW RAM at 0xbffffc84 (0x096f6c87) is tainted
  with other process. This is fuzz1 @0xc13fad28.
MASK: 0xff
COMPUTE NUMBER: 1
LABEL: 0x0040004a  which: 0  register: ???  syscall:  74
DATA: 0x64 d

FOUND CHANNEL! MW RAM at 0xbffffc88 (0x096f6c88) is tainted
  with other process. This is fuzz1 @0xc13fad28.
MASK: 0xff
COMPUTE NUMBER: 0
LABEL: 0x0041004a  which: 0  register: ???  syscall:  74
DATA: 0x65 e
 [...]
```

Listing 6.6: Output of demo replay (only the identification is listed)

In Listing 6.6 the identification output of the side-channels is listed. At first there are two entries of the priority channel. The first is at the end of the syscall *getpriority* and has a return value that is tainted with a label from *fuzz0*. As you can see, all bits are controlled by the other process and the value has been involved in a computing operation. Encoded in the label, we stored information about its source, in this case, which process, the register and the syscall number. Compared to the output of Listing 6.5 the register

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

and the syscall number match with information previously printed out. The new priority value is now 0x0d because the initial value was 20 and $20 - 7 = 13$, which is 0x0d in hexadecimal. There is more than one output for the value because it is used in user space more than once, for example, it is printed out in the program. The listing shows that, after the syscall returns, the value is only read in user space. This is indicated by *MR*, which means *memory read*.

The lower half is the output for the hostname side-channel. The output is like the finding described before, however, in this case, the kernel copies the hostname string to the user space. This is indicated by *MW*, which stand for *memory write*. Again, the bytes are under full control by an attacker and the compute number of 1 means that, in this case, the initial value was used as a pointer, which increases the compute number by one. This is part of the pointer patch discussed in Section 5.6. With PANDA one address corresponds to one compute number, thus all labels of one address share one compute number. There are two findings for the first letter because it has two different labels, one from the register tainting at the beginning of the syscall and one when the data is copied in. The second case is indicated by the "???" in the register output because this field, the number of the syscall argument register, is only available during a syscall enter. The last finding of Listing 6.6 shows the second letter of the hostname. The rest of the string is omitted for readability, but can be found in the real output of the demo.

The reason the virtual address of the hostname string is not consist (in contrast to the physical address) is that the detection is performed in the taint change callback, which only gives developers the physical address. However, as far as we know there is no efficient way to get a virtual address from a physical one, therefore we use the last address that was used by the write callback. We understand this value as a guess. In practice, this is not a real problem because we can determine the correct virtual address from the other findings and from the repeatability of the replay.

The demo shows both cases when memory will be marked, as a syscall argument and when it is copied to kernel space, and all three cases when it leaves the kernel or is processed in user space by the other process. Therefore, the demo shows that our system works as intended and can detect the described side-channels.

6.3 Early Fuzzing Attempts and False Positives

After the verification that the plugin is capable to find cross-container side-channels we continued with the fuzzing runs. These runs are intended to find the channels instead of developing and testing the software. We switched the hardware to a dedicated server with 64 GiB of RAM and an Intel i7-7700 CPU. We used the automation strategy implemented as discussed in Section 5.10. Initially, we used a recording size of 500 MiB, however, in some cases the server did run out of system memory and used the 32 GiB swap memory, which decreased the performance significantly. The automation script also stopped twice because there was not enough free memory left. Therefore, we decided to reduce the recording size to 100 MiB and to increase the swap to 32 GiB + 100 GiB. Furthermore,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

we periodically checked that the swap memory is not used extensively. If this should be the case, the analysis process is stopped and the loops starts from the beginning. This configuration turned out to be much more stable, as it did not stop running for six weeks.

During the development and early usage of the plugin we came across interesting false positive results. These are results that are reported as a successful side-channel but they are not. If applicable, we improved the implementation of the developed plugin such that false positives of the corresponding kind are not reported any more.

Write Operations that Use Two Pages

We produced a replay file that shows a wrong behavior during the after write callback and the taint change callback. The first callback provides the address and a buffer with the written data and the second gives information whenever a label of an address changes. The problem exists when a write operation affecting multiple bytes is performed and the destination memory resides on two memory pages. Only the first bytes are tainted correctly and instead of the latter bytes, the bytes following in the physical memory space are tainted.

Up to now, we have not figured out the origin of this behavior in the PANDA source code. It is probable a bug in the taint plugin of PANDA. This behavior resulted in a false positive because the taint engine of PANDA changed the label of all four bytes although only two are written successfully. The last two bytes are presumably mapped on another page in the other fuzzing process, which cause the developed plugin to report a finding when these bytes are read. This is a false positive because the data was in fact not written, so no information is exchanged between the two processes.

```
 Output buf of callback:
VMAWC buf phy: 0x0a3fbffe (vir: 0x04b83ffe): 0x30
VMAWC buf phy: 0x0a3fbfff (vir: 0x04b83fff): 0x78
VMAWC buf phy: 0x0bd17000 (vir: 0x04b84000): 0x30  <- Correct
                     "jump" of physical address (another page)
VMAWC buf phy: 0x0bd17001 (vir: 0x04b84001): 0x2c
 Output buf2, consecutive physical memory starting with
 0xa3fbffe:
VMAWC buf2 phy: 0x0a3fbffe: 0x30
VMAWC buf2 phy: 0x0a3fbfff: 0x78
VMAWC buf2 phy: 0x0a3fc000: 0x08  <- Consecutive bytes contain
VMAWC buf2 phy: 0x0a3fc001: 0x00                0x08 and 0x00
 taint_change phy: 0x0a3fbffe  size: 1
Mem: 0x30
 Labels: 0x00000666
 taint_change phy: 0x0a3fbfff  size: 1
Mem: 0x78
 Labels: 0x00000666
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

```
taint_change phy: 0x0a3fc000  size: 1 <- Should be 0x0bd17000
Mem: 0x08                                              and 0x30
Labels: 0x00000666
taint_change phy: 0x0a3fc001  size: 1 <- Should be 0x0bd17001
Mem: 0x00                                              and 0x2c
Labels: 0x00000666
```

Listing 6.7: Output of the test plugin including annotations

To further examine the issue, we created a test plugin that just works with the aforementioned recording. Listing 6.7 shows the output of the plugin. It consists of three parts. The first part shows the buffer that the after write callback retrieves with the corresponding physical address. Please note the first annotation. It marks the beginning of a new memory page, which is shown in the change of the physical address. The middle part shows the bytes following the physical address of the initial address translation. As you can see the memory content of these locations is `0x08` and `0x00`. The final part consists of the taint change callback. It shows that the wrong memory location is labeled. Instead of using the next page, the initial translation to the physical address space is taken and the required offset is added. However, this is wrong because in this case the latter bytes of the write operation would be written to another page.

We propose that this issue could be fixed by checking the taint operation after every write and correcting it. However, this would significantly decrease the performance of the analysis because for every byte written our plugin would perform the translation to the physical address again and compare it to the information of the taint change callback. The false behavior only occurred infrequent and is easy to spot, so we decided that it is not worth it to implement such a fix. We submitted an issue report to the repository of PANDA[14]. The report includes the test plugin which demonstrates the problem. The issue is still marked as *open* and there is currently[15] no fix available in the master or in an alternative branch.

Labeling of Read-Only Data

In an early version of the plugin, the source of a user space read operation was also tainted. We did this because the read callbacks only provide the source memory address and not the destination register. Therefore, we labeled the source before the read operation was executed such that the tainting plugin would take the labels from the source and copies them to the destination register. However, this leads to a false positive. We observed a finding where a file is opened in memory by both fuzzing processes at their startup phase. The data that is copied in during the opening (the file path and one or two bytes, presumably, for alignment) is tainted. We observed that the page with the constant strings is shared between the containers because they share the same physical address. When the first fuzzing process marks copied in addresses and the second one uses them

[14] `https://github.com/panda-re/panda/issues/441`
[15] As of August 11, 2019

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

too, then the result will be reported as a finding. However, this is a false positive finding because no information was exchanged.

We wrote a small program that changes its own data and program code as writeable. The program was then executed in two different containers. If no data is written by any instance the memory pages can be shared between the program. Once a write access is performed by one of the programs, the kernel makes a copy of the page for each process. Thus, it is not possible to exchange data between the two processes.

Furthermore, in cases, where the program already existed in the container image, the data is not written back to the file in the image. Each container has its own copy of the program. We tried writing to a memory address before the program's write with an attached debugger and the changes were written to the image file. Please note, that this was done outside the guest machine. This is not possible during normal (non-emulated) operation. This behavior shows that if no write is performed, containerized programs can share pages of the container image but changes create their own copy of the mapped file.

We partially fixed the reporting of this false positive by removing the applied label after the read operation, while leaving all other labels intact. This taints only the destination register with the new label and not the source memory address. For details on the implementation of this behavior, please see Section 5.4. During the final use of the system we discovered that in rare cases the label is not removed successfully. However, it would be quite easy to distinguish between real findings and these false positives because, for example, the examined syscall does not perform any write operation.

6.4 Results

The described procedure of Section 5.10 was running for 1030 hours, or about 42 days. The snapshot, which executed two fuzzing processes in two Docker containers, was configured to fuzz 316 distinct syscalls. During this run 193,982,020 emulated syscalls were executed and analyzed, although, the number of real *fuzzed* syscalls is significantly lower. We observed that many calls have the same syscall number, so we filtered out these very common syscalls and were left with only 9,932,990 syscalls. This indicates that Trinity has a rather large overhead in our setup. From these numbers, we calculate an average of 52.3 overall syscalls analyzed per second and 2.68 per second random syscalls with fuzzing data.

These syscalls were distributed in 100 MiB recordings. The average time to create such a recording was 6.25 minutes on the fuzzing server. During the 42 days 162 individual runs were created and subsequently analyzed. On average, the system needed 5.93 hours to completely analyze a recording. However, this varies significantly, with a sample standard deviation of $s = 3.98$ hours. This is caused by fact that the taint analysis is focused on the instructions in kernel mode. When a recording consists mainly of user program instructions the time to analyze it will be significantly shorter.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

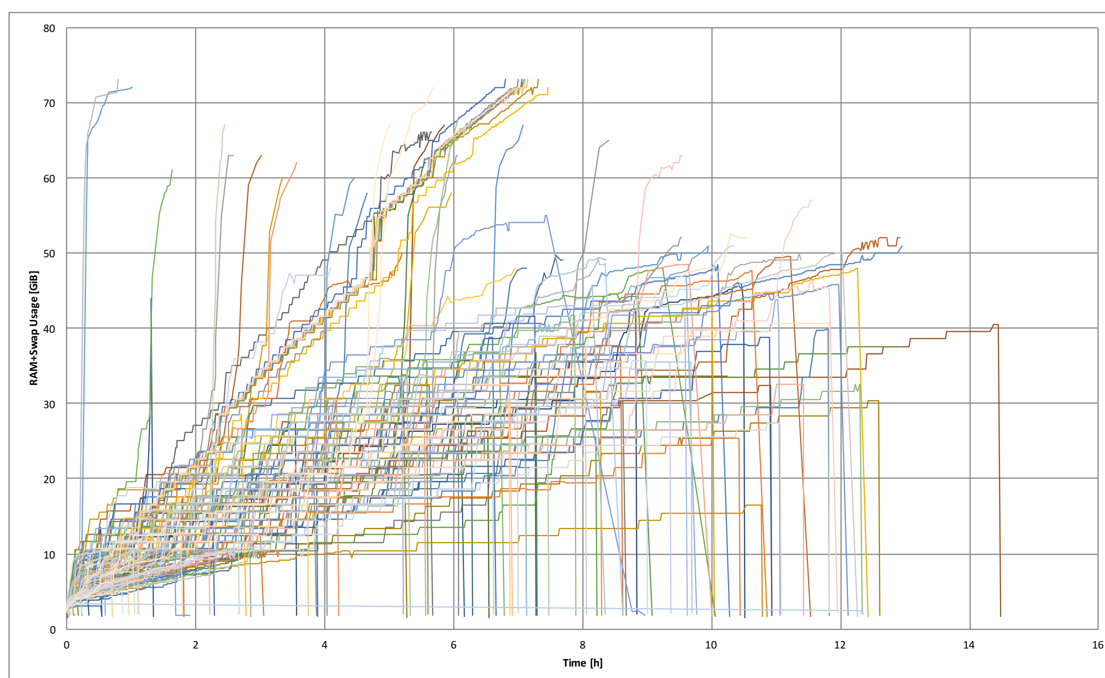


Figure 6.1: System RAM and swap usage over time.

The Figure 6.1 shows the memory consumption, RAM and swap usage, of all analysis runs. The maximum duration an analysis run needed was 14.48 hours and one recording did need the full system RAM in about 27 minutes. Furthermore, the figure shows that some runs reached the maximum memory usage limit and that at the end of a run the memory is freed before the process exits. The memory measurements were performed with a resolution of 30 seconds.

During the 6 weeks, the recording halted before reaching 100 MiB three times. We propose that a potential cause is the syscall fuzzer. It may have crashed the kernel in these cases, which halts the emulated machine and stops any further recording. To mitigate this problem, a timeout could be added to the recording stage that could continue the script although the recording has not reached the 100 MiB limit.

The implemented automation strategy stores the replay files, the log outputs of the plugin runs and a log file for the automation script. The plugin output logs contain the syscalls called, the data copied from user space into kernel space and the reported findings. We used their file size to calculate an average of 10.03 GiB per analyzed recording with a $s = 7.94$ GiB.

Overall we could not identify any issues that have a real impact on Docker container isolation.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Discussion and Future Work

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

We demonstrated that the developed system is capable of detecting side-channels over the kernel. This is shown in Section 6.2, where we use idealized versions of container setups. In these cases, we expected to see a reported finding. The idealized versions extend the containerized processes' default permissions, such that the test programs are allowed to change the state of the host system, which also influences the behavior of the other container under test. Thus, we can be sure that the system is indeed capable of reporting real-world information leakage between containers.

The false positives, which occurred during analysis runs, further increased the confidence by showing that the developed plugin does work but that in these cases the findings are not real vulnerabilities. This is especially true for the labeling of read-only data (Section 6.3), where the system did correctly detect access to a shared page, which cannot be influenced by an attacker.

The results of the automated runs indicate that the current isolation of a default Docker container is adequate. We propose that there are no "obvious" side-channels over the kernel because there are currently only two objects that are not namespaced, as discussed in Section 3.2.1, and these objects cannot be changed from within a default container. Therefore, we currently do not know of any side-channels of the type described in this thesis when the default configuration is used. However, in cases where additional permissions or namespaces are shared, the isolation quickly breaks down. We showed this with the setup of our idealized tests.

Due to time and resource constraints we decided to focus on a single container engine and system call fuzzer combination. Future work could improve this by running other combinations on multiple machines to increase the likelihood of finding side-channels. However, this would require increased expenses.

To improve the chance of finding a side-channel we could replace the random syscall fuzzer with a program that adds *human* knowledge, which tests, for example, only certain

syscalls with defined parameters. This reduces the search space and would therefore increase the likelihood that in a given time span a syscall combination occurs that results in a reported finding. However, it could also restrict the system in such a way that the more obscure combinations would never be found. The current implementation works unopinionated, which means that each syscall is equally likely to be called.

During the beginning of the fuzzing we had to stop and restart the automated process multiple times and adjust certain configurations, like the size of the recordings and the maximum memory an analysis run can to use. For the demonstrational purposes of this thesis, the configuration we chose is conservative to focus on stability instead of performance. In the future, these numbers could be improved to create a better performing system, which means in context of this thesis that overall more system calls are analyzed per time period.

Although the setup of the analysis runs did only include Docker as the container engine, many security properties are shared between different software products. This means that the results are also meaningful for all the software that uses the same configuration for process isolation. We examined that the default seccomp profile and the default list of capabilities of Docker can be found in other container engines. These findings are discussed in Section 3.1.3.

This work also acts as a basis for detecting information exchange in other implementations of process isolation. For example, to test the interaction of different processes when constrained by a Mandatory Access Control module (LSM), seccomp in filter mode or file permissions. To use the plugin just the program for trying to interact with another process must be exchanged in our current setup. Such a program could then focus on operations that, for example, test file access or inter-process communication.

Further improvements for the comprehensibility of found side-channels would include the increase in information stored in one label. For this, the PANDA source code could be changed in such a way that it supports more than the default 32 bits in one label. However, this would also increase the amount of memory needed to do a given analysis run. Alternatively, multiple labels can be used to store more information by using certain bits to indicate which type of label it represents. A two-stage approach would be possible too, where in the first stage the detection is performed with default sized labels and later parts of the recording are examined further with more information stored per label.

We demonstrated our system by creating idealized tests. We came up with our own idea on how to test the information leakage between containers by extending the default permissions and making changes of the default namespace configuration. As far as we know, there exists no extensive list of configuration options of Docker (or other container engines) and the potential vulnerabilities they imply. For example, if a user sets the PID namespace of containers to the host's namespace, then the containers could communicate unconfined by creating processes with certain names. The process name would then contain the data a container wants to share with others. Therefore, we propose further research of container configuration and the security impact they may entail.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

We also propose that there could be further improvements in how the taint analysis works in PANDA, although the current version of PANDA's taint2 plugin is about ten times faster than the first version [74]. Currently, the QEMU's TCG and the helper functions are translated to LLVM's intermediate representation. Hypothetically it should be possible to combine these steps to one intermediate representation, which would then be executed faster because the intermediate steps would be omitted. However, this would require a complete rewrite of the current implementation of QEMU and PANDA, which probably takes a large amount of time and effort.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

CHAPTER 8

Conclusion

We defined a potential vulnerability in the context of isolation of containers and presented a novel approach to automatically identify such side-channels. Due to the construction of Linux containers, our system tests the complete stack when running containers. This includes the setup profiles, with configuration about capabilities and the seccomp filter, and the operating system kernel itself. We showed that our system works, although the comprehensibility could be improved in the future. We evaluated the system by creating idealized tests that use extended (non-default) permissions and our software successfully identifies the test side-channels.

Furthermore, we designed and implemented a strategy to execute our system automatically. This fuzzing setup was then executed for six weeks, which showed that the system runs stable. It is therefore capable to run for any arbitrary timespan and will continuously report findings if any occur. We minimized the encountered false positives by developing countermeasures or reporting them to the corresponding project.

We presented the results of this fuzzing run and calculate certain metrics from the created log data. Overall, during the 42 days 9,932,990 random fuzzing syscalls were executed and analyzed. This results in 2.68 analyzed syscalls per second. Therefore, we propose that there is the need to execute our system on a larger scale and possibly with other types of software, like application sandboxes. We also show that other fields exist, which could benefit from our software. Furthermore, we argue that our results are not unique to Docker because other container software, like rkt and Apache Mesos, use the same default security profiles.

Although we did not discover vulnerabilities in the default configuration of Docker, this thesis shows how important it is to the reduce the number of added capabilities to the bare minimum, which is required for the executed task. Excessive capabilities or running in privileged mode can lead to an increase of damage once an attacker has gained control of a container. We showed with the idealized tests and in the demo that side-channels

over the kernel could easily occur when a container is started with certain extended permissions. If the isolation is a strong requirement for production application and the feature set of container engines is required, then Kata containers could be a viable solution. Users can deploy each container in a different virtual machine with the interface of container engines. Therefore, Kata containers combine the benefits of containers, the easier setup and administration, and virtual machines, the better isolation between instances.

Overall, the current situation in the context of isolation is not bad for users of Docker and other container engines on Linux. Many vulnerabilities found in the last year are problems related to the setup or interaction with containers, which is not this work's aim, where we solely focus on the case of already running containers. Information leaks and container breakouts have been found in container engines. Often such vulnerabilities are not unique to one engine, therefore a security bugfix in a shared component improves the security of all affected software products. Containers are an adequate solution when it comes to application isolation, especially in cases where different services used to be deployed directly on a single host.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Figures

3.1 Differences between a hypervisor-based deployment and a container-based deployment. Figures taken from [39]. 14
3.2 Overview of the Docker architecture. Figure taken from [44]. 15
3.3 Overview of the PANDA instruction translation. Figure taken from [33]. . 25

5.1 Overview of the analysis system. 34

6.1 System RAM and swap usage over time. 53

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

3.1 Capability Algorithm for Threads 23

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Listings

2.1 Trinity fuzzer code for the syscall rt_sigaction 10
5.1 Fetch information on which fuzzing process is currently running 35
5.2 Parts of the Dereferenced Pointer Patch 37
5.3 Reported information of a finding 39
6.1 Idealized priority test fuzz0 (include preprocessor directives omitted) . 44
6.2 Idealized priority test fuzz1 (include preprocessor directives omitted) . 45
6.3 Idealized hostname test fuzz0 (include preprocessor directives omitted) 45
6.4 Idealized hostname test fuzz1 (include preprocessor directives omitted) 45
6.5 Output of demo replay (only the labeling is listed) 46
6.6 Output of demo replay (only the identification is listed) 48
6.7 Output of the test plugin including annotations 50

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abbreviations

API Application Programming Interface. 15, 16

ASCII American Standard Code for Information Interchange. 35, 39

BPF Berkeley Packet Filter. 23, 24

CLI Command-line Interface. 15, 16

CPU Central Processing Unit. 2, 6–8, 26, 30, 35–37, 49

DoS Denial of Service. 3

DRAM Dynamic Random Access Memory. 7

GID Group ID. 22

IaaS Infrastructure as a Service. 3

IP Internet Protocol. 17

IPC Interprocess Communication. 17

KASLR Kernel Address Space Layout Randomization. 43

LLVM Low Level Virtual Machine. 9, 25–27, 38, 57

LSM Linux Security Module. 2, 24, 44, 56

MAC Mandatory Access Control. 24, 56

NIS Network Information System. 18

OCI Open Container Initiative. 15, 16

OSI Operating System Introspection. 9, 43

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

PaaS Platform as a Service. 6

PANDA Platform for Architecture-Neutral Dynamic Analysis. 9, 10, 24–27, 29, 31, 33–38, 44, 46, 49–51, 56, 57, 61

PID Process ID Number. 5, 18, 26, 44, 46, 47, 56

PIRATE Platform for Intermediate Representation-based Analyses of Tainted Execution. 9

POSIX Portable Operating System Interface. 17

PPID Parent Process ID Number. 26

RAM Random-Access Memory. 26, 31, 43, 49, 53

RSA Ron Rivest, Adi Shamir and Leonard Adleman (Cryptosystem). 7

S2E Selective Symbolic Execution. 25

SGID Set Group ID. 22

SSH Secure Shell. 7

SUID Set User ID. 21–23

TCG Tiny Code Generator. 25, 57

UID User ID. 22

UTS Unix Time Sharing. 18, 46

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[1] "Containers as the foundation for DevOps collaboration." https://docs.microsoft.com/en-us/dotnet/standard/containerized-lifecycle-architecture/docker-application-lifecycle/containers-foundation-for-devops-collaboration, 2019. [Online; accessed 14-April-2019].

[2] M. Ferranti, "2017 Annual Container Adoption Survey: Huge Growth in Containers." `https://portworx.com/2017-container-adoption-survey/`, 2017. [Online; accessed 19-April-2019].

[3] K. Buckley, "Featured Data: Application container market revenue expected to quadruple by 2021." `https://451research.com/blog/1657-featured-insight`, 2017. [Online; accessed 10-May-2019].

[4] C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[5] Y. Zhang, A. Juels, A. Oprea, and M. K. Reiter, "Homealone: Co-residency detection in the cloud via side-channel analysis," in *2011 IEEE symposium on security and privacy*, pp. 313–328, IEEE, 2011.

[6] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-tenant side-channel attacks in paas clouds," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pp. 990–1003, ACM, 2014.

[7] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[8] P. Kocher, J. Horn, A. Fogh, , D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[9] O. Weisse, J. Van Bulck, M. Minkin, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, R. Strackx, T. F. Wenisch, and Y. Yarom, "Foreshadow-NG: Breaking the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

virtual memory abstraction with transient out-of-order execution," *Technical report*, 2018. See also USENIX Security paper Foreshadow [24].

[10] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss, "ZombieLoad: Cross-privilege-boundary data sampling," *arXiv:1905.05726*, 2019.

[11] J. Hertz, "Abusing privileged and unprivileged linux containers," *Whitepaper, NCC Group*, vol. 48, 2016.

[12] "CAPABILITIES(7)." http://man7.org/linux/man-pages/man7/capabilities.7.html, 2019. [Online; accessed 30-March-2019].

[13] "OPEN_BY_HANDLE_AT(2)." http://man7.org/linux/man-pages/man2/open_by_handle_at.2.html, 2019. [Online; accessed 27-June-2019].

[14] J. Andre, "Docker breakout exploit analysis." https://medium.com/@fun_cuddles/docker-breakout-exploit-analysis-a274fff0e6b3, 2014. [Online; accessed 27-June-2019].

[15] Y. Yarom and K. Falkner, "Flush+ reload: a high resolution, low noise, l3 cache side-channel attack," in *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pp. 719–732, 2014.

[16] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' track at the RSA conference*, pp. 1–20, Springer, 2006.

[17] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-level cache side-channel attacks are practical," in *2015 IEEE Symposium on Security and Privacy*, pp. 605–622, IEEE, 2015.

[18] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, pp. 312–320, ACM, 2007.

[19] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*, pp. 225–242, Springer, 2007.

[20] O. Mutlu and J. S. Kim, "Rowhammer: A retrospective," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[21] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping bits in memory without accessing them: An experimental study of dram disturbance errors," in *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, (Piscataway, NJ, USA), pp. 361–372, IEEE Press, 2014.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[22] D. Gruss, C. Maurice, and S. Mangard, "Rowhammer. js: A remote software-induced fault attack in javascript," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 300–321, Springer, 2016.

[23] A. Kwong, D. Genkin, D. Gruss, and Y. Yarom, "Rambleed: Reading bits in memory without accessing them," in *41st IEEE Symposium on Security and Privacy (S&P)*, 2020.

[24] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. Strackx, "Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution," in *Proceedings of the 27th USENIX Security Symposium*, USENIX Association, August 2018. See also technical report Foreshadow-NG [9].

[25] M. Jurczyk, "Detecting kernel memory disclosure with x86 emulation and taint tracking," 2018.

[26] F. Wilhelm, "Tracing privileged memory accesses to discover software vulnerabilities," master thesis, Operating Systems Group, Karlsruhe Institute of Technology (KIT), Germany, Nov.30 2015.

[27] B. Gowda, "A peek into Extended Page Tables." `https://itpeernetwork.intel.com/a-peek-into-extended-page-tables/`, 2009. [Online; accessed 15-April-2019].

[28] A. Lanzi, M. I. Sharif, W. Lee, *et al.*, "K-tracer: A system for extracting kernel malware behavior.," in *NDSS*, pp. 255–264, 2009.

[29] H. Yin and D. Song, "Whole-system fine-grained taint analysis for automatic malware detection and analysis," *Technical paper. College of William and Mary & Carnegie Mellon University*, 2006.

[30] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*, pp. 116–127, ACM, 2007.

[31] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security Symposium*, pp. 321–336, 2004.

[32] R. Whelan, T. Leek, and D. Kaeli, "Architecture-independent dynamic information flow tracking," in *International Conference on Compiler Construction*, pp. 144–163, Springer, 2013.

[33] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable reverse engineering with panda," in *Proceedings of the 5th Program Protection and Reverse Engineering Workshop*, p. 4, ACM, 2015.

[34] "Fuzzing." https://www.owasp.org/index.php/Fuzzing, 2018. [Online; accessed 14-April-2019].

[35] "Trinity: Linux system call fuzzer.." https://github.com/kernelslacker/trinity/blob/master/README, 2017. [Online; accessed 14-April-2019].

[36] B. Garn and D. E. Simos, "Eris: A tool for combinatorial testing of the linux system call interface," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation Workshops*, pp. 58–67, IEEE, 2014.

[37] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, and T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels," in *USENIX Security Symposium*, 2017.

[38] "What is a Container?." https://www.docker.com/resources/what-container. [Online; accessed 8-April-2019].

[39] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.

[40] M. Eder, "Hypervisor-vs. container-based virtualization," *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, vol. 1, 2016.

[41] J. Cito and H. C. Gall, "Using docker containers to improve reproducibility in software engineering research," in *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pp. 906–907, May 2016.

[42] S. J. Vaughan-Nichols, "What is Docker and why is it so darn popular?." https://www.zdnet.com/article/what-is-docker-and-why-is-it-so-darn-popular/, 2018. [Online; accessed 8-April-2019].

[43] "8 SURPRISING FACTS ABOUT REAL DOCKER ADOPTION." https://www.datadoghq.com/docker-adoption/, 2018. [Online; accessed 8-April-2019].

[44] M. Crosby, "WHAT IS CONTAINERD ?." https://blog.docker.com/2017/08/what-is-containerd-runtime/, 2017. [Online; accessed 8-April-2019].

[45] "rkt vs other projects." https://github.com/rkt/rkt/blob/master/Documentation/rkt-vs-other-projects.md, 2017. [Online; accessed 15-June-2019].

[46] "Seccomp Isolators Guide." https://github.com/rkt/rkt/blob/a2f63178ee7749a4e90a2882744774dbd03bb425/Documentation/seccomp-guide.md, 2019. [Online; accessed 15-June-2019].

[47] "App Container Executor." https://github.com/appc/spec/blob/master/spec/ace.md, 2017. [Online; accessed 15-June-2019].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[48] "Linux Seccomp Support in Mesos Containerizer." http://mesos.apache.org/documentation/latest/isolators/linux-seccomp/. [Online; accessed 15-June-2019].

[49] "The current Flatpak sandbox." https://github.com/flatpak/flatpak/wiki/Sandbox#the-current-flatpak-sandbox, 2018. [Online; accessed 16-June-2019].

[50] "NAMESPACES(7)." http://man7.org/linux/man-pages/man7/namespaces.7.html, 2019. [Online; accessed 29-March-2019].

[51] "CGROUP_NAMESPACES(7)." http://man7.org/linux/man-pages/man7/cgroup_namespaces.7.html, 2019. [Online; accessed 30-March-2019].

[52] "NETWORK_NAMESPACES(7)." http://man7.org/linux/man-pages/man7/network_namespaces.7.html, 2018. [Online; accessed 29-March-2019].

[53] "MOUNT_NAMESPACES(7)." http://man7.org/linux/man-pages/man7/mount_namespaces.7.html, 2018. [Online; accessed 29-March-2019].

[54] "PID_NAMESPACES(7)." http://man7.org/linux/man-pages/man7/pid_namespaces.7.html, 2019. [Online; accessed 29-March-2019].

[55] "USER_NAMESPACES(7)." http://man7.org/linux/man-pages/man7/user_namespaces.7.html, 2019. [Online; accessed 29-March-2019].

[56] "CLONE(2)." http://man7.org/linux/man-pages/man2/clone.2.html, 2019. [Online; accessed 30-March-2019].

[57] "SETNS(2)." http://man7.org/linux/man-pages/man2/setns.2.html, 2019. [Online; accessed 30-March-2019].

[58] "UNSHARE(2)." http://man7.org/linux/man-pages/man2/unshare.2.html, 2019. [Online; accessed 30-March-2019].

[59] "IOCTL_NS(2)." http://man7.org/linux/man-pages/man2/ioctl_ns.2.html, 2019. [Online; accessed 30-March-2019].

[60] J. Frazelle, "Two Objects not Namespaced by the Linux Kernel." https://blog.jessfraz.com/post/two-objects-not-namespaced-linux-kernel/, 2017. [Online; accessed 22-June-2019].

[61] "CGROUPS(7)." http://man7.org/linux/man-pages/man7/cgroups.7.html, 2019. [Online; accessed 11-April-2019].

[62] "Docker security." https://docs.docker.com/engine/security/security/. [Online; accessed 1-April-2019].

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[63] "CAPGET(2)." http://man7.org/linux/man-pages/man2/capset.2.html, 2019. [Online; accessed 05-April-2019].

[64] "PRCTL(2)." http://man7.org/linux/man-pages/man2/prctl.2.html, 2019. [Online; accessed 05-April-2019].

[65] "SECCOMP(2)." http://man7.org/linux/man-pages/man2/seccomp.2.html, 2019. [Online; accessed 07-April-2019].

[66] J. Corbet, "BPF: the universal in-kernel virtual machine." https://lwn.net/Articles/599755/, 2014. [Online; accessed 1-July-2019].

[67] "Linux Security Module Usage." https://www.kernel.org/doc/html/latest/admin-guide/LSM/index.html, 2019. [Online; accessed 7-April-2019].

[68] "AppArmor security profiles for Docker." https://docs.docker.com/engine/security/apparmor/. [Online; accessed 7-April-2019].

[69] "PANDA User Manual." https://github.com/panda-re/panda/blob/master/panda/docs/manual.md, 2019. [Online; accessed 11-April-2019].

[70] B. W. Kernighan, *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd ed., 1988.

[71] "Plugin: osi." https://github.com/panda-re/panda/blob/master/panda/plugins/osi/USAGE.md, 2018. [Online; accessed 12-April-2019].

[72] "Plugin: osi_linux." https://github.com/panda-re/panda/blob/master/panda/plugins/osi_linux/USAGE.md, 2017. [Online; accessed 12-April-2019].

[73] "Plugin: syscalls2." https://github.com/panda-re/panda/blob/master/panda/plugins/syscalls2/USAGE.md, 2019. [Online; accessed 12-April-2019].

[74] "Plugin: taint2." https://github.com/panda-re/panda/blob/master/panda/plugins/taint2/USAGE.md, 2018. [Online; accessed 13-April-2019].

[75] "SYSCALLS(2)." http://man7.org/linux/man-pages/man2/syscalls.2.html, 2019. [Online; accessed 13-April-2019].

[76] "SYSCALL(2)." http://man7.org/linux/man-pages/man2/syscall.2.html, 2018. [Online; accessed 13-April-2019].

[77] "GETPRIORITY(2)." http://man7.org/linux/man-pages/man2/setpriority.2.html, 2017. [Online; accessed 13-April-2019].

[78] "GETHOSTNAME(2)." http://man7.org/linux/man-pages/man2/gethostname.2.html, 2017. [Online; accessed 13-April-2019].

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[79] "UNAME(2)." http://man7.org/linux/man-pages/man2/uname.2.html, 2019. [Online; accessed 13-April-2019].

[80] M. Jones, "User space memory access from the Linux kernel." https://developer.ibm.com/articles/l-kernel-memory-access/, 2010. [Online; accessed 14-April-2019].

[81] "NC(1)." https://manpages.debian.org/stretch/netcat-traditional/nc.1.en.html, 2014. [Online; accessed 19-May-2019].

[82] "SYSTEMD(1)." http://man7.org/linux/man-pages/man1/init.1.html, 2019. [Online; accessed 26-April-2019].

[83] "NICE(2)." http://man7.org/linux/man-pages/man2/nice.2.html, 2017. [Online; accessed 24-June-2019].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub