GENERATION OF SECURE RUNTIME ENVIRONMENTS FOR UNTRUSTED APPLICATIONS THROUGH MACHINE CODE ANALYSIS

Dissertation
zur Erlangung des Grades

Doctor rerum naturalium (Dr. rer. nat.)
Fachbereich Mathematik und Informatik
Institut für Informatik
Freie Universität Berlin

vorgelegt von:
Michael Witt

Berlin, 2021

Erstgutachter: Prof. Dr. Marian Margraf - Freie Universität Berlin
Zweitgutachterin: Prof. Dr. Dagmar Krefting - Universitätsmedizin Göttingen

Tag der Disputation: 20.09.2021

ABSTRACT

English

Infrastructure providers rely on the execution of third-party applications to offer their platforms to customers and researchers. The execution of each application without a preceded security review poses a risk for the system and the overall IT infrastructure. However, if the unreviewed application is available for execution, it is also available for an analysis to generate appropriate countermeasures to prevent unwanted behaviour. This work investigates the capabilities of an automated secure environment. This environment is generated based on the analysis of applications that are only available in their machine code format. The analysis focuses on the interaction of the application with the operating system through the *system call* interface. Therefore this work describes required technologies and mechanisms to collect data, process it and generate rules for a secure environment to protect assets from attacks. This process and the result environment is tested with real-world applications and attacks to determine its effectiveness and overall costs. It is shown that the described solution is able to decrease the rate of successful attacks against the system from 83% to 9% in selected use-cases. This is achieved with an execution overhead of 823 ms average. These results demonstrate that it is possible to utilise automatic software analysis pipelines to build restricted execution environments for pre-compiled applications. It also highlight the advantages and limitations of the selected approach to focus the analysis on the system call interface.

Deutsch

BetreiberInnen von Rechenzentren und IaaS-Systemen führen eine Vielzahl von Anwendungen unterschiedlichen Ursprungs von KundInnen und ForscherInnen aus. Die Ausführung dieser Anwendungen ist erforderlich, um die angebotene Dienstleistung zu erfüllen, kann jedoch erhebliche Schäden im System oder der gesamten Infrastruktur verursachen, wenn sie ohne vorherige Sicherheitsüberprüfung erfolgt. Da die Applikation jedoch zur Ausführung auf dem System vorliegen muss, kann sie auch analysiert werden, um geeignete Gegenmaßnahmen zu ergreifen und Schäden zu verhindern. Diese Arbeit untersucht die Möglichkeiten zur automatisierten Erstellung einer sicheren Ausführungsumgebung für Anwendungen aus nicht vertrauenswürdigen Quellen, welche nur als Binärcode vorliegen. Dabei fokussiert sich die Analyse der Anwendung auf dessen Interaktion mit dem Betriebssystem über das *System Call*-Interface. Hierfür werden die notwendigen Technologien zur Datenerfassung, -verarbeitung und -auswertung vorgestellt, sowie deren Verwendung zur Konfiguration einer gesicherten Ausführungsumgebung beschrieben. Die

entwickelte Lösung wird anschließend mit realen Testanwendungen und Angriffen ausgewertet. Es wird gezeigt werden, dass die generierte gesicherte Ausführungsumgebung die Quote erfolgreicher Angriffe auf das Testsystem von 83% auf bis zu 9% senken kann. Dabei wird die Gesamtlaufzeit der Anwendung im Durchschnitt um 823 ms erhöht. Diese Ergebnisse zeigen, dass es möglich sein kann, mit automatisierter Softwareanalyse eine effektive Pipeline zur Ausführung nicht vertrauenswürdiger Software zu erstellen. Es werden ebenfalls die Vorteile und Einschränkungen der System Call-basierten Analyse diskutiert.

PUBLICATIONS

Selected ideas and figures have appeared previously in the following publications:

[1] Björn Lindequist, Michael Witt, Marco Strutz, Dagmar Krefting, Hermann Heßling and Peter Hufnagl. 'Innovative Technologien für Biobanken'. In: *Biobanknetzwerke als Schrittmacher der medizinischen Forschung, Tagungsband des 4. Nationalen Biobanken-Symposiums*. Berlin: Akademische Verlagsgesellschaft AKA, Sept. 2015, pp. 177–179. ISBN: 978-3-89838-709-5.

[2] Christoph Jansen, Maximilian Beier, Michael Witt, Jie Wu and Dagmar Krefting. 'Extending XNAT towards a Cloud-based Quality Assessment Platform for Retinal Optical Coherence Tomographies'. In: *Scalable Computing: Practice and Experience* 16.1 (July 2015). ISSN: 1895-1767. DOI: 10.12694/scpe.v16i1.1062.

[3] M Strutz, B Lindequist, M Witt, H Hesüling, P Hufnagl and D Krefting. 'Application of Ki-67 analysis in a distributed computing infrastructure'. In: *Diagnostic Pathology* 1.8 (2016). ISSN: 2364-4893, pp. 1–2. DOI: 10.17629/www.diagnosticpathology.eu-2016-8:135.

[4] Michael Witt, Marco Strutz, Björn Lindequist, Christoph Jansen, Peter Hufnagl, Hermann Heßling and Dagmar Krefting. 'A comparison of distributed computing technologies for medical image processing in digital pathology'. In: *HEC 2016: Health — Exploring Complexity 2016 Joint Conference of GMDS, DGEpi, IEA-EEF, EFMI*. 2016, p. 1.

[5] Christoph Jansen, Michael Witt and Dagmar Krefting. 'Employing Docker Swarm on OpenStack for Biomedical Analysis'. In: *Computational Science and Its Applications : ICCSA 2016: 16th International Conference, Beijing, China, July 4-7, 2016, Proceedings, Part II*. Ed. by O. Gervasi, B. Murgante, S. Misra, A. Rocha, M. C. Torre, D. Taniar, O. Apduhan, E. Stankova and S. Wang. Cham: Springer International Publishing, 2016, pp. 303–318. ISBN: 978-3-319-42108-7.

[6] M. Witt, C. Jansen, D. Krefting and A. Streit. 'Fine-Grained Supervision and Restriction of Biomedical Applications in Linux Containers'. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2017, pp. 813–822. DOI: 10.1109/CCGRID.2017.53.

[7] Michael Witt, Christoph Jansen, Stefanie Breuer, Maximilian Beier and Dagmar Krefting. 'Artefakterkennung über eine cloud-basierte Plattform'. In: *Somnologie* 21.4 (Dec. 2017), pp. 311–318. ISSN: 1439-054X. DOI: 10.1007/s11818-017-0138-0.

[8] Maximilian Beier, Christoph Jansen, Geert Mayer, Thomas Penzel, Andrea Rodenbeck, René Siewert, Michael Witt, Jie Wu and Dagmar Krefting. 'Multicenter data sharing for collaboration in sleep medicine'. en. In: *Future Generation Computer Systems* 67 (Feb. 2017), pp. 466–480. ISSN: 0167739X. DOI: 10.1016/j.future.2016.03.025.

[9] Michael Witt, Christoph Jansen, Dagmar Krefting and Achim Streit. 'Sandboxing of biomedical applications in Linux containers based on system call evaluation'. de. In: *Concurrency and computation* 30.12 (2018). ISSN: 1532-0626. DOI: 10.1002/cpe.4484.

ACKNOWLEDGEMENTS

I would like to thank my advisors Marian and Dagmar who gave me the opportunity to accomplish this work and who believed in me during the process of the creation of this thesis. I would like to emphasize how grateful I feel about the time as a research associate at the CBMI where I was given the opportunity to strife for my research interest while being backed up by amazing colleagues and Dagmar herself in her role as mentor and friend. Therefore I would like to thank all members of the CBMI but especially Maryna, Christoph, Max, Anja, Björn and Marco who were great companions during this journey.

Finally I want to thank my wife Claudia, my best friend André and my dear friends Atanas and Enrico, who all endured my moods and self-doubt during the creation of this document. However, you never stopped to encourage me to keep on going and I am very grateful that you did.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS AND GLOSSARY

ABI Application Binary Interface

API Application Programming Interface

BAP Binary Analysis Platform

CBMI Centrum für Biomedizinische Bild- und Informationsverarbeitung

CFG Control Flow Graph, graph that represents possible application execution paths

CVE Common Vulnerabilities and Exposures

CVSS Common Vulnerability Scoring System

DFG Data Flow Graph, graph that represents the application state during execution

GOT Global Offset Table, part of application binaries that holds addresses of linked dynamic libraries

IaaS Infrastructure As A Service

IP Instruction Pointer, a pointer in the processor register to the next instruction to execute

IR Intermediate Representation, transfer language to group assembly instructions into meaningful groups

OS Operating System

PaaS Platform As A Service

PLT Process Linkage Table, part of application binaries that holds addresses of linked dynamic library functions

SAT problem Satisfiability problem

SMT Satisfiability Modulo Theory

Symbol (code) A compiled function present in the machine code of an application or dynamic library

Syscall Shorthand form of *System call*

System call Interface to access operating system managed resources from user-mode application.

WCET Worst Case Execution Time

Part I

PRELIMINARIES

1 INTRODUCTION

Infrastructure provider sell their computational capacities to customers to enable them to run custom applications within a fault tolerant and accessible environment. Scientific infrastructures offer computational resources to associated researchers which allow them to execute long running applications or data intensive calculation. Both use-cases depend on the execution of software, provided by the client (customer or researcher).

The execution of arbitrary software inside the infrastructure is potentially dangerous because of unwanted behaviour. Erroneous functionality like bugs or incorrectly implemented algorithms as well as malicious applications are threats to the providers assets. The compromisation of a computing system through the disguise of a benign application is one major path of infection for threats like viruses, trojan horses, spyware or ransomware [4].

Different approaches have been developed over time to protect computing systems from threats such as data loss, data leakage, hardware failures or malware infection. Multi-User systems, virtual memory management, separation of process spaces as well as different levels of virtualisation are components available for modern operating systems. However, their setup and configuration are tedious tasks that require profound knowledge of the target computing system, software executed and threats faced. Additionally, maintaining the security of the infrastructure is an ongoing process. Applications from new clients that access the computing systems need to be secured and newly discovered vulnerabilities must be mitigated.

1.1 PROBLEM STATEMENT

The execution of applications provided by a client or another untrusted source inside a computational powerful shared infrastructure is a substantial risk for their providers. Nonetheless, this risk cannot be avoided since the provision of computing resources is the business model of commercial IaaS/PaaS provider. Clients require them to run their applications and providers need to execute the clients software as part of their service offered.

The same problems from commercial providers apply for research institutions that offer their computing resources to associated partners. Authorised researchers transfer their algorithms bundled together with required dependencies as executable applications into the infrastructure. In big data, the transfer of the algorithms into the infrastructure that

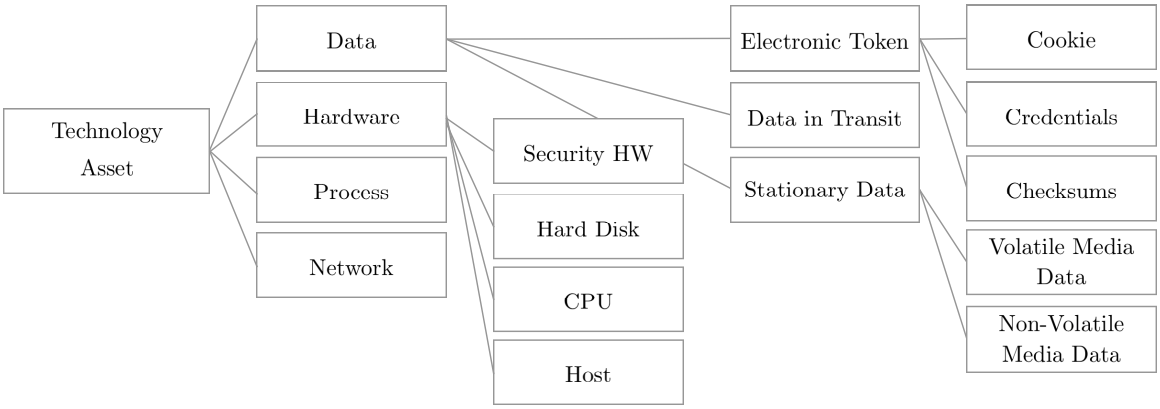


Figure 1: Examples for technical assets according to the ontology of Herzog et al. [6]

hosts the data is part of the paradigm [5] which makes the execution of the client application compulsory.

For both commercial and scientific infrastructure providers (*service provider*), a malicious or even an erroneous application poses a *threat* to one or more of their *assets* (see fig. 1). Thus, the implementation of appropriate security measures is mandatory. Virtualisation technologies like virtual machines or operating system based virtualisation help to isolate software from the computing system. Other mechanisms like access control lists, memory isolation or firewalls also provide mechanisms to protect the system against threats.

The implementation of effective and efficient security measures on the other hand is difficult. The foreign applications are complex and consist of multiple components that might interact with each other in different ways. A program itself (as well as its components) might also interact with the operating system. To guarantee that the execution poses no threat to the computing system is hard or even impossible [7]. Because the service provider controls the operating system and installed software that makes up the infrastructure, these components can be considered trustful and working correctly. For this reason this work focusses on securing of the untrusted application which is executed in a trusted environment.

The investigation of an application for threats is feasible if its source code is available. Such a source code analysis can be conducted to identify potential risks. It can also detect prerequisites that have to apply for a threat to become harmful. Because applications can incorporate additional third-party components (libraries, frameworks, toolboxes, etc.); these subsidiary code parts need to be investigated too. However, different issues can make a source code analysis of the application an its components impossible or produce incomplete results:

- The source code analysis fails to detect parts or interactions that pose a threat to the infrastructure.

- Closed-source software components prevent a thorough source code analysis at all.
- The user might by unable to investigate the application due to e.g. lack of time or insufficient expertise.
- Legal issues disallow a source code analysis.

So the guarantee of benign execution is impossible and if one or more of the issues above apply, the questionable application can not be verified to be harmless to the system either. Therefore it must be considered *untrusted*. It does not matter if an issue relates to the software as a whole or to only a selected part. As soon as there is uncertainty about the behaviour to anticipate throughout execution, it is advised to consider the application untrusted. Figure 2 illustrates the problem of untrusted parts inside the application where a user (e.g. a developer) might have high confidence in the trustworthiness of his/her algorithm, but can make no such assumptions for the other components.

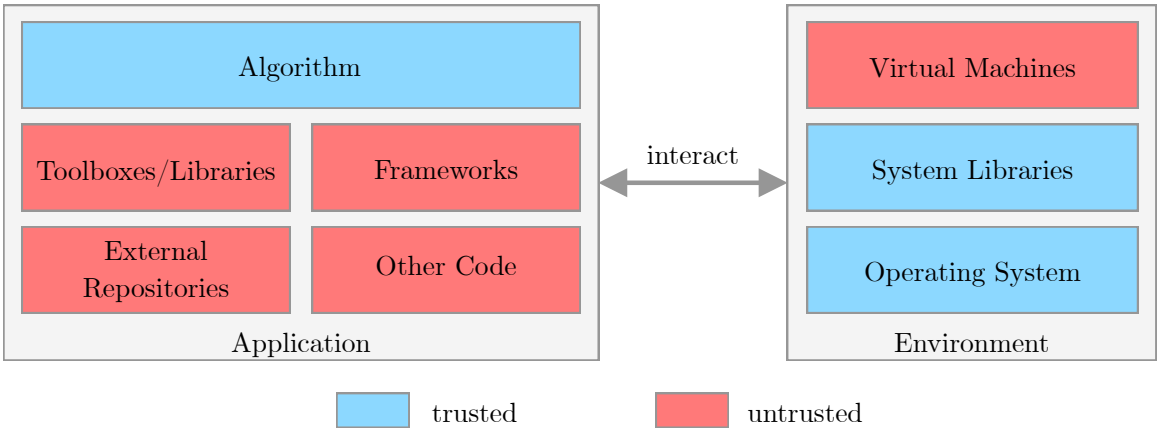


Figure 2: Visualisation of a complex application consisting of an analysis algorithm accompanied by third-party libraries and other components that is executed in an environment with operating system, library dependencies and virtualisation.

As described above, the service provider lacks domain knowledge of the untrusted application and for this reason has to consider the application altogether as untrusted. The service provider is required to offer an *application agnostic* infrastructure to the clients. Thus a manipulation of the client application itself through e.g. recompililation is not feasible. As a result, the infrastructure itself must be protected from threats that may result from the untrusted application. Appropriate *countermeasures* must be taken to mitigate threats that target the assets of the service provider.

These countermeasures need to be sophisticated enough to prevent all threats but at the same time still allow the computing system to execute benign applications in an expected fashion. Also, if it is considered that many untrusted applications with different purposes

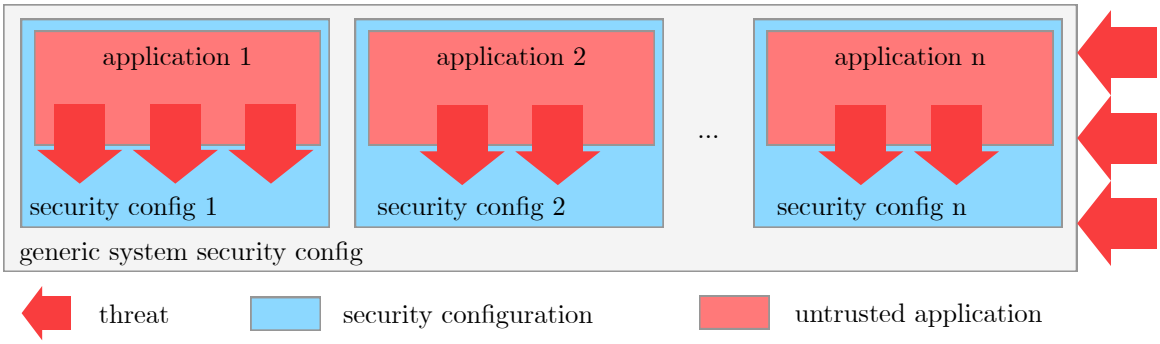


Figure 3: Design of a computer system with dedicated security settings for each executed application and overall system-wide security setup.

are executed on a system, there should be dedicated security mechanisms in place that are tailored for each application.

Figure 3 visualises a system with multiple executed application. Each of them is executed with a dedicated security configuration. These configurations allow only actions necessary for anticipated client application execution but deny other actions. Especially those actions that result in threats for the computing system have to be stopped. Additionally, due to threats from external sources, the computing system itself requires general security mechanisms for their mitigation.

Application agnostic execution of untrusted applications from different sources while simultaneously providing dedicated security configurations to protect assets is a difficult task. Application specific security configurations require information about the program that is run. These information might not be available due to the reasons described above. Furthermore, thorough knowledge about security mechanisms that can be employed for threat mitigation is required. As a result of the lack of information and configuration complexity, service provider usually do not employ security models as shown in figure 3. Instead, the general system security configuration is extended to protect the system from external threats *and* unwanted application actions.

In order to *maximise* the overall service infrastructure security, this work introduces methods based on the analysis of machine code of untrusted applications. These will be used to anticipate their behaviour during execution and extract the information required for dedicated security configurations as shown earlier.

1.2 RESEARCH QUESTIONS

As described above, the execution of arbitrary untrusted software by service providers is mandatory but dangerous. Different threats target assets of the infrastructure and can e.g. impair proper functionality of the computing systems. To establish dedicated per-application security configurations information about the client programs are required.

With the availability of static and dynamic code analysis as well as application emulation, powerful tools are available to collect the required information.

The untrusted application needs to be investigated for critical actions that might harm the computer system. However this requires a mechanism or interface to base the categorisation of actions as threatening or safe. Since most threats result from an unwanted or uncontrolled access to system-managed resources, a categorisation based on this resource access is feasible. The *system call* interface must be used to access resources that are managed by the operating system. Since it is mandatory for resource access, it is a promising interface to focus on during software analysis to identify threats. Using the system call interface to control or supervise an application is also feasible as shown first by Goldberg et al. in [8].

With the focus on the usage of the system call interface software analysis methods like static and dynamic code analysis can be used. This includes partial or full emulation as part of the dynamic analysis. To collect required information if the source code of the client application is unavailable, the analysis must be able to process machine code. With the collected data during analysis an automatic process should be able to extract the knowledge required to configure a dedicated application security system similar to figure 3. To achieve this, the following two main research questions will be discussed in this work.

> *Can a combination of static and dynamic machine code analysis techniques be used to extract exhaustive information about the behaviour of arbitrary untrusted applications? Furthermore, how can these information be used to generate an application security configuration that allows the execution of the untrusted program while protecting the assets of the service provider?*

These two questions themselves lead to several other problems related to software analysis and the configuration of isolated execution environments. The following paragraphs will further introduce these questions and outline how the research presented in this thesis addresses them. The solutions for the related problems are an integral part of the solution of the main problem described above.

1.2.1 *Machine Code Analysis*

This thesis will focus on untrusted applications that are available only in their compiled form. This form is called *machine code*, compiled application or binary and consists of a series of instructions that can be run by any processor of the programs target architecture. Additional application components as shown in figure 2 that are required to execute the application have to be considered too.

Code analysis itself (not limited to machine code analysis) can be done either statically or dynamically. Static machine code analysis investigates the application through the analysis of the given machine code instructions. The application itself is neither executed nor

emulated. Because of the limitation of static analysis (which will be described in detail in chapter 4), dynamic code analysis methods can be used. They are used in unit tests to test software modules, in integration tests to verify correct software component interaction and are the basis of memory leak detection. Dynamic program analysis is based on the observation and investigation of the application during its runtime. It is therefore either executed directly or emulated with different degrees of abstraction.

To extract the desired information about interaction with the operating system through the system call interface, two tasks need to be completed. First is the reconstruction of the *execution path* which led to the resource access that involves the system call interface. An execution path is the sequence of machine code instructions that led from a dedicated start instruction to another one in an arbitrary amount of instructions passed on the way. As a result of this principle, each time an instruction evaluates a conditional that affects the program flow (e.g. an `if`-clause), the execution path must be divided and all possible ongoing paths need to be investigated further. To evaluate conditionals the second task of constructing application *execution state* need to be considered. The execution state comprises the values of all inputs that are available during the application execution (processor registers, heap and stack memory). A condition in the execution path therefore refers to an execution state that evaluates it as *true*.

Static code analysis is able to reconstruct possible execution paths, but suffers from the fast growing complexity when investigating potential paths of execution. This is required because there is not sufficient information about the execution state. If the execution path is divided and the limited information about execution state does not allow to identify a subpath as unreachable, the analysis must follow the subpath for a complete analysis. Since dynamic program analysis actually executes (or at least emulates the execution of) the application it has a more complete execution state to work with. As a result it can rule out unreachable execution paths.

The execution state of the application is also important when a system call interface interaction is detected. The data provided to the interface about which resource to access is also part of the execution state.

In consequence, this work needs to investigate how methods from static and dynamic code analysis can be used to investigate applications in their compiled form. It is required to collect usage information of the system call interface as well as the execution state of the application at the time of the interface invocation.

1.2.2 *Security Configuration Generation*

The results of the machine code analysis should be used to generate the desired application specific security configuration. This configuration must be usable to set up a secure execution environment for the untrusted application. This environment must restrict the access to resources that are not authorised for access by the application. This restricted

environment is called a *sandbox* in this work. The sandbox is active during application execution and should not consume any more resources once the application finishes. This is important for multi-user systems like those offered by the service providers because stale resources are unavailable for current and new clients.

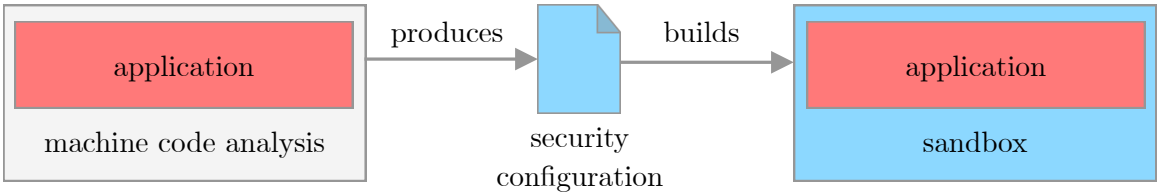


Figure 4: Generation of an execution sandbox based on the configuration generated from the data collected by machine code analysis.

This process is shown in figure 4. The result sandbox realises the computer system design with dedicated security configurations for each executed application as described in the problem statement. A benign application can function within the limits of the sandbox, whereas malicious or erroneous behaviour will try to access prohibited resources that will trigger the sandbox. A result of the latter behaviour is either the termination of the application or the denial of resource access. Either way the action is prevented and therefore the potential threat mitigated.

This procedure can be generalised for computing systems inside an infrastructure. Such a system can be described by a state S, which encompasses all its assets and their status. The generation of the security configuration for an application and subsequent execution in the derived sandbox will transfer the system into the new state S'. The process is working properly if all assets in state S' are in a status that is either equal of better than the one in state S.

1.2.3 *Threat, Asset and Result Evaluation*

The above description of system state S and the status of system assets leads to the next topic that are addressed in this work.

To compare two different states of the computing system a *metric* to actually measure the security of a system state is required. Since *security* is an intangible term according to [9], it is required to define a scheme to measure desired aspects of the system state and generate the metric with these measurements. As described above, this work focusses on the assets of a service provider, which are reflected as assets of computing systems. Therefore they are part of the system state and must be used in the evaluation metric.

Because the system state is made up of the status of the system assets, the metric to describe the system state needs to consider and evaluate these. However, this is not

enough as a system faces threats while it is in a certain state. Therefore the system state metric must consider and rate threats too.

Finally, only a metric that is able to rate the system state according to the principles described is usable to evaluate the effectiveness of the installed sandbox. For this reason a metric to achieve this security evaluation must be conceived alongside the process of application analysis and sandbox generation.

1.3 OBJECTIVES AND CONTRIBUTIONS

This work is focused on applied security research and will introduce and evaluate a framework that is capable of achieving the tasks outlined above. Since information security is an extensive field of research and overall computing system diversity is large, this work will focus on selected technologies and software components. Nonetheless it will employ generalisable approaches that can be transferred to similar use-cases like different operating systems, processor architectures or security frameworks.

To design a comprehensive solution for the secure execution of untrusted applications, this work focuses on applications in their compiled form as machine code files. Even if the source code of the applications or their parts are available, their analysis is not considered. There is already extensive research about static and dynamic source code-based application verification, testing and threat detection. Furthermore, the developed techniques focus on Linux-based operating systems and x86 64-Bit architecture processors. Nonetheless, since the mechanisms of interaction for applications with the operating systems are generalisable, the presented solution can be adapted for other operating systems. The system call interface is also slightly different for other processor architectures. Anyhow, similar to the adoption of other operating systems, the support for other processor architectures can be realised through adjustments in the machine code analysis framework.

The contribution of this work to the field of applied informatics research can be found in the developed algorithm to extract resource access information from an application without actual execution. Based on this information, a security configuration is generated that protects infrastructure assets from threats that originate from untrusted applications. Finally a metric to rate the security of a system state as well as threats and assets is a contribution to informatics research that can be used beyond the scope of this work.

1.4 STRUCTURE OF THE THESIS

The first chapter of the thesis will focus on the definition of assets present on computing systems and valuable to the service provider. Based on these assets, an overview of threats and threat classes is conducted to identify those relevant for this work. With the knowledge about assets and threats related to the infrastructure, different metrics are presented that can be used to measure the security of a system state.

The next chapter introduces the system call interface and the conceptional structure of application interaction with the operating system to access resources. Based on this introduction the association of assets and threats to system call interface utilisation is discussed. Such an abstraction between operating system-depended low-level interface invocations and abstract system assets is used to associate system calls to possible threats.

Chapter 4 focusses on the analysis of the untrusted application. The presented methods will be used to extract the required information of the system call interface utilisation from the program that is investigated. The structure of executable applications and dynamic libraries on Linux-systems is described and different methods for static and dynamic analysis are presented. The eligibility of these methods to work on applications only available in their machine code is also considered and discussed in this chapter.

Finally technologies and frameworks that are capable to generate sandboxes that allow the protection of the defined assets are presented in chapter 5. The presented technologies are also discussed based on their eligibility to be configured with the information extracted with the methods from the preceding chapter.

The final chapters of the thesis introduce the methods to analyse the untrusted application, generate the security configuration and set up the application sandbox. Based on the identified metrics for system state security evaluation, experiments are described to validate the effectiveness of the approach as a whole and the resulting sandbox. Finally a discussion of the achieved results and an outlook conclude the thesis.

Part II

INFRASTRUCTURE AND TECHNOLOGIES

2 SECURITY EVALUATION

As described in the introduction, a service provider has an interest to provide a secure infrastructure to the clients. This chapter investigates the abstract term of *security* and deduces concepts and technologies that helps to realise this *secure infrastructure*. This definition of security is required in later chapters to identify threats based on this definition in machine code analysis and to configure a sandbox that protects the computing system assets.

After the security definition is done, an analysis is conducted about threats that the infrastructures faces. This section includes related work about the categorisation and unification of the diverse classes of threats. After this categorisation is done, metrics to measure the severity of threats are introduced.

Finally, since this work requires an overall measurement of system security, methods and approaches for this task are presented in the last section.

2.1 SECURITY CONCEPTS

The definition of security is difficult as it is an intangible state that describes different things when assessed from different perspectives. However, there has been work to categorise security and to name secure features as well as system state that can be considered secure. Once these things are defined it becomes feasible to describe threats that attack the security of the system. This section provides an overview of established definitions of security and threats.

A disambiguation of the different core concepts related to system security is given by Herzog, Shahmehri and Duma in [6]. A figure with important concepts for this work is shown in figure 5.

The infrastructure of the service provider consists of multiple assets as shown in figure 1 that must be protected. These are threatened by threats and attacked by attacks. There is a minor difference between *attack* and *threat* regarding the state of realisation. A threat is an abstract concept of something that might somehow impact the infrastructure. An attack realises one or more threats and targets the assets of the infrastructure. Since the difference between these concepts lies solely in the state of realisation, they are often used synonymously in the literature [6, 10, 11]. Finally, in this work, an adversary or attacker is a person or entity, that uses attacks to target assets of the service provider. If the adversary is successful with the attack, the asset gets compromised.

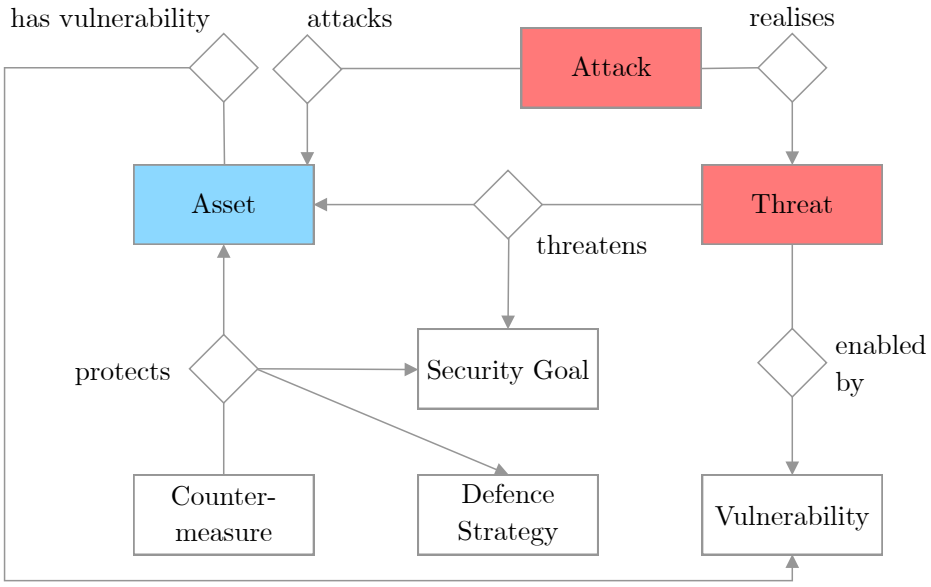


Figure 5: Overview of the core concepts and relations for an security ontology according to [6] extended with the *Attack* concept.

2.1.1 *Assets*

An asset is a useful or valuable thing for the service provider. Herzog, Shahmehri and Duma provide a complex ontology for the asset concept in [6]. The visualisation of this ontology is shown in appendix B. Their work divides assets into human, technological and credential-based assets. The protection of human assets is beyond the scope of this work. Table 1 shows the classes for credential-based assets and table 2 displays the classes of technology-based assets.

This asset definition is comprehensive and describes the different classes of valuable things for a service provider. Other approaches to the definition of system security like the one from Bates et al. focus on system provenance [10]. Based on this approach for system security, they identify *Scientific Computing*, *Access Control* and *Networks* as assets of the computing system. Their "network" assets can be matched to the network asset class shown in table 2. "Scientific computing" relates to the data asset class. This is shown by the description of the threat to this asset as an '(...) adversary may wish to manipulate provenance in order to commit fraud, or to inject uncertainty into records (...) [10, p. 322]'. Finally "access control" relates to the protection of assets of the credential class and describes a countermeasure for unauthorised access as well.

Other literature focuses on the description of threats and attacks that realise these threats. Therefore this work uses the ontology of Herzog et al. to reference different types of assets and to investigate attack target classes.

Class	Sub-Class	Asset-Classes
Credential	Electronic Token	Encryption Key (Symmetric, Private, Public Key)
		Password
		Login Name
		Cookie
		Message Digest (Authentication Code Data, Digital Signature Data, Certificate Data, Checksum Data)
	Biometric Token	Physical Biometric Credential (Facial pattern, Fingerprint, Retina, Iris, Hand Measurement)
		Behavioural Biometric Credential (Voice, Gait)
	Physical Token	Dongle
		Smart Card

Table 1: Classes and sub-classes of the *credential* asset concept [6]

2.1.2 *Threats and Attacks*

The ontology of Herzog et al. also contains classes of threats and summarizes them in the *Threat* concept of the ontology. A full concept definition of this part of the ontology can be found in appendix A. Bates et al. focus on the definition of system goals for their provenance-based system security approach [10]. Their work does not specify concrete threats that they try to mitigate. Instead they name the "system-goals" *Tamperproofness*, *Verifiability*, *Authenticated Communication* and *Attested Disclosure*. These relate to the *Security Goals* class rather than the threat class according to [6] and can therefore not be used for threat specification.

Sabahi names *Data leakage* and *Cloud security issues* like DDoS-attacks against or from the infrastructure as security threats [12]. A more broad description of threats against arbitrary computer systems was given by Myagmar, Lee and Yurcik in [11]. They name *spoofing*, *tampering*, *repudiation*, *information disclosure*, *denial of service* and *elevation of privilege* as threat classes based on the work introduced by Swiderski and Snyder [13]. Papp et al. focus their investigation on embedded systems and show different attack paths [14] (see figure 6).

The "attack method" that Papp et al. describe in their work relate to actual attacks that were investigated on embedded systems. Similar to the ontology presented in figure 5, attacks realise a threat that is enabled by different vulnerabilities as shown in figure 6. The attack is aimed at a certain target which can by related to the above described asset term. Therefore the attack method description of Papp et al. is also suited as a base to define threat classes.

Class	Sub-Class	Asset-Classes
Technology	Network	Untrusted Network
		Trusted Network
		Wireless Network
		Wired Network
		AdHoc Network
		Intranet
	Data	Symmetric Key
		Private Key
		Public Key
		Password
		Login Name
		Stationary Data (Stack, Heap, File Source Code File, Backup File, Database Data File, Configuration File, Program File)
		Data in Transit (IP Packet, TCP Packet, UDP Packet, HTTP Data, E-Mail)
	Hardware	Security Hardware (Dongle, Smart Card, Degausser, Encryption Hardware)
		Harddisk
		CPU
	Host	Unconnected Host
		Networked Host (Host on Intranet, Host on Wired Network, Bastion Host, Host on Internet, Client Host, Host on Wireless Network, Router, Wireless Access Point, Server Host)
	Process	

Table 2: Classes and sub-classes of the *technology* asset concept [6]

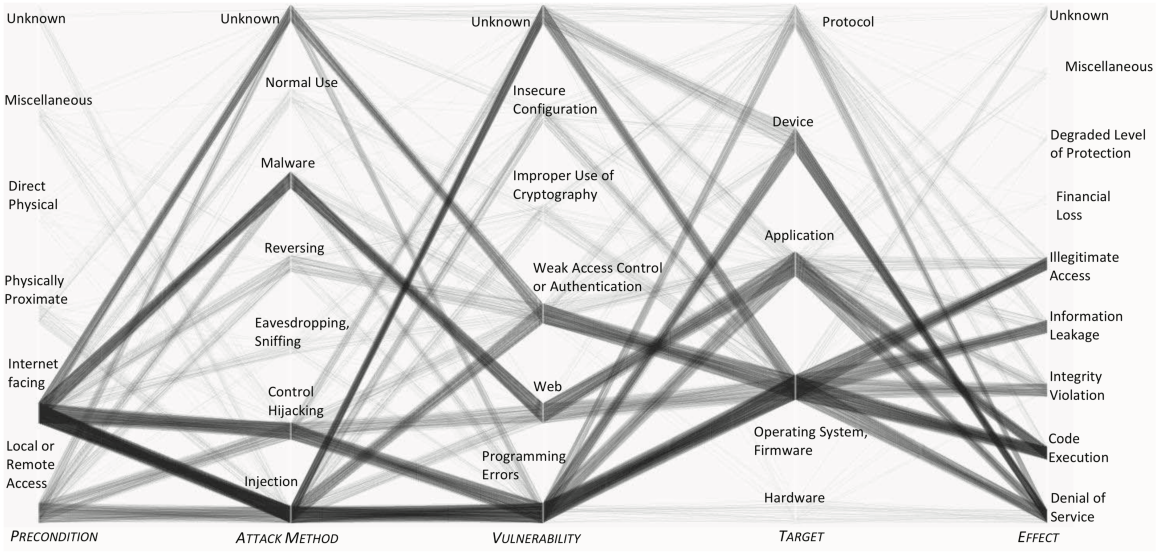


Figure 6: Common attack scenarios according to [14]. The thickness of lines in the diagram indicate the commonness of occurrences of attack paths.

Despite the focus on large computer networks and multi-stage attacks against them, Sheyner and Wing propose a general approach to model attack scenarios with graphs in [15]. The graph reflects the attack of an adversary against the computer system through a finite automaton. This is especially useful to model the chaining of several vulnerabilities that cause an asset to get compromised by an attacker. Besides the definition of the automaton, they also present a system for attack graph generation and subsequent mitigation through so-called *security properties*. The generation of the graph based on detected vulnerabilities is done with closed source and discontinued software products from MITRE's *Outpost* [16] or Lockheed Martin's *Advanced Technology Laboratory's Next Generation Infrastructure* [17].

The different work described above can be used to identify threat classes and associate them with assets that are threatened by them. The table 3 gives such an association of threats to assets of a single computing system or even an infrastructure.

Threat	Asset(s)	Description
Malformed Input [6]	all	Malformed input-based attacks like *buffer overflows*, *code injections* and *format string attacks* can be used to interrupt ordinary program flow and to force the algorithm to behave maliciously. In a worst case scenario, the attacker gains control over a privileged application and can target any asset based on this administrative system access.
Malicious Code [6, 14]	all	Identified as one kind of masquerading attacks by Herzog et al., malicious code like viruses, trojan horses, backdoors, spyware etc. target vulnerabilities to gain access to arbitrary assets.
System Modification [6]	all	Persistent configuration changes of the computing system that results in permanent asset compromisation beyond the actual runtime of attack.
Control Hijacking [14]	Stationary Data Process	Taking control over an otherwise benign application through manipulated data or targeted behaviour. Once a process has been taken over, further attacks can be launched.
Negative Acknowledgement [6]	Stationary Data Process	Negative acknowledgement attacks aim to attack a process asset when it is in a vulnerable state caused by an interrupt of an invalid action.
Passive Attacks [6], Eavesdropping [14]	Stationary Data Process	Attacks that are based on the continuous observation of the target system to gather data about the system itself, processed data etc. Examples for such threats are side channel attacks, scavenging, statistical attacks on e.g. cryptographic algorithms or eavesdropping.
Disruption, Usurpation [6], Repudiation[11]	Process, Network, Host	Attacks that employ physical threats like heat, theft or system overclocking as well as host compromisation attacks like DNS server compromisation can result in system takeover or service disruption due to failed server processes or network outage.

continued on next page

Threat	Asset(s)	Description
Internet Infrastructure Attack [6], Spoofing [11]	Network, Host, Data in Transit	Attacks like routing table or cache poisoning, spoofing or packet mistreatment aim to interrupt network and host assets or to perform unauthorized reads on transferred data.
Other Masquerading Attacks [6]	Network, Electronic Token Process	Other attacks that perform some kind of masquerade like man-in-the-middle attacks, spoofing, session hijacking etc. target a process or network assets.
Denial of Service [6, 11], Distributed DoS [12]	Network Process	The interruption of a service due to an exceeding amount of invalid requests.
Brute Force Attacks [6]	Electronic Token	The guessing of authentication credentials to gain access or to elevate current permissions.
Elevation of Privileges [13]	Stationary Data	General approaches to use vulnerabilities in permission evaluation and enforcement routines to elevate user privileges to acquire administrative permissions.
Tampering [11]	Data	Manipulation of system files, directories or other data to either gain further access to the system or destroy/ manipulate data records.
Data Leakage [12]	Data	Unauthorized extraction of data.

Table 3: Association of selected threats to targeted resource types

Some of the presented threats in table 3 are related to external attacks against the computing system. Nonetheless should these threats be considered because they may also be launched from inside the infrastructure against the same system or other member computing systems.

Outside of the focus of this work are threats against hardware assets like e.g. sabotage. Similarly, threats like men-in-the-middle-based attacks or algorithms that try to break secrecy through weaknesses of the cryptographic algorithm are also beyond the scope of this work. Since the presented framework focuses on the protection of the local computing system against unwanted behaviour, threats like network data traffic recording from an external adversary and subsequent information disclosure are also not considered here.

2.1.3 *Resources*

As described in the introduction, this work focuses on the supervision of access to resources managed by the operating system. Similar to the relation of threats and attacks, where an attack is the realisation of an abstract threat, is the correlation between asset and resource. An operating system managed resource (like a file, network connection or process) is a tangible realisation of an abstract asset.

An association of assets to resources is performed in chapter 3 after the different types of resources and their management by the operating is introduced.

2.2 THREAT RATING

The above section defined the essential terms that are required to describe the security of infrastructure, its assets and threats that target them. This section introduces mechanisms to assign a numerical factor to these threats. This is required to perform ratings of threats and generate a metric for the overall infrastructure security in the next section. It should be emphasized that this is difficult because of the intangible nature of a threat which targets one or more abstract vulnerabilities.

One approach to assign a numerical value to application behaviour is described in the patent *US 7,530,106B1* by Zaitsev et al [1]. Assigned values range from 0% (no risk of an attack at all) to 100% (application is malicious and threatens the system). This risk-score of an application is calculated in the beginning but can be updated over time if it behaves suspiciously. If the risk-score exceeds certain boundary values (50% and 75%) the application is stopped. The user is then asked for approval to continue the execution or, in the case of a very high risk, the application is immediately terminated. A flow graph for this process is shown in figure 7.

This process is understandable and allows to identify an application as malicious before it is executed. It also performs an ongoing investigation of actions that might result in a revision of the security decision from the beginning. As a result, this requires the system to define attributes to check at the beginning and define behaviour to rate during runtime. Zaitsev et al. specify an example for such a behaviour supervision rule in their work that illustrates this. Listing 1 shows a rule that describes how the security rating is changed if the supervised application checks for an installed anti-virus software on the operating system. The first check will increase the rating by 10%, a second and third probe causes the rating to increase by 30% and every other check causes an addition of 60% to the calculated score. This rules reflects the hypothesis, that a malicious application will continuously check if there are countermeasures present in the operating system that might prevent certain attacks. However this might also be valid behaviour for other types of software. Therefore a definite ruling for malicious intent based on this behaviour is problematic.

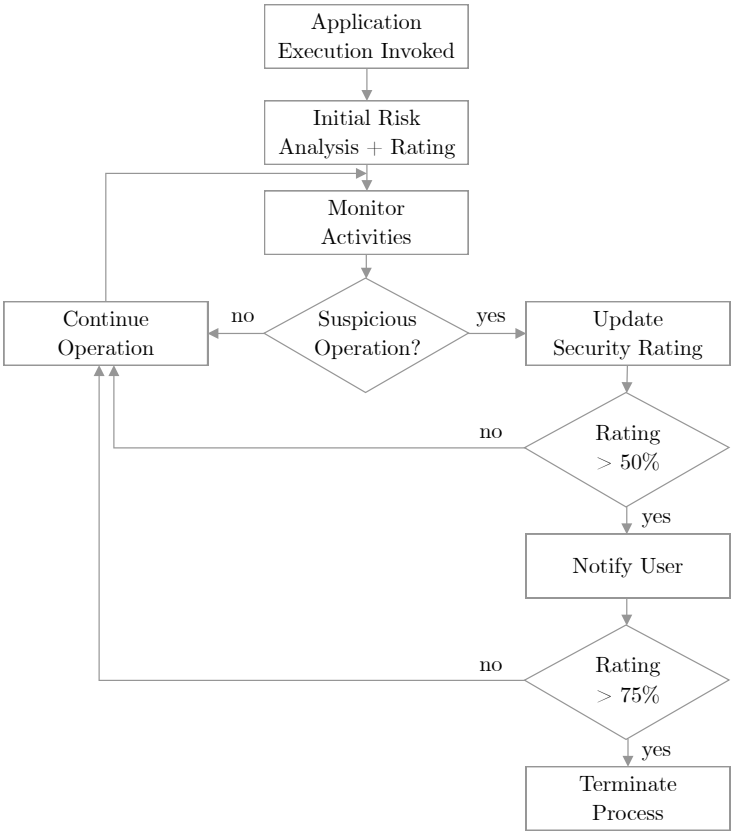


Figure 7: Simplified flow diagram of the continuous application rating and supervision process as patented by [1].

```
Rule 'Checking status of antivirus service'
Rule identifier: 8
API function: Checking the status of antivirus services
              (QueryServiceStatus)
Rating: single operation - 10%
          2-3 operations - 30%
           >3 operations - 60%
}
```

Listing 1: Security rating modification rule based on application behaviour [1]

The system from Zaitsev et al. is a closed source system and designed to operate on *Microsoft Windows* systems. It suggests an easy-to-understand scale with associated actions. A more complex and generic system is proposed by the *Common vulnerability scoring system* (CVSS) [18]. This industry standard enables the calculation of scores for vulnerabilities based on *Base*, *Temporal* and *Environmental* metrics [19]. A score determined with the CVSS

Metric	Scored Properties
Base	Attack Vector (AV), Attack Complexity (AC), Privileges Required (PR), User Interaction (UI), Scope (S), Confidentiality (C), Integrity (I), Availability (A)
Temporal	Exploit Code Maturity (E), Remediation Level (RL), Report Confidence (RC)
Environmental	Confidentiality Requirement (CR), Integrity Requirement (IR), Availability Requirement (AR), Modified Attack Vector (MAV), Modified Attack Complexity (MAC), Modified Privileges Required (MPR), Modified User Interaction (MUI), Modified Scope (MS), Modified Confidentiality (MC), Modified Integrity (MI), Modified Availability (MA)

Table 4: CVSS scoring categories [19]

is a standalone value in the range from 0.0 (low) to 10.0 (high) that describes the severity of a single investigated vulnerability.

The rating is done manually for each vulnerability and should be executed by a IT security professional based on the CVSS scoring categories. The professional assigns ordinal values to each of these categories (e.g. high, medium, low or exploitable via network, local access or physical access). Please refer to table 4 for a brief description of the scoring parameters according to CVSS version 3.0 [19]. The CVSS defines numerical values to assign to the selected categories and how to build a final score out of them. Additionally, a unique string is generated that can be assigned to public vulnerabilities to describe their properties based on the CVSS.

A textual rating is assigned to the vulnerability based on the calculated score. The defined labels are *None* ($\text{score} = 0.0$), *Low* ($0.1 \leqslant \text{score} \leqslant 3.9$), *Medium* ($4.0 \leqslant \text{score} \leqslant 6.9$), *High* ($7.0 \leqslant \text{score} \leqslant 8.9$) and *Critical* ($\text{score} \geqslant 9.0$). This score is regularly used to indicate the severity of a newly discovered vulnerability and is commonly provided alongside a CVE *Common Vulnerabilities and Exposures*) number. The U.S. NIST provides a publicly available list of all CVE vulnerabilities in the national vulnerability database [20]. Alongside the CVE identifier is the associated CVSS score and a reference to the affected software.

Other threat assessment techniques are *STRIDE* from Microsoft [21] or *DREAD* used by OpenStack [22]. Both systems describe categories that are reflected in their names: *STRIDE* addresses threats like *Spoofing*, *Tampering*, *Repudiation*, *Information Disclosure*, *Denial of Service* and *Escalation* whereas *DREAD* generates a score out of values indicating the impact on *Damage*, *Reproducibility*, *Exploitability*, *Affected Users* and *Discoverability*. For each category in the DREAD system a rating guideline is specified to assign a category score.

Potential for: Tampering, Escalation		
Category	Score	Rationale
Damage	6	Significant Disruption
Reproducability	8	Code path is easily understood, condition exists as standard
Exploitability	2	Very hard to exploit without specific conditions
Affected Users	8	All cloud compute users
Discoverability	10	Discoverability always assumed to be 10
DREAD SCORE: $31/5 = 6.2$ - Important, fix as a priority		

Table 5: Example DREAD score calculation [22]

The final score is determined by the mean value of the five categories. Table 5 shows this calculation for a tampering or escalation threat. Based on this score a priority for mitigation is assigned to the vulnerability that enables the threat. The STRIDE system does not provide a numerical evaluation of threats at all.

Another approach to threat rating is described in the *HMG Information Assurance Standard No.1* (IS1) [23]. This rating method uses the concept of assets as well as the *Risk Assessment Scope*, *Business Impact Level*, *Threat Source and Threat Actor*, *Threat Level*, *Compromise Method*, *Risk* and *Risk Level*. This system enables an organisation to build and maintain a working security asset management by providing example forms and guidelines. Comparable to the CVSS, IS1 bases the rating on an expert review of the aforementioned categories. The major difference to CVSS is the focus on assets rather than vulnerabilities/threats. IS1 assigns an ordinal rating to an asset which determines actions to secure it. Related systems like the international *ISO 27001* guideline [24] or the German *BSI Grundschutz* [25] also aim to implement security management systems based on asset analysis.

This work requires the rating of severity of threats rather than the importance of assets. Therefore the CVSS is suited best for the investigated use-case. It is able to assign numerical values to arbitrary vulnerabilities based on an expert rating and calculated with an open well-known formula ([19], [26]). Because of the relationship between threats and vulnerabilities (threats are *enabled by* vulnerabilities, see figure 5), the score for a vulnerability will also be used to rate enabled threats and their realised attacks.

2.3 SYSTEM SECURITY RATING

The above section described metrics to measure the severity of a single threat. Although this is a vital components for the analysis conducted in this work, it is not sufficient for the evaluation of the overall system security. As described in 1.2.3, a metric for a system state

S is required. Only such a quantifiable measure of S allows the comparison of system states at different times. This makes it feasible to evaluate whether the overall system security has become better, worse or is at least equal. Finally this evaluation enables the proof of effectiveness of the application security configuration and generated sandbox.

This section will introduce related work in the field of system security evaluation as well as metrics that can be used in this thesis. Methods from the previous section reappear in this section and are investigated again. Anyhow, this investigation now focuses on their capabilities for overall system security rating rather than single threat assessment.

System Security Metrics

Khudhair and Ahmed describe metrics related to information security as something that involves the application of measurement to multiple entities of a system and generate further knowledge through the combination of these data-points [27]. They also name four examples for metrics available for organisations that are required to perform a security assessment. These examples are shown in table 6 together with a brief description.

Metric Standard	Definition
ISO 27002	This standard is meant to be used together with the ISO 27001 document. It describes information security controls and their objectives. Advices are given based on best practice examples [28].
ISO 27004	Finished in 2016, this standard describes guidelines to assist with the evaluation of security risks based on the requirements defined in ISO 27001 [29].
IS 1	As described in section 2.2 this UK government standard provides ordinal measures for risk assessments. It also suggests forms and processes to employ these risk assessment tools in a continuous organisational workflow [23].
USA NIST	The *Security Metrics Guide for Information Technology Systems* is similar to the IS1 guidelines and is used for US federal government entities to implement a security and risk assessment [30].

Table 6: Security metrics standard according to [27]

The aim of the described metrics is to implement, verify and continuously evolve a security management within organisations. They describe the comprehensive tasks involved in the operation of service infrastructures. The assessment of the overall system security

is a vital part of these tasks. After all, due to the advisory and process oriented character, these systems do not provide a method to determine the desired measurement for the security of a computing system. A more concrete and tangible metric is required for this. Houngbo and Hounsou found the same problem in their work presented in [9]. They describe, that '(...) one can say that there is no scarcity in security metrics. The challenge is to find one's way, to select those of the measurements that impact the business' ([9, p. 116]). They elaborate two main issues one encounters when a security metric should be described: *Selecting the measures* and *Ensuring accuracy of measures*.

The selection of an appropriate measure for security threats has been addressed in section 2.2. Assuring accuracy is a different problem that closely relates to the threat rating problem, but is not the same. Accuracy addresses the confidence that can be put on a measurement. To achieve a high confidence the NIST published a guideline with factors that should be therefore considered in [31]:

- Measures must yield quantifiable information (percentages, averages, and numbers);
- Data that supports the measures needs to be readily obtainable;
- Only repeatable information security processes should be considered for measurement; and
- Measures must be useful for tracking performance and directing resources.

Besides the listing of this considerations, the NIST specification does not name any concrete security metrics to apply in a generic or even in a specific case. Houngbo and Hounsou also do not provide a method to define a metric based on single threat measures in [9]. Shaikh and Haider compiled a comparison of eleven different security evaluation systems in [32]. Yet, none of the introduced systems provides a mechanism to calculate the desired overall system security rating.

Formal Security Metrics

After the review of the presented security metrics it becomes clear that a more formal approach is required to find a suitable metric for system security evaluation. Such an approach is presented by Krautsevich, Martinelli and Yautsiukhin in [33] as well as by Wang in [34]. Both methods are described and evaluated here.

Krautsevich et al. define an abstract security system that can be used for the presented use-case. A system state S describes the state of the assets of the computing system. Let Q be a set of all possible system states with r and q denoting arbitrary exemplary system states $(r, q) \in Q$. A measurement function is required to assign a real value (the overall system security rating) to a system state in Q: $M : Q \to \mathbb{R}$.

With this definition Krautsevich et al. define the relation $r \sim_s q$ as r is equally secure q and $r \succ_s q$ as r is more secure as q. With this definition it is possible to state the representativeness of M if the following relation holds:

$$\begin{aligned} \forall r, q \in Q \qquad & (r \sim_s q) \Leftrightarrow (M(r) = M(q)) \quad \text{and} \\ & ((r \succ_s q) \Leftrightarrow (M(r) > M(q)) \quad \text{xor} \\ & (r \succ_s q) \Leftrightarrow (M(r) < M(q))) \end{aligned} \tag{1}$$

Equation 1 shows that M must be monotonic to suffice the condition. With this it is possible to define the security rating metric R as a function that describes the distance between two members of Q ($R : Q \times Q \to \mathbb{R}$) and satisfies the following properties.

1. $R(r, q) \geqslant 0 \quad \forall r, q \in Q$ (positivity)
   The distance between two states of the system can never be negative. It should be emphasized, that R does not measure the change in overall system security.
2. $R(r, q) = 0 \iff r = q, \forall r, q \in Q$ (identity).
3. $R(r, q) = R(r, q) \quad \forall r, q \in Q$ (symmetry).
4. $R(r, p) \leqslant R(r, q) + R(p, q) \quad \forall r, q, p \in Q$ (triangle inequality).

To finally define the metrics based on the established system a function C is defined that decides if a system state contains compromised assets ($\Omega : Q \to \text{True} \,|\, \text{False}$). For a system S let Γ be a set of all possible actions on the computing system. An adversary will try to perform a chain of actions $\gamma \in \Gamma$ that will compromise one or more assets of the system. Therefore Krautsevich et al. define that a system is perfectly secure if new system states S' that are reached by arbitrary actions γ are all considered not compromised [33]:

$$\forall \gamma \in \Gamma \qquad S \xrightarrow{\gamma} S' \implies \Omega(S') = \text{False} \tag{2}$$

The following metrics are a selection of the ones presented in [33].

NUMBER OF ATTACKS (N_{att}) simply counts the number attacks that compromised the system ($\Omega(S) = \text{True}$). A result from this definition is the criterion, that a system is considered more secure if there are less successful attacks on it .
$r \succ_s q$ iff $N_{att}(r) < N_{att}(q)$

MINIMAL COST OF ATTACK (C_{att}^{min}) describes the minimum effort an attacker has to take to exploit a system. If an action γ can be used to take over the system its costs are defined as C_{att}. The minimal cost of attack is then defined as action with minimal costs that results in the compromisation of the system:

$C_{att}^{min}(S) = \min\{C_{att}(\gamma) | S \xrightarrow{\gamma} S' \implies \Omega(S') = \text{True}\}$
Based on this, a system is considered more secure if the minimal costs of an attack are higher than for another system.
$r \succ_s q$ iff $C_{att}^{min}(r) > C_{att}^{min}(q)$

MINIMAL COST FOR REDUCTION OF ATTACK (C_{red}) are the minimal costs that are required to transform the system S into a state S∗ that does no longer allow exploitation through *any* attack. Under this metric a system is more secure than another if these costs are lower.
$r \succ_s q$ iff $C_{red}(r) < C_{red}(q)$

SHORTEST LENGTH OF ATTACK (L^{min}) characterises the length of the shortest action that leads to a compromisation of the systems assets. The length of an action is defined as the number of steps that it takes to execute this action. Systems that require longer attack paths are considered more secure than other ones.
$r \succ_s q$ iff $L^{min}(r) > L^{min}(q)$

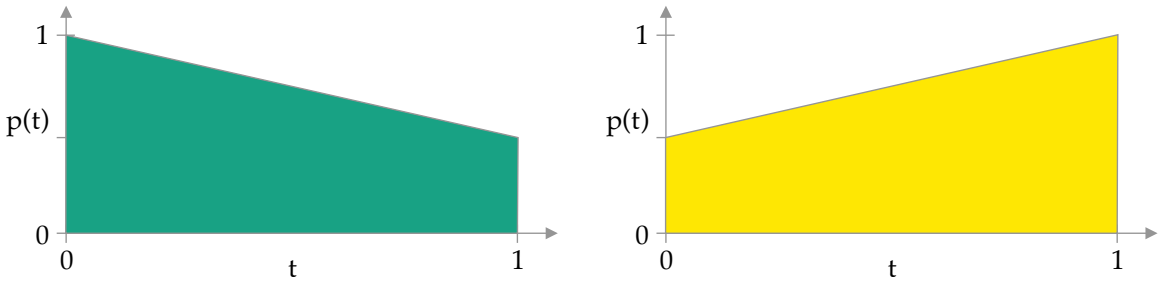
MAXIMAL PROBABILITY OF ATTACK (P^{max}) and

OVERALL PROBABILITY OF ATTACK (P^{suc}) are probability based characterisations of a system. The probability of an attack is defined as the product of the probabilities of the steps that make up the attack. P^{max} denotes the maximum of all probability values for successful attacks. Because this metric is not affected if any other security issue is mitigated besides the most probable one, the overall probability of attack (P^{suc}) is more commonly used. It is defined as the product of all probabilities of successful attacks against the system. In both metrics a lesser probability indicates a more secure system: $A \succ_s B$ iff $P^{max}(A) < P^{max}(B)$ and $A \succ_s B$ iff $P^{suc}(A) < P^{suc}(B)$

Besides the described metrics above, an *Attack surface metric* and a *Percentage of compliance* metric is introduced in [33]. However these two does not comply to the equations 1 - 2 and therefore do not allow to evaluate the change in security of two systems states.

From the other metrices only N_{att} and P^{suc} allow distinct checks of two system states if one is "as secure as" or "more secure than" the other. The remaining metrices can only answer checks for "equally secure or if one is more secure". This is because these metrices do not allow distinguishing between higher or equal security based on the number of successful compromising actions. The mathematical reasoning behind this criterion is found in [33, p. 163]. Because of the requirement to perform a rating of costs introduced by the employment of selected security technologies the C_{red} metric is of interest for this work.

Another different metric is proposed by Wang in [34] that tries to consider the change in security of an asset over time. Given a security feature $f(t)$ at a given time t, the security metric S_f for a time interval $[t_1, t_2]$ is a real number in $[0, 1]$ determined with the following formula:



(a) Security feature function $f(t) = 1 - 0,5t$

(b) Security feature function $f(t) = 0,5 + 0,5t$

Figure 8: Example security feature functions that model the decrease and increase of a security feature over time resulting in the same S_f value of 0.75

$$S_f = \frac{\int_{t_1}^{t_2} f(t)dt}{t_2 - t_1} \tag{3}$$

However this metric has the major disadvantage that it does not reflect an increase or decrease of security of the investigated feature over time, which is essential for the use-case investigated here. An untrusted application is executed and the system/or the application should be monitored; constantly evaluated and decreases or increases in security should be detected. If the formula in 3 is used, a decline and incline might result in the same value like shown in figure 8.

SUMMARY

This chapter provided definitions for system security, assets, threats, attacks and resources that will be used throughout this thesis. It was shown that the definition of security is difficult because of its intangible nature. Anyhow this becomes possible if system security is defined by the assets of the system. With the help of the ontology of Herzog et al. the asset concepts and dependent classes like threats and attacks could be defined [6]. Additionally threats of interest for this theses were selected based on related literature.

To proof the effectiveness of security measurements, the definition of threats is not enough. To evaluate abstract threat classes and their realised attacks numerical ratings are required. Therefore different methods were investigated that enable the assignment of real values to arbitrary threats based on a deterministic function and/ or objective criteria. In this work the CVSS is used to assign numerical values to the defined threads [18].

Finally the formal metrics for *Number of attacks* and *Minimal cost for reduction of attack* were selected to provide a system security rating of computer systems states at distinct times. They also produce comparable ratings that actually allow to perform a mathematically founded statement about the security of the system.

3 SYSTEM CALL INTERFACE

The previous chapter provided a definition of the essential security terms: assets, threat and attack. It also selected metrices to rate threats and the overall system security. This chapter introduces the system call interface of Linux-based operating systems. This interface is mandatory for applications to access *resources* that are managed by the operating system.

The chosen approach to investigate untrusted applications for the utilisation of this system call interface requires a definition of the relationship between *assets* and *resources*. The first section of this chapter groups the different resources into categories and establishes this relationship to the asset classes described in 2.1.1.

The next section introduces the system call interface of Linux-based operating systems for the x86-64 reference architecture. This gives an overview about the design and overall functionality of this interface.

The last section presents the challenges when it comes to the supervision or investigation of actions that are executed via this interface. It also investigates reference systems that utilise the system call interface for suitable approaches as well as their limitations.

3.1 RESOURCES AND ASSETS

A resource in this work describes an object that is managed by the operating system. The operating system mediates access to resources through the system call interface described in 3.2.

Resources can be grouped into different categories based on the description of operating system concepts, computer hardware history and operating system tasks described by Tanenbaum [35].

The following classes are used throughout this work to categorise resources:

FILESYSTEM RESOURCES (FS_1) Resources that are located in filesystems like files, directories, links etc.

FILESYSTEMS (FS_2) A filesystem is the higher organisation structure that holds information about resources described for class FS_1. Filesystems reflect different storage technologies (hard disk, solid state, in-memory, etc.) as well as organisation strategies (journaling and indexing, physical and logical layout, filesystem resource meta-data etc.).

MEMORY (MEM) Modern operating system abstract physical memory into virtual memory resources to protect, separate and possibly relocate applications.

PROCESS/ CPU MANAGEMENT/ IPC (CPU) With the availability of multi-processor and multi-process operating systems, the computing time resource has to be managed by the operating system too. Additionally, due to the possibility of multiple active applications at the (apparently) same time, the communication between them and the management of these shared communication is a managed resource too.

NETWORK (NET) The network resource group summarise all resources that relate to the communication of two parties over a (potentially package switched) network. This includes the establishment and control of connections as well as the management of client and server-sockets that reflect communication endpoints handled by the operating system.

DEVICE MANAGEMENT (DEV) Whereas resources like files, directories and memory are presented to applications independently of the hardware used for their provision, these hardware components are also resources managed by the operating system. The access, management and data exchange with hardware resources must for this reason also be mediated by the operating system.

TIME Similar to the management of processing time in resource class CPU is the management of actual time. Although date, time and timezone are simple resources, their importance for the system is high. Mishandling those can inflict time-critical operations like the validation of temporary access token or the validity of certificates.

PERMISSION MANAGEMENT (ACCESS) Multi-user operating systems require the enforcement of an access and permission model. This model is realised via the user- and group-based permission model in POSIX-compliant operating systems. Besides the enforcement of access rules, the operating system provides mechanisms to query and manipulate permissions on items it manages. As a result, the possession of a permission can be considered a resource too.

KERNEL MANAGEMENT (KERN) Finally the operating system itself and its kernel are resources present in the system. An application can interact with them e.g. to configure runtime behaviour or load extension features.

With the more comprehensive resource classes it is possible to associate assets to them. Although a one-to-one association is desired, some assets belong to more than one resource class. The association of assets to resource classes is done based on the character of the asset and the resources reflected by the class. An excerpt of this association can be seen in table 7. The full table is found in appendix C.

Through the association of assets to their corresponding resource classes, the link to technologies that support resource based supervision is established. Hence the usage of

Resource Class	Assets
FS_1	Stationary Data, Data on Non-Volatile Media, File, Program Source Code File, Backup File, Database Data File, Configuration File, Program File

Table 7: Excerpt of the association of resource classes with asset concepts from [6]

low level resource access analysis to associate it with assets through the usage of resource classes is feasible and used in this work.

Although this approach is able to detect many different attacks, it is not suited for threats that target assets that do not involve the system call interface. The following attacks are examples for such threats which are beyond the scope of this work.

MEMORY CORRUPTION ATTACKS Manipulation of memory owned by the application which does not involve access permission changes do not result in system calls. An attacker might override readable, writeable or executable parts in memory with malicious instructions and force the application to execute these parts. However the configuration of such vulnerable memory areas is rare and subsequent malicious actions can be detected.

HARDWARE MEMORY ATTACKS Attacks against hardware related issues like the presented *Rowhammer* attack by Kim et. al [36] are beyond the scope of system call supervision. The presented approach assumes that there are no hardware related side-effects that leads to information disclosure or manipulation.

HARDWARE PROCESSOR SIDE-CHANNEL ATTACKS The exploitation of processor features like predictive branch execution as shown by Kocher et al. in [37] also bypasses the system call API because such hardware memory attacks cache reads do not issues any system calls.

APPLICATION BEHAVIOUR MANIPULATION Attacks against the execution paths of the application are hard to detect during code analysis and impossible to prevent through the envisioned sandbox system. An attacker might manipulate calculation results, weaken secure communication protocols or cause an application to never terminate to consume resources for an infinite amount of time.

3.2 LINUX SYSTEM CALL INTERFACE

Conceptually, access to resources requires administrative privileges since it involves e.g. access to hardware components like hard drives or network adapters. As a result, these

syscall number	arg 1	arg 2	arg 3	arg 4	arg 5	arg 6
`RAX`	`RDI`	`RSI`	`RDX`	`R10`	`R8`	`R9`

Table 8: x86-64 syscall invocation interface [39, p. 147]

resources are only accessible through the operating system which possesses these privileges. However modern operating systems run applications in an unprivileged *user-mode*, whereas the operating system components run in *kernel-mode*[35]. Therefore, to enable user-mode applications to access resources, a well-defined procedure to transfer control to the operating system, perform the desired action, and return the result is required.

The system-call interface specifies this procedure and is described by Tanenbaum in [35, p. 50] and shown in figure 9. An application in user-mode pushes all required parameters for the operation onto the stack (step 1. to 3.) and performs the system call usually via a system library function (step 4.). The library prepares the processor registers with the stack-passed data and places the integer identifier of the desired system call to execute into the identifying register (step 5.). To transfer control to the operating system, a so-called `trap` is invoked (step 6.). Modern 64-Bit x86 architectures use `SYSENTER/SYSEXIT` (Intel) or `SYSCALL/SYSRET` (AMD) machine code instructions for this [38]. A privilege context switch occurs and the application now operates in kernel-mode. A lookup is performed to select the correct handler for the passed system call identifier (step 7.). If such a handler is found, it is executed (step 8.). Once this process finishes, machine code instructions `SYSEXIT` or `SYSRET` return control to the library function and the privileges are dropped as the application flow returns back into user-mode (step 9.). Finally the library function returns the result to the user application (step 10.) and execution continues (step 11.). Analogous to the distinction between user-mode and kernel-mode for instruction execution is the distinction of memory segments into user space and kernel space as shown in the figure.

The invocation interface utilised in step 5. is very basic and supports the transfer of the system call identifier together with up to six arguments. The system call identifier is an integer number that uniquely identifies each system call. The passed arguments are used to pass further information to the system call handler (e.g. a file descriptor number or the address of a memory region to map). When executed on a 64-bit x86 architecture, the system call identifier is placed in the `RAX` register whereas arguments are stored in the registers shown in table 8. This standardised API allows the syscall handler to read passed arguments as well as an analysis to investigate these parameters. This is essential for the technologies introduced in section 3.3.

The result to indicate the success of failure of the system call handler passed in step 9. is again an integer number passed via the `RAX` register. The system call issuer can evaluate it to determine if the intended resource access was executed as desired [39, p. 148]. Due to

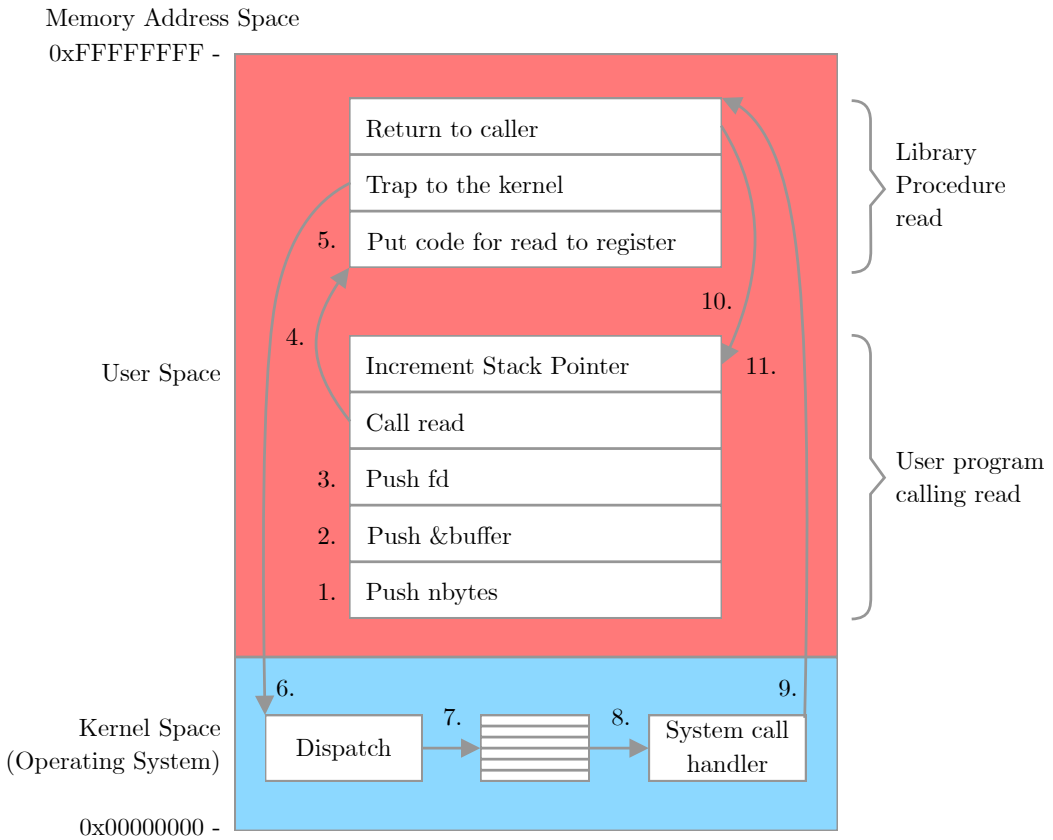


Figure 9: The eleven steps in making the system call `read(fd, buffer, nbytes)` [35, p. 52]

the register size of 64-bit in 64-bit processors, system call arguments are limited to values of the same size. For data structures that require more space, pointers to user-memory are passed as arguments. For example a string that specifies the path of a file that should be accessed with the `open` syscall is passed as such a pointer. This mechanism is also used to pass more complex return values back to the system call issuer. A memory address passed as argument can be interpreted by the kernel as the place to write result data to. Since this address is in user memory, it can afterwards be accessed by the issuer.

3.3 SUPERVISION AND INVESTIGATION TECHNIQUES

The strict definition of the system call interface makes it possible to investigate system calls issued at runtime. Whenever a system call occurs, its identifier can be evaluated as described above and its type as well as arguments passed in the defined registers can be read. This allows an investigation for resources accessed by the application.

Technology	Description
`ptrace()`	System call to register a monitoring process running in user-mode to supervise system call issuance of a monitored process.
`Kprobes` [41]	Mechanism that allows to break arbitrary kernel routines (e.g. the `syscall` routine) and investigate state of processor, memory, etc.
`Utrace`	`Utrace` was an attempt to implement a more sophisticated interface into the Linux kernel to take over tracing tasks from `ptrace`. However the interface was not integrated into the mainline kernel [42].
`SystemTap` [43]	With an own `awk`-like scripting language `SystemTap` allows users to define *probes* that are inserted at desired points into kernel routines. If the kernel reaches such a probe the associated script is executed.
`strace` [44]	Command line tool that utilizes `ptrace` to show issued syscalls during runtime.

Table 9: Linux- and Unix-system based system call tracing technologies [45]

The Linux kernel supports several interfaces to supervise issued system calls. Table 9 displays different technologies and lists their availability [40]. The most common interface is the `ptrace` interface. It allows a monitoring process to attach itself to the process or thread that should be supervised (monitored process). From this point on, for each system call that is issued, the operating system informs the monitoring process before it starts the dispatching process (see step 6. in figure 9) and after it finishes the syscall handler (see step 9. in figure 9). The monitoring process is therefore able to evaluate the system call identifier and passed arguments at these points in time. It can furthermore perform arbitrary actions like filtering dedicated system calls or additional activity logging. Figure 10 illustrates the attachment of the monitoring process to the monitored process and shows the interception of an example syscall.

The monitoring process is also able to access the memory of the monitored process through the `ptrace` interface. It can therefore follow memory pointer addresses specified in system call arguments. This is required to collect e.g. information about files access with the `open` system call where the path of the desired file is passed as a pointer. As described in 3.3, the usage of pointers to user memory allows the passing of complex data types that are otherwise to big for the processor registers. On the other hand, it makes it *impossible* to built security mechanisms based on system call supervision. This is shown by [47] and [48]. They prove that user-controlled memory can't be used for mutually independent security checks and resource access. If performed this way, an attacker is able to time a manipulation of the user memory to change its contents exactly after the security check has succeeded, yet before the actual resource access is executed by

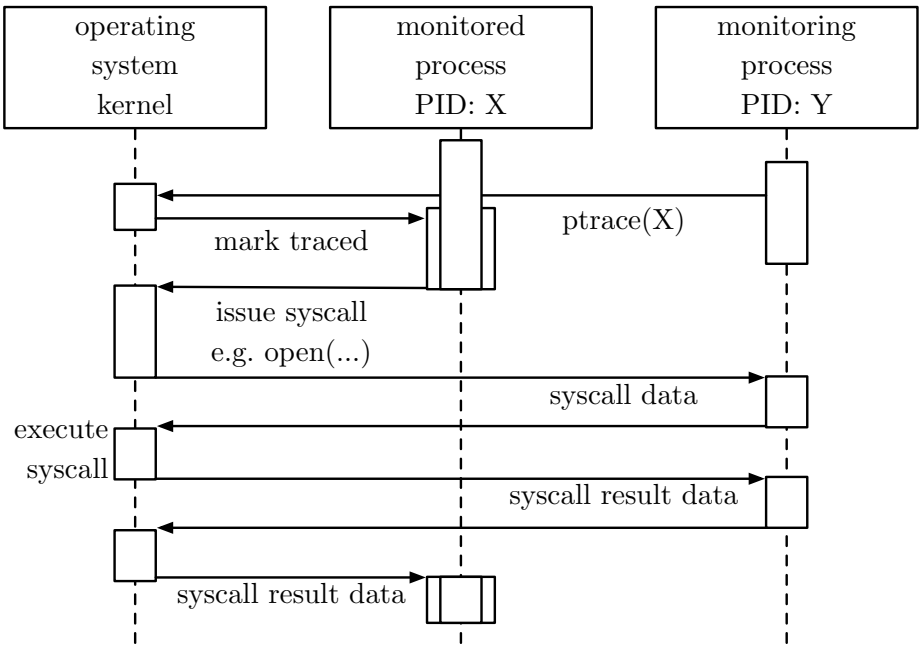


Figure 10: Activity diagram of a monitoring process that starts supervision and syscall interception of a process through the `ptrace` interface [46].

the kernel. This process is illustrated in figure 11. The only solution to overcome this issue is an atomic uninterruptible operation that performs security check and resource access. However such a system call-based interface is not available through a stable kernel ABI as of today.

Finally, besides the problems due to timed manipulation of user memory, supervising system calls with `ptrace` introduces a large management overhead into the system call execution process. As shown in [46], [47] and [49] the overhead introduced increases execution time by up to 420%. This is due to the additional full process context switches required by the kernel when it switches execution from the monitored process to the monitoring process (see fig. 10). The `ptrace` interface requires one context switch to the monitoring process and another one when switching back to the kernel/monitored process. Because of the two-times invocation of the monitoring process in the system call execution process, this results in four additional context switches per system call.

SUMMARY

This chapter established the required relationship between low-level operating system managed resources and abstract assets according to the definition specified in 2.1.1. Re-

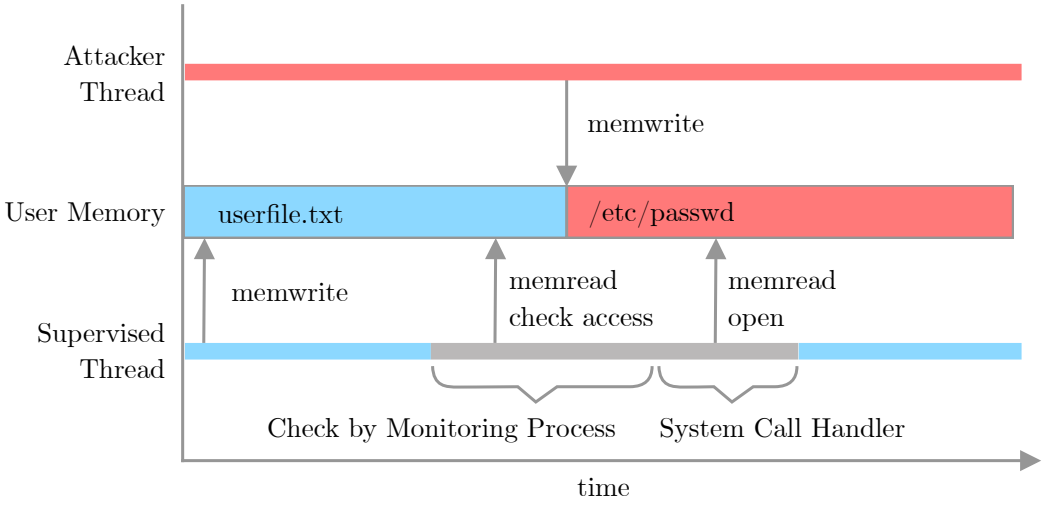


Figure 11: Example attack on non-atomic security mechanisms where a timed manipulation of user memory circumvents checks. When the monitoring process checks the path of the filename that should be accessed, a valid value is read (`userfile.txt`). Before the system call handler is executed, the attacker changes this value to the normally inaccessible filename (`/etc/passwd`).

source classes are used in follow-up chapters to investigate how application analysis mechanisms can detect resource access and how sandbox mechanism can protect them.

The described specifics of the system call interface are important for the application analysis mechanisms. Resource access should be detected and all relevant resources are accessed through this interface. Consecutively the detection of system call interface utilisation allows the application analysis to deduce a resource access.

Finally this chapter illustrated the challenges for security technologies based on system call supervision and interception. Mechanisms based on these technologies operate while an untrusted application is executed. Nonetheless, since there is no mechanism that allows appropriate protection of resources while maintaining system performance, the solely utilisation of these mechanisms is discouraged. Nonetheless, since this work focuses on the detection of resource access before the untrusted application is executed, these technologies can still be utilised in a comprehensive sandbox approach.

4 APPLICATION EVALUATION

The investigation of the system call interface has shown that it is possible to detect resource access through this interface. Based on these results, this chapter introduces the techniques available to investigate untrusted applications for their system call interface utilisation.

In the first section, the *Executable and Linkable Format* is introduced. Since this is the standard file format for executable applications on Linux-based systems, knowledge about the file structure and contained information is useful for the analysis. The section will also introduce the basic mechanisms of application loading and runtime linking of dynamic libraries.

The context of this work enables an investigation of the untrusted application only based on its compiled form. The specifics of this machine code format are introduced in the second section. Additionally, an overview of techniques that can be applied to transfer the binary representation into an easier processable form is given.

With the knowledge about the file format for applications and an interpretation mechanism for machine code, an analysis of the program flow becomes available. Techniques for *static* and *dynamic* code analysis are introduced in the last section to perform this analysis.

4.1 EXECUTABLE AND LINKABLE FORMAT

Applications that are created for Linux-based operating systems must be distributed in the *Executable and Linkable Format* (ELF) [45]. The ELF describes the structure for *executables*, *dynamic libraries* (or sometimes named shared objects) and *relocatable binaries*. If an application is compiled from source code to machine code, the *linker* collects the generated machine code, validates references to dynamic libraries in the system and generates an executable that conforms to the ELF. The linker also adds all required information to enable the operating system to run the executable.

An ELF compliant file consists of a main file header with basic structural information. Important fields are displayed and described in table 10. The file is structured by two different concepts. *Segments* describe memory segments that are required during runtime execution of an application. The file header refers to the location of segments headers in the file via the field `e_phoff`. The total number of segments is stored in the file header field `e_phnum`. Each segment is described by its dedicated header. This segment header contains its type, the offset of the segment content offset into the file, its size and desired location

Field	Size	Content
`e_ident`	4 Byte	4-byte magic number to identify ELF formatted file
`e_type`	2 Byte	ELF filetype (none, executable, library, relocatable)
`e_machine`	2 Byte	Architecture this ELF file was created for
`e_entry`	8 Byte	Application execution start address
`e_phoff`	8 Byte	Offset in bytes to segment headers
`e_phnum`	2 Byte	Number of segment headers in the file
`e_shoff`	8 Byte	Offset in bytes to section headers
`e_shnum`	2 Byte	Number of section headers in the file

Table 10: Relevant 64-Bit ELF file header fields [45]

in memory during runtime. It also holds information about permissions that should be assigned to the segment in memory (readable, writeable, executable). Segments might overlap as shown in figure 12.

Sections are used beside segments to further structure the file. Similarly, sections are also described by their headers, which usually are placed at the end of the file. Their location is again referenced in the file header by the `e_shoff` field. The number of sections is given in `e_shnum`. Sections do not overlap and provide a more fine-grained structure of the file than segments. Since they are not required to execute the application, section headers can be absent. However, if an application is linked to dynamic libraries, the required linking information are stored in dedicated sections. Therefore sections are mandatory for dynamically linked executables.

Relevant sections and their purpose during application loading and execution are shown in table 11.

4.1.1 *Application Loading*

If an application is about to run, the *Linux loader* must set up the memory layout required for execution. This process starts with the application to execute and is visualised in figure 12. The loader inspects the application if it is a statically or dynamically linked executable. If the application is dynamically linked, the loader has to find all libraries in the system that are referenced in the `.dynamic` section of the executable. Since dynamic libraries can reference other dynamic libraries, this library collection is a recursive process. This step is skipped for statically linked executables, because they define no dependencies to dynamic libraries.

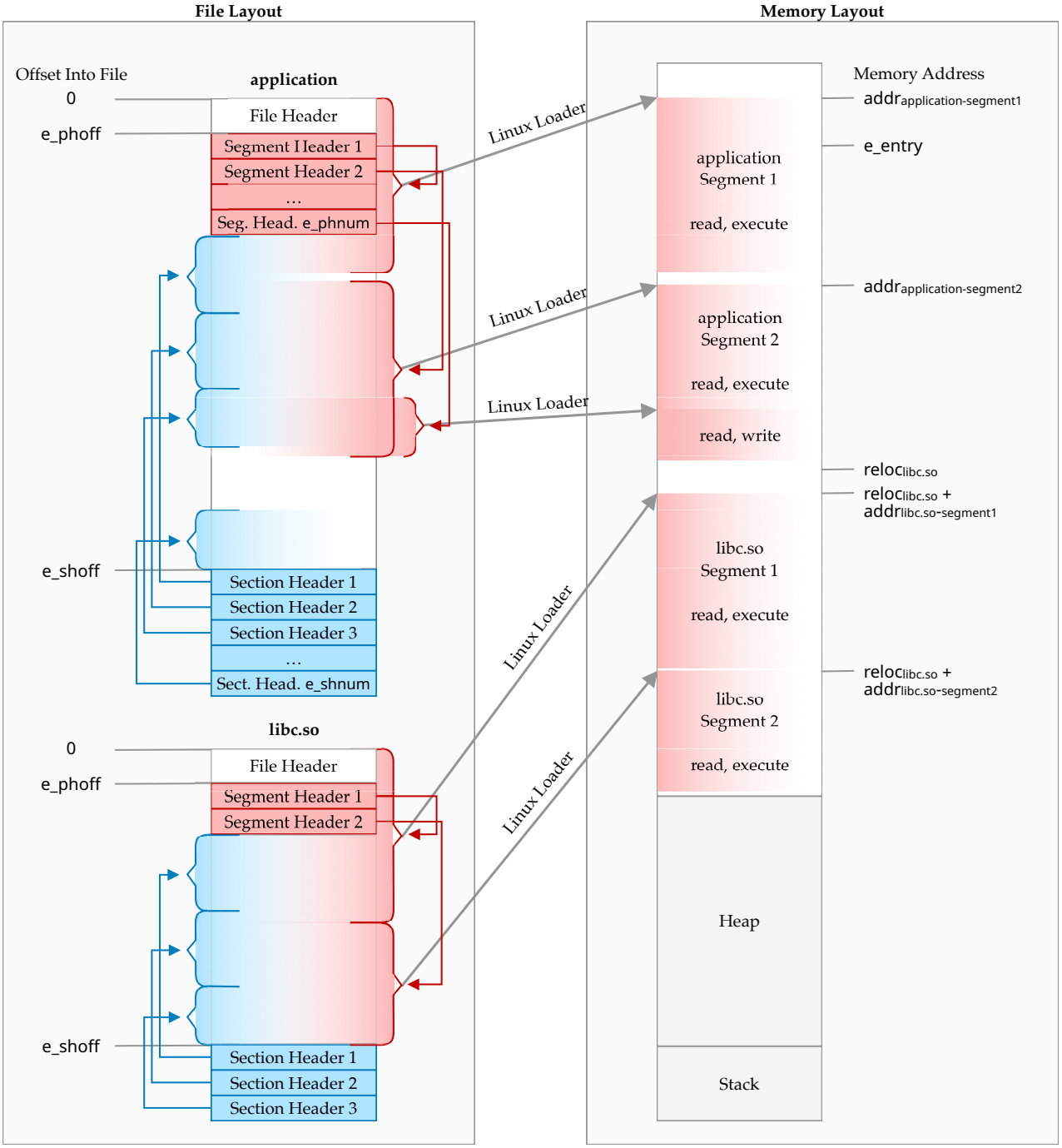


Figure 12: File and Memory layout of two ELF compliant files *application* and *libc.so*. The left side shows the file structure and the location of the fixed sized file header, segment headers (red) and section headers (blue), as well as their references to dynamically sized content areas. The right side shows a schematic view of a memory layout created by the Linux loader for the two input files. Segments from *application* get loaded to their desired absolute address ($\texttt{addr}_{\texttt{application-segment1}}$, $\texttt{addr}_{\texttt{application-segment2}}$). The segments of *libc.so* are relocated based on the free address $\texttt{reloc}_{\texttt{libc.so}}$ determined by the loader.

Name	Purpose
`.text`	Contains the machine code of the application.
`.init` and `.fini`	Sections with executable instructions for application initialisation and termination.
`.data`	Initial and static variable values
`.symtab`	A table of all functions (called symbols) defined in the local file and referenced from dynamic libraries.
`.dynsym`	A table of only those symbols referenced from dynamic libraries.
`.dynamic`	Table with names of referenced libraries and their required versions
`.got`	*Global Offset Table* with information of symbol locations in the memory layout the Linux loader created.
`.plt`	*Procedure Linkage Table* that enables dynamic linking during runtime.
`.rel` and `.rela`	List of addresses of values that need to be updated once the relocation process finished.
`.debug`	Debug information

Table 11: Relevant sections specified in the ELF standard [45]

For each loadable segment in the executable, the loader places it to the desired position. Overlapping segments are used to adjust memory permissions. This is utilised to enable the load of large chunks of the file into memory at once and adjust memory permissions of a part of these chunks afterwards. See *application segment 2* in figure 12, where the load of the segment and memory permission switch is visualised.

The executable specifies desired addresses for its segments in virtual memory. Dynamic libraries on the other hand are *position independent*. This means that their machine code is oblivious of its position in memory at runtime. However, this requires the loader to determine an appropriate position in memory for each dynamic library and set it up there. This process is called *relocation*. For each dynamic library the loader picks a base address (see $\texttt{reloc}_{\texttt{libc.so}}$ in figure 12) and places its segments based on this address. After this process is finished the loader updates all values specified in `.rel` and `.rela` according to the base address of the file that is now known after placing it into memory.

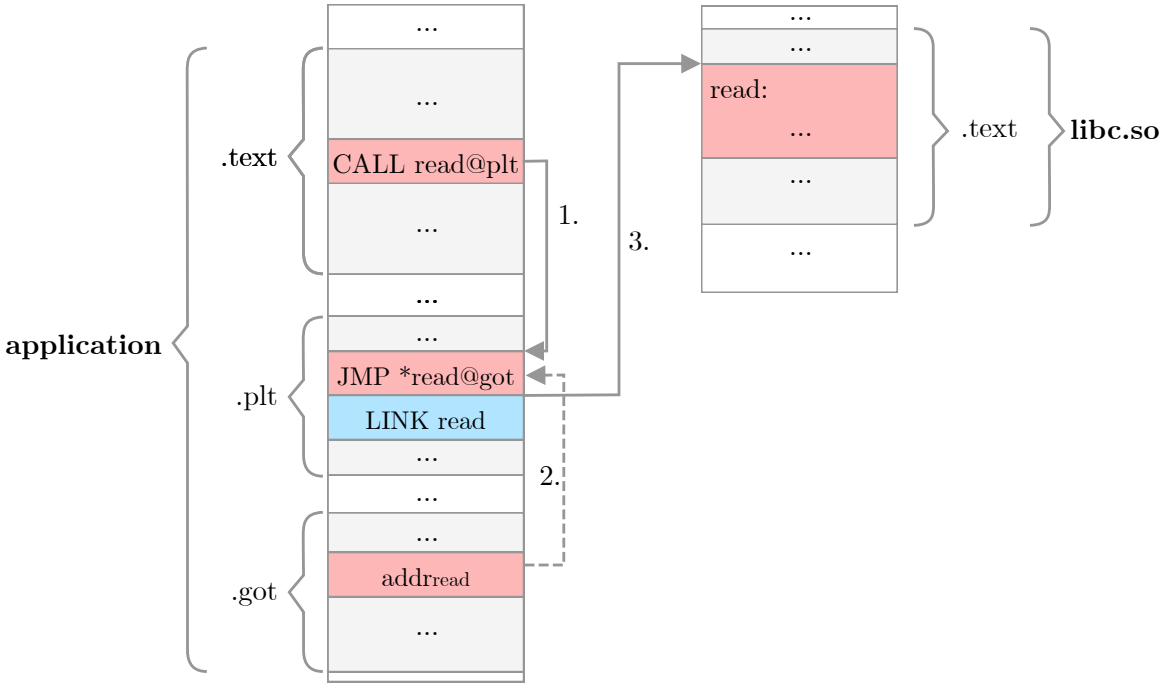


Figure 13: Schematic control flow of the invocation of the dynamically linked symbol `read` from `libc.so`. The reference from the machine code section `.text` leads to the `.plt` section (step 1.). The relocated address of `read` is read from the `.got` (step 2.) and the jump is executed to start the symbol execution (step 3.). If the symbol is not yet linked the value in `.got` contains the address of the linkage routine shown as *LINK read*.

4.1.2 *Symbol Linkage*

The linkage of symbols (basically exported functions) in the dynamic libraries to the invocations in the executable is normally done at application runtime (but can be forced to be done at application load time via environment variables). If the application invokes a symbol from a dynamic library, the compiler implements this as a call to the process linkage table (PLT, see figure 13, step 1). The machine code in the `.plt` tries to read the address of the relocated symbol in the dynamic library from the global offset table (step 2.). If the symbol was already located, its address is stored there and the jump to the symbol is performed (step 3.). If the application is executed with dynamic linking at runtime, it is possible that the symbol has not yet been located. In this case the address in the `.got` points to the location of the linkage routine that will search for the symbol in the dynamic libraries. Once found, the address in the `.got` is updated and consecutive calls to this symbol can be executed directly as described by step 1.-3 in figure 13.

4.2 MACHINE CODE INTERPRETATION

Besides the information required to set up the memory layout for an application, the most important part in the ELF is the executable machine code. Machine code is the form of executable code that is ready to be run on a certain processor. Unlike low-level languages like assembly or high-level programming languages, machine code is highly processor dependent[1]. It consists of instructions that are coded in binary form specifically tailored for an architecture (like *arm*, *x86*, *ppc*, *ia64*, etc.).

For high-level languages, a compiler translates source code into assembly code. Afterwards the assembler translates this code into machine code instructions for the desired architecture. Processors might offer some kind of compatibility mode (e.g. to execute x86 32-Bit code on x86 64-Bit processors). Anyhow in general, processor architectures are incompatible to each other.

Machine code instructions can easily be converted back into assembly code, since their relationship is bijective. A disassembler with the knowledge about the used processor architecture can rebuild the assembly code instruction for a machine code one. An x86-assembler can translate the code in listing 2 to the machine code in listing 3, whereas a x86-disassembler can reverse this action.

```
1 mov    %edi,%eax
2 mov    $0xcccccccd,%ecx
3 imul   %rax,%rcx
4 shr    $0x26,%rcx
5 imul   $0x50,%ecx,%eax
6 sub    %eax,%edi
7 mov    %edi,%eax
8 retq
```

Listing 2: Example assembly code

```
1 89 f8
2 b9 cd cc cc cc
3 48 0f af c8
4 48 c1 e9 26
5 6b c1 50
6 29 c7
7 89 f8
8 c3
```

Listing 3: x86 64-Bit machine code for listing 2

The following properties illustrate the characteristics of machine code that have to be considered if it is used for application analysis.

TYPELESS Machine code and reconstructed assembly code holds no type information. Registers and memory locations can be accessed and manipulated with arithmetical, logical or bitwise operations. These operations are performed on numbers and no type information is stored in the machine code. To reconstruct application behaviour, all types of variables or complex structures must be inferred [50].

1 Applications written in assembly code might also be processor dependent if they use architecture specific registers or instructions. However it is also possible to create architecture-agnostic programs.

REDUCED INSTRUCTION SET While some assembly languages offer loops or conditionals like `if/else` or `switch`, there are no such instructions that reflect this behaviour on machine code level. Instead these control flow structures are translated to comparisons and conditional jumps [51]. While this is equally powerful in solving algorithmic problems, it complicates application behaviour reconstruction.

SUBROUTINE IDENTIFICATION Machine code is not executed in a linear fashion. Jump instructions affect the *instruction pointer register* (IP) that stores the address of the upcoming instruction of a running application. A jump causes the program to continue execution at a different location in memory. Such jump instruction may be conditional. Another method of integrating subroutines are `CALL/RET` constructs that indicate standalone functions. One task for machine code interpretation is to find such functions with their code boundaries. This can be difficult in binaries, where symbol boundary information has been removed e.g. by the compiler [52].

If high-level language source code is compiled, unsupported features like loops, objects and classes, inheritance or structured memory are translated to multiple machine code instructions. The original code structure and type annotations are lost during this translation. For this reason, the reconstruction of the originating source code is difficult.

Beside the loss of information due to the described machine code characteristics, different compiler interpret the same source code instructions in different ways. Therefore two different compilers generate two versions of assembly code from the same source code. Additionally, compiler perform optimisations on the processed code to remove unused instructions or to optimize instructions for the designated architecture they build the application for. This can be seen in listing 4, 5 and 6.

```
1 int f(int p) {
2         return p % 80;
3 }
```

Listing 4: Example `C` function

```
1 int f(int a) {
2         int x = a / 80;
3         x = x * 80;
4         return a - x;
5 }
```

Listing 5: Reconstructed function from assembly code 6

```
1  mov    %edi,%eax
2  mov    $0xcccccccd,%edx
3  mul    %edx
4  mov    %edx,%eax
5  shr    $0x6,%eax
6  lea    (%rax,%rax,4),%edx
7  mov    %edi,%eax
8  shl    $0x4,%edx
9  sub    %edx,%eax
10 retq
```

Listing 6: Assembly generated by `gcc` for listing 4

Listing 4 shows the original `C` code for function `f` and listing 6 the result of a compilation with the `gcc` compiler with maximum optimisation requested. Listing 5 shows

the reinterpreted assembly code which is clearly different from listing 4, but behaves the same. It should also be noted, that the division a/80 in listing 5 is done in machine code as a multiplication and shift instruction (line 2-5, listing 6) which is more efficient on 64-Bit x86 architectures than the division instruction.

High-level language source code can be analysed with various techniques to detect errors or unwanted behaviour. Nonetheless, as described above, the reconstruction of originating source code from the machine code is hard. This problem is discussed in the literature e.g. in [53] and [50]. It has also been shown in [54] that the source code reconstruction is an NP-hard problem.

For basic reconstruction of assembly and machine code there are tools available like e.g. *IDA Pro* [55], *OllyDbg* [56], *gdb* [57], *Binary Ninja* [58], *radare* [59] or *Panopticon* [60]. Some of them also provide features like call-graph generation and limited static analysis as described in 4.3. All of the mentioned software-products support debugging of a running binary execution, which enables the inspection of memory and processor registers at arbitrary points in time as well as runtime code traversal.

Because of the focus of the system call interface usage in this work, it is important to detect this usage in the machine code. The required kernel trap, to initiate the dispatch of the system call in kernel space (see 3.2), is done with a single machine code instruction. As a result, filtering or search for these instructions is possible.

4.3 CODE ANALYSIS

4.3.1 *Static Code Analysis*

Static code analysis or static program analysis describes the investigation of source code or assembly code without actually executing it. It is performed to find errors already visible at this point in time (like possible divisions by zero or buffer overflows) or to do code audits. For small programs with known restricted classes of input data, it is even possible to formally prove that a software acts like it is described. Despite that, programming language features like scoped variables or pointers can introduce undecidable or uncomputable issues to the static source code analysis as shown by Landi in [61]. Furthermore is the formal proof of an application to be free of error not possible. This is because of the undecidable nature of the halting problem which is a generalisation of this proof. Nonetheless techniques are available that can be applied to find approximate solutions [62].

The goals for static code analysis when performed on machine code are the same as for source code. However they are harder to achieve due to the characteristics of machine code described earlier. One of the goals of static code analysis if to collect information about application behaviour to anticipate. The application behaviour describes possible sequences of instructions that might be executed during runtime.

Control Flow Graphs (CFG) are an approach to anticipate application behaviour and to reconstruct subroutines from the machine code and are described below. Another possibility besides graph-based solutions is to transfer the assembly code instructions to an intermediate representation (IR). A single assembly code instructions does not provide much information. The IR groups together logical sequences of instructions to make analysis easier [63–65]. Brumley et al. provide with their *Binary Analysis Platform* a tool that allows CFG construction and identifies function boundaries through IR [66].

The rebuild of type information from assembly code was already addressed in 1999 by Mycroft [67]. This process also involves the reconstruction of primitive and composite datatypes [50, 68].

Primitive types can be reconstructed as shown by Dolgova and Chernov in [69]. Their approach works for primitive `C` programming language types. It infers the orginal datatype through the investigation of assembly code instructions performed on a certain point in memory. Each variable that is located in the machine code is recognised to be of an arbitrary type at the start of the analysis. Their algorithm assigns different properties to the variable based on its *core* behaviour, *size* and *sign*. *Core* describes if the functions applied related to an `integer`, `pointer` or `float`. If e.g. the investigation of a type returned the properties core = integer, size = 8 and sign = signed it was a `long int` in the originating `C` application. `C` strings are reconstructed as pointers and single characters are equal to integer number of one byte size.

Complex type reconstruction is more difficult and was investigated with different approaches by Troshina et al in [68], van Emmerik in [70] or Lee et al in [71]. While all these approaches differ from each other in detail, they all utilize the fact, that complex types store their structural data in subsequent memory locations. This is visualized in figure 14, that shows the layout in stack memory of the `C` structure in listing 7.

```
1 struct s_user {
2 // signed integers
3 int uid; // offset +0
4 int gid; // offset +4
5 // unsigned integer
6 time_t last_login; // offset +8
7 // 16 character string
8 char[16] name; // offset +12
9 }
```

Listing 7: Complex datatype with four fields

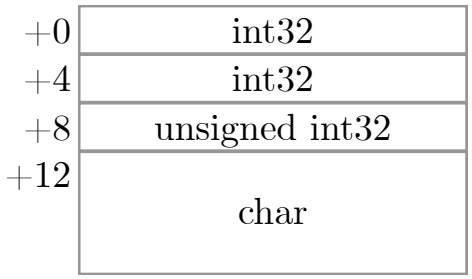


Figure 14: Stack layout from complex datatype 7 based on [68].

Buinevich et al. describe a toolkit for the detection of vulnerabilities in [53] that is capable of type inference and control flow reconstruction. It was tested to detect malicious behaviour with these techniques. Although the performed test was not able to detect or

prevent the exemplary unwanted write to a read-only section in memory, it shows that application behaviour based on machine code investigation is feasible.

Recently, machine learning has become popular for tasks that require the analysis of behaviour as well as pattern recognition. Due to the fact that common attacks on memory-related assets as well as programming mistakes are well known, there is current research that tries to detect them. Work like those of Omri et al. focus on pattern detection in source code as described in [72] to aid software development. There the goal is to find erroneous patterns that can lead to vulnerabilities in the result application. The work presented by Popov in [73] also employs machine learning but for machine code analysis. As a result, the target is no longer software quality improvement but rather malware detection. The approach presented uses a convolutional neural network that works based on the `word2vec` algorithm which is used to predict neighbouring words in natural language processing. The assumption made is that machine code can also be used as input, where the assembly instructions are used in place of the natural text. The presented classifier was trained with malicious and benign Windows-based applications and showed promising results. To produce acceptable result rates the required amount of programs to train the classifier has to be very large.

Christoderescu and Jha used a different approach than machine learning for the detection of attacks in executables with static analysis as presented in [74]. They found that common anti-virus scanners can be spoofed when virus code is obfuscated. Their proposed solution annotates machine code instructions to identify unused, irrelevant or equivalent parts and harmonizes them in an annotated control flow graph. To detect a malicious pattern a finite-state machine is employed that was built based on the vanilla virus. The annotated control flow graph of a suspicious application is then used as input for the finite-state machine. If the machine accepts the graph, it is concluded that the application is malicious.

Control and Data Flow Graphs

To reconstruct application behaviour from machine code, graph-based approaches can be used as described above. Static code analysis can build these graphs through the investigation of each instruction found in the assembly code [75]. Based on found sequences, conditions and jumps, a graph is generated where each node represents an instruction with its location in memory. Edges are generated between consecutive instructions, for jumps or function calls [76, p. 305]. The result graph displays the control flow (*Control Flow Graph*). It is directed and may contain cycles. Tools like IDA Pro or work presented in [77] and [64] can be used to generate such CFG.

Another type of graph focuses on the presentation the *application state* or *universe* [76, p. 306] which consists of the memory and register state. The graph shows the alteration of the application state throughout application execution. In this *Data Flow Graphs* (DFG)

each node represents an application state. Executed instruction that manipulate the application state are displayed as edges in the graph [78].

Applications for CFGs are the identification of paths with worst-case execution time (WCET) as shown in [79]. It can also be used to optimize the compiler result if the source code is still available like described in [80]. It is also possible to enforce integrity with a priori determined CFGs as shown in [81]. This can secure a system against code injection attacks.

To examine possible execution paths during static code analysis the conditionals at edges can be concatenated. Figure 15 shows a very simple CFG graph where the conditionals are show at the edges. To reach the node with the "`return 100`" instruction the code path condition can be formulated as $x \leqslant 0 \wedge x = 0$.

To analyse these formulas, to simplify them and to find unsolvable conditions in code paths *satisfiability solvers* (SAT solver) or more complex solvers used in *satisfiability modulo theory* (SMT solver) can be employed like the ones presented in [82–85]. The *Binary Analysis Platform* (BAP) integrates a conditional builder for traversed graph paths and employs SMT solver to produce comprehensive formulas [66]. Nonetheless, due to the fact that the SAT problem is proven to be NP-complete and the conditionals built along longer control-flow graph paths get complex very fast, it is not possible to analyse arbitrarily sized applications this way.

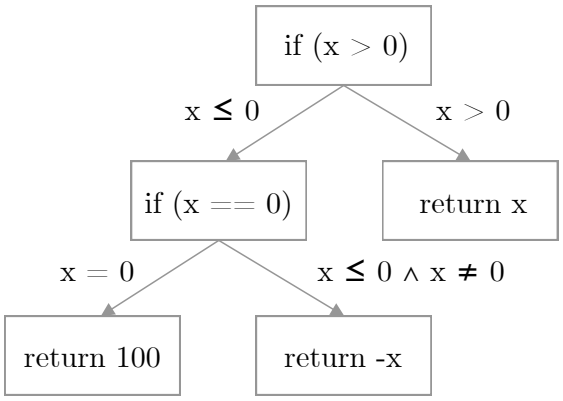


Figure 15: Example control-flow graph from [2].

CFG and DFG can become very large even for small applications. As a result, different optimisation approaches have been introduced to reduce the overall amount of nodes and edges in the graphs. Muchnick and Jones describe an algorithm for DFG in [76, p. 306]. Each node that does not represent an exit or erroneous application state has either an out-degree of 1 or 2. Out-degree 1 indicates that the instruction is an assignment whereas an out-degree of 2 identifies conditional nodes. The evaluation of the condition decides which edge is taken to reach the next application state. The condition can only evaluate to true or false. Although unconditional jumps are not mentioned by Muchnick and Jones, they can be interpreted as simple assignments to the instruction pointer register that is a member of the application state and can therefore be modified through an instruction.

Mohnen introduces an optimization approach to this DFG, where he describes a simplification of the graph by combining consecutive nodes that are connected by assignments (out-degree of 1) to *basic blocks* [86, p. 50]. This results in a simplified graph that contains nodes representing one or more consecutive assignment instructions and nodes that

indicate jumps or conditions (out-degree of 2). To avoid ambiguity e.g. when a jump is connected to an application state that is part of a building block the condition for combining nodes is extended. The ambiguity can be avoided if all nodes of the basic block (except the first and last) have an out-degree *and* in-degree of 1. This results in basic blocks where each member node must have exactly one predecessor (except for the first node) and one successor (except for the last node).

Su et al. performed a survey focused on data flow testing in [87] and introduced different techniques like search-based [88], random testing-based [89], collateral coverage-based [90] and symbolic execution-based approaches [91].

Whereas the described principles are designed for higher level programming languages they can also be applied to machine code. Considering the principles of compilers to translate high-level programming language features into multiple machine code instructions, these chained instructions can be grouped together again as long as they follow the requirements above basic blocks.

To create processor architecture independent CFG or DFG the machine code must first be disassembled into assembly code. On Linux-based system the machine code is extracted from the dedicated segments of the ELF compliant application file. Figure 16 shows a control flow graph for symbols in the Debian application `nice`. The color of the nodes indicate the file (executable or dynamic library) the symbols are located in.

The root of the graph is the application execution start address that is specified in the ELF header of the executable. This entry point will invoke the Linux loader to set up the runtime environment for the application as shown in 12. This process is not relevant for the application analysis. To overcome this issue the address of symbol `main` can be used too. This symbol is the entry point after the Linux loader finished the memory setup.

Each disassembled instruction is evaluated and the graph is build according to the principles described earlier. Memory addresses used by the assembly code instructions can refer to absolute and relative addresses in the virtual address space of the application. Since static analysis performs no actual execution, these memory locations need to be translated to referred instructions and data in the used ELF files. For ELF binaries that are marked as executables no relocation is performed before execution. Therefore, the virtual memory addresses can be resolved with the segments headers of the executable. Let H_i be the header of segment i, then be $VAddr_i$ the address in virtual memory space of this segment, $VSize_i$ the memory size and $FAddr_i$ the beginning of the segment data in the ELF file. The function R_{abs} determines the absolute address for a virtual address v in executables with the following calculation.

$$R_{abs}(v) = FAddr_i + (v - VAddr_i) \tag{4}$$

where i denotes the segment where $VAddr_i \leqslant v < (VAddr_i + VSize_i)$. This suffices to resolve absolute virtual addresses. Another special form are relative addresses. These addresses denote the location relative to the current value of the instruction pointer register.

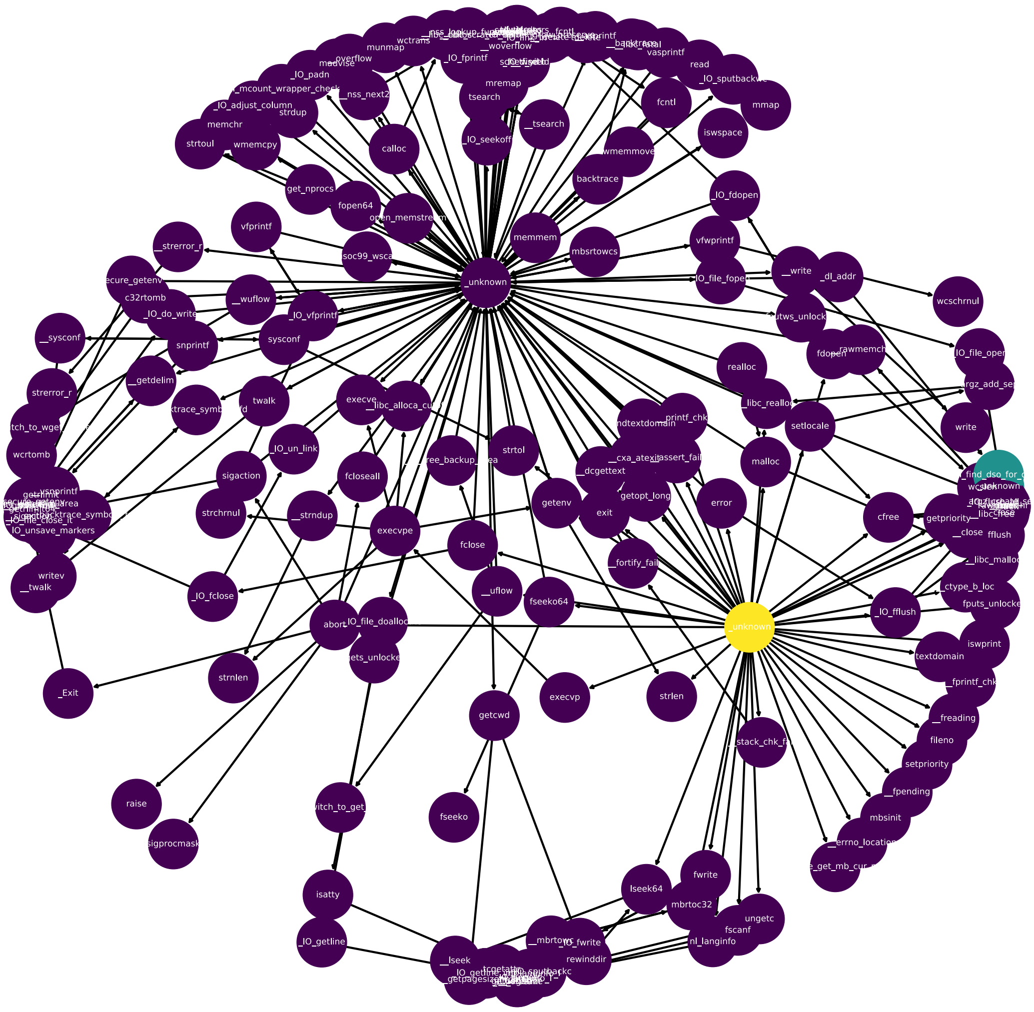


Figure 16: Control flow graph with 192 symbol nodes of `/usr/bin/nice`.

As an instruction is executed, the instruction pointer register holds the virtual address of the **next** instruction to execute. The listing 8 shows a memory reference based on the instruction pointer register (named `rip` on 64-bit processors) with an offset of 0x20b4b3.

```
...
0x411fb9: e8 a2 06 ff ff        callq  0x402660
0x411fbe: 4c 8b 3d b3 b4 20 00  mov    0x20b4b3(%rip),%r15
0x411fc5: 48 85 c0              test   %rax,%rax
...
```

Listing 8: Excerpt of the disassembly of the Debian 9 `/bin/ls` binary

The instruction at virtual address 0x411fb9 is a `CALL` instruction to an absolute address that can be resolved with $R_{abs}(0x411fb9)$. However, the `MOV` instruction at position 0x411fbe accesses the relative address 0x20b4b3 to copy the content to register `R15`. To determine the absolute address for relative address r it is required to know the state of the instruction pointer register. The definition for the instruction pointer register is to always point to the next instruction can be used for this. During static analysis, the address of the current instruction i is known ($IAddr_i$) as well as the instruction size ($ISize_i$). Hence the instruction pointer register value for i is $IAddr_i + ISize_i$. Based on this a relative address r for an arbitrary instruction i can be converted to an absolute address using the following function R_{rel}.

$$R_{rel}(i, r) = IAddr_i + ISize_i + r \tag{5}$$

The result of a DFG construction using static analysis can be seen in figure 17 or in its complete form in appendix E. Basic blocks have been built to increase readability. Unconditional and conditional jumps are still visible in the graph.

While the graph is built according to the principles described above, jumps into the `.plt` section can be observed. As described in section 4.1 this indicates the usage of dynamically linked symbols from other libraries. To integrate these symbols in the graph the static analysis must be continued in the referenced file. The analysis algorithm has to locate the dynamic library file in the filesystem and the referenced symbol based on the information in the ELF header and section headers of the library file. If this succeeds the graph construction can be resumed at the entry point of the symbol in the dynamic library. This procedure has to be done for all jumps into the `.plt` section of an ELF executable or dynamic library.

The introduced mechanisms enable static analysis mechanisms to build CFG and DFG without actual execution of the application. Absolute and relative virtual addresses can be translated to corresponding locations in ELF binaries. Dynamically linked symbols can be detected by jumps to the `.plt` section and can be integrated into the graph. However, one problem remains for jumps and calls to addresses that are stored in or referenced by

registers. Since the content of these registers is unknown if the application is not executed, the destination of the jump or call can not be determined. An example for an instruction that uses an address referenced by a register is shown in listing 9. These assembly code instructions are found e.g. for function pointers passed in the source code.

```
1 ...
2 0x405c0f: ff e0                    jmpq   *%rax
3 ...
```

Listing 9: Jump to an address referenced by a register RAX

This problem can not be solved with the mechanisms of plain static analysis. Further interpretation of instructions, their contextual properties and possible runtime values have to be considered. This can be done with *abstract interpretation* or *symbolic execution* methods that are described below.

Abstract Interpretation

The principles of abstract interpretation extend the possibilities of application analysis based on CFGs and DFGs. The deduction of program flow depending on application state can be achieved to a certain degree with this technique. As described by Cousot and Cousot in [92], the generated graph refers to conditionals that relate to the application state. If a condition becomes unsatisfiable the follow-up nodes are unable to reach. These unsatisfiable conditions are detected via the continuous evaluation of conditionals found during code analysis. If the evaluation returns an empty set of values that would satisfy a condition it can be safely ignored for the ongoing analysis.

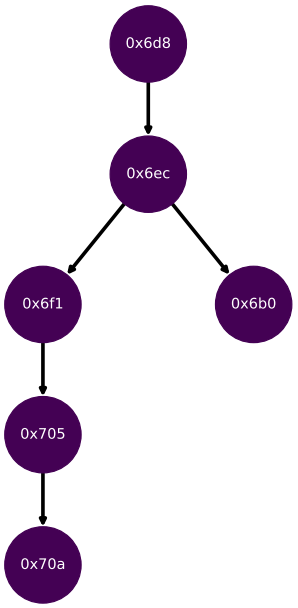


Figure 17: Example CFG with grouped blocks

This interpretation of conditionals is achieved with mathematical lattices as e.g. shown by Cousot and Cousot as well as Nielson [92, 93]. Each variable of an application state can be described as a lattice. Consequently, infimum and supremum can be determined that would satisfy a conditional. However, during the execution of the analysis these conditionals can become very complex and SAT-solver are required to determine the conditions for which a branch is executed. Figure 15 illustrates the combination of two conditions. Mauborgne et al. introduce a more efficient approach in [94] that allows the partitioning of so called *traces*. A trace is a sequence of instructions that is executed subsequently similar to the grouping of basic blocks in CFG. The

proposed technique enables *dynamic partitioning* which helps to reduce the number of misinterpreted trace paths as well as the overall condition complexity.

Besides the introduced work are other publications that focus e.g. on logic programming languages and the abstract interpretation of their source code to enable better compiler optimisation techniques as shown by Muthukumar and Hermenegildo in [95]. Anyhow since their approach is too narrow for the investigated use-case of this work it is presented here for a more complete overview of the field of abstract interpretation.

The main problem for abstract interpretation remains the SAT-solving of conditions for either tracing branch instructions or to identify potential misuse of memory areas etc. It is proven that SAT is a NP-hard problem and thus not computable for arbitrary problem sizes. Additionally, since abstract instruction works on the creation of value groups that satisfy conditions it is by design not able to identify jumps to arbitrary code locations based on variable content.

Symbolic Execution

Abstract interpretation uses complex semantic equations to describe application behaviour at a certain point in time. Given a concrete application state, the actual behaviour can be determined through the evaluation of the built conditionals.

Symbolic execution (or symbolic evaluation) uses *symbolic* values for variables which state is unknown without actual execution of the application. While source code or machine code is investigated for tests of identified symbolic values, *symbolic constraints* are identified that apply. The result of this analysis is a graph (G), that contains executions paths ($p \in P$), for which certain symbolic constraints apply (ϕ_p) [2]. A simple example is given in listing 10 and figure 18. It can be seen that for $p = \text{return} - x$ (line 7) the constraint $\phi_p = x \leqslant 0 \wedge x \neq 0$ applies.

```
1 int abs(int x) {
2   if (x > 0)
3     return x;
4   else if (x == 0)
5     return 100;
6   else
7     return -x;
8 }
```

Listing 10: Example application for symbolic execution as shown in [2]

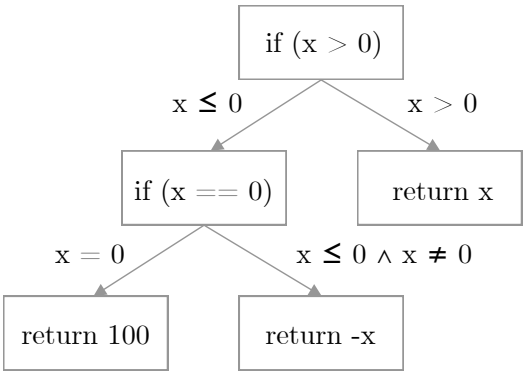


Figure 18: Fully discovered execution tree based on listing 10 [2].

The iteration of all possible p is feasible for small programs and suitable inputs and testcases can be deduced from the calculated ϕ_p [96]. But, if a program grows larger, the number of p grows exponentially. This so-called *path explosion* is a problem addressed by [97] and [98] and is a general problem of static analysis techniques.

Selective symbolic execution in test generation according to [99] can help to overcome the problem of path explosion. With this technique introduced by Chipounov et al. applications can be tested selectively on the machine code level. Frameworks like `manticore` [100] and `angr` [101] are able to generate CFGs and to perform symbolic execution based on them. It is possible to either perform whole application symbolic execution or to specify a dedicated entry point in the binary where execution should start.

Where symbolic execution can find symbolic constraints for selected paths, it is also based on CFGs. As a result, it is also difficult for this method to deduce referenced addresses in registers unknown without execution (see listing 9). The symbolic analysis can be able to reconstruct the value that is stored in the addressed register. Nonetheless, it is generally uncomputable to determine all possible paths that need to be traced back for this [61].

4.3.2 *Dynamic Code Analysis*

In contrast to static code analysis, the task of dynamic code analysis summarises techniques and investigations that involve the execution of an application on a real or virtual processor [102, p. 14]. In the latter case one can speak of *emulation* instead of execution. It is possible to either run the application as a whole or only certain parts the investigation is interested in.

The major advantage of dynamic code analysis is that a real or emulated input is given to the application and the control or data flow with this given input can be evaluated. This enables the distinct modelling of register values and memory at a certain point in time and allows the investigator to calculate or emulate application behaviour based on them. The goal of dynamic code analysis is to collect more exhaustive information about possible execution paths and constraints applied to them than static analysis.

The emulation of machine code instructions requires a virtual processor. This processor must be capable to execute the instructions in the machine code. The lightweight multi-architecture emulator `unicorn` is able to interpret arbitrary machine code provided in a supported architecture [103]. The 64-Bit x86 architectures used in this work is supported by the emulator too. The virtual processor possesses the architecture specific registers and understands its operations. Partial and full application emulation are supported by `unicorn`.

Code Coverage and Fault Localisation

Code coverage is a metric that can be used to identify rarely used or even unused code segments. Large unreachable code segments can indicate functionality that is not required for the investigated input or even malicious code that is activated only under certain preconditions. To analyse code coverage the application is executed with representative input data for all use-cases that should be handled. This includes invalid input data which must be recognized to cover error handling routines. Wong et al. introduced an effective framework for fault localization based on code-coverage tests in [104]. They use the information about covered code segments during unit tests to determine which segment is associate to which test cases. Based on the execution success of the test case they present three heuristics that associate e.g. ratio of appearance of a code segment in a test case to test failure rate with the probability that an error is located in the concerned segment.

A different approach is introduced by Tikir and Hollingsworth in [105] that allows code coverage recording in machine code. Their technology installs hooks in the dynamic linking mechanism that is done by the Linux loader. This is done to collect information about external symbols that are invoked. Additionally, each step of the application is traced using the `ptrace` library. If a symbol from a dynamic library function is invoked their injected code is executed and the bookkeeping tasks to record code coverage information are performed (see figure 19).

Chen et al. show in their work that the analysis of source code or machine instructions is not always required to collect information about code coverage [106]. They demonstrate that for an application with sufficient distinct logging output and knowledge about the code instructions which produced these messages, a thorough coverage analysis can be done through log message analysis alone. While this is an interesting approach, it does not fit the use-case targeted in this work, where neither sufficient log information nor code knowledge is available.

Memory Error Detection

Hicks provides a good description about memory errors that cause problems during application execution [107]:

ACCESS TO UNALLOCATED MEMORY describes a spatial error, where an application tries to access memory that is located outside of the boundaries of the allocated memory. These errors can be the result of type confusion, missing boundary checks for buffer or array access or a general insufficient memory allocation.

VIOLATION TO THE CAPABILITY OF A POINTER instead describe a temporal error that is caused by either an access too early in time (when a memory segment has not been allocated yet) or too late during execution (when a memory segment has already been freed).

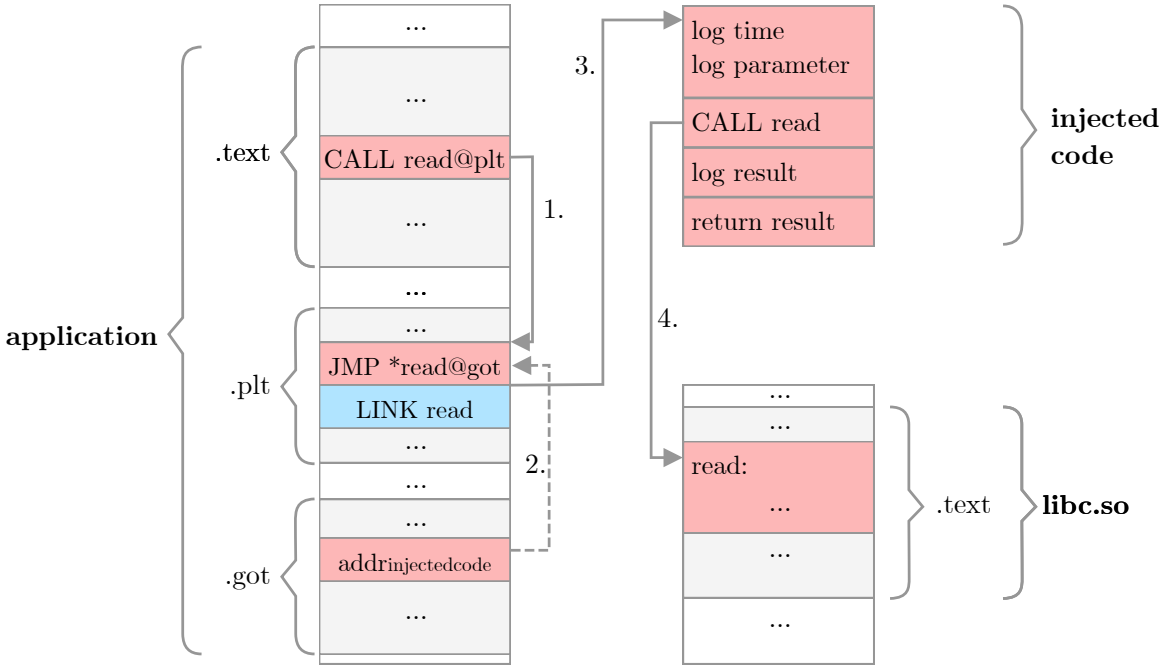


Figure 19: Visualisation of code injection based on the work of [105] to record data about invoked symbols. The address of the logging code is placed in the `.got` instead of of the actual symbol address. The resulting jump leads the execution to the injected code (step 3.). The desired log information is stored and the real symbol is invoked (step 4.). Afterwards the result is saved and returned to the callee.

Several techniques exist for the detection of memory errors, such as *redzone-based detection*. Redzones are special areas that are placed between orderly allocated objects. If an out-of-bounds access results in the access to a redzone, the invalid access can be traced back to the originating instruction [108–111].

Approaches to detect pointer violations caused by *use-after-free* or *double-free* can be found in [112–114]. There are also techniques for memory error detection with static analysis described in [115].

Concurrency Errors

Concurrency errors can be classified into race-condition related problems and deadlock-/infinite waiting problems. The first class of problems result from independent threads that change the application state in an unanticipated way. These problems are very hard to detect with dynamic code analysis tools. This is due to the interference of the analysis tools with the execution of the application. A race-condition might not even occur when the program is executed in emulated or supervised mode.

Deadlock and infinite waiting problems can result from mutual exclusion problems. These exclusions often relate to an unconsidered application state where mutexes, locks, barriers or other operating system locking mechanisms are employed. The detection of several problems related to these issues is possible with the *valgrind* toolkit introduced in [110].

Dynamic Security Analysis of Source Code

A field for the security analysis of source code is the detection of threats in *JavaScript* code that is downloaded and executed in a user browser. Tripp et al. investigated thedetection of unwanted and insecure action redirections in [116] and presented a vulnerability detection framework in [117]. One step further beyond detection is done by Magazinius et al in [118] where insecure sections are identified and dynamically fixed.

Russo and Sabelfeld compare several frameworks for security analysis that either use a static or dynamic code analysis approach [119]. They define four characteristics of dynamic code analysis frameworks:

- The monitor/emulator that runs the application does not look ahead when it decides if the instruction that is about to get executed is save (not look ahead).
- The monitor/emulator does not consider conditional control flows that were not executed due to the current application state of the program when it decides the current safety (not look aside).
- The execution of the application is deterministic.
- Finally the monitor/emulator must be permissive. Therefore it does not change the application behaviour itself if public information is only read and outputted. The emulator is not allowed to write to the application state if the emulation is not requesting it.

Their works also show, that all these four properties can only be achieved by a system that incorporates static and dynamic code analysis capabilities [119, p. 10].

Partial and Full Emulation

Dynamic code analysis mechanisms are based on techniques of application emulation. This can be done for applications provided in their machine code form. Depending on the data that should be collected, a partial or full emulation is required. If the control or data flow for a given input should be investigated, the emulation must start at the application entry point. If the behaviour of single symbols inside the machine code is of interest, the partial execution starting at the beginning of the symbol is sufficient.

The application state (registers and memory) must be emulated too. This is required for both partial and full emulation. For full application emulation, only the initial memory

state must be set up the same way as it would normally be by the Linux loader. For partial emulation however, the application state must be valid for the desired part of application that should be emulated. The construction of this state is difficult. Therefore testing methods that perform partial execution rely on *fuzzing* methods [110, 119]. Fuzzing determines the dependent variables for the desired application part and emulates its execution with all possible variable variants. However if the application state depends on many variables, fuzzing methods are not applicable because of exponential complexity.

SUMMARY

This chapter introduced the challenges of machine code analysis of applications executed on Linux-based systems by a processor with the 64-bit x86 architecture. Besides the machine code itself, the used ELF as a file format provides information about the application, linked libraries and the runtime set up. This information is usable to identify application dependencies and to extract required information for the dynamic code analysis.

The major part of the chapter introduced the principles of static and dynamic code analysis as well as techniques based on these principles. Static code analysis is a powerful tool to investigate the application and to generate control and data flow graphs without actual execution. However, it suffers from the complexity that is introduced due to unknown application state during the analysis. Conditions are introduced to model dependencies between application paths and application state. Nonetheless these conditions become very complex with continuous analysis. As a result, application flow that is heavily depended on the application state can not be investigated.

Dynamic code analysis tries to overcome this issue through the actual or emulated execution of the application. This is feasible for machine code applications if a virtual processor is employed and if the application state is emulated too. Using this approach more complete/ use-case relevant CFG and DFG can be computed based on the input data selected for emulation. Nonetheless, the selection of appropriate input data is crucial for these techniques.

5 RESTRICTED EXECUTION ENVIRONMENTS

The previous chapters provided the definition of assets and threats as well as a description of resources which can be associated with them. Technologies were described that can be used to investigate an application for certain resource access. The introduced technologies allow such a resource-access analysis even for machine code application.

In this work, the result data from the application analysis is used to configure and set up a secure execution environment. This environment must allow the resource access to those resources that the analysis has identified and which pose no threat to any assets. Furthermore it must deny any other resource access attempts by denying them or terminating the application. The term sandbox is typically used to describe such an isolated environment.

The first part of this chapter will introduce the requirements and different implementation options for restricted execution environments with a trusted operating system. The second part describes different technologies available for Linux-based operating systems that are suitable to secure resources and assets from unauthorised access.

5.1 REQUIREMENTS

A threat can target the computing system in several ways. Different assets can be targeted by an attack. Venter and Eloff give an abstract overview over security and sandbox technologies and associated threats in [120]. They categorise information security technologies into *proactive* and *reactive* classes. These classes are partitioned again where each branch contains security measures relating to *network level*, *host level* and *application level* as shown in figure 20.

The machine code analysis of an application can be placed into the application level category of proactive technologies. Yet the result data is afterwards used to configure software components from the reactive category that work on different levels as introduced later. Figure 20 highlights the large overlap between technologies for the reactive category. Technologies for *access control*, *biometrics*, *logging*, *firewalls* and *passwords* all aim to mitigate threats that try to access protected resources. Resource-classes like memory, filesystem resources or network are protected using these technologies. Additionally, separation of user-specific data is a goal for technologies like *passwords*, *access control* and *biometrics*.

The principles of isolating different user-mode applications from each other to prevent unwanted resource access is also described as a requirement for secure infrastructures by Keahey et al. [121]. To create secure dynamic virtual environments they identify that

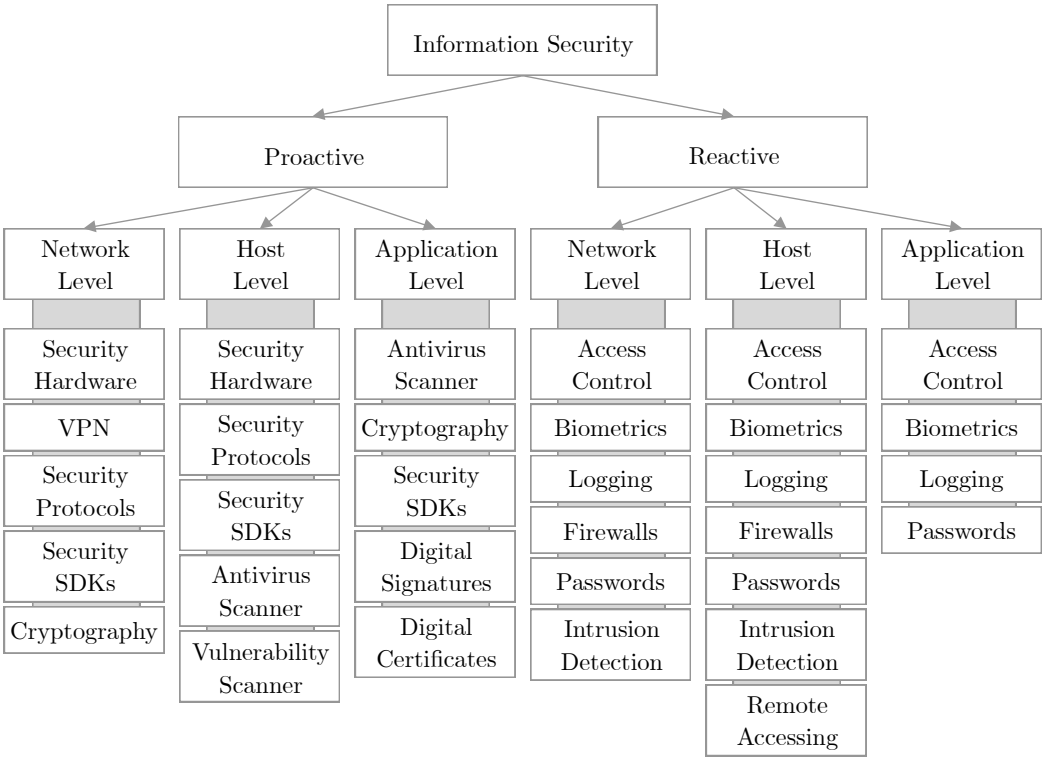


Figure 20: Taxonomy for information security technologies based on Venter and Eloff [120].

thorough separation of resources from all resource-classes is required and can be achieved with virtualisation technologies. Aside from this technological statement, they also point out five criteria for any technology used for sandbox provision:

GENERALITY A solution for sandbox provisioning should work for a large amount of applications and should not require adjustments for new payload software.

NON-INVASIVE The modification of software is discouraged. Although the results of a machine code analysis could be used to remove unused code paths from the binary. The non-invasive pattern prohibits such modifications to the application.

PROTECTION The solution should 'provide suitable levels of protection between users (not allowing users read each other's files for example) as well as between the user and resource (not allowing a user to gain superuser privileges on a resource)' ([121]). Consequently the sandbox should support a hierarchical user-management where the application is executed with non-administrative privileges.

ENFORCEMENT This requires a solution to support limitation of resources to consume (e.g. amount of CPU-time or disk space). It must also be capable to enforce these re-

strictions through a denial of resource allocation which exceeds the limit or through concealment of inaccessible resources.

APPLICATION STATE The preservation of application state when sandbox mechanisms are executed is an important requirement in cloud environments. Especially when it comes to capabilities like migration of sandboxes with confined applications between compute node in the infrastructure. Anyhow state preservation techniques do not relate to threat mitigation strategies. As a result, these technologies are not considered in this thesis.

The restriction of applications based on their resource access through the system call interface can be tested with the described criteria. If all of them apply a system call-based sandbox is feasible. *Generality* is given by the fact that all applications must use the system call interface to access resources. The supervision of the system call interface and a modification of system call results if necessary does not modify the application itself. Therefore the *non-invasiveness* criterion is satisfied too. Either the sandbox technology or the kernel provide mechanisms to enforce implemented rules which satisfies the *enforcement* criterion. A sandbox can check the utilisation of the system calls interface with a layer between operating system and user-space application. If such a layer is employed, its behaviour has to match the original interface. This is because of the *application state* criterion that requires the same application state handling with or without sandbox mechanisms. Finally the *protection* criterion is satisfied if the sandbox enforces user separation and supplies a hierarchical permission model.

These criteria, the argumentation for reactive security technologies based on resource classes and the definition of resource classes for assets in 3.1, can now be combined. The result is an association of threats with system calls which can be found through machine code analysis. The final step to achieve this is to associate system calls with resource classes. This can be done through an investigation of available and standardized system calls and referring to their manual pages or implementation [122]. Table 12 shows an excerpt of this association. The full table with the over 300 system calls on a modern Linux system is given in appendix D.

Resource Class	System Calls
FS_1	`read`, `write`, `open`, `close`, `stat`, `fstat`, ...
FS_2	`statfs`, `fstatfs`, `sysfs`, `syncfs`, `mount`, `umount2`, ...
MEM	`mmap`, `mprotect`, `munmap`, `brk`, `mremap`, `msync`, ...
CPU	`poll`, `pause`, `clone`, `fork`, `execve`, `exit`, ...

Continued on next page

Resource Class	System Calls
ACCESS	`getuid`, `getgid`, `setuid`, `setgid`, `geteuid`, `getegid`, ...
NET	`read`, `write`, `close`, `accept`, `sendto`, `recvfrom`, ...
DEV	`ioctl`, `iopl`, `ioperm`, `io_setup`, `io_destroy`, `getcpu`, ...
KERN	`create_module`, `init_module`, `delete_module`, ...
Time	`gettimeofday`, `settimeofday`, `time`, `clock_settime`, `clock_gettime`, ...
ROOT	`dup`, `dup2`, `dup3`, `fcntl`, `sync`, ...
Other	`uname`, `sysinfo`, `syslog`, `vhangup`, `_sysctl`, `reboot`, ...

Table 12: Selection of 64-Bit Linux system calls and their associated resource classes

The table shows that certain system calls can relate to multiple resource classes like e.g. `read`, `write` or `close` relate to FS_1 and NET. An investigation of the system call arguments is required to perform a disambiguation. The passed arguments allow a distinct association of the concrete system call to a resource class. System calls that can relate to more than two resource classes are grouped into a new class (ROOT). The resource class *other* is made up of system calls which do not fit in any of the other classes.

The more than 300 system calls are now associated with resource classes, assets and therefore with threats that originate from them. But, as described in 3.1, there are attack classes that can not be detected with this approach because they do not utilise the syscall interface. These threats will be considered in the final evaluation but are excluded from this work due to its focus on the system call interface.

5.2 PRINCIPLES FOR SANDBOXES

Different architectural possibilities exist to establish a sandbox on a computing system. Resources outside the sandbox (or those only provided as read-only items) are protected or unreachable for the limited application. However a sandbox might also be considered as a secure enclave where data produced and stored inside the sandbox is inaccessible from outside of the sandbox. Because the computing system is a complex layered architecture as shown in figure 21, there are sandbox technologies available at each of these layers. Significant research and state-of-the-art concepts will be introduced in this section that allow to sandbox an application [123].

5.2.1 *Hardware Sandboxes*

The security and isolation starts at the hardware layer for CPU, memory and devices. Whereas most of the technologies work in cooperation with the operating system (like process scheduling, permission levels, memory mapping etc.), there are those which allow the creation of secure enclaves for data or calculations. The latter ones implement the concept of *Trusted Execution Environments*.

Intels *Software Guard Extensions* (SGX, [124]) and *ARM TrustZone* [125] are features available in selected processors and provide hardware sandboxes. The TrustZone architecture enables the separation of a so-called *Normal world* from a *Secure world*. Dedicated bus signal lines indicate to attached devices and memory if the system operates in a normal or secure world. SGX on the other hand does not require additional hardware layout changes but is not as strict with the separation of devices and memory. The only thing SGX provides is the creation of *secure areas* where code and data in memory is protected from unauthorised access by the processor.

These hardware sandboxes are especially useful in cloud computing environments, where the client has high security demands which require the effective protection of data and/or code from the infrastructure provider with administrative access to the system [126, 127].

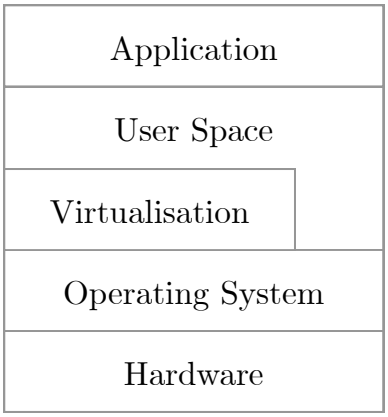


Figure 21: OS layers for sandbox technologies

5.2.2 *Operating System Sandboxes*

Modern operating systems provide several mechanisms for resource separation between users, processes and kernel managed functions. These mechanisms include e.g. user- and group-based access control, virtual memory management and process scheduling [35].

This separation is enforced by the operating system kernel. Therefore an attack aimed at the kernel is especially interesting for an attacker on monolithic kernel operating systems. This type of kernel runs all resource management and permission enforcement routines in the same kernel space. If an attacker succeeds to exploit such a kernel, they obtain administrative privileges and security mechanisms become useless. Only hardware sandboxes would remain effective in such a scenario.

To overcome this issue, Dautenhahn et al. suggest a nested kernel approach in [128] where an additional separated nested kernel is used only for security related tasks. Only the outer kernel interacts with the applications through system calls and communicates with the nested kernel through secure so-called entry and exit *gates*. With this setup, a

tainted outer kernel does not inflict the secure resources managed by the nested kernel component. The presented benchmarks show that this approach causes kernel-related task to take up to 3.69 times as long as in normal operation.

Resource separation is closely related to resource scheduling. This is shown by Rajkumar et al. in [129] and extended by Chang et al. in [130]. The introduced OS kernel mechanism handles resource allocations for CPU, memory and network and assigns static and asserted contingencies to running processes. As a result, the process is restricted to these assignments like placed in a sandbox.

Another approach is to provide the kernel with more granular features to specify permissions for applications. By default each process of an executed application possesses a real and an effective user ID. The effective user ID is checked by the kernel upon resource access. Technologies that extend these simple access verification patterns are *Linux Security Modules*, *capabilities* or *namespaces* which are introduced later in this chapter. A similar technique for Windows-based operating systems is presented in [131].

5.2.3 *Sandboxes Using Virtualisation*

Virtualisation is an established method to provide thorough separation of resources between an operating system running on the *host machine* and a simulated *guest*. The virtualisation software is responsible for the simulation of required hardware components for the guest. This allows the separation of real hardware resources from the untrusted guest through the provision of only simulated hardware.

The so-called *hypervisor* or *virtual machine monitor* is the software component that is capable to emulate the environment required for the guest and is executed on the host machine [132]. Solutions can be divided into *native* and *hosted* hypervisors. A native hypervisor runs directly on the host machine hardware, whereas a hosted hypervisor is executed by an operating system that is installed on the host machine.

Besides the administrative and organisational advantages, security features are of high interest for infrastructure providers. The hypervisor provides an effective security layer that shields the hardware or the host machine operating system from attacks. However, there are also security issues related to hypervisors. Turnbull and Shropshire present possible attacks against a native hypervisor in [133]. They considered threats like network traffic redirection, system library call hijacking, hypervisor API rewriting and system call hooking. All of these attacks require a full compromise of the native hypervisor. These threats can also be used for attacks against hosted hypervisors if an attacker has gained administrative privileges on the executing operating system. In these cases the host system as well as all guests must be considered potentially compromised.

More common threats are attacks from a guest machine against either the hypervisor, the host machine or other guests that are executed by the same hypervisor. Szefer et al. investigate these threats in [134]. They found that the so-called *VM exits* (or *hypercalls* on

paravirtualised guests) can be used to execute attacks. VM exists occur when a guest is stopped and the hypervisor takes over control to execute an event that originated from the guest. With each VM exit, the processor (that has to support hardware virtualisation) evaluates a dedicated register that describes the occurred event. Intels 64-bit architecture specifies 56 reasons for VM exits [135]. To reduce this attack surface Szefer et al. show a system that eliminates the pre-allocation of processor cores and memory resources, removes access to hardware I/O devices, manipulates the system discovery process and avoids additional memory mappings between guest and host system. Yet this requires changes in the guest operating system and reduces flexibility.

Several other approaches exist that try to mitigate attacks against the hypervisor. Tahir et al. introduce a detection framework based on *Cuckoo filters* in [136]. Cuckoo filters are key-value data structures with especially high performance for insertion and lookup operations [137]. The presented framework is integrated as an intermediate layer between a guest machine and hypervisor to collect data about issued system calls by the guest. These patterns are collected and used to built a cuckoo filter. This filter is then sent through anomaly detection to find irregular behaviour of a guest. The anomaly detection relies on a machine-learned pattern recognition mechanism that was trained beforehand. Because of the high-performance nature of Cuckoo filters the imposed processing overhead is small (4%). Quality of the detection however depends on the size of the filter, the learning time of the anomaly detector and monotonic guest behaviour. Promising results are shown for training periods of 60 minutes and more with Cuckoo filters of 1 MB size.

The work by Xia et al. in [138] and Zhang et al. in [139] evaluate attacks that originate from a compromised hypervisor. A mechanism is introduced that enables the guest machine to defend against *rollback attacks*. A rollback can occur when the hypervisor stops the guest and restores an arbitrary disk and memory image to return the guest to a state chosen by the attacker. While this is a feature that enables easy guest migration, backup and restore, it also enables a compromised hypervisor to stop, rollback and restart a guest to e.g. monitor confidential operations multiple times to collect secret data. The system from Xia et al. employ technologies also used for hardware sandboxes to create a secure space on the system. This is used to provide tamper-proof logging that enables a guest to detect a maliciously intended rollback.

5.2.4 *User Space Sandboxes*

Another way to restrict an application in its possible actions is to enable unprivileged users to establish rules for its execution. These rules are then enforced by an independent trusted component that is capable to supervise those rules. Such technologies are prominently featured in interpreted programming language environments or application virtual machines (like the *Java Virtual Machine*). These software components are executed with user permissions and execute the provided source code or an intermediate representation

generated from the source code. Besides tasks like memory management and process scheduling they can also employ sandboxing capabilities.

One example is the *Native Client* software (NaCl) which describes a sandbox for executing untrusted JavaScript and other binary modules inside a web-browser [140]. The software validates the code flows, memory layout permissions and intercepts system calls send to the operating system. It also defines "unsafe" operations that are disallowed altogether.

Other technologies for the restriction of applications that do not require administrative privileges can be based on system call supervision. This is a promising approach because unprivileged users can be allowed to supervise their own processes with trusted monitoring processes. Whereas [46] and [141] intercept system calls and make runtime decisions for the monitored process based on the system call, the Apple sandbox implementation described by [142] is a two-layered system. A user space sandbox-daemon evaluates system calls from the monitored process, whereas operating-system kernel components enforce actions like application termination or resource filtering.

The approach of Liang et al. in [143] proposes a one-way sandbox where sensitive resources might be readable (if permitted), but writeable operations are only present in the sandboxed environment. Once the application is finished and the sandbox is disposed all changes are deleted. This is similar behaviour to virtual machines or lightweight containers, but happen in user space.

5.2.5 *Application Sandboxes*

Application sandboxes are different from the other introduced technologies from the point of problem assessment. Whereas the other technologies consider the application untrusted and as a result construct a sandbox to limit damage to the system, application sandboxes are a way for a developer of a benign program to complicate attacks which might compromise it or to protect sensitive data inside the application.

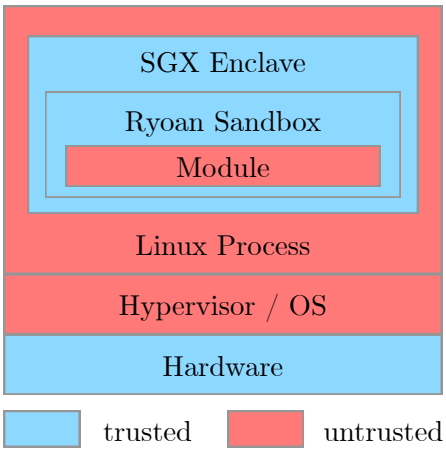


Figure 22: Ryoan sandbox according to [144]

The software developer is in the best position to know exactly what the developed application is expected to do and what system resources are required or not for a successful execution. With this knowledge, the developer is able to describe an application sandbox that only contains the required functionality and resources and might therefore mitigate attacks like privilege escalations and unanticipated resource accesses.

Seccomp [145] and *Capsicum* [146] are two examples of such technologies. An application that utilises these functionality is started like normal programs on the operating system. During runtime, the application is able to enable one of the mentioned technologies to drop selected privileges. Such privileges are e.g. the read or write to memory areas, access to file descriptors or the interaction with processes. Once these privileges are dropped, neither the process that activated the application sandbox nor spawned child processes or threads can acquire them back. Seccomp supports further capability restriction during runtime, whereas Capsicum is not able to reconfigure a once enabled sandbox. Rule enforcement of these technologies is done by the operating system kernel, which makes these technologies dependent on Linux distributions and kernel configuration.

To protect assets from unauthorized access the application faces the same problems as virtual machines that might be attacked by a compromised hypervisor. The difficulty of a compromised operating system that tries to extract secret information from the application can only be mitigated with hardware security assurances like Intels SGX technology or other trusted execution environments. The *Ryoan* system introduced by Hunt et al. in [144] or *Minibox* by Li et al. [147] use such set ups as shown in figure 22.

5.3 PLATFORMS AND TECHNOLOGIES

This section will investigate the different technologies and platforms available to provide secure runtime environments for applications [148], [149]. All candidates will be evaluated according to the following scheme:

RESOURCES PROTECTED As described in the prior section, an attacker can target different types of resources. This criterion lists the resources that can be protected with the technology and how sophisticated these capabilities are.

RUNTIME CONFIGURATION ABILITY The aim of this work is the generation of a one-time execution environment. The possibilities to execute multiple of such environments should not be ruled out. Therefore a technology must be configurable during operation system runtime and should neither require service or even system reboots nor any other system-wide rule reload mechanism that can impact overall availability or other already running execution environments.

REQUIRED PERMISSION This criterion will evaluate whether the technology can be used and configured by non-privileged users or if some kind of system administration capabilities are required. Due to the fact that security frameworks can be affected by security issues this is important as it determines if an attacked might gain administrative access to the overall system in case of such an issue.

CROSS-DISTRIBUTION COMPATIBILITY Different technologies pursue different approaches when it comes to the realisation of the provided security features. This criterion

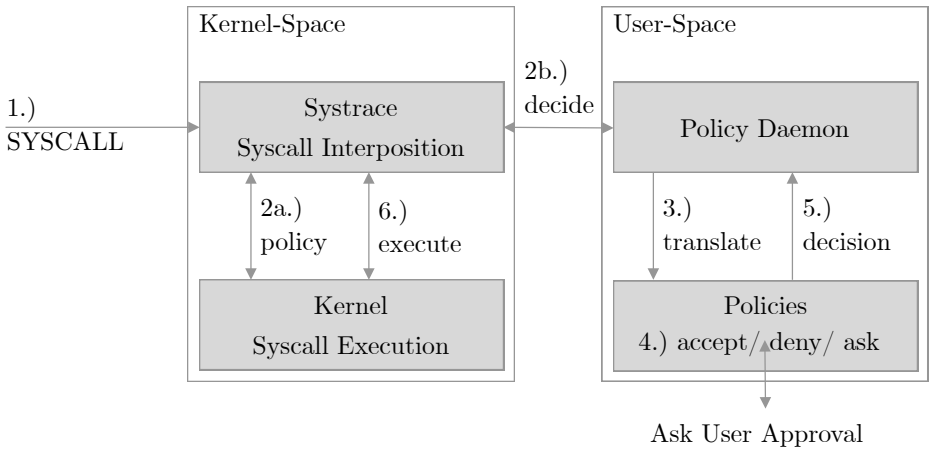


Figure 23: *Systrace* architecture based on the description in [150].

evaluation whether the technology is either easily portable between different Linux distributions or if it uses standardised or near-standardised features available on common distributions.

PERFORMANCE It is obvious that additional checks and the implementation of extra security mechanism negatively impact the application performance. This point in the evaluation will try to compare the size of the impact of the different investigated solutions.

At the end of this section a tabular comparison will be given for an overview of the described results in the following sections.

5.3.1 *System Call-based Filtering*

Systrace

Systrace is a technology introduced by Niels Provos in [150] to improve system security through the interposition of system calls. The system is designed as a hybrid architecture that works with a component residual in kernel space for system call interception and a so-called *policy daemon*. This daemon process runs in user-mode and is consulted by the kernel component to check an occurring syscall against the policies specified. Figure 23 illustrates this architecture. The Systrace system can work fully without any user interaction but is also able to leave certain decisions about an execution permission to him/her. In this case the abstract captured syscall is translated into a human readable form and the user is asked if he or she wants to allow the requested action (see 3 and 4 in figure 23).

Because the Systrace framework works based on system call interception it is able to protect all the introduced resources that might be targeted using them.

The system can be configured per application and the policy daemon is able to load and unload new rules without being restarted. The capability of user-based decision making is of no interest for the use-case scenario in this work, as it aims to provide a fully automatic solution for unsupervised systems [151]. Extensions exist that would also allow monitored applications to communicate with the Systrace policy daemon [152].

The kernel module is executed in privileged kernel mode and loaded at runtime. The policy daemon is started for an arbitrary system user and does not require additional administrative capabilities.

Systrace was available for *NetBSD* and *OpenBSD* (but was removed due to unfixed software issues or was replaced with the `pledge` framework [153]). The Linux-port always relied on `ptrace` as a backend for system call interposition rather than a dedicated kernel module which improves portability but significantly impairs performance [154].

The performance loss of system call interposition on Linux-systems is significant as shown by [47] and [46]. Furthermore, Watson has shown in [155] that the interposition mechanism is not suited to provide sufficient resilience against timing attacks. Finally, the development of Systrace appears to have ceased in 2009 with the last software release from May 2009 [154].

Seccomp

The *Seccomp* technology (shorthand for *Secure computing*) is a technology available in the mainline vanilla kernel since 2005 [156]. It also uses system call supervision like Systrace, but enforces rules solely in a kernel-based automata. All major distributions have seccomp support enabled in their kernels.

All users and application can activate seccomp enforcement for an application and specify rules to apply. As mentioned earlier, once the seccomp sandbox is activated it can not be disabled for the contained application. Also, once a rule-set has been activated these rules can only be further restricted not loosened.

It was shown that sandboxes designed with this technology based on a prior syscall tracing is feasible and does not significantly inflict performance even if a large amount of rules are applied for each syscall [46]. Figure 24 shows these results by a comparison of application runtimes for executions without confinement, with a blacklist and a whitelist operational Seccomp mode. The work highlights that even for the worst-case-scenario where more than 2400 rules had to be checked each time a system call occurred, the runtime does not increase significantly.

However, based on the flaws described by Watson in [155], seccomp maintainers refrained from offering seccomp-rules that would allow evaluation of user-controlled memory. Only the six passed arguments to a syscall are available when rules for filtering are defined. This makes it impossible to implement more sophisticated checks like filename comparisons or network address validation.

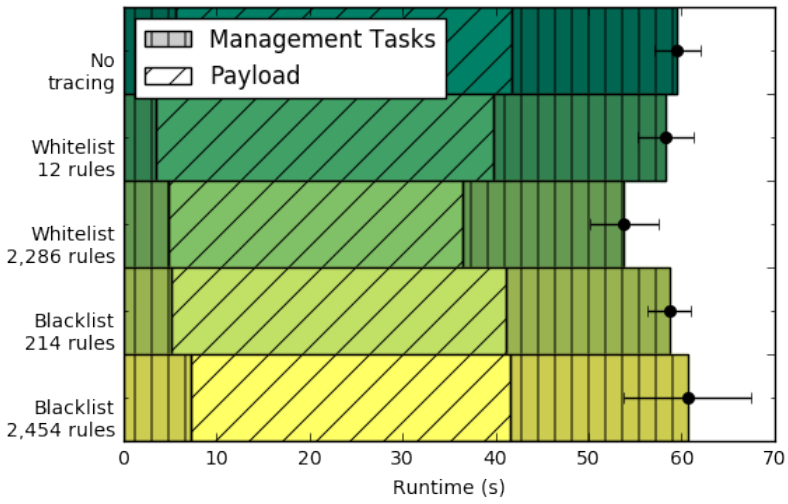


Figure 24: *Seccomp* rule-set performance evaluation [46].

5.3.2 *Linux Security Module*

The *Linux Security Module* (LSM) extension became available in the mainline kernel in 2002 [157, 158]. LSM was moved into the kernel to provide an interface for more abstract kernel modules to perform security checks. These checks are integrated into the system call interface of the kernel as exemplarily shown in figure 25 for the `open` syscall.

To overcome the timing attack issues Systrace suffers from, or the reduced functionality due to the lack of missing user-memory evaluation like Seccomp, LSM does not provide syscall arguments to the modules but rather complete kernel context structures. These structures are set up from the data provided by the syscall up until the point where plausibility checks have been run for correct input values but right before the actual resource access is executed. The modules registered for LSM hooks can access all fields in these kernel structures to check if permission to the requested resource should be granted and return the decision to the kernel.

5.3.2.1 *SELinux*

The *SELinux* security framework was introduced by Stephen Smalley in [159]. It is an implementation to provide secure Linux application processing and is now based on the LSM infrastructure. SELinux also employs *policies* to describe actions that are allowed for specified resources and those who aren't. The policies building blocks are `types`. Types are either a `domain` which encapsulates a user and his or her role or an `object` like e.g. a file, socket or service.

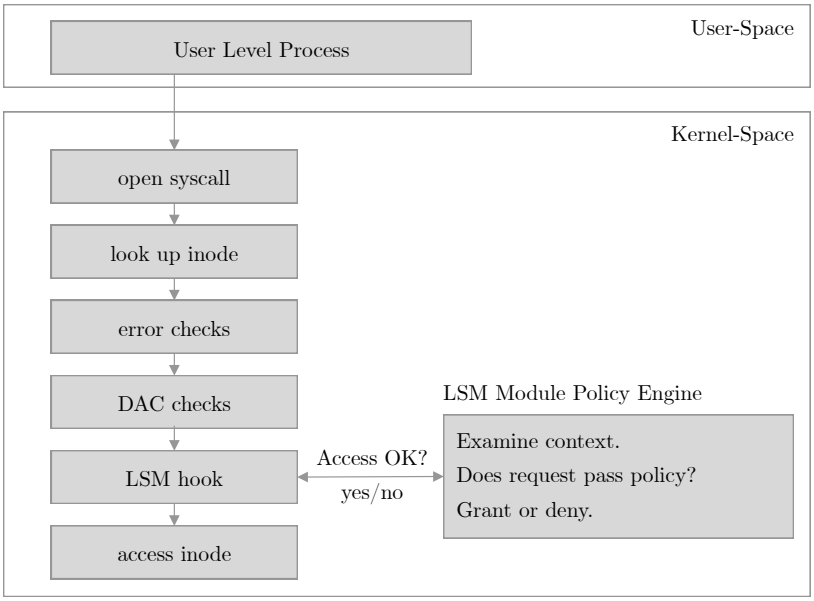


Figure 25: *Linux Security Module* (LSM) Hook Architecture [157].

Every user or process is associated with a SELinux security context that contains its user, role, domain and a *Multi-Level Security* identifier [160]. The latter can be used to reflect a mandatory access model according to *Bell-La Padula* and has to be enabled separately [161]. Access to an object is granted if a specific rule in the policy allows access to the requested object.

SELinux rules can be very fine-grained and be dependent on each other [162]. There is no dedicated pattern that describes a hierarchy of rules and tracing errors can be tedious. As a result, other systems to define policies have been suggested like behaviour-based policies [163], model-based policy generation [164] or even learning-based approaches to optimize the overall policy structure [165].

As mentioned above, SELinux hooks are integrated within the LSM architecture, which itself is used to protect arbitrary syscalls. Therefore SELinux is able to be used to secure all of the resource types identified in 5.1. Yet, some restrictions apply from the `object`-focused security pattern. E.g. filter policies for network connections to or from a specific host are not possible directly. Instead `iptables` rules must be used to redirect and label traffic into a specific `security` chain that is filtered by the kernel and the LSM architecture. This allows SELinux to perform policy enforcement for this use-case [166].

New *policy modules* can be defined by any user on a Linux system. However, compilation and installation into the system requires administrative privileges. A restart of the SELinux system daemon is also required to activate the new policy. Finally, if the new policy module defines new `types` for objects in the filesystem, a *relabelling* needs to be done to propagate these new information to the concerned `objects`.

Descriptions of overall performance drawbacks for SELinux use-cases range from a general 7% as reported in [167], to 75% for `read`-syscalls, 98.7% for `write`-syscalls in [162]. Comparable metrices are hard to find because there is no standard for testing SELinux use-case scenarios. Nonetheless, there has also been steady improvement of the SELinux performance over the last year as shown in [166].

5.3.2.2 *AppArmor*

Like SELinux, *AppArmor* uses the capabilities of LSM to implement its security features [162]. AppArmors goal is to provide an easy-to-use system that '(...) proactively protects the operating system and applications from external or internal threats, even zero-day attacks, by enforcing good behaviour and preventing even unknown application flaws from being exploited' ([168]).

AppArmor enables users to specify a list of file system paths or patterns with POSIX-like permissions for each item. It is also valid to specify rules for inter-process communication. This list of rules is associated with an application, which is identified by the path of its executable binary [3].

Although AppArmor also uses LSM, it not as sophisticated as SELinux. Configuration beyond objects located in the file-system is very limited. Also the usage of absolute paths as shown in the example profile in listing 11 is error-prone. This may result in security issues if e.g. that restricted application is moved to a different filesystem location without adjusting the rule. The labelling-technique of SELinux is a more persistent approach, which on the other hand requires filesystem support to store the required labels.

```
/usr/bin/example {
    /etc/passwd r,
    /home/*/** rwl,
    /home/*/bin/ ix,
    /home/likewise/*/*/** rwl,
    /{usr,}/bin/** px,
}
```

Listing 11: Example AppArmor profile with rules for `/usr/bin/example` [3]

Likewise SELinux is the integration of new AppArmor rules into the runtime system. New policy files may be created by every user known to the system. However translation into a configuration and loading it into the backing hybrid finite automata of the AppArmor framework requires administrative permissions to the OS.

Kernel modules for AppArmor are available for all major distributions and can be enabled in the operating system through accompanying management tools [168]. To improve policy generation, tools are available to supervise applications during runtime and generate tailored rule sets. Additionally, graphical and text-based dedicated editors can ease policy file creation for users.

Users described AppArmor to be more usable than SELinux. The reduced complexity results in better configuration and therefore improved sandboxes as shown by [169]. Again performance evaluations are difficult because of the heterogeneous infrastructures employed. Nonetheless, Helmy et al. did not find any significant runtime increase when employing AppArmor in their sandbox tests [170].

5.3.2.3 *Other LSM-based or similar technologies*

Several other technologies are available that are based on the LSM infrastructure. [169] compares SELinux and AppArmor with an implementation of *functionality-based application confinement* (FBAC-LSM). Functionalities in this framework are high level descriptions of applications like "Web Browser" or "Image Editor" that should be granted or denied certain access to selected resources. This makes FBAC-LSM easier to configure than low level security context and object access policies in SELinux. But they are not that flexible and fine-grained.

Bates et al. supposed a system that hooks into the system call processing pipeline right after the LSM-hooks were executed (see figure 25) to run so-called *Linux Provenance Modules* (LPM) [10].

5.3.3 *Namespaces*

A different approach rather than supervision of actions and reactively denying them is the creation of restricted environments, that contain only a subset of resources of the operating system. The capabilities available in these environments might either be linked to real resources in the actual operating system or only exists in this *sandboxed* environment at all.

To create such an environment which is separated from the main operating system, Menage proposed that *namespace isolation* is required [171]. He also defined and implemented different subsystems to provide the required amount of resource separation. This included an additional CPU accounting subsystem, CPU sets, resource grouping and management frameworks. Parallel to the work of Menage, Biederman investigated the measures necessary to integrate whole-purpose namespace separation capabilities into the Linux kernel in [172]. He identified that about 7% to 15% of the linux kernel had to be modified to provide namespaces for the following resources:

FILESYSTEM A sandboxed instance can configure and handle own mount points to filesystems reachable in the environment.

UTS UTS-capabilities characterize and identify the system that a program is executed on (host name, domain name). These information is retrieved through the `uname` system call.

IPC Inter-process communication involves the management of shared memory, semaphores and message queues on the linux system.

NETWORK A network namespace provides stand-alone routing tables, ARP resolution tables, network adapters and address information.

PROCESS ID Processes, process hierarchy, threads as well as process and thread group management has to separated from the rest of the system.

USER AND GROUP ID Analogous to process ID management is the provision and management of arbitrary user and group IDs in namespaces without causing collisions within the hosting operating system.

SECURITY MODULES AND SECURITY KEYS These considerations focus on the provision of security modules in namespaces and key exchange mechanisms between namespaces and the difficulties that arise from these concepts.

DEVICE

TIME The OS kernel provides several clocks to the system. Namespaces simply needs a time offset from the "real" system time to realise own time namespaces.

From this proposed specification there are namespaces for network, process ID, UTS, filesystem, user and groups ID as well as control groups present in the linux kernel [173]. The other namespaces mentioned above are either planned or their functionality was not considered suited for namespaces at all by the kernel development community.

5.3.3.1 *LxC*

Reshetova et al. compared several namespace isolation techniques that are based on the concepts described above in [149]. One technology introduced are *Linux Containers* (LxC). LxC consists of a collection of user space tools which enable the creation of sandboxed environments solely through the employment of namespace technology and upstream kernel features. Hence this technology is highly portable between different Linux distributions [174]. The improved toolchain to manage linux containers is named *LXD* [175].

Resource separation for all the above mentioned types of resources is possible with LxC/LXD. Filessystem resources and Filesystems are protected via mount namespaces. Process management and IPC are also constrained through their dedicated namespaces. Network devices are also created in a separate namespace and virtual interfaces are used in the OS to isolate sandbox application traffic from the OS until it has to enter the physical network, where the container traffic has to pass the physical OS managed interface. Finally memory, CPU and I/O loads can't be controlled through namespaces, but can be restricted with control groups (*cgroups*, [173]) [149].

Performance of linux containers is comparatively good in relation to other proactive IT-security techniques which rely on resource separation and concealment (like e.g. virtual machines). Joy [176] and Bernstein [177] show that for every tested scenario, container outperform virtual machines. Felter et al. conducted a more thorough investigation in [178] and tested performance loss for different resource types. They also found no significant negative impact when an application is executed inside a container compared to native execution except for network traffic handling. The additional steps required to route traffic from the OS to the container and vice versa increased by 80% from 38µs to over 72µs [2, p. 6].

5.3.3.2 *Docker*

Docker uses the same techniques as linux containers and has become increasingly popular for application sandboxing in Linux environments. One major difference between docker and LxC/LXD is the ecosystem that comes with the Docker software. Sandboxed applications can be bundled together with all required dependencies into *Docker images* and are distributes via the *Docker Hub*. A locally hosted so-called registry can be used if users want to limit access to their images [179]. Although these more comfortable ecosystem ([180], [181]) led to the rise of Docker over the older LxC technology, it also caused security issues related to the handling of foreign container images. The images are run as an encapsulated Docker container, but might contain old and vulnerable software [182] or the image itself might be compromised [183].

As mentioned above, Docker possesses the same properties as LxC. Yet it is able to use additional security features on the operating system to add reactive IT security technologies to the otherwise solely proactive character of container-based sandbox generation. The Docker daemon (which is executed with administrative permissions) is able to use *Linux capabilities*, LSM-based security modules (SELinux or AppArmor) as well as Seccomp filter rules to improve security of the system against attacks from the application executed inside the container [179]. It was shown in [184] that the additional employment of capability dropping and syscall filtering is feasible.

A concluding difference between Docker and LxC should be emphasized again. Docker runs a daemon process system administration privileges. By default, each user that wants to interact with this daemon also requires administrative privileges. It is shown by [183] that enabling otherwise unprivileged user to interact with the Docker daemon is equal to granting system administration capabilities to them. Even the official Docker manual states 'The docker group grants privileges equivalent to the root user)' ([185]).

5.3.3.3 *Other Namespace-based Technologies*

There are other technologies that rely on namespaces, control groups and the employment of standard Linux tools. These include *Rocket* [186], *Mesos Containerizer* [187], *OpenVz* [188]

or *containerd* [189]. Because of their similarity to Docker and LxC they won't be explained in further detail.

5.4 SUMMARY

Technology	Resources protected	Runtime configurable	Required permissions	Compatibility	Performance	Base Technology
Systrace	all classes	yes	user	compile required	bad [47]	syscall monitoring
Seccomp	all classes - no syscall attribute checks	yes	user	mainline kernel	good [46]	syscall monitoring
SELinux	FS_1, FS_2, CPU, MEM (partial), ACCESS, NET	no	admin	major distributions	good [167], medium [162]	LSM & iptables
AppArmor	like SELinux, NET (partial)	no	admin	major distributions	good [170]	LSM
LxC/LXD	FS_1, FS_2, NET (partial), CPU (partial), MEM (partial), ACCESS	yes	admin	mainline kernel	very good [178], good [2]	Namespaces, cgroups
Docker	all classes - no syscall attribute checks	yes	admin	major distributions	good [184]	Namespaces, cgroups

Table 13: Technology capability overview

After the detailed description of the different technologies available to achieve application sandboxing, this section summarises their capabilities according to the requirements introduced in section 5.1. The results are displayed in table 13. It is visible that all technologies have advantages in certain fields of the described requirements. The most promising result give technologies that utilise namespaces in combination with cgroup, iptables and other capabilities.

Part III

RESULTS

6 EXPERIMENT SETUP

This chapter describes the utilisation of the techniques for application analysis described in chapter 4. The collected data are used to generate a configuration for the used sandbox technologies to protect the identified resource classes from chapter 3.

In the first part of this chapter methods and technologies to generate the sandbox are introduced. The second part describes the datasets that are used to verify the selected approach. The result data from this datasets is used in the upcoming chapter to measure the improvements on the overall system security. This measurement is done with the metrics presented in 2.3.

6.1 METHODS

The processing pipeline that is described here is designed to investigate the machine code of applications that are built for Linux-based operating systems. Although other distributions utilise different approaches to implement interaction between user-space applications and the operating system, these technologies are similar and this work focuses on the system call interface to anticipate resource access of an application. The list at [122] is used as a reference for available system calls.

Furthermore, the introduced pipeline assumes that the applications to analyse are conform to the ELF standard as described in 4.1. Attacks through manipulation of the information stored in the ELF headers are beyond the scope of this work (see [190]).

6.1.1 *ELF File Analysis*

The basic information about the application, its size in memory, execution starting point and linked dynamic libraries are read from the structured information of the ELF file (see 4.1). The information is used to initially locate required libraries and ensure that the analysis pipeline is able to execute the investigated application. The pipeline ensures that the application target architecture (must be 64-bit x86 processors), operating system (must be Linux) and file type (must be executable) are set to valid values.

6.1.2 *Static Analysis*

A static analysis is performed according to the principles described in section 4.3. The entry point of the application serves as the root node for the constructed CFG. The entry

point is determined from the ELF program header. It is common that the application entry address points to a subroutine which invokes the Linux loader to set up the desired application layout described in section 4.1. This invocation is not of interest for the analysis and can be skipped. If it is detected, the invoking entry address symbol is disassembled and the Linux loader invocation is searched (line 11 of listing 12). Since the actual application entry point is always passed in register `rdi` to the Linux loader on 64-bit systems [39], it can be easily extracted from the subroutine machine code (see line 10 of listing 12 that pushes the address into the register).

```
1  xor    %ebp,%ebp
2  mov    %rdx,%r9
3  pop    %rsi
4  mov    %rsp,%rdx
5  and    $0xfffffffffffffff0,%rsp
6  push   %rax
7  push   %rsp
8  mov    $0x412560,%r8
9  mov    $0x4124f0,%rcx
10 mov    $0x4028a0,%rdi
11 call   4024f0 <__libc_start_main@plt>
12 hlt
```

Listing 12: Disassembly of the application entry subroutine of `/bin/ls` with the invocation of the Linux loader routine `__libc_start_main`. The address of the actual application entry point is passed in the `rdi` register in line 10.

When the correct entry point is identified the CFG is constructed. The used disassembler [191] rebuilds the instruction from the machine code of the application binary. Each instruction is analysed and added to the CFG. Instructions that form a sequence are grouped together based on the methods of [76] to reduce the complexity of the CFG. The analysis follows `JMP` instructions or subroutine invocations through a `CALL` if the destination address can be resolved.

The address resolution problem was described in section 4.3 as a major drawback of static analysis. Static analysis is not capable to determine memory addresses based on content stored in CPU registers with high certainty. Therefore these invocations are ignored and the graph is not continued for these instructions.

The static analysis however is capable to resolve dynamically linked function invocations which is especially important for common applications that are linked against standard libraries of the Linux system. Dynamically linked function invocations are detected through their specific pattern of instructions leading into the `.plt` section and dereferencing memory from the `.got` (see 4.1.2). If such a pattern is detected the referenced library is searched, opened and analysed the same way as the application binary. This enables the static analysis to construct a CFG with instructions from the originating binary as well

as from referenced library files. However, this requires the analysis to store the name of the file that contains the instructions of the CFG alongside their addresses for each node in the result data.

The result of the static analysis toolchain is a CFG with nodes representing a single or sequence of instructions. Each node stores information about the file that contains the instructions, the instructions itself as well as their virtual and absolute addresses. Additionally, if function boundaries could be reconstructed, the name of the function a graph node belongs to, is stored alongside the other information. Edges between nodes represent sequences, jumps or subroutine calls.

Because of the focus on system calls in this work, no further efforts are taken to e.g. rebuild complex datatypes from the analysed machine code. System call invocations are identified by the `SYSCALL` instruction if they are found in the CFG. The location of a system call in the machine code is stored alongside the instruction in CFG. Furthermore, possible paths that lead to the invocation of the `SYSCALL` instruction are examined in the CFG. Static analysis is not able to reconstruct datatype and program flow information with a sufficient accuracy. To collect these missing information application emulation is used.

6.1.3 *Emulation*

The application emulation is realised with the `unicorn` processor emulator [103]. The software requires the application to be loaded into memory and is able to handle all 64-Bit processor commands of the x86_64 architecture. Since `unicorn` provides only capabilities to emulate the processor, the emulation and association of other resources must be done separately. The analyser that employs the emulator has to acquire resources that are requested by the analysed binary (e.g. files to be opened or memory blocks to be allocated). The emulator provides a thin layer to map memory from the emulating application address space to the address space of the executed analysed binary. This process is shown in figure 26. Memory blocks are acquired by the analyser and set up to a specific size ($\mathtt{size}_N$) and permission set (`read`, `write`, `execute`, $\mathtt{perm}_N$). These blocks are assigned to a specific address ($\mathtt{addr}_N$) to the analysed binary executed in the emulator. This enables the emulator to work on an independent address space during the emulation. Additionally, since the memory blocks are accessible from the emulating application, residual data from the analysed binary can be investigated easily.

To evaluate the application state (consisting of the values of all processor registers as well as the content of assigned memory blocks) a so-called `hook` is used. A hook for `unicorn` is a callback which is invoked before a machine code instruction is executed.

This hook checks the application state and saves important information in the CFG that is build during the emulation. During emulation, the analysis pipeline is required to also emulate system calls that are issued by the investigated application. The emulator

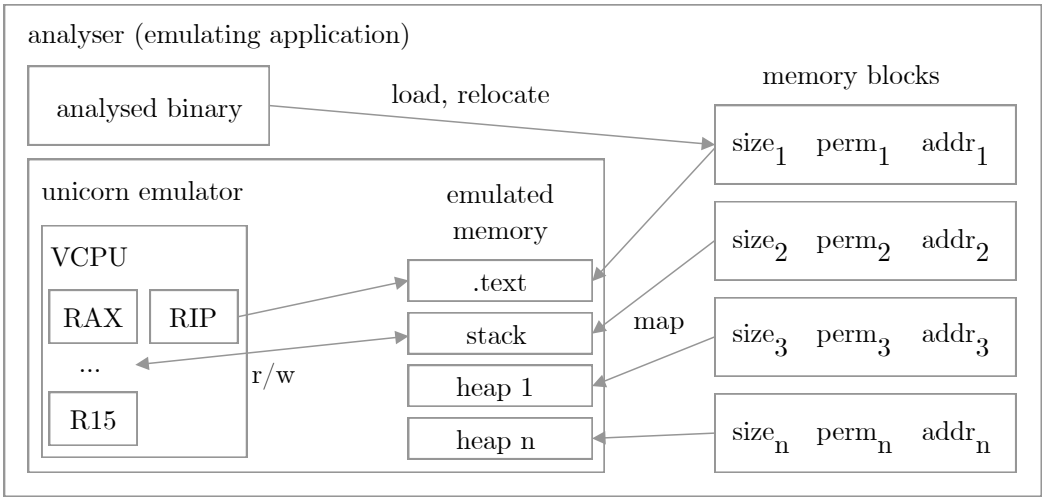


Figure 26: Transparent bi-directional mapping of memory blocks from the emulating application into the unicorn emulator space.

can analyse and detect the `SYSCALL` instruction. However, it is not capable of actually emulating system calls. Therefore this has to be done by the analysis pipeline.

For the analysis of the machine code applications the common system calls were implemented and behave the same way as their operating system counterparts. An additional logging of system call parameters is integrated to keep track about accessed resources (similar to SysTrace introduced in 5.3.1). All this data is added to the CFG during emulation. More rarely used system calls are implemented without any functionality but return appropriate return codes to the emulated application.

The emulation starts at the specified entry point given in the ELF file header. However, for correct emulation there is no workaround possible to skip the Linux loader. If the Linux loader is required by the application it must be invoked in the emulated environment. Its actions are recorded as part of the CFG and occurring system calls are analysed. The emulated Linux loader sets up a similar layout as described in 4.1.1 in the emulated memory space.

The results of the emulation is the CFG built during execution, a list of mapped memory blocks and their access permissions, recorded system calls with passed parameters and result codes, as well as a list of ELF binaries loaded and analysed during emulation.

6.2 CONFIGURATION GENERATION

The data collected with the static analysis and emulation are processed to generate configurations for the selected sandbox technologies in section 6.3. This data processing step is the crucial action that converts the abstract behaviour data from a CFG with additional

information about utilised resources and memory images into actual technology dependent configuration options. This process and its adaptability to arbitrary technology stacks is described in this section.

6.2.1 *Data Processing*

The collected data contain information about analysed application paths, system call code locations and execution during emulation. Additionally information about linked libraries and basic memory layout are available. These information must be preprocessed to deduce the required resources of the application and furthermore create a configuration for the execution sandbox.

The following resource classes were identified in chapter 3 with associated syscalls:

- Filesystem FS_1, FS_2
- Memory MEM
- Process and CPU Management, IPC CPU
- Permission Management ACCESS
- Network NET
- Device Management DEV
- Kernel Management KERN
- Time

The goal of the data processing step is to build rules for the sandbox technologies that are restrictive enough to prevent attacks but authorise anticipated behaviour. To reduce complexity of the system, the described preprocessing steps rely on basic system security rules. It is e.g. assumed that secret information can be stored in filesystem objects that deny `read` access to unauthorised users. It is not reasonable to move these tasks of general system security employment to the application sandbox too, as this would significantly increase system dependent configuration workload and complexity and is not required at all since the operating system is considered trusted (see 1.1).

Rules for Filesystem and Filesystem Objects

Interaction with filesystems is handled through the syscalls described in group FS_2. The system calls `statfs`, `fstatfs` and `sysfs` deliver information about a mounted filesystem (e.g. block size or free space) and are not harmful to the system. If there are high demands to prevent information disclosure, these syscall should be prohibited on an optional basis

(rule R_{FS1}). The flush of outstanding write buffers using `sync` poses no threat to the system (rule R_{FS2}). However, to bind or release filesystems from the computing system as well as the manipulation of the location of the `root` filesystem is potentially dangerous and should be prohibited. Therefore the syscalls `mount`, `umount`, `pivot_root` and `chroot` are disabled (rule R_{FS3}). System calls related to `inotify` indicate, that the application performs some kind of filesystem event monitoring [192]. It should be left to the system administrator if these supervision should be allowed or not. Nevertheless must this configuration be reflected in the sandbox configuration (rule R_{FS4}).

Access and interaction with filesystem objects like files, directories or links are handled through a lot more syscalls than filesystems are. However, syscalls that operate on file descriptors like `read`, `write` or `close` don't pose much of a threat to the system if there are appropriate checks in place that would perform authorisation checks on file descriptor creation time. Therefore, all filesystem resource syscalls that result in new file descriptors are crucial for the sandbox configuration. These syscalls include file-related ones like `open`, `openat`, `creat` and `open_by_handle_at` as well as directory-related interactions like `mkdir`, `rmdir` or `mkdirat`. The rule that should be enforced on these syscalls should allow file creation in temporary and write-enabled directories as well as read-access to anticipated user-readable system resources. Given that all read- or write-enabled filesystem resources are configured in sets Res_r (readable) and Res_w (writeable) and the application analysis determined accessed filesystem objects for reading as set Emu_r and writing access as set Emu_w, the rule R_{FS5} needs to restrict read actions to resources located in Emu_r and write action to elements in Emu_w. It should also be noted that overall execution is denied if emulated resource access contradicts the configured allowed resources. Therefore if $Emu_r \nsubseteq Res_r$ or $Emu_w \nsubseteq Res_w$, the application is not allowed to run at all on the computing system.

The manipulation of filesystem objects by renaming or moving them in the filesystem is also subject to rule R_{FS5}.

Similar to the R_{FS1} is the read access to filesystem object information with syscalls like `access`, `getxattr` or `stat`. These information should only be accessible if the administrator allowed such access, except if these resources are members of set Emu_r or Emu_w (rule R_{FS6}). However manipulation of such information for filesystem resources not in Emu_w must be denied (`chmod`, `setxattr` or `fchmod`, rule R_{FS7}).

Rules for Memory

Secure memory management is a key feature for secure application execution as security issues can result from invalid memory handling. The operating system performs several tasks to protect the system from malfunctioning as a result of erroneous behaviour of memory management. With the information available through the system call emulation there are only limited options available which itself might impact overall application execution [155].

Rule	Purpose	Enforcement
R_{FS1}	Read filesystem status information	optional
R_{FS2}	Allow filesystem write buffer synchronisation	mandatory
R_{FS3}	Disallow filesystem mount and `root` manipulation	mandatory
R_{FS4}	Configure filesystem event setup and monitoring	optional
R_{FS5}	Allow read/write access to only those resources in Emu_r and Emu_w	mandatory
R_{FS6}	Read filesystem object status information outside of Emu_r and Emu_w	optional
R_{FS7}	Change filesystem object status information outside Emu_w	mandatory

Table 14: Rules related the filesystem resource class

The data collected from `mmap`, `munmap`, `brk`, `mremap` and `madvise` provide an overview about memory layout during runtime and show readable, writeable and executable areas. Additionally, an estimate about the required amount of memory can be made. However using this estimate to establish rules that enforce the determined memory bounds might result in unwanted application termination. Therefore rule R_{M1} restrict overall memory consumption to a maximum of $N \times Emu_{Mem}$ where Emu_{Mem} holds the amount of required memory estimated during emulation. The factor N can be chosen freely. For this work, N is set to the number of sub-processes and threads observed during emulation.

A restriction of memory layout is hard to impose onto the untrusted application due to the different memory layout strategies of the operating system kernel. The emulator used for data collection pursues different memory layout strategies than the runtime OS. Therefore it is not possible to generate rules for this task from the collected data.

The remaining memory management system calls relate to the setup and usage of shared memory (`shmget`, `shmat`, `shmctl`, `shmdt`) and memory pages management (`shmdt`, `migrate_pages`, `move_pages`, `remap_file_pages`). These syscalls are rarely used, but a malicious or acceptable intend is hard to derive if they are registered. Nonetheless, in order to reduce the overall attack surface of potential threats these system calls should be disabled if the application analysis did not detect any of them. Therefore an attacker can't utilise their functionality (rule R_{M2}).

Rules for Process and CPU Management, Inter-Process Communication

Process management groups together many system calls with very different purposes. Syscalls that pose no threat to the system relate to waiting for certain events like checks

Rule	Purpose	Enforcement
R_{M1}	Restrict overall memory consumption to $N \times Emu_{Mem}$	mandatory
R_{M2}	Disable shared memory and memory page management if the analysis didn't find any related system calls	mandatory

Table 15: Rules related to the memory resource class

for available I/O (`poll`, `ppoll`, `select`), for other processes (`wait4`, `futex`), for a defined time (`nanosleep`, `alarm`) or a signal (`rt_sig*`, `pause`).

The creation of process-bound timers, messaging queues as well as I/O event notification facilities through the `epoll` interface are also reasonable syscalls that might occur in benign applications, especially if these applications spawn child processes or threads and require communication between these instances. No specific rules can be outlined for the presence of these features.

However the creation of child processes or threads itself might be utilised to generate rules for the sandbox. Rule R_{CPU1} describes the availability of an application to create new processes or threads using the `fork`, `clone` or `vfork` interface. If the analysis hasn't found any evidence for the requirement of multiprocessing inside the application, it can be safely disabled. Furthermore, even if child instances of the original application are spawned, the replacement of the application image inside the main process or a child item using the `execve` system call might indicate malicious behaviour. Even if the analysis registered this system call, caution is advised, as this enables arbitrary follow-up applications to be run by the program (rule R_{CPU2}).

The interaction of the application with other processes (already running on the computing system or spawned by the application itself) through `kill`, `tkill`, `tgkill` or `prctl` must be limited to the originating process and child items. Interference with other process items in the system must be averted (rule R_{CPU3}).

An application may also request the operating system to change overall scheduling parameters to increase the priority with which it is executed. To allow or deny these syscalls is optional, as often there are already operating system mechanisms in place that restrict these manipulation in a reasonable way (rule R_{CPU4}).

Similar to filesystem and memory management, there are syscalls related to process management that read process, kernel or IPC information (e.g. `getpid`, `getppid`, `ksym`). If these information are considered sensitive, these syscalls should be denied (rule R_{CPU5}).

System calls that weren't recorded during analysis should be disabled for this resource group too (rule R_{CPU6}).

Finally the supervision of other processes using the `ptrace` interface must be prohibited or at least be limited to child processes of the restricted application (rule R_{CPU7}). Gener-

ally, the presence of `ptrace` syscalls is an indicator of malicious behaviour if the analysed application does not focus on some sort of debugging or application analysis.

Rule	Purpose	Enforcement
R_{CPU1}	Disable the creation of child processes or threads if emulation indicated single-process/single-thread execution	mandatory
R_{CPU2}	Disable process image replacement at all or limit the interface to required linked applications	mandatory
R_{CPU3}	Allow signalling and configuration of other process items only for the application itself and spawned child items.	mandatory
R_{CPU4}	Disable process priority manipulation	optional
R_{CPU5}	Read process, kernel or IPC status information	optional
R_{CPU6}	Disable process management system calls that weren't anticipated during application analysis	mandatory
R_{CPU7}	Disable or thoroughly restrict the `ptrace` interface	mandatory

Table 16: Rules related to the process management resource class

Rules for Process Permission Management

Process runtime permission management allows a process to drop or elevate capabilities during its execution. Since this work focuses on applications that should run in a restricted environment where administrative tasks are not anticipated actions, permission elevation must be prohibited (rule $R_{ACCESS1}$). Additionally, the manipulation of user- or group-ownership of the running application must be prevented (rule $R_{ACCESS2}$). Furthermore, if a process item is bound to a certain namespace to limit its resource access, it is forbidden to undo or circumvent this security mechanism be re-associating itself with a different namespace (rule $R_{ACCESS3}$).

Again, to prevent information disclose, rule $R_{ACCESS4}$ limits syscalls to query the current process permission configuration.

Rules for Networking

Networking system calls behave similarly like filesystem object related ones. Once a network resource (a socket) is opened, it is assigned a file descriptor and syscalls for receiving/reading or sending/writing data can be used. That is the reason why the system calls `read`, `write`, `readv`, `writev`, `pread64`, `pwrite64` and `close` are also present in this category. However limiting these syscalls is not advised for this category either.

Rule	Purpose	Enforcement
$R_{ACCESS1}$	Disable permission elevation	mandatory
$R_{ACCESS2}$	Disable user- or group-ownership manipulation of the running process or its children	mandatory
$R_{ACCESS3}$	Prevent resource namespace re-association	mandatory
$R_{ACCESS4}$	Disable user- or group-ownership information retrieval of the running process or its children	optional

Table 17: Rules related to the runtime permission management resource class

Instead, the creation and management of client or server sockets is of interest for rule generation. For this work it is considered that the system allows connections to remote clients specified in the set Res_{CSock} and the creation of server sockets declared in the set Res_{SSock}. The rule R_{NET1} should ensure that only those connections to host and port combinations are permitted that are present in set Emu_{CSock} which was determined during application analysis. Additionally, if $Emu_{CSock} \nsubseteq Res_{CSock}$ the application is not allowed to be executed at all because connection attempts were found during emulation that are not white-listed in set Res_{CSock}.

Analogously to R_{NET1} rule R_{NET2} should only allow the creation of server side sockets for ports determined in Emu_{SSock}. The condition of $Emu_{SSock} \nsubseteq Res_{SSock}$, which would prohibit the software execution in the first place, applies here too.

Due to the fact that a large class of applications do not require network communication at all, rule R_{NET3} disallows those system calls if the emulation determined that no data transfer with external hosts is anticipated.

Finally the manipulation of the assigned host- or domain-name by the application should be prohibited (`sethostname`, `setdomainname`, rule R_{NET4}) .

Rules for Device Management

System calls related to device management operate on a low level interface to mostly interact with hardware components. The interface can be used for arbitrary input and output operations. For the described use-case of execution of untrusted application in shared infrastructures the access to direct I/O channels is a high risk. Infrastructure providers typically employ virtualisation or container technologies to separate user-space applications from the actual hardware the host operating system is executed on. Therefore the utilisation of these system calls are restricted in R_{DEV1}.

The `getcpu` system call is an exception from the described syscall family for direct I/O. This system call provides information about which CPU currently executes the calling thread. Since this system call allows only read access to a single information about

Rule	Purpose	Enforcement
R_{NET1}	Allow client socket creation only for host and port combinations in Emu_{CSock}	mandatory
R_{NET2}	Allow server socket creation only for address and port combinations in Emu_{SSock}	mandatory
R_{NET3}	If no entries are present in Emu_{CSock} and Emu_{SSock}, disable network communication features altogether.	mandatory
R_{NET4}	Prevent the manipulation of assigned host- and domain-name.	mandatory

Table 18: Rules related to the network resource class

the executing system the risk ensuing from it is small. However rule R_{DEV2} enables the prohibition of this system call to deny access to this information.

Rule	Purpose	Enforcement
R_{DEV1}	Prevent low level input/output channel creation, utilisation and destruction	mandatory
R_{DEV2}	Deny access to information returned by `getcpu`	optional

Table 19: Rules related to the device resource class

Rules for Kernel Management

Kernel management functions that are exposed to applications through the system call interface require administrative privileges. Therefore a utilisation of theses syscalls can be prohibited for the use-case targeted in this thesis.

Rule	Purpose	Enforcement
R_{KERNEL}	Prevent system calls that manipulate the kernel or kernel extension modules	mandatory

Table 20: Rules related to the kernel management resource class

Rules for Time Management

Finally, time related system calls can be divided into *read* and *write* categories. Syscalls like `gettimeofday`, `time`, `clock_gettime` and `clock_getres` read information about the current system time or the resolution of system timers. Because time information is often required for cryptographic operations or application tasks that involve random numbers it is reasonable to allow theses system calls. However, the exposition of information about clock resolution might enable an attacker to use high-precision timing attacks against the system ([193], [194]). Therefore the two rules R_{TIME1} and R_{TIME2} allow administrators to configure access to high-precision time information and more sensible clock resolution data.

System calls with write intentions can manipulate the system time of the operating system (`settimeofday`, `clock_settime`) or the way the system time is managed (`clock_adjtime`, `adjtimex`). Since these syscalls pose a risk to the system they normally require time manipulation capabilities on the operating system. These are prohibited by rule R_{TIME3} by default.

Finally the `clock_nanosleep` system call causes the system to suspend the invoking thread for the given amount of time. This system call poses no threat other than causing an infinite or undesirable long application runtime. Since this attack should be mitigated by other techniques no rule is defined that restricts the usage of `clock_nanosleep`.

Rule	Purpose	Enforcement
R_{TIME1}	Allow read access to system time information	optional
R_{TIME2}	Allow read access to information about system clock resolution	optional
R_{TIME3}	Deny the manipulation of system time and system time adjustment behaviour	mandatory

Table 21: Rules related to the device resource class

6.3 SANDBOX GENERATION

The configuration generated for the defined rules in 6.2 is used for the selected sandbox technology stack. This thesis uses the introduced `Namespaces` for this purpose. Additionally, `iptables` is employed to limit network access, `ulimits/prlimits` to protect against overconsumption of system resources and `seccomp`.

The selected technologies are combined to provide a comprehensive sandbox and to eliminate drawbacks of each other. Their overall impact to application performance as well as their effectiveness will be shown in chapter 7. Although this thesis relies on the

selected technologies the overall approach allows single or multiple security components to be exchanged. As long as the sandbox technology stack contains a virtual environment technology, a network filter and a resource restricting security framework, the configuration generation described below can be adapted to match the selected software components.

6.3.1 *Namespaces*

Namespaces provide a thorough mechanisms to shield system components. As described in section 5.3.3, they are the basis of virtualisation technologies like LxC Linux containers or Docker and capable to protect different classes of resources.

To protect information about the filesystems of the system as well as their manipulation (R_{FS1}, R_{FS3}), a namespace for mounted filesystems is employed. This namespace holds only information about the filesystem the application is running in. To shield the filesystem resources against attacks a filesystem namespace is created based on the `chroot` technology (R_{FS4}, R_{FS5}, R_{FS6}, R_{FS7}). The contents of this namespace are taken from the configuration determined by 6.2.1. An additional filtering is applied to ignore temporarily created files detected during the analysis as they must not be copied to the sandbox (R_{FS5}, R_{FS6}).

A namespace for user- and group-ids is used to separate the operating system information from the executed application ($R_{ACCESS2}$, $R_{ACCESS4}$). The administrative user inside the sandbox is mapped to an unprivileged user account to prevent access privileges escalation ($R_{ACCESS1}$, $R_{ACCESS3}$).

The separation of inter-process-communication and process manipulation is also achieved via two distinct namespaces (*IPC* and *process ids* namespace). The created process sandbox limits access to other processes (R_{CPU2}, R_{CPU3}, R_{CPU6}, R_{CPU7}) and therefore potential malicious interference. Due to the combination of limited inter-process-combination and process visibility restriction the rules to prevent these attacks can be realised (R_{CPU5}).

An *UTS* namespace is employed for the sandbox to protect the system against unwanted naming manipulations (R_{NET4}).

Finally a separate namespace for networking is created that is described in the following section.

A process running in the Linux system can be a member of different namespaces for the described purposes. It is possible for each of the namespaces described in 5.3.3 to select whether a process should share this namespace with the operating system or if it should be placed into a separated one. This work assigns new namespaces for all described components to the sandboxed process. This is realised with *namespace entering*. When the sandbox is configured, a new process with a `sleep` command is requested from the operating system with dedicated namespaces for the aforementioned purposes (*Namespace Provider*). Afterwards, the sandbox configurator creates a child process which will serve

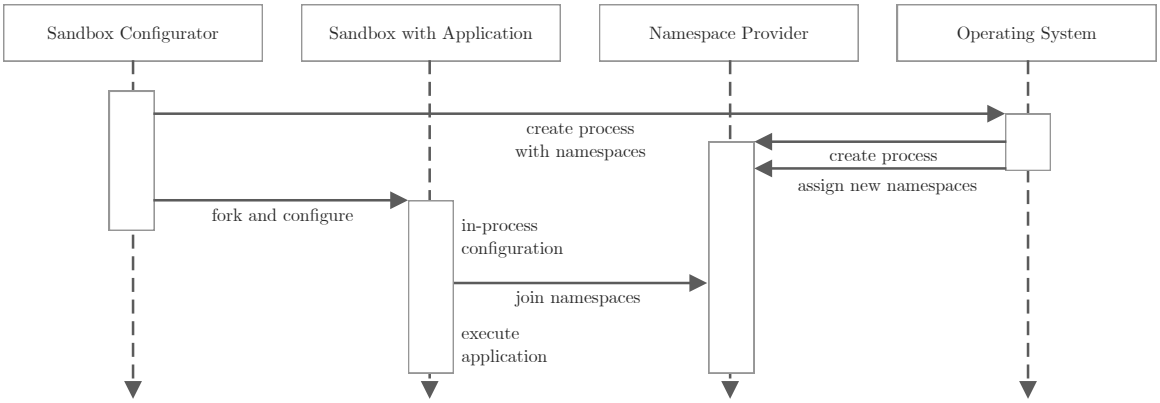


Figure 27: Utilisation of a dummy process as a *Namespace Provider* to use operating system created separated namespaces for the sandbox process.

as the sandbox. This child joins the set up namespace of the namespace provider and afterwards starts the sandboxed application. This process is shown in figure 27.

6.3.2 *Networking and* `iptables`

As described above, a network namespace is used to separate networking of the sandbox from the main operating system. Additionally, an `iptables`-based firewall is employed to discard all traffic from and to the sandbox that was not configured.

The realisation of R_{NET3} is done by a virtual network adapter that connects the network device inside the namespace of the sandbox with a bridge and the physical network device in the main operating system (28). If the configuration does not require networking at all this virtual device is not created and therefore the network namespace lacks the capability to receive or send data via any IP-based network connection.

If networking is required the configuration is transformed into a whitelist for `iptables` that allows connections from the sandbox to remote hosts according to rule R_{NET1} or from remote hosts according to rule R_{NET2}. All other traffic not matching this resulting whitelist is discarded. The filtering is performed inside the network namespace to prevent interference with any firewall rules present outside the namespace.

6.3.3 *Limits*

The configuration to mitigate issues concerning resource overconsumption are translated to a setup for the Linux technologies `ulimits` and `prlimits`. These limits are enforced on kernel level upon resource acquisition and actively prevent a resource consumption beyond the configured bounds.

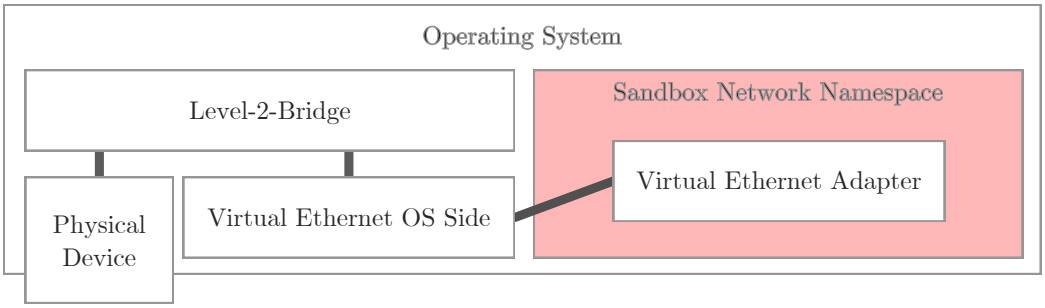


Figure 28: Setup to connect the isolated network adapter in the sandbox networking namespace to the physical device of the operating system.

Limits are configured with soft and hard boundaries. An attempt to consume more than the configured hard limit is denied by the operating system. The configuration specifies the amount of memory required. Yet, the fact that the CFG was constructed with example input data and that the application might behave different (but not maliciously) with different input data must be taken into account. The configured factor N is used to for this. In this work N is set to the total number of processes registered during emulation as described above.

The determined amount of memory is set as hard limits for the application (R_{M1}). To prevent attacks against the swapping mechanism of the operating system the total amount of lockable (and therefore non-swappable) memory is also limited to the amount of locked memory determined by the emulator (R_{M2}).

Configured CPU usage and priority manipulation is limited too for the selected test-cases (R_{CPU4}). However, it has to be considered to relieve this restriction for computational complex applications as a high CPU load alone is no indicator of malicious behaviour in their cases. Additionally to the CPU load the determined number of created processes and threats is also limited to the configured amount (R_{CPU1}).

Finally the filesystem and process management system can be protected against attacks that consume irregular amounts of resources through the limitation of open file descriptors (R_{FS4}, R_{FS7}). The number of detected open files is taken from the emulation results and a safety margin similar to the memory handling is added. This is required because additional file descriptors are used by the sandbox to transfer input data to the sandboxed application and to record its output.

6.3.4 *seccomp*

To protect the system against attacks `seccomp` is used in the generated sandbox to restrict system calls to those recorded by the emulator. The seccomp filter works with a whitelist of the recorded system calls. It is configured and enabled before the application is started.

Due to the design of seccomp, once established restrictions can not be disabled. Therefore the application is not able to circumvent these restrictions. The effectiveness of this filtering is stated in 5.3.1.

Although it is possible to provide more sophisticated rules to the seccomp filter to e.g. filter the amount of memory to acquire in a single syscall, these refinements are not employed in this work. This is due to the fact that these restrictions either can not prevent attacks mitigated by the formerly described methods or the emulation does not provide data to set up reliable and effective rules.

Seccomp is used to implement rules that require the restriction of functionally closely coupled to system calls like time/data management and operating system kernel interference (R_{TIME1}, R_{TIME2}, R_{TIME3}, R_{KERNEL}). It is also suited to prevent low-level access to devices if the configuration requires its limitation (R_{DEV1}, R_{DEV2}).

To protect the employed namespaces from manipulation by an attacker or a re-association of the sandboxed application with another namespace ($R_{ACCESS3}$) seccomp uses a dedicted rule to prevent this.

6.4 DATASETS AND SECURITY MEASUREMENT

The described techniques for the analysis of untrusted applications and the generation of sandboxes must be tested on representative example to verify their effectiveness. Furthermore, this work aims to provide insight about the efficiency of the designed solution. Therefore appropriate metrics must be selected from the introduced mechanisms in 2.3.

6.4.1 *Benign Testcases*

The designed solution must be able to analyse benign applications and provide an execution environment that allows these applications to run normally. To verify this, 100 example command line applications from the Debian distribution are used. For each application a testcase is specified. Such a testcase consists of a setup, execution and cleanup phase. The setup and cleanup phases are introduced to allow an automatic provisioning and tear down of analysis environments for this work. During the execution step the application is analysed and emulated with the techniques described above. All data that is used to generate the execution environment is collected during this step.

The list of benign testcases is shown in appendix F. The applications are taken from the `coreutils` [195] and `ntpdate` [196] package of the Debian distribution.

No explicit access to filesystem objects is preconfigured (Res_r and Res_w are empty). Only filesystem object access detected during emulation (Emu_r and Emu_w without file creation) result in filesystem objects present in the sandbox. The same principle applies to preconfigured accessible network hosts or server ports (Res_{CSock} and Res_{SSock}). Only

detected connections and server ports registered during emulation are allowed in the result sandbox.

6.4.2 *Malicious Testcases*

To verify the functionality and effectiveness of the generated sandboxes it is also required to test their protection against attacks. The execution environment must secure the described resource classes against different kind of attacks. To simulate such attacks 100 example applications are used that target different kind of resource classes. The 100 samples of malicious applications are divided into three categories:

EXPLOITS Malicious applications can target known vulnerabilities in operating system components or software parts to attack the described resource classes. The utilisation of so-called *exploits* rely on certain software and operating system versions. For this work ten exploits were chosen that use known vulnerabilities described in the *Open Vulnerability and Assessment Language* [197] for Debian at [198] and published at the *Common Vulnerabilities and Exposures* list available at [199]. If such a vulnerability is reported a CVSS score is also assigned.

VIRUSSHARE APPLICATIONS VirusShare.com [200] provides a database with malicious applications such as viruses, worms, trojan horses or rootkits. The database assigns each sample a unique identifier and links detection results from different anti-virus software products. Samples are sorted by operating system and type. Binary 64-Bit ELF samples were selected that are executable on the Debian test operating system. For the tests performed in this work 45 samples were picked from the database.

OWN TESTCASES To simulate attacks using dedicated system calls 45 own testcases are used. Each of these testcases focuses on the utilisation of a selected system call to attack a related resource of the operating system.

Each of the sample testcases is executed in a newly installed Debian system to rule out interference between them.

The selected samples ensure that real-world threats as well as specifically crafted applications are tested to prove the effectiveness of the running sandbox. A complete list of the utilised samples as well as references to their exploits or their VirusShare description are given in appendix G. The CVSS rating for each threat was calculated or taken from the source description and is used for the calculation in the applied metrics described below.

6.4.3 *Applied Metrics*

The selected 100 benign test cases are used to investigate the applications and create a execution sandbox. Each of these 100 result sandboxes is afterwards tested with the 100 malicious examples.

To calculate a score for the effectiveness of the sandbox approach the score is build with the N_{att} metric. As described in section 2.3 this metric counts the number of successful attacks against the system or its resource classes. The score calculated from the execution of the malicious application without any sandbox mechanism applied is used as the base value to compare the results against. Additionally, to take the CVSS of a threat into account, another metric based on N_{att} is used. Instead of counting the number of successful attacks this work defines N_{cvss} as the sum of their CVSS scores and uses this metric.

The cost metric C_{red} from section 2.3 is also used to quantify the overall costs of the applied metric. This work will focus on execution time to deduce the costs of the employed solution. Each additional millisecond (ms) the execution requires compared to the unrestricted run is calculated with virtual costs of 1.0.

7 RESULTS

This chapter presents the results of the chosen approach that was described in chapter 6. The first part describes the data used to perform an evaluation about the effectiveness of the designed solution. It presents the data from unmonitored and unrestricted executions of the benign testcases as well as collected information about the malicious testcases.

The second section shows the results of the analysis of the testcases that were collected using static analysis and emulated execution as described. It explains the data collected, their size as well as performance key points like runtime and resource consumption.

The last two sections present the runtime data collected from the execution of the benign and malicious testcases inside the generated sandboxed environment described earlier. This data is afterwards used for an overall evaluation with the metrics defined in 6.4.3 to finally rate the effectiveness of the sandboxing solution.

7.1 BASELINE EXECUTION

The tested applications are executed as described by a Debian 9.0 operating system. To differentiate phases in the execution process different timestamps were taken and will be explained in detail in the following section. To eliminate runtime variations caused by other factors that are not in the focus of this work each testcase was executed multiple times. The shown runtime results represent the mean values of these executions.

7.1.1 *Benign Testcases*

The benign testcases serve as the exemplary set of applications that need to be sandboxed. As described in 6, each of the 100 testcases consist of the setup, execution and cleanup phase. Once executed, the result code of the application was stored alongside the different execution phase timestamps. The return code of an application indicates if its execution was successful. Therefore a matching return code of the execution with and without the sandbox also indicates that the application performed the same way inside the sandbox as it would have without it.

The data shows a mean execution of the testcases of 574 ms (median = 505 ms). The setup time to prepare the execution take 13.5 ms on average (median = 4 ms) and the average time taken to cleanup any created results is 0.1 ms. Therefore the cleanup time can be safely ignored for further purposes. It is also visible in table 22 and H, that the return code 0 is the most common (90%). However this is not always the case, as some of

#	Command	t_{setup} [ms]	t_{exec} [ms]	$t_{cleanup}$ [ms]	Return Code
0	/usr/bin/chcon	2	503	0	256
1	/bin/ls	9	508	0	0
2	/bin/bash	5	505	0	0
3	/usr/bin/apt	3	1050	0	0
4	/usr/bin/hostid	4	506	0	0

Table 22: Excerpt of the execution results of the first five benign testcases. The full table can be found in appendix H.

the applications like `timeout` or `false` end with a non-zero return code even though they were executed as expected. To take this fact into account, the determined return codes presented here will be used to verify the a successful execution of the testcases inside the sandbox in 7.3.

7.1.2 *Malicious Testcases*

Each of the 100 malicious testcases was also executed in the same environment as the benign testcases. However the attack success is of interest for these testcases instead of result codes or setup and cleanup times. Several testcases ran as a hidden service for an infinite amount of time if not stopped manually. Therefore the execution of each malicious testcase was limited to 30 seconds. If the application did not terminate itself, it was terminated by the runtime framework and marked as forcefully ended (*killed*). After the execution of a testcase the system was investigated if the attack was successful. A description about each malicious testcase and the conditions to evaluate a successful infection are shown in appendix G.

#	CVSS	Return Code	Killed	Successful Attack
0	6.6	-11	no	yes
1	3.3	-	yes	yes
2	5.8	-	yes	yes
3	6.5	0	no	yes
4	3.8	-	yes	no

Table 23: Excerpt of the execution results of the first five malicious testcases. The full table can be found in appendix I.

Out of the 100 testcases, 83 attacks were successful and compromised the system. This value will serve as a baseline for the evaluation of the designed sandbox solution. The value of N_{att} for the unprotected Debian system is 83 and N_{cvss} is 448.2.

The unsuccessful attacks result from either patched software that appears to be no longer exploitable (e.g. testcase 94 and 97) or from virus samples that fail to connect to control servers before they attempt to compromise the system. Even thought a testcase might fail in a newly installed Debian system it is kept in the test set. This is done to ensure that the software required inside the sandbox does not re-enable former unavailable vulnerabilities. A malicious testcase that failed in the baseline execution but succeeded inside the sandbox can show the utilisation of new attack vectors introduced by the sandbox itself.

7.2 ANALYSIS

After the description of the execution results of the benign testcases in an unrestricted and unsupervised environment the results of the analysis of the given binary applications are presented. The first part shows the execution runtimes of the static analysis and the emulated execution in a similar manner to the unmonitored execution. The second part investigates the collected results and built rules for the sandbox.

7.2.1 *Execution Runtime*

The analysis of the benign testcases has to be done only once to collect the data required for the sandbox generation. However, runtime required for the analysis is a crucial factor for a selected metric in the evaluation for the solution. For this reason, the runtime of the static analysis as well as the emulation of the testcase is of interest and shown in tables 24 and 25.

#	Command	t_{setup} [ms]	t_{exec} [ms]	CFG Blocks	Syscalls	Resolving Errors
0	/usr/bin/chcon	3	504066	32655	123	471
1	/bin/ls	9	205358	34049	116	504
2	/bin/bash	6	2270471	78425	142	514
3	/usr/bin/apt	9	841	65	0	16
4	/usr/bin/hostid	7	47585	28217	105	421

Table 24: Excerpt of the analysis runtime results of the first five benign testcases. The full table can be found in appendix J. Execution times are given in ms.

The runtime results of the static analysis clearly show the complexity that is involved in this process. The execution times of the static analysis are significantly higher than the normal runtime. The mean ratio for the execution time compared to the testcase runtime (calculated by t_{exec} of the testcase analysis devided by t_{exec} of the testcase runtime) is 201.52 and the median is 104.86. This is due to the fact that the static analysis has to follow every possible execution path when the control flow graph is generated, whereas the plain executed application runs only a single code path.

#	Command	t_{setup}	t_{exec}	Processes	Threads	Syscalls
0	/usr/bin/chcon	3	4592	1	1	92
1	/bin/ls	5	9454	1	1	212
2	/bin/bash	2	3823	1	1	54
3	/usr/bin/apt	3	313911	1	1	766
4	/usr/bin/hostid	4	1571	1	1	40

Table 25: Excerpt of the emulation runtime measurements of the first five benign testcases. The full table can be found in appendix K. Execution times are given in ms.

Compared to the static analysis, the emulation of the application is much faster. However, it is also slower than the unmonitored execution of the testcase. The average execution time ratio between emulation end plain execution is 8.16 and the median equals 3.44. The increase in execution time is expected as the emulator is required to perform additional steps to analyse the instructions and virtualise system calls. Especially the introduced overhead through the emulation of system calls can be seen in the data in appendix K, where a higher number of recorded system calls correlate with a higher execution time.

The mean and median execution time ratios appear to be reasonable since emulation does not have to follow every possible execution path as static analysis does. But if each testcase and its execution time during unmonitored execution, static analysis and emulation are investigated anomalies can be seen (see fig. 29 and 30). E.g. the static analysis of testcase 3 takes less time than the emulation and is even lower than the unmonitored execution. These are indicators that the static analysis failed for this testcase. A further investigation showed that the command executed did not use the anticipated way of application entry point resolution as described for the static analysis (6.1.2). Another problem can be seen for testcase 19 where the static analysis is again faster than the application emulation. This was caused by a shared library, which name and path could only be determined during runtime and was unreachable for the static analysis due to its inability to anticipate runtime-memory contents as described in section 4.3.

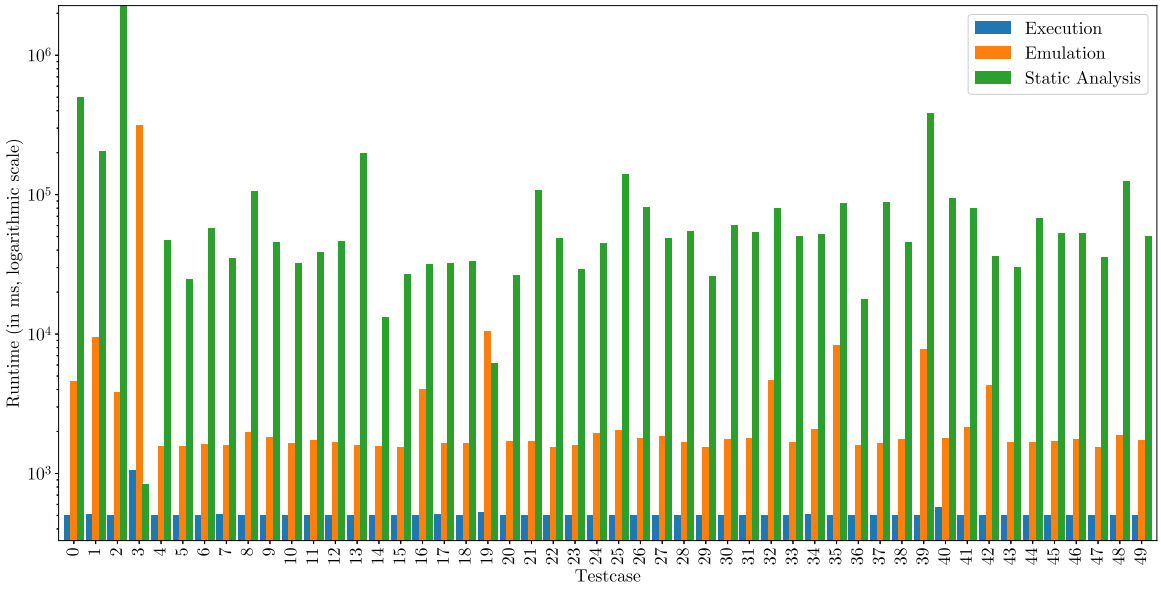


Figure 29: Runtime comparison of the benign testcases 0 to 49.

Finally the runtime of testcase 61 shows a potential problem with the emulator due to its faster execution than the unmonitored execution itself. The command for this testcase is the `timeout` command which causes the system execution to wait for a given amount of time. The emulator receives the desired system call but has chosen to ignore the timeout command because no other emulated processes or threads where pending execution. This finally results in an emulation runtime lower than the execution runtime.

7.2.2 *Results*

This section briefly describes the result data from the static analysis and the emulator that are used for the configuration generation process to build the sandbox. The static analysis generated the CFG for all testcases. The size of the graph with the number of nodes is given in appendix J. Each node represents a sequence of instructions. Instructions that are of special interest are those, which cause system calls to occur.

The analysis of the CFG of all 100 benign testcases has shown that their investigated machine code binaries do not directly contain any system call machine code instructions at all. Instead, system calls are issued by linked dynamic libraries like the `libc` that is provided by the operating system. This seems reasonable as a developer aims for platform independence and relies on implementation specific abstractions that are provided by such dynamic libraries. System calls, their identification numbers and parameter handling might be operating system dependent. It is therefore common practice to use the standardised API provided by OS libraries.

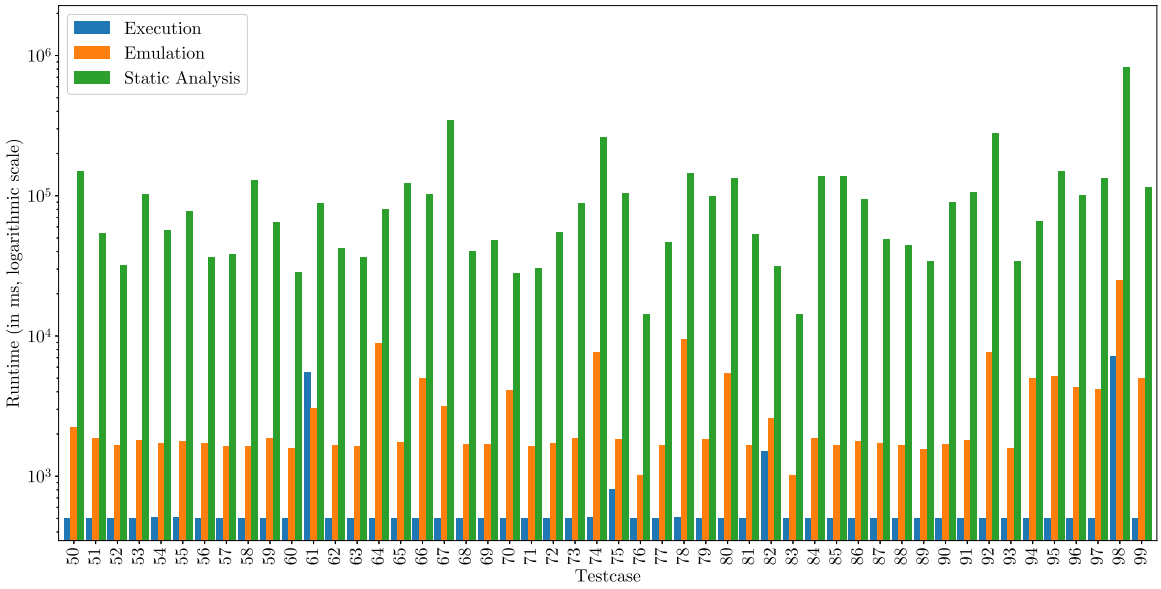


Figure 30: Runtime comparison of the benign testcases 50 to 99.

However, even though the investigated testcases do not issue system calls themself, this can not be used as a criterion to identify benign applications. The system call interface itself is a well documented API and might be utilised especially for low-level applications for e.g. hardware interaction like drivers or security frameworks.

The static analysis provides a list of files that are linked with the originating application. These list of files is used for the sandbox. This list is cross-checked and extended with the list of binaries and libraries that were loaded by the emulator. The result set of required binaries and libraries are used to configure the application in the filesystem namespace of the sandbox (Emu_r). Beside files required to run the application are files accessed (also Emu_r) and created by the observed testcase (Emu_w).

```
1 mmap(0x0000000, 0x002000, RW)
2 mmap(0x0000000, 0x229b20, RX)
3 mmap(0x0e27000, 0x005000, RW)
4 mmap(0x0000000, 0x2030f0, RX)
5 mmap(0x102e000, 0x002000, RW)
6 mmap(0x0000000, 0x39e960, RX)
7 mmap(0x13c5000, 0x006000, RW)
8 mmap(0x13cb000, 0x003960, RW)
9 mmap(0x0000000, 0x002000, RW)
```

Listing 13: Example memory mapping system calls captured by the emulator for testcase 2. The first parameter holds the address of the memory to map or 0x0000000 to acquire new memory. The second parameter specifies the size of the memory block to map/acquire. The third parameter shows the assigned access permissions.

The emulator captured the related system calls and was able to read the associated memory segments that hold the referenced filesystem object names/paths. This was not possible for the static analysis and requires the analysis through application emulation. The detected additional filesystem objects are added to Emu_r and Emu_w to make them available in the sandbox filesystem namespace.

Furthermore the emulator has detailed information about the number of open files, processes, threads and acquired memory to generate the configuration for the *Limits* technology. The produced output could be evaluated with the methods described in section 6.2 to extract the required data for the rule generation. An example for this process can be seen in listing 13. This listing shows the memory mapping system calls captured for benign testcase 2. If a `mmap` system call is invoked with target address `0x0000000` (first parameter) a new memory block is requested from the operating system with the given size in the second parameter. Therefore, a simple filtering for this system call with a first argument set to zero can be used to determine the amount of memory required to run the application.

7.3 SANDBOXED EXECUTION

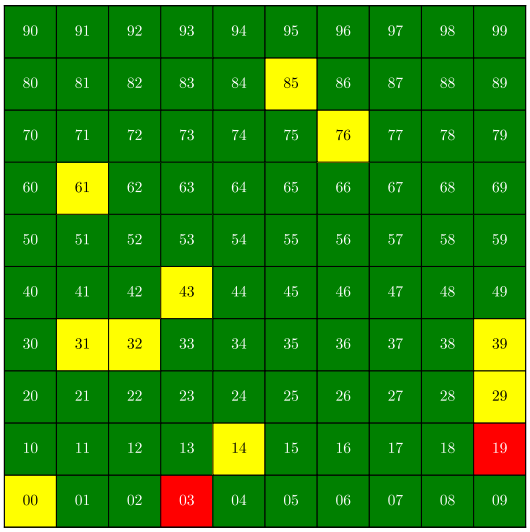
As stated before, the overall success of the sandbox solution is evaluated based on three aspects. The benign applications must continue to run as expected, malicious attacks must be prevented and the overall increase in runtime has to be measured. If a runtime increase is within acceptable bounds depends on the specific security needs of a infrastructure provider. After the results of static analysis and emulation were shown, this section displays the results from the execution of the benign and malicious testcases.

7.3.1 *Benign Testcase Results*

As described in the methods chapter a sandbox was created based on the data collected from each application in the benign testcase set. The application was executed inside the sandbox with the same input data. If the result codes from the baseline execution matches the result code from the execution inside the sandbox the execution is considered successful. If the return codes differ from each other the testcase was investigated further and is described below.

Figure 31 gives an overview of the results of the 100 testcases. Since the sandbox was built with enabled system call filtering via the seccomp technology might restrict the application too much and cause its termination two setups were tested. The left figure 31 a) shows the results for the sandbox with namespaces, iptables and limits technology enabled. The right figure 31 b) displays the results with the mentioned technologies and seccomp system call filtering enabled. The number of each testcase is shown in each square and its color represents the match of the investigated return codes. A green square

a) Namespaces, iptables and Limits



b) Namespaces, iptables, Limits and seccomp

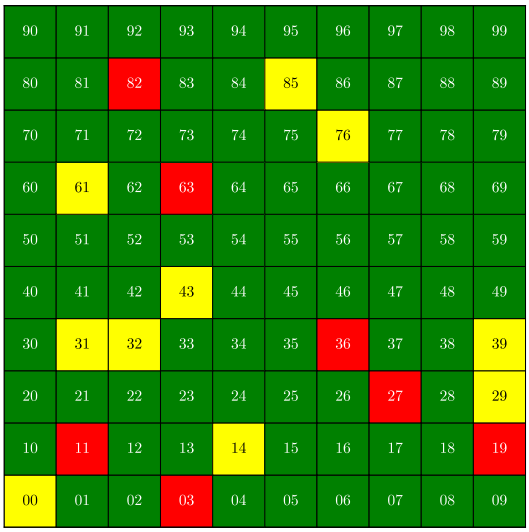


Figure 31: Visualisation of return code matches for executions of benign testcases with the sandbox mechanism and the baseline results from section 7.1. Subfigure a) shows the results with namespaces, limits and iptables enabled and subfigure b) shows the results with the additional filtering of system calls using the seccomp technology.

indicates a match, a yellow square shows a virtual match and red squares display a mismatch. A virtual match occurs due to the different technologies that recorded the return codes. Whereas the baseline application was executed without any framework in place the sandboxed application is embedded inside the processes of application supervision and sandbox creation 27. This results in different representations for the same return code e.g. for testcase 1, 14 and 29. Their baseline execution terminated with return code 256 (complement on two of -1 for one-byte sized return codes) and the sandbox execution reported an unsuccessful execution with error number 1. These return codes are considered to effectively match since they represent the same error in different ways through the return code.

It can be seen in figure 31 that 98% of the testcases match or virtually match the baseline execution. This rate drops to 93% for an execution with a sandbox additionally restricted with seccomp. Testcase 3 and 19 fail in both setups. Testcase 3 shows a problem of the `apt` application to write a temporary file inside the filesystem namespace. The file is part of $\mathrm{Emu}_{\mathrm{r}}$ because the file analysis of the emulator has determined its utilisation during execution as read only. Since the file is present in the original filesystem it is transferred to the filesystem namespace of the sandbox causing the application to fail when trying a write access. The application is executed as expected if the file is excluded manually. However, since manual white- or blacklisting of filesystem objects for single applications is not considered in this approach this testcase is considered as failed. Testcase 19 quits

with an error and a warning that the required command `dpkg-query` could not be found in the sandbox namespace. The file is also not present in the list of required files from the analysis ultimately causing the testcase to fail. A further investigation is required to determine the reasons why this file was not found in the analysis process.

The testcases 27, 36, 63 and 82 end inside the seccomp enabled sandbox with return code −31 which indicates a termination through the operating system. The reason for this termination can be an overconsumption of limited resources or the usage of a disallowed system call. Since a violation against the resource limits in place is ruled out due to the successful execution in the non-seccomp sandbox a violation against the seccomp-rules is assumed. To determine which rule was violated requires an investigation of each testcase and is beyond this scope. For this work it is assumed that the designed sandbox with system call filtering fails in the execution of the four mentioned testcases due to an over-restriction.

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]
0	/usr/bin/chcon	874	574	835	587	2	503
1	/bin/ls	839	597	876	579	9	508
2	/bin/bash	1404	602	1458	643	5	505
3	/usr/bin/apt	1661	583	1437	593	3	1050
4	/usr/bin/hostid	718	583	730	594	4	506

Table 26: Excerpt of the sandbox runtimes for the benign testcases. The full table can be found in appendix L. Times are given in ms. Setup and execution time for the sandbox without seccomp filtering are given in t_{nsetup} and t_{nexec} whereas the time information with the seccomp-enabled sandbox are displayed in t_{ssetup} and t_{sexec}. Setup and execution times from the baseline execution are given for comparison in baseline t_{setup} and t_{exec}.

The runtimes of the execution of the benign testcase in the sandbox are displayed partially in table 26 and fully in appendix L. The table shows the time required to setup and execute the application. Similar to figure 31 the runtimes for a sandbox without system call filtering (t_{nsetup}, t_{nexec}) and with system call filtering (t_{ssetup}, t_{sexec}) are given. It can be seen that no significant increase or decrease in overall runtime between the two versions is present. This is highlighted in figure 32 and 33 where the runtimes of the two sandbox versions are plotted compared to the baseline runtime of the testcases.

Table 26 also shows, that the setup time introduced by the sandbox build process (t_{ssetup}, t_{nsetup}) significantly increases the overall runtime. Furthermore, to get a better understanding of the impact of the sandbox on the application performance, the figures in 34 show the increase of t_{sexec} and t_{nexec} compared to the baseline execution time.

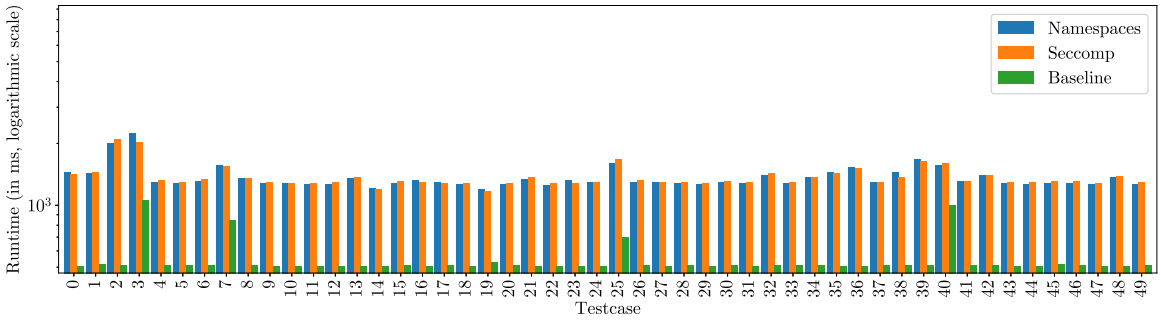


Figure 32: Runtime comparison between the execution of benign testcases inside the sandbox with namespaces, iptables and limits (*Namespaces*), sandbox with additional system call filtering (*Seccomp*) and baseline execution time for testcase 0-49.

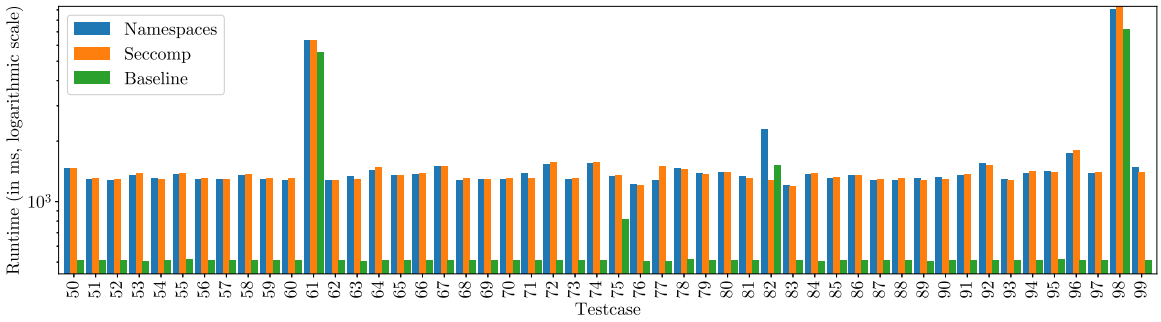


Figure 33: Continued sandbox runtime comparison for testcase 50-99.

Figure 34 indicates that the overall overhead due to the sandbox for all testcases has a median value of 12.9% (both with and without seccomp). The mean value for the execution overhead without seccomp is 14.7% and with seccomp 13.8%. These minor variations result from execution influences of the operating system that impact the comparatively small execution time. It can be seen that the efficient implementation of system call filtering has no impact on the execution time of the application inside the sandbox. However, it might prevent the overall functionality of the application as described earlier.

7.3.2 *Malicious Testcase Results*

The prior results have shown that the sandbox generated allows 93% to 98% of the benign test applications to run as expected with a median overhead of 12.9% on the execution time. This part investigates the effectiveness of the sandbox against the described 100 malicious testcases. The runtime of the malicious application is no longer of interest, since the overall prevention of the attack should be achieved. Results are shown for the seccomp enabled sandbox.

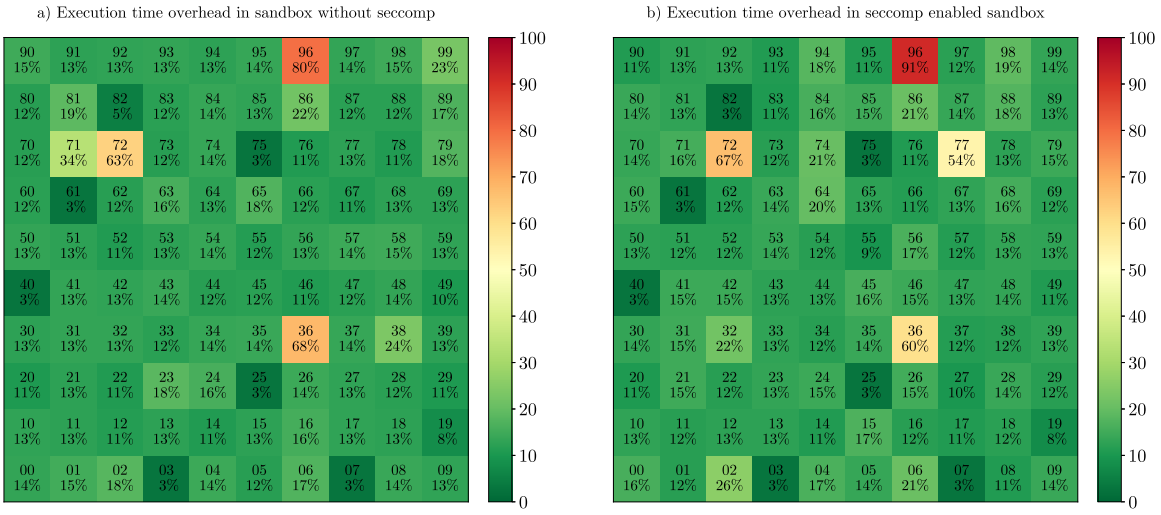


Figure 34: Execution time overhead in percent based on the baseline execution time in 7.1. Subfigure a) shows the results for the namespace, limits and iptables enabled sandbox. Subfigure b) shows the results with the additional filtering of system calls using the seccomp technology.

A visualisation of these results are shown as an overview in figure 35. For each sandbox that was build based on the analysis of a benign testcase (y-axis shows the number of the testcase) all 100 malicious testcases were executed (x-axis shows the number of the malicious testcase). The attack success against the system was plotted as a coloured square. A green square indicates a prevented attack. It can be seen that selected attacks were prevented by all sandboxes showing that they were effective against these attacks. A grey square indicates an unsuccessful attack, that has also been unsuccessful when executed without a sandbox due to the reasons described in the chapter 6.

The yellow and red squares in the figure are of special interest for the evaluation. A red square indicated a successful attack against the system. All detectors that are in place to detect the compromisation have been activated (see app. G for a description of these detectors). Yellow squares indicate a partially successful attack where some detectors evaluated the attack as successful whereas others detected no compromisation of the sandbox or the host system.

Finally a white square is used for benign testcase 3 and 12 where the application itself failed to execute in the sandbox.

Figure 36 a) shows the overall success rate of each attack against the system. It can be seen that a lot of attacks are prevented by all generated sandboxes. These attacks show a success rate of 0% in the figure. A total number 77 out of the 100 attacks were prevented successfully for all benign testcases. 8% of the attacks were always successful, whereas 11% of the attacks were successful in at least 90% of the sandboxed testcases. Table 27 shows these success rates with the associated attacks.

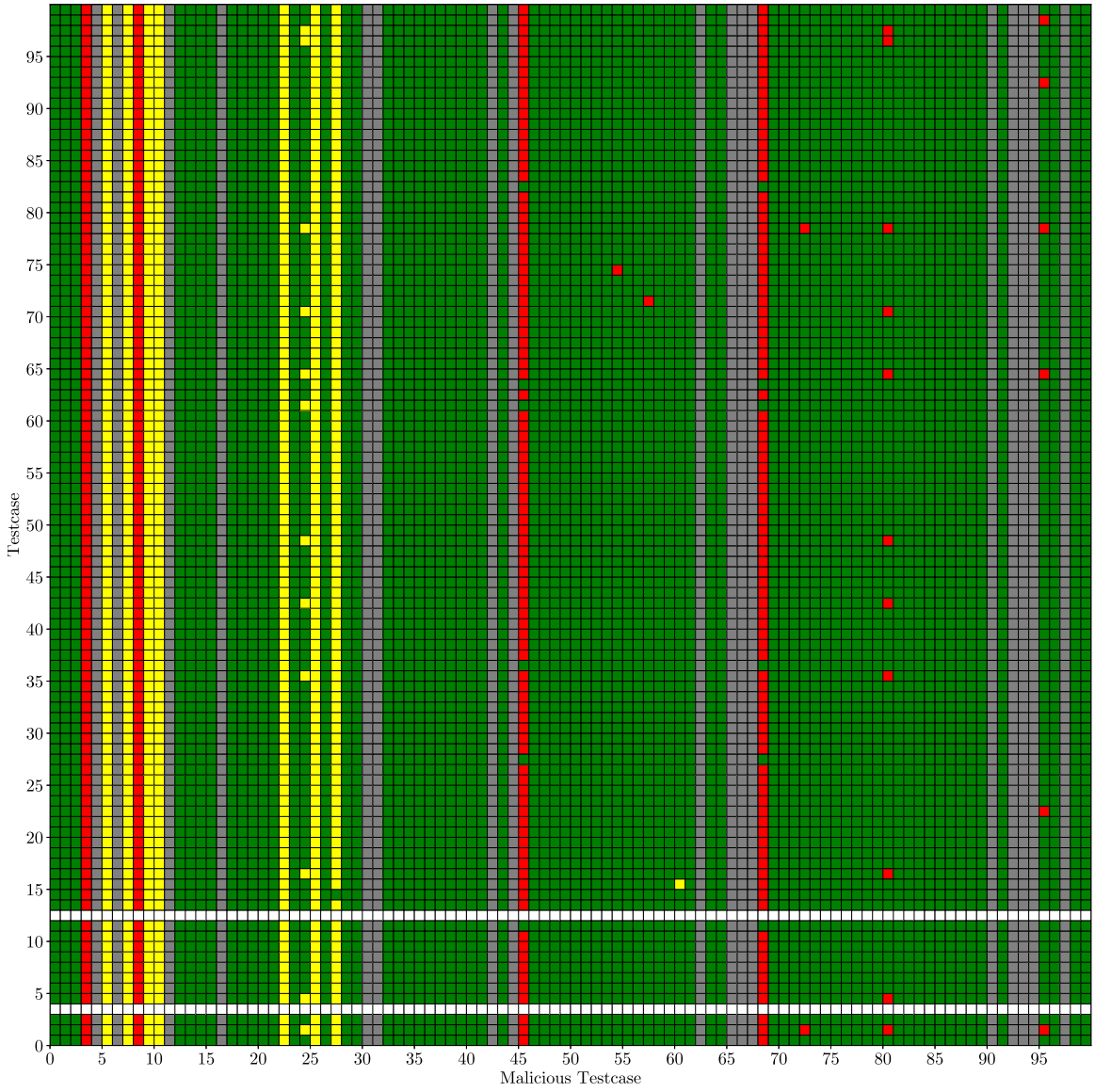


Figure 35: Visualisation of the protection of the sandbox build for each benign testcase against malicious testcases. A prevented attack is shown as a green square, a yellow square shows a partially prevented attack and a red square indicates the ineffectiveness of the sandbox. Grey squares indicate attacks that were unsuccessful in an unrestricted environment and are still unsuccessful inside the sandbox.

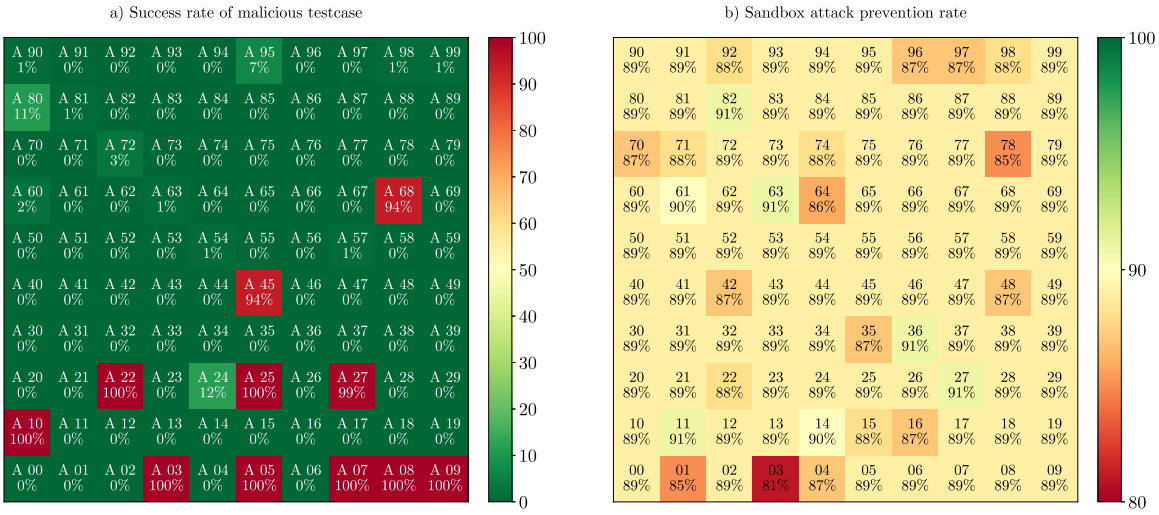


Figure 36: a) Attack success rate of the malicious testcases for the tested sandboxes. The percentage indicated the number of sandboxes for which the attack could be executed (partially) successfully. b) Attack mitigation rates of the sandboxes. The percentage indicated the number of attacks there were successfully prevented through the sandbox

Before the successful malicious testcases are described and the reasons why the attack was not mitigated are investigated, it is of interest to show the overall success rate of each sandbox. Since each benign testcase was used to build an individually configured sandbox, its individual effectiveness against the attacks can be measured as the number of attacks that were prevented successfully and therefore failed to compromise the system (inverse of N_{att}). An overview for the 100 sandboxes generated is given in figure 36 b). This attack failure rate ranges from 81% (testcase 3) to 91% (testcase 11, 27, 36, 63 and 82). The average attack failure rate is 88.7%. Compared to the baseline execution of the malicious testcases, where 83 attacks were successful, giving an attack failure rate of 17%, the effectiveness of the sandbox is clearly shown. This effectiveness results from the fact that a sandbox is based on the analysis of a specific application. Executing a different application inside such a sandbox can result in application termination. However the reuse of a sandbox for a class of applications similar to the analysed one used for the sandbox configuration is a promising approach but beyond the scope of this work.

Finally the attacks with a high success rate are of special interest. Their analysis can help to improve the sandbox to prevent those attacks. Malicious testcases 3, 7, 8, 9, 22, 25 and 27 were considered (partially) successful, because a malicious process was started that kept running until forcefully terminated by the runner process after a timeout. These processes disguised itself with various fake names in the list of running processes. Mitigating such an attack is difficult, since the execution of a process has to be enabled to run benign applications. Process renaming does not require any interaction with the operating system through system calls since it is execute solely in application memory. Therefore these

Success Rate	Malicious Testcases
100%	3, 5, 7, 8, 9, 10, 22, 25
99%	27
94%	45, 68
12%	24
11%	80
7%	95
3%	72
2%	60
1%	54, 57, 63, 81, 90, 98, 99

Table 27: List of malicious testcases that were successful against at least one sandbox of a benign testcase. The success rate shows the percentage of sandboxes of the benign testcases that were compromised by the malicious testcases given in the second column.

attacks were not able to be detected beforehand by the analysis or be mitigated by any of the taken security mechanism. However, it can be argued that the execution of a process in a disposable sandbox that is terminated after a given timeout is not harmful to the system even if it attempts to run forever, renames itself and plants infected files inside the sandbox. These threats are mitigated by the removal of the sandbox namespace and the runtime limit placed onto the application. Nevertheless, since the malicious process could execute itself and manipulate the process tree, its malicious behaviour is considered successful for this use case and has shown that a mitigation with the chosen approach of system call restriction is not possible. Nonetheless, the sandbox was able to restrict further malicious actions like connecting to command and control servers for testcases 7, 9, 22, 25 and 27.

Testcase 5 and 10 are bitcoin mining threats that occupy large amounts of computational capacity. The execution of tasks with high processor load does not involve system calls or other operating system resource access besides actual processing resources and memory. As a consequence, the imposed restrictions based on system calls did not prevent the mining application from execution. Even the restriction to the filesystem namespace with only files necessary to execute the benign application did not prevent the execution. This is because of the fact that the malicious testcases are statically linked binaries that carry all required dependencies with them to run inside environments that does not provide them. The attacks were evaluated as successful because they produced a high system load and did not terminate by its own. A more rigorous limit on processing resources could have prevented the attack, but this might cause benign applications to stop functioning correctly as described in 6.3.3.

The custom testcases 45 and 68 simulate attacks that cause a buffer overflow through a write out of bounds (45) and a stack overflow (68). Both threats target the overall availability of an application since they either force the operating system to terminate the application or use the normally inaccessible memory to perform further malicious actions. Somehow similar to testcase 5 and 10 these applications attack the memory associated with an application. These attacks do not issue system calls or try to break limits or the namespace established by the sandbox. If the attacker targets the availability of a benign application the attack is successful because of its termination by the operating system as a result of the detected memory access error. However ongoing attacks, that use the memory access problem as a starting point, are limited by sandbox and it is therefore considerably harder for the attacker to permanently compromise to operating system. These testcases emphasize the limits of system based detection as described in 5.3.1.

7.4 EVALUATION

After the presentation of the results from the sandbox execution, the overall evaluation of effectiveness and costs of the sandbox mechanism. The selected metrics N_{att}, N_{cvss} and C_{red} (see section 6.4.3) are used.

For a better overview and to account for the fact that a time consuming application analysis has to be performed only once to build a sandbox that can be used multiple times, C_{red} is given separately for application analysis and actual sandbox runtime. The complete list of results is given in appendix M.

Without the employment of the sandbox, the N_{att} was determined as 83 and N_{cvss} as 448.2. With the used sandbox the value of N_{att} decreases to a value in range 9 (testcases 11, 27, 36, 63, 82) to 15 (testcase 78). The corresponding N_{cvss} values drops down to 50.1. The average values for $avg(N_{att}) = 11.3$ and $avg(N_{cvss}) = 61.0$ show a significant increase in system security that was achieved with the employed sandbox. The required analysis to build the sandboxes causes average virtual costs for the selected testcases of $avg(C_{red}, analysis) = 109201$ and median costs of $median(C_{red}, analysis) = 55706$ as defined by the metric in section 6.4.3. This highlights the computational complexity and therefore virtually expensive costs of the approach. However, it was shown that these costs can be reduced if the system refrains from using static analysis and instead relies only on the application emulation. The virtual execution costs inside the sandbox account for an average value of $avg(C_{red}, execution) = 823$ and median costs of $median(C_{red}, execution) = 797$. These costs are comparatively small and highlight the efficiency of the selected technologies and therefore the sandboxes that are built upon these technologies.

8 CONCLUSION

The results presented in the previous chapter show that a significant increase in overall system security can be achieved when the introduced sandbox mechanism is used. A discussion of these results if given in this chapter. Furthermore, an outlook into future work is given to further develop the introduced sandbox mechanism.

8.1 DISCUSSION

The research questions given in 1.2 can be answered with the results given in chapter 7. It is feasible to generate a secure sandbox based on the analysis of system calls of applications. This sandbox is effective against attacks that aim for different assets of the service provider. This effectiveness was shown in this work by the investigation of 100 benign and malicious application that showed a decrease of the N_{cvss} metric from 448.2 to an average value of $avg(N_{cvss})$ of 61.0. This is a strong indicator of an overall increase of system security.

This work has also shown that the required data to build such a sandbox can be gathered automatically by a pipeline of steps. These steps include ELF file analysis, static analysis and application emulation. It was shown that the required sandbox configuration can even be assembled when the desired application is only available in its machine code form. The application of interest can be disassembled and emulated to collect the information required for employed sandbox technologies. Although a combination of ELF file analysis, static analysis and application emulation is used in this work it was shown that the static analysis is very time consuming and does not provide any data that is not available through file analysis or emulation. Therefore it can be concluded that the static analysis can either be removed from the analysis pipeline or should remain for cross validation purposes of the results between pipeline steps.

The introduced analysis and configuration pipeline made assumptions about the target operating system (Debian Linux) and processor architecture (64-Bit x86). However, the pipeline presented rules for associating resource accesses with different resource classes, assets and threats that are independent from these assumptions. The resource access supervision paradigm through the system call interface can be transferred to different operating systems and processor architectures. If a transformation of collected data to the specified rule sets in 6.2 is given and an implementation of these rules with suitable technologies in the target OS is performed, the effectiveness of the sandboxed environment presented in this work can be assumed.

Each sandbox that is generated with the presented pipeline is unique for the application on which analysis data it is build upon. The sandboxes used in this work differ from each other in the configuration of the utilised namespaces, resource limits and explicit system call filters. Therefore only those resource accesses and their associated system calls are allowed that are explicitly configured as permissible and provided resources are additionally separated from the operating system by the namespaces. A sandbox is reusable for consecutive executions of the same application but might not be used for other programs that perform different tasks.

Although the presented solution did improve the overall system security, it was not able to mitigate all threads. This highlights the problem of computing system provider to offer their capacities for multi-purpose application. A strong restriction of applications in their capabilities can result in benign application to seize to operate as expected, whereas less restrictions can result in successful malicious attacks or unwanted behaviour from applications. Service availability requirements oppose system security needs. This dichotomy can be seen in the presented results. The sandbox pipeline is too restrictive for selected benign testcases but to permissive for some malicious applications.

It should also be noted that the designed sandbox is rather a building block for a sophisticated security infrastructure than a comprehensive stand-alone solution. The presented solution suffers from the presented weaknesses that need to be compensated with other technologies.

However, the introduced questions had been answered through thorough tests with a large amount of benign and malicious testcases. Static analysis and application were successfully combined to anticipate the behaviour of unknown binary application. System calls were successfully used as the medium to analyse application behaviour. As described above and proved with the discussed results, the generation of a secure execution environment to protect service provider assets was also successful.

8.2 FUTURE WORK

The testcases used for the benign application consisted of basic Linux application. These command line tools serve dedicated purposes. Further research should be conducted to prove the effectiveness of the approach for high complex applications. Their analysis and runtime supervision is supposed to be more difficult. Nonetheless, this should be possible as long as the mechanism to collect the data for the sandbox configuration are suited to analyse the application.

Another approach to further improve this work lies in the reusability of sandboxes. The application analysis can required to configure a sandbox can take a considerable amount of time. A reuse of sandboxes for a class of applications that are similar to a sample program used for the analysis is a promising approach and can reduce the overall

analysis time overhead. But such an approach must still ensure that the sandbox protects the system against the different kinds of threats.

Finally a port of the described pipeline to other processor architectures and sandbox technologies like e.g. Docker could further improve the acceptance and overall effectiveness of the solution.

Part IV

APPENDIX

A

ONTOLOGY OF THE THREAT CONCEPT

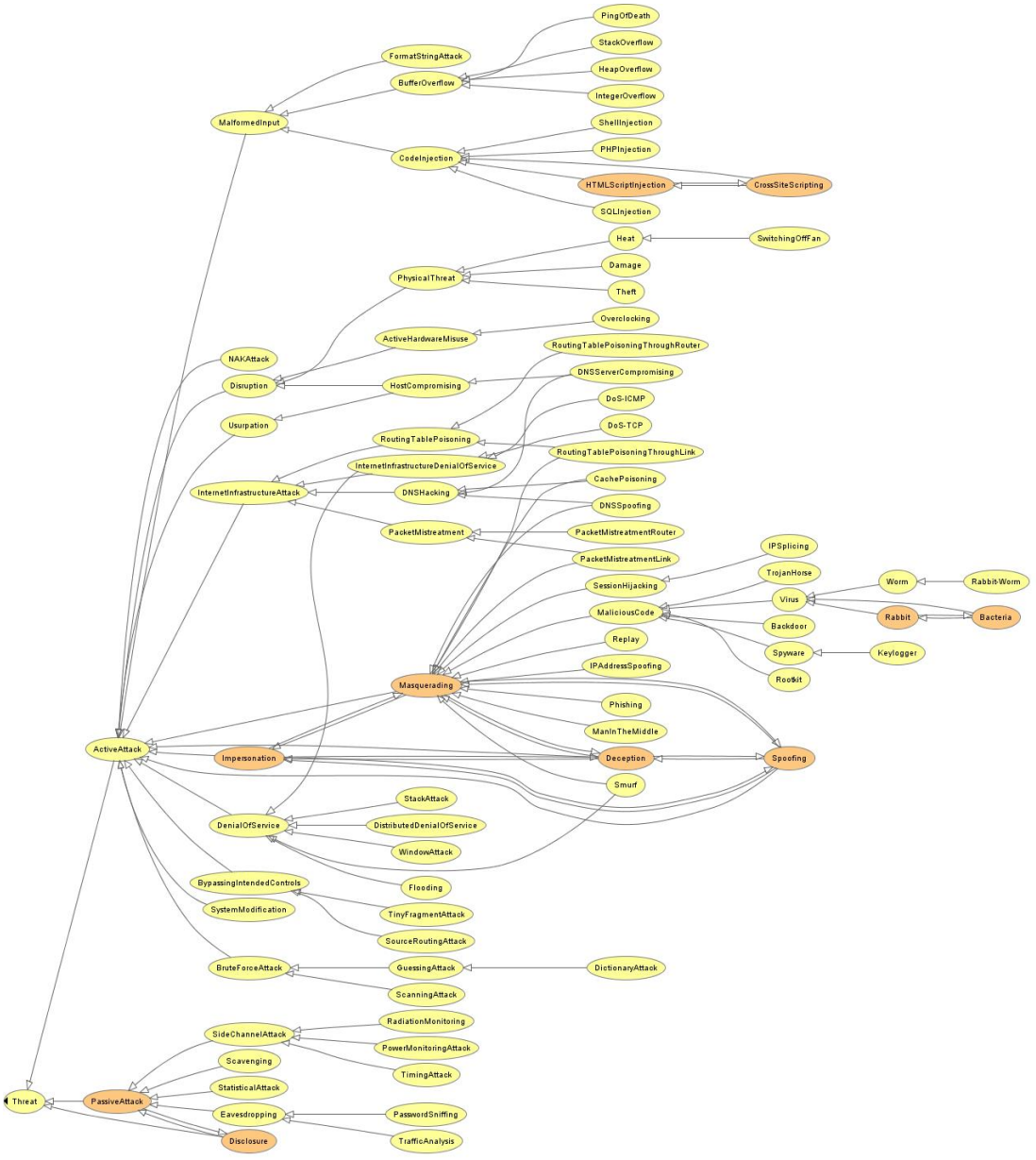


Figure 37: *Threat* ontology with sub-concepts according to [6] and published at `https://www.ida.liu.se/divisions/adit/security/projects/secont/Threat.jpg`.

B

ONTOLOGY OF THE ASSET CONCEPT

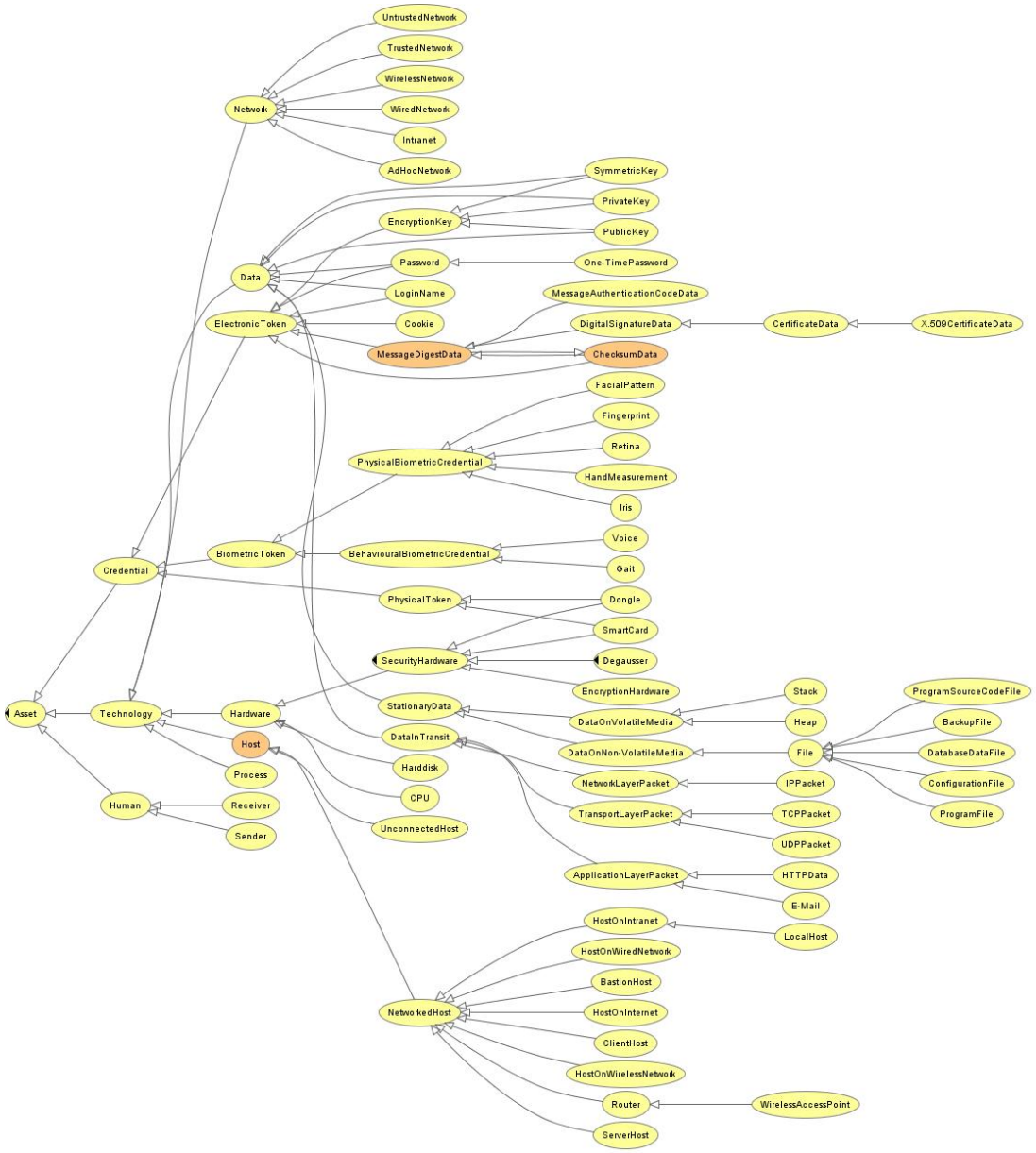


Figure 38: *Asset* ontology according to [6] and published at https://www.ida.liu.se/divisions/adit/security/projects/secont/Asset.jpg.

C

RESOURCE CLASS AND ASSET ASSOCIATION

Resource Class	Assets
FS_1	Stationary Data, Data on Non-Volatile Media, File, Program Source Code File, Backup File, Database Data File, Configuration File, Program File
FS_2	Stationary Data, Data on Non-Volatile Media
MEM	Data on Volatile Media, Stack, Heap, Data in Transit, Application Layer Packet, HTTP Data, E-Mail
CPU	Hardware, Host, CPU, Process
NET	Network, Untrusted Network, Trusted Network, Wireless Network, Wired Network, Intranet, AdHoc Network, Host, Host on Intranet, Host on Wired Network, Bastion Host, Host on Internet, Client Host, Host on Wireless Network, Router, Wireless Access Point, Server Host, Data in Transit, Network Layer Packet, IP Packet, Transport Layer Packet, TCP Packet, UDP Packet
DEV	Hardware, Harddisk, Security Hardware, Encryption Hardware
TIME	Host, One-Time Password, Certificate Data
ACCESS	Credential
KERN	Host

Table 28: Association of resource classes with asset concepts described by Herzog et al. [6]

D

SYSTEM CALL ASSOCIATION TO RESOURCE CLASSES

Resource Class	System Calls
FS_1	read, write, open, close, stat, fstat, lstat, lseek, pread64, pwrite64, readv, writev, preadv, pwritev, access, msync, flock, fsync, fdatasync, fallocate, truncate, ftruncate, getdents, getdents64, getcwd, chdir, fchdir,rename, mkdir, rmdir, creat, link, unlink, symlink, readlink, chmod, fchmod, chown, fchown, lchown, utimes, fchmodat, faccessat, openat, mkdirat, mknodat, fchownat, futimesat, newfstatat,unlinkat, renameat, linkat, symlinkat, readlinkat, utimensat, umask, utime, mknod, uselib, ustat, readahead, fadvise64, setxattr, lsetxattr, fsetxattr, getxattr, lgetxattr, fgetxattr, listxattr, llistxattr, flistxattr, removexattr, lremovexattr, fremovexattr, lookup_dcookie, sync_file_range, name_to_handle_at, open_by_handle_at
FS_2	statfs, fstatfs, sysfs, syncfs, mount, umount2, pivot_root, chroot, quotactl, inotify_init, inotify_add_watch, inotify_rm_watch
MEM	mmap, mprotect, munmap, brk, mremap, msync, mincore, madvise, shmget, shmat, shmctl, shmdt, mlock, munlock, mlockall, munlockall, swapon, swapoff, remap_file_pages, set_mempolicy, get_mempolicy, mbind, migrate_pages, vmsplice, move_pages, process_vm_readv, process_vm_writev

continued on next page

Resource Class	System Calls
CPU	poll, ppoll, rt_sigaction, rt_sigprocmask, rt_sigreturn, rt_sigpending, rt_sigtimedwait, rt_sigqueueinfo, rt_sigsuspend, rt_tgsigqueueinfo, pipe, pipe2, select, pselect6, sched_yield, pause, nanosleep, alarm, clone, fork, vfork, execve, exit, wait4, waitid, kill, semget, semtimedop, semop, semctl,msgget, msgsnd, msgrcv, msgctl, times, ptrace, sigaltstack, sched_setparam, sched_getparam, sched_setscheduler, sched _getscheduler, sched_get_priority_max, sched_get_priority_min, sched_rr_get_interval, getpriority, setpriority, personality, modify_ldt, prctl, arch_prctl, setrlimit, prlimit64, acct, getitimer, setitimer, getpid, setpgid, getppid, gettid, kcmp, tkill, tgkill, exit_group, futex, sched_setaffinity, sched_getaffinity, set_thread_area, get_thread_area, timer_create, timer_settime, timer_gettime, timer_getoverrun, timer_delete, timerfd_create, timerfd_settime, timerfd_gettime, epoll_create, epoll_create1, epoll_ctl_old, epoll_wait_old, epoll_wait, epoll_ctl, epoll_pwait, set_tid_address, mq_open, mq_unlink, mq_timedsend, mq_timedreceive, mq_notify, mq_getsetattr, unshare, set_robust_list, get_robust_list, splice
NET	read, write, close, pread64, pwrite64, readv, writev, sendfile, socket, connect, accept, accept4, sendto, recvfrom, sendmsg, sendmmsg, recvmsg, recvmmsg, shutdown, bind, listen, getsockname, getpeername, getsockopt, setsockopt, socketpair, sethostname, setdomainname
DEV	ioctl, iopl, ioperm, io_setup, io_destroy, io_getevents, io_submit, io_cancel, ioprio_set, ioprio_get, getcpu
TIME	gettimeofday, adjtimex, settimeofday, time, clock_settime, clock_gettime, clock_getres, clock_nanosleep, clock_adjtime
ACCESS	getuid, getgid, setuid, setgid, geteuid, getegid, getpgrp, setsid, setreuid, setregid, getgroups, setgroups, setresuid, getresuid, setresgid, getresgid, getpgid, setfsuid, setfsgid, getsid, capget, capset, setns

continued on next page

Resource Class	System Calls
KERN	`create_module`, `init_module`, `delete_module`, `get_kernel_syms`, `query_module`, `finit_module`, `nfsservctl`, `restart_syscall`, `kexec_load`, `add_key`, `request_key`, `keyctl`
Multiple Resources (ROOT)	`dup`, `dup2`, `dup3`, `fcntl`, `getrlimit`, `getrusage`, `signalfd`, `signalfd4`, `eventfd`, `eventfd2`, `tee`, `sync`
Other	`uname`, `sysinfo`, `syslog`, `vhangup`, `_sysctl`, `reboot`, `inotify_init1`, `fanotify_init`, `fanotify_mark`, `perf_event_open`

Table 29: 64-Bit Linux system calls and their corresponding resource classes

E

EXAMPLE SIMPLE CONTROL-FLOW GRAPH

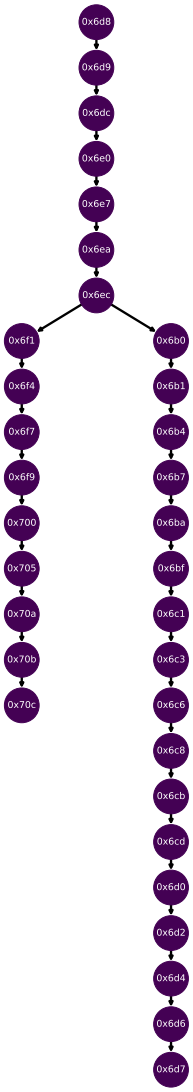


Figure 39: Example control-flow graph where each machine code instruction is represented as a node.

F

BENIGN APPLICATION TESTCASES

To collect representative information about benign applications and to assemble an overview of actions taken, 100 examples of common system tools from the Debian operating system were selected and analysed. The selected testcases were executed within the environment described in 6.1. The used programs and a short description for each application is given in the following table.

	Command	Description
0	/usr/bin/chcon	change file SELinux security context
1	/bin/ls	list contents of a directory
2	/bin/bash	start a new command line shell
3	/usr/bin/apt	search, install or remove software packages
4	/usr/bin/hostid	get the host identification number for the system
5	/usr/bin/link	call the link function to create a link to a file
6	/usr/bin/printf	format and print data to the command line
7	/usr/bin/truncate	shrink or extend the size of a file to the specified size
8	/usr/bin/pr	convert text files for printing
9	/usr/bin/sha1sum	calculate checksums using the SHA1 algorithm
10	/usr/bin/nice	set the scheduling priority of a process
11	/usr/bin/tee	redirect data to both the command line and other file related resources
12	/usr/bin/realpath	determine the absolute real path of an argument
13	/usr/bin/tac	output the specified file or stream line by line in reverse order
14	/usr/bin/printenv	output the current execution environment variables
15	/usr/bin/arch	print machine hardware name
16	/usr/bin/logname	print user's login name
17	/usr/bin/fold	wrap each input line to fit in specified width

Continued on next page

	Command	Description
18	/usr/bin/users	print the user names of users currently logged in to the current host
19	/usr/bin/dpkg	manage installed software packages
20	/usr/bin/paste	merge lines of files
21	/usr/bin/factor	prime factor numbers
22	/usr/bin/pathchk	check whether file names are valid or portable
23	/usr/bin/basename	return the last path component of a path
24	/usr/bin/dircolors	set colors for directory listings
25	/usr/bin/du	show statistics about the disk usage on the system
26	/usr/bin/shuf	generate random permutations
27	/usr/bin/sha224sum	calculate checksums using the SHA224 algorithm
28	/usr/bin/head	output the first lines of a stream or file
29	/usr/bin/tty	print the file name of the terminal connected to standard input
30	/usr/bin/join	join lines of two files on a common field
31	/usr/bin/test	check file types and compare values
32	/usr/bin/runcon	run command with specified security context
33	/usr/bin/base64	base64 encode/decode data and print to standard output
34	/usr/bin/sha512sum	calculate checksums using the SHA512 algorithm
35	/usr/bin/id	print real and effective user and group IDs
36	/usr/bin/dirname	return the directory components of a path
37	/usr/bin/numfmt	convert numbers from or to human-readable strings
38	/usr/bin/nl	number lines of files
39	/usr/bin/install	copy files and set attributes
40	/usr/bin/split	split a file into pieces
41	/usr/bin/od	dump files in octal and other formats
42	/usr/bin/groups	print the groups a user is in
43	/usr/bin/env	run a program in a modified environment
44	/usr/bin/tr	translate or delete characters

Continued on next page

	Command	Description
45	/usr/bin/comm	compare two sorted files line by line
46	/usr/bin/md5sum	calculate checksums using the MD5 algorithm
47	/usr/bin/nproc	print the number of processing units available
48	/usr/bin/pinky	find information about computer users
49	/usr/bin/uniq	remove duplicates in a given stream
50	/usr/bin/ptx	produce a permuted index of file contents
51	/usr/bin/sha256sum	calculate checksums using the SHA256 algorithm
52	/usr/bin/cksum	checksum and count the bytes in a file
53	/usr/bin/who	show who is logged on
54	/usr/bin/cut	remove sections from each line of files
55	/usr/bin/csplit	split a file into sections determined by context lines
56	/usr/bin/expand	convert tabs to spaces
57	/usr/bin/unexpand	convert spaces to tabs
58	/usr/bin/seq	print a sequence of numbers
59	/usr/bin/stdbuf	run a command with modified buffering operations for its standard streams
60	/usr/bin/unlink	remove an item from the filesystem
61	/usr/bin/timeout	run a command with a time limit
62	/usr/bin/tsort	perform topological sort
63	/usr/bin/expr	evaluate expressions
64	/usr/bin/stat	get filesystem information for a specified item
65	/usr/bin/tail	print the last lines of an input stream
66	/usr/bin/mkfifo	create a named pipe in the system
67	/usr/bin/sort	sort the given input
68	/usr/bin/nohup	run a command immune to hangups, with output to a non-tty
69	/usr/bin/fmt	simple optimal text formatter
70	/usr/bin/whoami	print effective userid
71	/usr/bin/sum	checksum and count the blocks in a file
72	/usr/bin/wc	count words, characters and lines in input

Continued on next page

	Command	Description
73	/usr/bin/shred	save delete files
74	/bin/cp	copy filesystem items
75	/bin/dd	dump content to the disk
76	/bin/false	return logical false values
77	/bin/readlink	return information about a link in the filesystem
78	/bin/vdir	list directory contents
79	/bin/rm	remove filesystem items
80	/bin/df	report file system disk space usage
81	/bin/rmdir	remove an empty directory from the disk
82	/bin/sleep	wait for the specified amount of seconds
83	/bin/true	return logical true values
84	/bin/date	print or set the system date and time
85	/bin/stty	change and print terminal line settings
86	/bin/ln	create soft- or hardlinks in the filesystem
87	/bin/mktemp	create a temporary file or directory
88	/bin/cat	concatenate files and print on the standard output
89	/bin/uname	print system information
90	/bin/chmod	change permissions of filesystem items
91	/bin/touch	change file timestamps
92	/bin/mv	move filesystem items to another place
93	/bin/sync	empty pending buffers in the system (filesystem, network, etc.)
94	/bin/mkdir	create a new directory
95	/bin/dir	list directory contents
96	/bin/chgrp	change the group ownership of a filesystem item
97	/bin/chown	change the ownership of a filesystem item
98	/usr/sbin/ntpdate	sync time via NTP with a remote server
99	/bin/mknod	make block or character special files

Table 30: Benign Application Testcases

G

MALICIOUS APPLICATION TESTCASES

For this work 100 testcases with real world and exemplary computer system threats were selected to verify the functionality of the developed solution. The table below lists a description of all 100 testcases, their source as well as a explanation how a compromisation of the system is detected. This work uses samples from VirusShare.com [200], known Debian exploits and own testcases to verify the effectiveness of the sandbox.

	Type	Source	CVSS
0	Virus	VirusShare [201]	6.6
	Backdoor that downloads and executes additional malicious files from internet resources. Files are written to the `/tmp` directory and executed. It is detected if the source downloader is present in the `/tmp` directory with the name `*.pl`		
1	Virus	VirusShare [202]	3.3
	A small torrent-based client and server that is used for file sharing. Infection is detected if the local server port 5001 is opened.		
2	Virus	VirusShare [203]	5.8
	Trojan horse which establishes a minimal python environment in the `/tmp` directory. It afterwards scans the network for control servers and other systems to further spread through the network. It is found via the detection of the suspicious python installation.		
3	Virus	VirusShare [204]	6.5
	DDoS client that listens on the local machine for instructions from command-and-control servers to attack desired targets. The process creates a disguised executable `dbuspm-session` to appear as a legitimate process. The detection of an executable with this name is used to detect a system compromisation.		
4	Virus	VirusShare [205]	3.8
	DNS denial of service virus. The application scans and connects to discover local and public network DNS servers and spams them with non-valid requests. An infection of the system is detected through monitoring of UDP traffic for unusual DNS-related ports `5353` and `5354`.		

Continued on next page

	Type	Source	CVSS
5	Virus	VirusShare [206]	3.2
	A mining virus that infects the system to harvest its calculation capacities. The miner connects to a specified mining pool and sends results to the connected server. The virus is detected through high system load and network connection through the configured mining port `5555`.		
6	Virus	VirusShare [207]	6.6
	A linux based trojan horse that creates a payload binary subfolders inside the `/tmp` directory. The detection of a successful infection scans for files with the `.elf` ending that are created by the trojan horse.		
7	Virus	VirusShare [208]	5.8
	Trojan horse virus that connects to control server `89.34.97.210` to get additional malicious payload. The connection attempt is done in a forked subprocess that disguises itself as a nameless process in the system. An infection detection utilises this fact and scans for unnamed processes and connections to the specified address.		
8	Virus	VirusShare [209]	6.5
	Another DDoS client that listens on the local machine for instructions to attack targets. Process creates a disguised executable `dbuspm-session` to appear as a legitimate process. The detection of an executable with this name is used to detect a system compromisation.		
9	Virus	VirusShare [210]	5.4
	Malware that connects to control server `206.81.11.231` to receive malicious payload. The virus works similarly to threat 7. An infection detection can again scan for unnamed processes and connections to the specified address.		
10	Virus	VirusShare [211]	3.2
	BitCoin mining variant virus that uses the target computational resources for mining purposes. The miner can connect to different mining pools and is detected through high processing load and utilization of the OpenCL computational library.		

Continued on next page

	Type	Source	CVSS
11	Virus	VirusShare [212]	4.3
	NTP server attacking virus. The application connects to NTP servers on the local network and tries to compromise them. The attack is detected on the system by looking for processes running in the `/tmp` directory that connect to other hosts via the NTP port.		
12	Virus	VirusShare [213]	6.6
	A Python-based trojan horse. Installs a minimal python execution environment and connects to command and control servers via HTTPS to receive further instructions and malware for the target system. The trojan horse is detected by searching the python environment.		
13	Virus	VirusShare [214]	6.6
	Simple backdoor that opens local port `6661` or `6662` on the target machine to enable other malicious applications to connect and cause further damage. An infection is detected by scanning for the given port.		
14	Virus	VirusShare [215]	6.6
	Simlar to threat 12 a minimal python implementation that starts a bash-like command line and allows command and control servers to execute arbitrary commands on the target system. A detection of the suspicious python environment is used to proof an infection with this virus.		
15	Virus	VirusShare [216]	6.6
	Variant of threat 14 with different attack vector used. A detection of the suspicious python environment is used to proof an infection with this virus.		
16	Virus	VirusShare [217]	8.3
	Backdoor that opens a private SSH server and connection to a TOR server to register the compromised system with generated random credentials. A successful compromisation of the system can be found by detecting connections to the TOR server `w4gfzjunvynjhpj6.onion.guide`.		
17	Virus	VirusShare [218]	3.2
	Mining virus that consumes resources of the target system for mining. Results are send to `pool.supportxmr.com` which is used to detect an infection of the system.		

Continued on next page

	Type	Source	CVSS
18	Virus	VirusShare [219]	8.3
	Linux system backdoor that connects to a command and control server at `luoxkexp.com`. Created subprocesses sleep until commands are received though the encrypted connection. An infection is detected by scanning the network traffic for connections to the mentioned control server address.		
19	Virus	VirusShare [220]	6.6
	Python-based backdoor that disguises itself as an SSH daemon process. The spawned and disguised process connects to the command and control server `89.248.172.165` for additional instructions and malware. Connections to this address are used to detect an infection with this virus.		
20	Virus	VirusShare [221]	7.3
	ELF trojan horse that creates several ELF binaries in the `/tmp` directory. These applications are executed and connect to address `185.81.158.47`. To detect an infection the system is searched for the described binaries.		
21	Virus	VirusShare [222]	7.3
	Similar to treat 20 this trojan horse installs an application in the `/tmp` directory. The control server is located at `5.189.153.241` and commands are exchanged using HTTPS. The system is searched for the described binaries in `/tmp` to detect an infection.		
22	Virus	VirusShare [223]	5.4
	Trojan horse virus that connects to control server `185.165.29.25` to download payload. The connection attempt is done in a forked subprocess that disguises itself again as a nameless process in the system similar to threat 7. An infection detection utilises this fact and scans for unnamed processes and connections to the specified address.		
23	Virus	VirusShare [224]	7.3
	Another variant of threat 20 with control server `208.67.1.57` that is also used to detect an infection.		
24	Virus	VirusShare [225]	7.3
	Another variant of threat 20 with control server `173.199.71.172` that is also used to detect an infection.		

Continued on next page

	Type	Source	CVSS
25	Virus	VirusShare [226]	5.8
	Another variant of threat 7 with an unnamed child process that connects to the command and control server at `217.61.16.74`. The infection detection can again scan for unnamed processes and connections to the specified address		
26	Virus	VirusShare [227]	3.9
	Ambiguous malware that connects to several different servers through port `37215`. Because the list of servers seems to change upon each execution, a connection attempt to a remote host with port `37215` is used to detect an infection.		
27	Virus	VirusShare [228]	8.8
	Backdoor implementation that relies on the busybox minimal execution environment system. The application establishes a connection to the control server at `45.32.1.44`. Due to the fact that the busybox application might not be present on the system or is used for a valid application the connection attempt to the control server is used to detect an infection with the virus.		
28	Virus	VirusShare [229]	9.0
	Complex virus and rootkit that installs several exploits into the system. It can be detected through searching for the hidden ELF files in the `/tmp` directory and the file `/etc/.zl`.		
29	Virus	VirusShare [230]	5.8
	Another variant of threat 7 but this one does not use an unnamed child process that connects to the command and control server. However a connection attempt to `167.99.107.136` can be used to detect the infection.		
30	Virus	VirusShare [231]	6.6
	Similar threat as the one described for threat 19. This DDoS virus also disguises itself as an SSH daemon process. The command and control server connected is located at `197.164.232.57`. Connections to this address are used to detect an infection with this virus.		
31	Virus	VirusShare [232]	9.0
	Hacking toolkit with interactive shell to launch several attacks. For the purpose of this test the installation and start of a SSH backdoor service is tested. The system is considered infected if the SSH service can be downloaded from the remote source and can be started on port 22223.		

Continued on next page

	Type	Source	CVSS
32	Virus	VirusShare [233]	7.3
	Another variant of threat 20 with control server `185.29.9.180` that is also used to detect an infection.		
33	Virus	VirusShare [234]	7.3
	Another variant of threat 20 with control server `62.4.24.136` that is also used to detect an infection. The virus prints a `TELNET` header to disguise itself as a valid application.		
34	Virus	VirusShare [235]	3.9
	Variant of threat 26 that operates with connections to control servers at port `52869`. A successful execution of the virus causes the system to seize accepting SSH connections which makes it unresponsible for management access. A successful attack is detected though outgoing connections to server with port `52869`.		
35	Virus	VirusShare [236]	8.3
	Worm variant of threat 16. Connections are made to TOR service `w4gfzjunvynjhpj6.onion.cab`. These connection attempts are used to detect infections.		
36	Virus	VirusShare [237]	6.6
	Variant of threat 19. However the process disguises itself as a cron daemon process. The spawned and disguised process connects to the command and control server `179.43.141.235`. Connections to this address are used to detect an infection with this virus.		
37	Virus	VirusShare [238]	7.7
	A stand alone backdoor virus. Upon execution the virus forks a child process and connects to the command server at `80.211.40.234`. Infections are detected through a detection of these connections.		
38	Virus	VirusShare [239]	7.3
	Another variant of threat 20 with control server `206.189.167.201` that is also used to detect an infection. The virus prints a `BUILD RAZER` output line.		
39	Virus	VirusShare [240]	3.2
	Mining virus variant of threat 17. Results are send to `pool.minexmr.cn` which is used to detect an infection of the system.		

Continued on next page

	Type	Source	CVSS
40	Virus	VirusShare [241]	6.5
	Simple backdoor virus that connects to `62.4.24.135` on port `6667` to fetch additional payload and instructions. The backdoor is considered active if such connections are found on the system.		
41	Virus	VirusShare [242]	9.0
	Backdoor that installs itself into the system into the `/bin` directory and sets up cron jobs using the cron daemon and the new systemd infrastructure. This is done to ensure persistent restarts of the virus in the case of a termination through the user or any other security mechanisms. After ensuring repetitive restarts, the virus connects to `121.42.144.22` to read further commands. This connection is used to detect the virus, because of the fact that the names and entries made to cron vary between runs.		
42	Virus	VirusShare [243]	3.2
	Mining virus variant of threat 17. Results are send to `pool.minexmr.com` which is used to detect an infection of the system.		
43	Virus	VirusShare [244]	7.3
	Another variant of threat 20 with control server `120.52.120.11` that is also used to detect an infection. The host mac address is shown on the command line upon execution of the virus.		
44	Virus	VirusShare [245]	6.4
	Backdoor variant that is loosely based on threat 20 with control server `222.186.56.25`. Connections to this address are used to detect an infection.		
45	Buffer overflow	Own testcase	5.0
	A buffer overflow due to an out of bounds memory buffer write by a string copy operation. This attack is detected by a segmentation fault by the system.		
46	UDP package spam	Own testcase	3.5
	The threat creates UDP packages to random targets (IP address and destination ports are chosen at random) of random size. A successful attack is detected, if UDP packages are send by the host system that do not belong to the benign application.		

Continued on next page

	Type	Source	CVSS
47	Custom made rootkit	Own testcase	8.7
	Infect the cron table of the system with a binary and send usage data to a remote server. The application re-schedules itself using the cron service of the system and installs itself in several locations of the system. A successful infection is detected if one of the malicious files is detected.		
48	Rootkit	Own testcase	8.7
	Variant of threat 47 that infects the `ls` command of the system.		
49	Unauthorized file read	Own testcase	5.5
	The attack opens all files in the system in read only mode. The attack is considered successful, if a system file can be read that does not belong to the application that is intended to run.		
50	Unauthorized file deletion	Own testcase	5.5
	When the attack is executed, it removes all files in the system. The attack is considered successful, if a file can be deleted that does not belong to the application that is intended to run.		
51	Rename and truncate system log file	Own testcase	3.3
	The application tries to empty `/var/log/auth.log` and to rename `/var/log/kern.log`. This can be used to disguise possible system infiltrations. The attack is evaluated as successful if one of the mentioned files is changed.		
52	Permission manipulation	Own testcase	4.4
	If the application is executed, all files in the `/home` directory will be assigned read, write and execute permissions for all users. This will emulate an attack against private files that are not accessible by default. If the access permission are changed by the application the attack is considered successful.		
53	Filesystem hardlink creation	Own testcase	3.3
	The attack creates filesystem hardlinks for all files found in the system. This can result in an filesystem inode consumption or may circumvent sandbox restriction imposed on absolute file paths. The attack is successful if at least one hardlink can be created.		

Continued on next page

	Type	Source	CVSS
54	File encryption attack	Own testcase	7.1
	During execution, the application opens all files on the system and encrypts them with a random key. The attack succeeds if a file that is not allowed to change is manipulated.		
55	Read extended file attributes	Own testcase	3.3
	The application reads all extended file attributes of all files in the filesystem. The unauthorized access to extended file attributes can leak information to an attacker. The threat is successful if attributes of files can be read.		
56	Process termination	Own testcase	5.5
	The attack emulates an attack against all running processes on the system. It tries to forcefully end any existing process. If this succeeds for any process present on the system, the threat is considered successful.		
57	System scheduler attack	Own testcase	3.3
	The application creates multiple new processes and threads until the system scheduler becomes overloaded and the system is no longer able to run. The attack is evaluated as successful if more than two additional processes are created.		
58	System limit removal	Own testcase	1.3
	The threat targets the `prlimit` security mechanism and tries to cancel any imposed restriction onto any running process. The attack is successful if limitations can be lifted by the attack.		
59	Process memory corruption	Own testcase	3.3
	For all running processes, the attack executes a write operation to the process memory. The attack is successful if the operating system reference to the memory of a process can be opened and the write operation can be executed.		
60	System resource consumption	Own testcase	5.5
	The attack forks several child processes and targets the system-wide entropy pool to consume available random data. The attack is successful if the unneeded processes can be created and the random data is consumed.		
61	Application deadlock	Own testcase	3.3
	The application creates several threads and forcefully creates a thread deadlock that causes the application to lock and never terminate. The attack is successful if it never terminates.		

Continued on next page

	Type	Source	CVSS
62	Unauthorized application supervision	Own testcase	6.1
	The attack tries to start a `ptrace`-based process supervision. If the application can attach to an arbitrary process with the `ptrace` interface the attack is successful.		
63	Memory consumption attack	Own testcase	3.3
	The attack tries to acquire all available system memory. If more system memory than 2 GB is consumed by the attack it is evaluated as successful.		
64	Shared memory consumption attack	Own testcase	3.3
	The attack tries to acquire shared system memory until no more free memory is available. If more memory than 2 GB is consumed by the attack it is evaluated as successful.		
65	Instantiate unwanted swap partition	Own testcase	3.3
	The attack tries to generate a new swap partition to force the operating system to swap out over-provisioned memory to an attacker-controlled data structure. The attack is successful if the new swap space is installed into the system.		
66	Read installed memory mapping	Own testcase	3.3
	The attack tries to read system wide present memory mappings. These mappings might leak sensitive information from running processes. If the application can access these information the attack is successful.		
67	Memory protection circumvention	Own testcase	6.1
	The attack tries to rewrite memory permissions. This enables an attacker to load malicious executable data into normally non-executable parts of the memory and run it. The attack is successful if memory permissions can be rewritten by the application.		
68	Stack overflow	Own testcase	3.3
	The attack causes a stack overflow. It is successful if the application consumes all available memory and is not terminated by the operating system.		

Continued on next page

	Type	Source	CVSS
69	Heap overflow	Own testcase	3.3
	The attack causes a heap memory overconsumption. It is successful if the application consumes all available memory and is not terminated by the operating system.		
70	INode resouce consumption	Own testcase	3.3
	The attack acquires all available inodes of the root filesystem and causes the operating system to become unable to create any new filesystem objects. The attack is successful if inodes are created that does not belong to the desired application.		
71	CHROOT escape	Own testcase based on [246]	5.5
	A `chroot` environment can be used to lock malicious applications into a separated space of the operating system. The attack tries to circumvent the imposed restrictions.		
72	Filesystem information disclosure	Own testcase	3.3
	The attack tries to read possibly sensitive filesystem information. The malicious application is successful if the information of any of the filesystems can be read.		
73	inotify filesystem watchdog attack	Own testcase	3.3
	This attack tries to install a filesystem supervision hook that utilises the `inotify` kernel interface. If the attacker is successful, the application is informed by the operating system whenever filesystem related operations are executed and may therefore access sensitive information. The attack is considered successful if the hook can be installed.		
74	Process information disclosure	Own testcase	3.3
	The attack tries to read information about itself. This information may contain information about user- and group-ids and therefore disclose system-dependent information to an attacker. The attack is successful if real information are read by the application.		
75	Effective user and group ID manipulation	Own testcase	4.4
	The application tries to elevate its own privileges be setting its effective user and group IDs that is used by the operating system to perform access checks. The attack is successful if permissions can be elevated by the application.		

Continued on next page

	Type	Source	CVSS
76	Application capability assignment	Own testcase	3.3
	The attacker tries to elevate its permissions by manipulating its assigned capabilities. Using this technique, the attacker can circumvent restrictions imposed by the operating system. If the capabilities can be extended, the attack is successful.		
77	Increase process scheduling importance	Own testcase	3.3
	This attack tries to elevate its system `nice` level and therefore heighten its affinity by the operating system scheduler, resulting in more computational time. If the nice level can be increased, the attack is considered successful.		
78	User and group ID manipulation for filesystem access	Own testcase	4.4
	This is a variant of threat 75 where the attack tries to manipulate user and group IDs that are used for filesystem write operations. If the attacker is able to create filesystem objects for a different user or group the attack is successful.		
79	System shutdown	Own testcase	5.5
	The attack tries to shut down the operating system. If the system reboots, the attack is successful.		
80	Malicious file download	Own testcase	6.1
	The attack tries to download a malicious file from `198.11.181.184`. If a connection to the desired address can be made the attack is successful.		
81	System socket information disclosure	Own testcase	3.3
	During its execution the application tries to execute the `lsof` command to read information about sockets (network and UNIX) present in the system. The attack is successful if these information can be accessed.		
82	Network information disclosure	Own testcase	4.4
	The application tries to send a file via UDP to a random destination. If the application succeeds to establish a connection to a remote resource via the desired port, it is evaluated as successful.		

Continued on next page

	Type	Source	CVSS
83	Server port creation	Own testcase	5.8
	The test tries to open a server socket on port `80` to emulate the creation of a backdoor. The attack succeeds if the server socket can be created and the application can bind to it.		
84	Socket pair creation	Own testcase	3.3
	If the attack is successful, it is able to create a pair of sockets that are used to communicate between two processes.		
85	Manipulate system hostname	Own testcase	4.4
	The application utilises the kernel interface to reset the system host name. If the host name of the system can be changed the attack is successful. These changes can lead to man-in-the-middle attacks by subsequent manipulations.		
86	UNIX socket corruption	Own testcase	5.8
	This attack simulates an attack against all active applications on the host that use UNIX sockets to listen for input data. During runtime the application connects to all available sockets and sends random data to corrupt the connected application. The attack is considered successful if data is send through a socket that does not belong to the application.		
87	System time manipulation	Own testcase	3.3
	A successful attack uses the kernel interface to manipulate the operating system time.		
88	System timetzone manipulation	Own testcase	3.3
	Similar to threat 87, this attack tries to manipulate the operating system timezone.		
89	DDoS attack	Own testcase	9.0
	The attacker establishes a multitude of connections to a specified target. The attack is successful if more than 50 connection can be established per second.		
90	Multiplication Overflow	Exploit [247]	5.5
	CVE-2000-1219; Force a multiplication overflow to terminate the `gcc`. Subsequent attacks may use the read after bounds to execute malicious code. The attack is successful, if the gcc is terminated by the operating system.		

Continued on next page

	Type	Source	CVSS
91	Out of bounds write	Exploit [248]	5.5
	CVE-2018-20376 - illegal 8 byte out of bounds write when using Tiny C Compiler. Subsequent attacks may use the read after bounds to execute malicious code. The attack is successful, if the application is terminated by the operating system.		
92	Git branch naming attack	Exploit [249]	8.8
	CVE-2014-9938 - `contrib/completion/git-prompt.sh` in Git before 1.9.3 does not sanitize branch names in the PS1 variable, allowing a malicious repository to cause code execution. If the code stored in the crafted PS1 variable is executed the attacker gain control over the executing shell and might initiate further attacks.		
93	KVM use-after-free	Exploit [250]	8.1
	CVE-2019-6974 - In the Linux kernel before 4.20.8, kvm_ioctl_create_device in virt/kvm/kvm_main.c mishandles reference counting because of a race condition, leading to a use-after-free. The attack is successful, if the application is terminated by the operating system.		
94	heap memory buffer overflow	Exploit [251]	9.8
	CVE-2018-0500 - `curl` might overflow a heap based memory buffer when sending data over SMTP and using a reduced read buffer. The attack is successful, if the application is terminated by the operating system.		
95	LD_LIBRARY_PATH injection	Exploit [252]	7.8
	CVE-2017-1000366 - `glibc` contains a vulnerability that allows specially crafted LD_LIBRARY_PATH values to manipulate the heap/stack, causing them to alias, potentially resulting in arbitrary code execution. If the crafted LD_LIBRARY_PATH variable is executed this attack is considered successful.		
96	Dirty COW	Exploit [253]	7.8
	CVE-2016-5195 - Race condition in mm/gup.c in the Linux kernel 2.x through 4.x before 4.8.3 allows local users to gain privileges by leveraging incorrect handling of a copy-on-write (COW) feature to write to a read-only memory mapping. If the exploit is executed successfully the attack is counted as successful too.		

Continued on next page

	Type	Source	CVSS
97	NULL-address mapping	Exploit [254]	9.0
	By following a certain codepath it is possible for userspace on a normal distro to map virtual address 0, which on an X86 system without SMAP enables the exploitation of kernel NULL pointer dereferences.		
98	Heap-based over-read	Exploit [255]	5.5
	CVE-2019-7665 - In elfutils 0.175, a heap-based buffer over-read was discovered in the function elf32_xlatetom in elf32_xlatetom.c in libelf. A crafted ELF input can cause a segmentation fault leading to denial of service (program crash) because ebl_core_note does not reject malformed core file notes.		
99	Invalid Address dereference	Exploit [256]	6.5
	CVE-2019-7663 - An Invalid Address dereference was discovered in TIFFWriteDirectoryTagTransferfunction in libtiff/tif_dirwrite.c in LibTIFF 4.0.10, affecting the cpSeparateBufToContigBuf function in tiffcp.c. Remote attackers could leverage this vulnerability to cause a denial-of-service via a crafted tiff file.		

Table 31: Selected System Threat Testcases

H

BENIGN TESTCASES BASELINE EXECUTION

#	Command	t_{setup}	t_{exec}	$t_{cleanup}$	Return Code
0	/usr/bin/chcon	2	503	0	256
1	/bin/ls	9	508	0	0
2	/bin/bash	5	505	0	0
3	/usr/bin/apt	3	1050	0	0
4	/usr/bin/hostid	4	506	0	0
5	/usr/bin/link	5	507	0	0
6	/usr/bin/printf	3	506	0	0
7	/usr/bin/truncate	336	508	0	0
8	/usr/bin/pr	8	505	0	0
9	/usr/bin/sha1sum	3	505	0	0
10	/usr/bin/nice	2	506	0	0
11	/usr/bin/tee	3	505	0	0
12	/usr/bin/realpath	3	505	0	0
13	/usr/bin/tac	5	504	0	0
14	/usr/bin/printenv	3	505	0	256
15	/usr/bin/arch	5	505	0	0
16	/usr/bin/logname	4	504	0	0
17	/usr/bin/fold	4	509	0	0
18	/usr/bin/users	2	504	0	0
19	/usr/bin/dpkg	3	527	0	0
20	/usr/bin/paste	6	507	0	0
21	/usr/bin/factor	3	505	0	0
22	/usr/bin/pathchk	3	506	0	0
23	/usr/bin/basename	5	503	0	0

Continued on next page

#	Command	t_{setup}	t_{exec}	$t_{cleanup}$	Return Code
24	/usr/bin/dircolors	2	503	0	0
25	/usr/bin/du	191	507	0	0
26	/usr/bin/shuf	5	506	0	0
27	/usr/bin/sha224sum	4	505	0	0
28	/usr/bin/head	6	506	0	0
29	/usr/bin/tty	3	504	0	256
30	/usr/bin/join	8	506	0	0
31	/usr/bin/test	3	505	0	256
32	/usr/bin/runcon	6	506	0	256
33	/usr/bin/base64	6	505	0	0
34	/usr/bin/sha512sum	4	508	0	0
35	/usr/bin/id	3	503	0	0
36	/usr/bin/dirname	2	503	0	0
37	/usr/bin/numfmt	4	506	0	0
38	/usr/bin/nl	4	505	0	0
39	/usr/bin/install	7	505	0	256
40	/usr/bin/split	424	576	0	0
41	/usr/bin/od	3	507	0	0
42	/usr/bin/groups	3	506	0	0
43	/usr/bin/env	4	504	0	32512
44	/usr/bin/tr	4	504	0	0
45	/usr/bin/comm	8	506	0	0
46	/usr/bin/md5sum	5	507	0	0
47	/usr/bin/nproc	3	503	0	0
48	/usr/bin/pinky	4	504	0	0
49	/usr/bin/uniq	6	505	0	0
50	/usr/bin/ptx	4	506	0	0
51	/usr/bin/sha256sum	5	506	0	0
52	/usr/bin/cksum	4	507	0	0

Continued on next page

#	Command	t_{setup}	t_{exec}	$t_{cleanup}$	Return Code
53	/usr/bin/who	3	504	0	0
54	/usr/bin/cut	5	507	0	0
55	/usr/bin/csplit	8	508	0	0
56	/usr/bin/expand	5	506	0	0
57	/usr/bin/unexpand	6	505	0	0
58	/usr/bin/seq	3	505	0	0
59	/usr/bin/stdbuf	4	507	0	0
60	/usr/bin/unlink	5	505	0	0
61	/usr/bin/timeout	4	5506	0	31744
62	/usr/bin/tsort	6	506	0	0
63	/usr/bin/expr	2	504	0	0
64	/usr/bin/stat	5	507	0	0
65	/usr/bin/tail	4	506	0	0
66	/usr/bin/mkfifo	3	506	0	0
67	/usr/bin/sort	4	507	0	0
68	/usr/bin/nohup	3	505	0	0
69	/usr/bin/fmt	4	507	0	0
70	/usr/bin/whoami	3	506	0	0
71	/usr/bin/sum	6	506	0	0
72	/usr/bin/wc	4	505	0	0
73	/usr/bin/shred	5	506	0	0
74	/bin/cp	5	508	0	0
75	/bin/dd	3	816	0	0
76	/bin/false	2	506	0	256
77	/bin/readlink	4	503	0	0
78	/bin/vdir	6	509	0	0
79	/bin/rm	4	505	0	0
80	/bin/df	3	506	0	0
81	/bin/rmdir	4	506	0	0

Continued on next page

#	Command	t_{setup}	t_{exec}	$t_{cleanup}$	Return Code
82	/bin/sleep	4	1504	0	0
83	/bin/true	3	505	0	0
84	/bin/date	3	503	0	0
85	/bin/stty	3	505	0	256
86	/bin/ln	6	504	0	0
87	/bin/mktemp	3	505	0	0
88	/bin/cat	4	505	0	0
89	/bin/uname	2	504	0	0
90	/bin/chmod	5	505	0	0
91	/bin/touch	3	505	0	0
92	/bin/mv	7	505	0	0
93	/bin/sync	4	507	0	0
94	/bin/mkdir	4	504	0	0
95	/bin/dir	8	507	0	0
96	/bin/chgrp	4	506	0	0
97	/bin/chown	4	504	0	0
98	/usr/sbin/ntpdate	3	7230	1	0
99	/bin/mknod	3	505	0	0

Table 32: Execution results of the benign testcases in an unrestricted environment. Execution times are given in ms.

#	CVSS	Return Code	Killed	Successful Attack
0	6.6	-11	no	yes
1	3.3	-	yes	yes
2	5.8	-	yes	yes
3	6.5	0	no	yes
4	3.8	-	yes	no
5	3.2	-	yes	yes
6	6.6	0	no	no
7	5.8	0	no	yes
8	6.5	0	no	yes
9	5.4	0	no	yes
10	3.2	1	no	yes
11	4.3	0	no	no
12	6.6	-	yes	yes
13	6.6	-	yes	yes
14	6.6	-	yes	yes
15	6.6	0	no	yes
16	8.3	-	yes	no
17	3.2	0	no	yes
18	8.3	0	no	yes
19	6.6	0	no	yes
20	7.3	0	no	yes
21	7.3	0	no	yes
22	5.4	0	no	yes
23	7.3	0	no	yes

Continued on next page

#	CVSS	Return Code	Killed	Successful Attack
24	7.3	-	yes	yes
25	5.8	0	no	yes
26	3.9	0	no	yes
27	8.8	0	no	yes
28	9.0	0	no	yes
29	5.8	0	no	yes
30	6.6	0	no	no
31	9.0	-	yes	no
32	7.3	0	no	yes
33	7.3	0	no	yes
34	3.9	0	no	yes
35	8.3	-	yes	yes
36	6.6	0	no	yes
37	7.7	0	no	yes
38	7.3	0	no	yes
39	3.2	0	no	yes
40	6.5	0	no	yes
41	9.0	-	yes	yes
42	3.2	0	no	no
43	7.3	0	no	yes
44	6.4	0	no	no
45	5.0	-11	no	yes
46	3.5	0	no	yes
47	8.7	-	yes	yes
48	8.7	0	no	yes
49	5.5	-	yes	yes
50	5.5	0	no	yes
51	3.3	0	no	yes
52	4.4	0	no	yes

Continued on next page

#	CVSS	Return Code	Killed	Successful Attack
53	3.3	-	yes	yes
54	7.1	0	no	yes
55	3.3	0	no	yes
56	5.5	0	no	yes
57	3.3	-	yes	yes
58	1.3	0	no	yes
59	3.3	0	no	yes
60	5.5	-	yes	yes
61	3.3	-	yes	yes
62	6.1	-	yes	no
63	3.3	-	yes	yes
64	3.3	-	yes	yes
65	3.3	0	no	no
66	3.3	0	no	no
67	6.1	-11	no	no
68	3.3	-11	no	yes
69	3.3	-	yes	yes
70	3.3	-	yes	yes
71	5.5	-	yes	yes
72	3.3	0	no	yes
73	3.3	-	yes	yes
74	3.3	0	no	yes
75	4.4	0	no	yes
76	3.3	1	no	yes
77	3.3	0	no	yes
78	4.4	0	no	yes
79	5.5	0	no	yes
80	6.1	0	no	yes
81	3.3	0	no	yes

Continued on next page

#	CVSS	Return Code	Killed	Successful Attack
82	4.4	0	no	yes
83	5.8	-	yes	yes
84	3.3	-	yes	yes
85	4.4	0	no	yes
86	5.8	0	no	yes
87	3.3	0	no	yes
88	3.3	0	no	yes
89	9.0	-	yes	yes
90	5.5	0	no	no
91	5.5	-11	no	yes
92	8.8	0	no	no
93	8.1	1	no	no
94	9.8	-	yes	no
95	7.8	-	yes	yes
96	7.8	-	yes	yes
97	9.0	-	yes	no
98	5.5	1	no	yes
99	6.5	1	no	yes

Table 33: Execution results of the malicious testcases in an unrestricted environment.

J

BENIGN TESTCASES STATIC ANALYSIS RESULTS

Table 34 shows the results of the static analysis of all benign testcases. The number of blocks in the generated optimised CFG are shown in the table as well as the number of system call invocations registered. Finally the number of graph generation failures during the analysis are also displayed. Such failures occur if a `JMP` or `CALL` instruction can't be resolved due to the problems described in chapter 4.

#	Command	t_{setup}	t_{exec}	CFG Blocks	Syscalls	Resolving Errors
0	/usr/bin/chcon	3	504066	32655	123	471
1	/bin/ls	9	205358	34049	116	504
2	/bin/bash	6	2270471	78425	142	514
3	/usr/bin/apt	9	841	65	0	16
4	/usr/bin/hostid	7	47585	28217	105	421
5	/usr/bin/link	10	24823	24793	86	373
6	/usr/bin/printf	9	57654	25878	86	379
7	/usr/bin/truncate	70	35163	25271	87	373
8	/usr/bin/pr	9	106614	29526	91	406
9	/usr/bin/sha1sum	10	45589	26936	86	377
10	/usr/bin/nice	10	32254	25103	88	373
11	/usr/bin/tee	9	39049	25450	87	382
12	/usr/bin/realpath	9	46165	26503	93	389
13	/usr/bin/tac	10	198080	34285	91	537
14	/usr/bin/printenv	9	13242	17407	79	250
15	/usr/bin/arch	9	26902	24905	87	378
16	/usr/bin/logname	9	31829	26112	103	395
17	/usr/bin/fold	10	32308	25279	87	380
18	/usr/bin/users	9	33119	25545	86	396
19	/usr/bin/dpkg	9	6241	5098	1	356

Continued on next page

#	Command	t_{setup}	t_{exec}	CFG Blocks	Syscalls	Resolving Errors
20	/usr/bin/paste	9	26388	24809	85	373
21	/usr/bin/factor	10	108576	29033	86	386
22	/usr/bin/pathchk	11	48874	25649	91	383
23	/usr/bin/basename	11	29427	24941	86	378
24	/usr/bin/dircolors	11	44803	25923	88	408
25	/usr/bin/du	44	141668	38253	99	565
26	/usr/bin/shuf	11	82134	27600	95	411
27	/usr/bin/sha224sum	11	49069	28223	86	377
28	/usr/bin/head	12	54376	26146	85	375
29	/usr/bin/tty	13	26216	24948	85	376
30	/usr/bin/join	11	60597	26989	87	380
31	/usr/bin/test	11	53782	24879	82	365
32	/usr/bin/runcon	12	79891	29097	111	416
33	/usr/bin/base64	12	50840	25555	86	376
34	/usr/bin/sha512sum	11	51825	29013	86	377
35	/usr/bin/id	12	86728	30661	113	461
36	/usr/bin/dirname	11	17895	24004	85	369
37	/usr/bin/numfmt	11	89044	28122	86	407
38	/usr/bin/nl	12	45879	33165	87	536
39	/usr/bin/install	14	384832	47471	167	740
40	/usr/bin/split	89	94725	27536	94	387
41	/usr/bin/od	12	80703	27736	86	398
42	/usr/bin/groups	12	36240	28177	102	444
43	/usr/bin/env	13	30339	25194	86	376
44	/usr/bin/tr	14	67376	26230	87	386
45	/usr/bin/comm	14	52686	26110	86	373
46	/usr/bin/md5sum	14	53174	26408	86	377
47	/usr/bin/nproc	14	35915	24951	86	373
48	/usr/bin/pinky	15	126074	34999	115	534

Continued on next page

#	Command	t_{setup}	t_{exec}	CFG Blocks	Syscalls	Resolving Errors
49	/usr/bin/uniq	14	50288	26601	88	378
50	/usr/bin/ptx	14	150063	37657	87	559
51	/usr/bin/sha256sum	15	53907	28270	86	377
52	/usr/bin/cksum	14	31986	25156	87	383
53	/usr/bin/who	14	102193	34262	105	494
54	/usr/bin/cut	13	57149	26469	86	392
55	/usr/bin/csplit	15	77682	34338	88	537
56	/usr/bin/expand	14	36337	25234	87	378
57	/usr/bin/unexpand	15	38281	25330	87	379
58	/usr/bin/seq	15	129124	29773	85	385
59	/usr/bin/stdbuf	15	64771	26686	89	386
60	/usr/bin/unlink	17	28777	24898	87	379
61	/usr/bin/timeout	15	89387	29066	101	396
62	/usr/bin/tsort	15	42525	25629	89	385
63	/usr/bin/expr	16	36675	27861	86	453
64	/usr/bin/stat	15	79782	30476	101	404
65	/usr/bin/tail	16	123128	28794	97	388
66	/usr/bin/mkfifo	16	103229	30940	114	483
67	/usr/bin/sort	17	348875	42780	159	487
68	/usr/bin/nohup	17	40060	25369	88	373
69	/usr/bin/fmt	15	48138	25780	86	374
70	/usr/bin/whoami	17	27895	25611	95	392
71	/usr/bin/sum	15	30302	24935	85	377
72	/usr/bin/wc	15	55140	26175	88	395
73	/usr/bin/shred	17	88565	27814	103	386
74	/bin/cp	17	263247	39392	141	594
75	/bin/dd	18	105307	29223	95	385
76	/bin/false	17	14233	16444	80	239
77	/bin/readlink	19	46910	26125	92	383

Continued on next page

#	Command	t_{setup}	t_{exec}	CFG Blocks	Syscalls	Resolving Errors
78	/bin/vdir	20	145584	33914	110	502
79	/bin/rm	17	99021	28885	103	445
80	/bin/df	18	133611	30655	99	394
81	/bin/rmdir	20	53059	26257	91	380
82	/bin/sleep	17	31637	24984	88	380
83	/bin/true	17	14358	16441	80	239
84	/bin/date	17	138253	32472	91	417
85	/bin/stty	19	138688	27788	90	379
86	/bin/ln	35	94667	28359	103	402
87	/bin/mktemp	23	49424	26078	94	385
88	/bin/cat	25	44234	25417	88	378
89	/bin/uname	24	34238	24903	87	378
90	/bin/chmod	26	90772	28406	98	441
91	/bin/touch	30	107154	29776	92	393
92	/bin/mv	26	281757	39865	144	598
93	/bin/sync	25	34276	25035	92	378
94	/bin/mkdir	27	66425	28044	102	443
95	/bin/dir	33	148890	33894	110	502
96	/bin/chgrp	26	100810	30837	107	489
97	/bin/chown	30	132745	32270	111	505
98	/usr/sbin/ntpdate	1	831738	17386	81	632
99	/bin/mknod	28	115154	31291	115	484

Table 34: Static analysis runtime results of the benign testcases. Execution times are given in ms.

K

BENIGN TESTCASES EMULATION RUNTIME RESULTS

Table 35 presents the runtime and selected data for the emulation of all benign testcases. The number of processes and threads emulated are shown as well as the total number of system calls. A system call was detected if the emulator issued the `x86_64` instruction `SYSCALL`.

#	Command	t_{setup}	t_{exec}	Processes	Threads	Syscalls
0	/usr/bin/chcon	3	4592	1	0	92
1	/bin/ls	5	9454	1	0	212
2	/bin/bash	2	3823	1	0	54
3	/usr/bin/apt	3	313911	1	0	766
4	/usr/bin/hostid	4	1571	1	0	40
5	/usr/bin/link	3	1564	1	0	36
6	/usr/bin/printf	3	1614	1	0	38
7	/usr/bin/truncate	53	1612	1	0	38
8	/usr/bin/pr	4	1990	1	0	53
9	/usr/bin/sha1sum	3	1822	1	0	46
10	/usr/bin/nice	3	1661	1	0	40
11	/usr/bin/tee	3	1749	1	0	50
12	/usr/bin/realpath	3	1667	1	0	41
13	/usr/bin/tac	3	1612	1	0	44
14	/usr/bin/printenv	3	1561	1	0	35
15	/usr/bin/arch	3	1556	1	0	40
16	/usr/bin/logname	2	4030	1	0	94
17	/usr/bin/fold	3	1663	1	0	45
18	/usr/bin/users	2	1657	1	0	46
19	/usr/bin/dpkg	3	10439	1	0	102
20	/usr/bin/paste	3	1708	1	0	55

Continued on next page

#	Command	t_{setup}	t_{exec}	Processes	Threads	Syscalls
21	/usr/bin/factor	2	1703	1	0	37
22	/usr/bin/pathchk	2	1558	1	0	36
23	/usr/bin/basename	3	1587	1	0	38
24	/usr/bin/dircolors	2	1962	1	0	39
25	/usr/bin/du	29	2053	1	0	42
26	/usr/bin/shuf	4	1780	1	0	47
27	/usr/bin/sha224sum	2	1849	1	0	46
28	/usr/bin/head	3	1691	1	0	42
29	/usr/bin/tty	3	1557	1	0	40
30	/usr/bin/join	4	1754	1	0	52
31	/usr/bin/test	2	1807	1	0	35
32	/usr/bin/runcon	3	4678	1	0	89
33	/usr/bin/base64	3	1685	1	0	45
34	/usr/bin/sha512sum	2	2066	1	0	46
35	/usr/bin/id	2	8387	1	0	165
36	/usr/bin/dirname	3	1594	1	0	38
37	/usr/bin/numfmt	2	1652	1	0	38
38	/usr/bin/nl	2	1754	1	0	45
39	/usr/bin/install	5	7838	1	0	120
40	/usr/bin/split	63	1786	1	0	42
41	/usr/bin/od	2	2142	1	0	44
42	/usr/bin/groups	2	4313	1	0	100
43	/usr/bin/env	3	1670	1	0	41
44	/usr/bin/tr	3	1682	1	0	41
45	/usr/bin/comm	4	1711	1	0	51
46	/usr/bin/md5sum	2	1765	1	0	46
47	/usr/bin/nproc	2	1545	1	0	42
48	/usr/bin/pinky	5	1871	1	0	49
49	/usr/bin/uniq	3	1740	1	0	45

Continued on next page

#	Command	t_{setup}	t_{exec}	Processes	Threads	Syscalls
50	/usr/bin/ptx	2	2241	1	0	47
51	/usr/bin/sha256sum	3	1885	1	0	46
52	/usr/bin/cksum	3	1668	1	0	47
53	/usr/bin/who	3	1808	1	0	46
54	/usr/bin/cut	2	1720	1	0	45
55	/usr/bin/csplit	3	1771	1	0	42
56	/usr/bin/expand	3	1730	1	0	45
57	/usr/bin/unexpand	2	1651	1	0	45
58	/usr/bin/seq	3	1647	1	0	38
59	/usr/bin/stdbuf	2	1872	1	0	43
60	/usr/bin/unlink	2	1598	1	0	40
61	/usr/bin/timeout	2	3086	1	0	0
62	/usr/bin/tsort	5	1660	1	0	49
63	/usr/bin/expr	2	1653	1	0	38
64	/usr/bin/stat	3	8932	1	0	163
65	/usr/bin/tail	3	1762	1	0	44
66	/usr/bin/mkfifo	3	5038	1	0	93
67	/usr/bin/sort	4	3148	1	0	101
68	/usr/bin/nohup	3	1700	1	0	41
69	/usr/bin/fmt	4	1699	1	0	46
70	/usr/bin/whoami	2	4115	1	0	92
71	/usr/bin/sum	3	1650	1	0	41
72	/usr/bin/wc	3	1716	1	0	43
73	/usr/bin/shred	3	1871	1	0	49
74	/bin/cp	4	7725	1	0	117
75	/bin/dd	3	1838	1	0	41
76	/bin/false	3	1025	1	0	27
77	/bin/readlink	5	1680	1	0	39
78	/bin/vdir	5	9581	1	0	197

Continued on next page

#	Command	t_{setup}	t_{exec}	Processes	Threads	Syscalls
79	/bin/rm	4	1848	1	0	48
80	/bin/df	2	5473	1	0	182
81	/bin/rmdir	3	1656	1	0	40
82	/bin/sleep	2	2605	1	0	36
83	/bin/true	2	1024	1	0	27
84	/bin/date	3	1857	1	0	45
85	/bin/stty	2	1682	1	0	40
86	/bin/ln	3	1780	1	0	46
87	/bin/mktemp	2	1734	1	0	49
88	/bin/cat	5	1660	1	0	44
89	/bin/uname	3	1562	1	0	40
90	/bin/chmod	5	1708	1	0	42
91	/bin/touch	2	1813	1	0	43
92	/bin/mv	3	7663	1	0	116
93	/bin/sync	3	1581	1	0	36
94	/bin/mkdir	2	5008	1	0	87
95	/bin/dir	4	5191	1	0	97
96	/bin/chgrp	5	4310	1	0	98
97	/bin/chown	3	4214	1	0	98
98	/usr/sbin/ntpdate	2	25247	1	0	192
99	/bin/mknod	3	4977	1	0	91

Table 35: Emulation runtime results of the benign testcases. Execution times are given in ms.

L

BENIGN TESTCASES SANDBOX RUNTIME RESULTS

The table 36 presents the runtime results from the benign testcases in the generated sandboxes. The results for the execution inside the sandbox with enabled namespaces, iptables and limits are shown as well as the results from the sandbox with additional system call filtering using seccomp. The time required to set up the sandbox is displayed in t_{nsetup} (without seccomp) and t_{ssetup} (with seccomp). The execution time of the application is shown in t_{nexec} (without seccomp) and t_{sexec} (with seccomp).

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]
0	/usr/bin/chcon	874	574	835	587	2	503
1	/bin/ls	839	597	876	579	9	508
2	/bin/bash	1404	602	1458	643	5	505
3	/usr/bin/apt	1661	583	1437	593	3	1050
4	/usr/bin/hostid	718	583	730	594	4	506
5	/usr/bin/link	703	573	711	582	5	507
6	/usr/bin/printf	719	597	723	616	3	506
7	/usr/bin/truncate	991	578	973	574	336	508
8	/usr/bin/pr	776	586	792	570	8	505
9	/usr/bin/sha1sum	715	574	722	580	3	505
10	/usr/bin/nice	702	574	709	575	2	506
11	/usr/bin/tee	696	573	711	568	3	505
12	/usr/bin/realpath	706	566	720	572	3	505
13	/usr/bin/tac	785	574	805	572	5	504
14	/usr/bin/printenv	645	564	633	565	3	505
15	/usr/bin/arch	705	575	708	596	5	505
16	/usr/bin/logname	732	588	730	569	4	504
17	/usr/bin/fold	708	581	713	570	4	509

Continued on next page

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]
18	/usr/bin/users	702	571	715	569	2	504
19	/usr/bin/dpkg	625	572	598	571	3	527
20	/usr/bin/paste	705	568	711	567	6	507
21	/usr/bin/factor	765	571	783	584	3	505
22	/usr/bin/pathchk	695	565	705	570	3	506
23	/usr/bin/basename	723	601	703	575	5	503
24	/usr/bin/dircolors	708	587	720	580	2	503
25	/usr/bin/du	1019	588	1051	631	191	507
26	/usr/bin/shuf	712	584	739	589	5	506
27	/usr/bin/sha224sum	717	576	733	558	4	505
28	/usr/bin/head	708	575	715	584	6	506
29	/usr/bin/tty	708	562	715	568	3	504
30	/usr/bin/join	723	578	724	583	8	506
31	/usr/bin/test	707	576	715	585	3	505
32	/usr/bin/runcon	817	581	810	625	6	506
33	/usr/bin/base64	707	574	718	577	6	505
34	/usr/bin/sha512sum	791	581	797	571	4	508
35	/usr/bin/id	862	579	859	579	3	503
36	/usr/bin/dirname	692	848	704	807	2	503
37	/usr/bin/numfmt	716	579	727	569	4	506
38	/usr/bin/nl	824	630	800	570	4	505
39	/usr/bin/install	1094	575	1058	578	7	505
40	/usr/bin/split	1006	563	1013	594	424	576
41	/usr/bin/od	727	579	731	584	3	507
42	/usr/bin/groups	830	577	819	585	3	506
43	/usr/bin/env	699	580	719	572	4	504
44	/usr/bin/tr	698	569	723	572	4	504
45	/usr/bin/comm	705	579	712	597	8	506

Continued on next page

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]
46	/usr/bin/md5sum	708	569	724	587	5	507
47	/usr/bin/nproc	704	567	703	573	3	503
48	/usr/bin/pinky	796	577	809	577	4	504
49	/usr/bin/uniq	707	563	723	568	6	505
50	/usr/bin/ptx	880	579	883	576	4	506
51	/usr/bin/sha256sum	722	576	733	574	5	506
52	/usr/bin/cksum	706	568	711	575	4	507
53	/usr/bin/who	786	571	799	578	3	504
54	/usr/bin/cut	721	584	714	572	5	507
55	/usr/bin/csplit	789	575	816	564	8	508
56	/usr/bin/expand	709	578	715	595	5	506
57	/usr/bin/unexpand	704	585	717	570	6	505
58	/usr/bin/seq	768	584	796	576	3	505
59	/usr/bin/stdbuf	718	578	727	578	4	507
60	/usr/bin/unlink	701	572	716	585	5	505
61	/usr/bin/timeout	766	5573	769	5564	4	5506
62	/usr/bin/tsort	702	572	713	571	6	506
63	/usr/bin/expr	745	588	719	577	2	504
64	/usr/bin/stat	862	576	859	615	5	507
65	/usr/bin/tail	759	599	786	574	4	506
66	/usr/bin/mkfifo	804	571	821	567	3	506
67	/usr/bin/sort	926	568	915	578	4	507
68	/usr/bin/nohup	710	573	711	589	3	505
69	/usr/bin/fmt	714	575	723	573	4	507
70	/usr/bin/whoami	730	568	731	578	3	506
71	/usr/bin/sum	699	683	710	592	6	506
72	/usr/bin/wc	713	829	713	849	4	505
73	/usr/bin/shred	722	571	738	572	5	506

Continued on next page

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]
74	/bin/cp	971	584	950	619	5	508
75	/bin/dd	760	581	778	577	3	816
76	/bin/false	657	563	636	565	2	506
77	/bin/readlink	711	573	715	779	4	503
78	/bin/vdir	890	573	865	580	6	509
79	/bin/rm	781	599	777	584	4	505
80	/bin/df	834	569	821	580	3	506
81	/bin/rmdir	729	606	732	577	4	506
82	/bin/sleep	698	1580	711	570	4	1504
83	/bin/true	630	571	633	562	3	505
84	/bin/date	790	575	794	587	3	503
85	/bin/stty	727	574	733	584	3	505
86	/bin/ln	733	622	738	615	6	504
87	/bin/mktemp	713	571	718	578	3	505
88	/bin/cat	703	572	711	600	4	505
89	/bin/uname	712	593	712	571	2	504
90	/bin/chmod	733	586	730	567	5	505
91	/bin/touch	779	574	795	574	3	505
92	/bin/mv	973	578	942	579	7	505
93	/bin/sync	707	578	710	566	4	507
94	/bin/mkdir	811	572	817	599	4	504
95	/bin/dir	826	587	834	572	8	507
96	/bin/chgrp	814	916	822	975	4	506
97	/bin/chown	805	577	834	571	4	504
98	/usr/sbin/ntpdate	741	8330	748	8640	3	7230
99	/bin/mknod	854	626	816	577	3	505

Continued on next page

#	Command	t_{nsetup} [ms]	t_{nexec} [ms]	t_{ssetup} [ms]	t_{sexec} [ms]	Baseline t_{setup} [ms]	Baseline t_{exec} [ms]

Table 36: Sandbox runtimes for the 100 benign testcases. The times for setup and execution are given in ms. Timings for the sandbox without seccomp filtering are given in t_{nsetup} and t_{nexec} whereas the time information with the seccomp-enabled sandbox are displayed in t_{ssetup} and t_{sexec}. Setup and execution times from the baseline execution are given for comparison in baseline t_{setup} and t_{exec}.

M

TESTCASE EVALUATION DATA

The table 37 shows the rating of the results from the benign and malicious testcase evaluation using the metrics discussed in 6.4.3.

#	Command	N_{att}	C_{red} (analysis)	C_{red} (execution)
0	/usr/bin/chcon	11	508664	918
1	/bin/ls	15	214826	937
2	/bin/bash	11	2274302	1591
3	/usr/bin/apt	19	314763	977
4	/usr/bin/hostid	13	49167	815
5	/usr/bin/link	11	26401	781
6	/usr/bin/printf	11	59281	829
7	/usr/bin/truncate	11	36898	702
8	/usr/bin/pr	11	108616	849
9	/usr/bin/sha1sum	11	47424	794
10	/usr/bin/nice	11	33927	776
11	/usr/bin/tee	9	40809	771
12	/usr/bin/realpath	11	47844	784
13	/usr/bin/tac	11	199705	869
14	/usr/bin/printenv	10	14814	689
15	/usr/bin/arch	12	28469	794
16	/usr/bin/logname	13	35870	791
17	/usr/bin/fold	11	33984	769
18	/usr/bin/users	11	34788	778
19	/usr/bin/dpkg	11	16691	639
20	/usr/bin/paste	11	28109	766
21	/usr/bin/factor	11	110292	859
22	/usr/bin/pathchk	12	50445	766

Continued on next page

#	Command	N_{att}	C_{red} (analysis)	C_{red} (execution)
23	/usr/bin/basename	11	31028	771
24	/usr/bin/dircolors	11	46778	794
25	/usr/bin/du	11	143793	984
26	/usr/bin/shuf	11	83929	818
27	/usr/bin/sha224sum	9	50930	783
28	/usr/bin/head	11	56082	787
29	/usr/bin/tty	11	27789	776
30	/usr/bin/join	11	62365	793
31	/usr/bin/test	11	55602	792
32	/usr/bin/runcon	11	84584	922
33	/usr/bin/base64	11	52540	784
34	/usr/bin/sha512sum	11	53905	857
35	/usr/bin/id	13	95129	932
36	/usr/bin/dirname	9	19502	1005
37	/usr/bin/numfmt	11	90709	787
38	/usr/bin/nl	11	47647	860
39	/usr/bin/install	11	392688	1125
40	/usr/bin/split	11	96663	608
41	/usr/bin/od	11	82860	805
42	/usr/bin/groups	13	40568	895
43	/usr/bin/env	11	32024	784
44	/usr/bin/tr	11	69074	787
45	/usr/bin/comm	11	54415	794
46	/usr/bin/md5sum	11	54955	799
47	/usr/bin/nproc	11	37476	770
48	/usr/bin/pinky	13	127964	878
49	/usr/bin/uniq	11	52044	780
50	/usr/bin/ptx	11	152320	948
51	/usr/bin/sha256sum	11	55810	796

Continued on next page

#	Command	N_{att}	C_{red} (analysis)	C_{red} (execution)
52	/usr/bin/cksum	11	33671	775
53	/usr/bin/who	11	104018	870
54	/usr/bin/cut	11	58884	774
55	/usr/bin/csplit	11	79471	864
56	/usr/bin/expand	11	38084	800
57	/usr/bin/unexpand	11	39949	776
58	/usr/bin/seq	11	130789	864
59	/usr/bin/stdbuf	11	66661	794
60	/usr/bin/unlink	11	30393	791
61	/usr/bin/timeout	10	92490	822
62	/usr/bin/tsort	11	44205	772
63	/usr/bin/expr	9	38346	790
64	/usr/bin/stat	14	88731	963
65	/usr/bin/tail	11	124909	850
66	/usr/bin/mkfifo	11	108286	879
67	/usr/bin/sort	11	352044	982
68	/usr/bin/nohup	11	41780	791
69	/usr/bin/fmt	11	49857	786
70	/usr/bin/whoami	13	32030	801
71	/usr/bin/sum	12	31970	790
72	/usr/bin/wc	11	56874	784
73	/usr/bin/shred	11	90456	798
74	/bin/cp	12	270993	1056
75	/bin/dd	11	107166	535
76	/bin/false	11	15278	693
77	/bin/readlink	11	48613	987
78	/bin/vdir	15	155190	930
79	/bin/rm	11	100890	852
80	/bin/df	11	139103	893

Continued on next page

#	Command	N_{att}	C_{red} (analysis)	C_{red} (execution)
81	/bin/rmdir	11	54738	800
82	/bin/sleep	9	34262	-227
83	/bin/true	11	15402	687
84	/bin/date	11	140130	875
85	/bin/stty	11	140390	808
86	/bin/ln	11	96485	844
87	/bin/mktemp	11	51183	788
88	/bin/cat	11	45924	802
89	/bin/uname	11	35827	777
90	/bin/chmod	11	92512	787
91	/bin/touch	11	109000	860
92	/bin/mv	12	289449	1009
93	/bin/sync	11	35885	766
94	/bin/mkdir	11	71462	908
95	/bin/dir	11	154118	890
96	/bin/chgrp	13	105151	887
97	/bin/chown	13	136993	898
98	/usr/sbin/ntpdate	12	52701	775
99	/bin/mknod	11	120163	885

Table 37: Result ratings of the implemented sandboxes for all benign testcases using the N_{att} and N_{red} operators.

BIBLIOGRAPHY

[1] Oleg V. Zaitsev, Nikolay A. Grebennikov, Alexey V. Monastyrsky and Mikhail A. Pavlyushchik. 'System and method for security rating of computer processes'. US7530106B1. May 2009. URL: https://patents.google.com/patent/US7530106B1/en (visited on 22/03/2019).

[2] Saswat Anand, Patrice Godefroid and Nikolai Tillmann. 'Demand-Driven Compositional Symbolic Execution'. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 367–381. ISBN: 978-3-540-78800-3.

[3] *TechnicalDoc · Wiki · AppArmor / apparmor*. en. URL: https://gitlab.com/apparmor/apparmor/wikis/TechnicalDoc (visited on 18/03/2019).

[4] Giuseppe Serazzi and Stefano Zanero. 'Computer virus propagation models'. In: *International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems*. Springer. 2003, pp. 26–50.

[5] Min Chen, Shiwen Mao and Yunhao Liu. 'Big Data: A Survey'. In: *Mobile Networks and Applications* 19.2 (Apr. 2014), pp. 171–209. ISSN: 1572-8153. DOI: 10.1007/s11036-013-0489-0. URL: https://doi.org/10.1007/s11036-013-0489-0.

[6] Almut Herzog, Nahid Shahmehri and Claudiu Duma. 'An Ontology of Information Security'. en. In: *International Journal of Information Security and Privacy (IJISP)* 1.4 (Oct. 2007), pp. 1–23. ISSN: 1930-1650 DOI: 10.4018/jisp.2007100101. DOI: 10.4018/jisp.2007100101. URL: https://www.igi-global.com/article/ontology-information-security/2468 (visited on 22/03/2019).

[7] Robert I. Soare. 'Turing oracle machines, online computing, and three displacements in computability theory'. In: *Annals of Pure and Applied Logic*. Computation and Logic in the Real World: CiE 2007 160.3 (Sept. 2009), pp. 368–399. ISSN: 0168-0072. DOI: 10.1016/j.apal.2009.01.008. URL: http://www.sciencedirect.com/science/article/pii/S0168007209000128 (visited on 29/05/2019).

[8] Ian Goldberg, David A. Wagner, Randi Thomas and Eric A. Brewer. 'A secure environment for untrusted helper applications confining the Wily Hacker'. In: 1996.

[9] Perpetus Jacques Houngbo and Joël Toyigbé Hounsou. 'Measuring information security: understanding and selecting appropriate metrics'. In: *International Journal of Computer Science and Security (IJCSS)* 9.2 (2015), p. 108.

[10] Adam Bates, Dave (Jing) Tian, Kevin R. B. Butler and Thomas Moyer. 'Trustworthy Whole-System Provenance for the Linux Kernel'. en. In: 2015, pp. 319–334. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/bates (visited on 21/03/2019).

[11] Suvda Myagmar, Adam J Lee and William Yurcik. 'Threat modeling as a basis for security requirements'. In: *Symposium on requirements engineering for information security (SREIS)*. Vol. 2005. Citeseer. 2005, pp. 1–8.

[12] F. Sabahi. 'Cloud computing security threats and responses'. In: *2011 IEEE 3rd International Conference on Communication Software and Networks*. May 2011, pp. 245–249. DOI: 10.1109/ICCSN.2011.6014715.

[13] Frank Swiderski and Window Snyder. *Threat Modeling*. Redmond, WA, USA: Microsoft Press, 2004. ISBN: 978-0-7356-1991-3.

[14] D. Papp, Z. Ma and L. Buttyan. 'Embedded systems security: Threats, vulnerabilities, and attack taxonomy'. In: *2015 13th Annual Conference on Privacy, Security and Trust (PST)*. July 2015, pp. 145–152. DOI: 10.1109/PST.2015.7232966.

[15] Oleg Sheyner and Jeannette Wing. 'Tools for Generating and Analyzing Attack Graphs'. en. In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf and Willem-Paul de Roever. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 344–371. ISBN: 978-3-540-30101-1.

[16] *The MITRE Corporation*. URL: https://www.mitre.org/ (visited on 22/03/2019).

[17] *Advanced Technology Laboratories | Lockheed Martin*. URL: https://www.lockheedmartin.com/en-us/capabilities/research-labs/advanced-technology-labs.html (visited on 22/03/2019).

[18] Peter Mell, Karen Scarfone and Sasha Romanosky. 'A complete guide to the common vulnerability scoring system version 2.0'. In: *Published by FIRST-Forum of Incident Response and Security Teams*. Vol. 1. 2007, p. 23.

[19] *CVSS v3.0 Specification Document*. URL: https://www.first.org/cvss/specification-document (visited on 23/03/2019).

[20] *NVD - Vulnerabilities*. URL: https://nvd.nist.gov/vuln (visited on 23/03/2019).

[21] Michael Howard and Steve Lipner. *The security development lifecycle*. Vol. 8. Redmond, WA, USA: Microsoft Press, 2006. ISBN: 978-07356-2214-2.

[22] *Security/OSSA-Metrics – OpenStack*. URL: https://wiki.openstack.org/wiki/Security/OSSA-Metrics#DREAD (visited on 23/03/2019).

[23] *HMG IA Standard No. 1 Technical Risk Assessment*. May 2012. URL: https://web.archive.org/web/20120526213309/http://www.cesg.gov.uk/publications/Documents/is1_risk_assessment.pdf (visited on 23/03/2019).

[24] 14:00-17:00. *ISO/IEC 27001:2013*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/45/54534.html (visited on 23/03/2019).

[25] *BSI - IT-Grundschutz*. URL: https://www.bsi.bund.de/EN/Topics/ITGrundschutz/itgrundschutz_node.html (visited on 23/03/2019).

[26] JA Wang, Min Xia and Fengwei Zhang. 'Metrics for information security vulnerabilities'. In: *Journal of Applied Global Research* 1.1 (2008), pp. 48–58.

[27] Rana Khudhair Ahmed. 'Security Metrics and the Risks: An Overview'. In: *International Journal of Computer Trends and Technology (IJCTT)* 41 (Nov. 2016), pp. 106–112. DOI: 10.14445/22312803/IJCTT-V41P119.

[28] International Organization for Standardization. *ISO/IEC 27002:2013*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/05/45/54533.html (visited on 24/03/2019).

[29] International Organization for Standardization. *ISO/IEC 27004:2016*. en. URL: http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/06/41/64120.html (visited on 24/03/2019).

[30] Marianne Swanson, Nadya Bartol, John Sabato, Joan Hash and Laurie Graffo. *Security Metrics Guide for Information Technology Systems*. en. Tech. rep. NIST Special Publication (SP) 800-55 (Withdrawn). National Institute of Standards and Technology, Aug. 2003. DOI: https://doi.org/10.6028/NIST.SP.800-55. URL: https://csrc.nist.gov/publications/detail/sp/800-55/archive/2003-08-01 (visited on 24/03/2019).

[31] Elizabeth Chew, Marianne M. Swanson, Kevin M. Stine, N. Bartol, Anthony Brown and W. Robinson. 'Performance Measurement Guide for Information Security | NIST'. en. In: *Special Publication (NIST SP) - 800-55 Rev 1* (July 2008). URL: https://www.nist.gov/publications/performance-measurement-guide-information-security (visited on 24/03/2019).

[32] and S. Haider. 'Security threats in cloud computing'. In: *2011 International Conference for Internet Technology and Secured Transactions*. Dec. 2011, pp. 214–219.

[33] Leanid Krautsevich, Fabio Martinelli and Artsiom Yautsiukhin. 'Formal Approach to Security Metrics.: What Does "More Secure" Mean for You?' In: *Proceedings of the Fourth European Conference on Software Architecture: Companion Volume*. ECSA '10. event-place: Copenhagen, Denmark. New York, NY, USA: ACM, 2010, pp. 162–169. ISBN: 978-1-4503-0179-4. DOI: 10.1145/1842752.1842787. URL: http://doi.acm.org/10.1145/1842752.1842787 (visited on 24/03/2019).

[34] Andy Ju An Wang. 'Information Security Models and Metrics'. In: *Proceedings of the 43rd Annual Southeast Regional Conference - Volume 2*. ACM-SE 43. event-place: Kennesaw, Georgia. New York, NY, USA: ACM, 2005, pp. 178–184. ISBN: 978-1-59593-059-0. DOI: 10.1145/1167253.1167295. URL: http://doi.acm.org/10.1145/1167253.1167295 (visited on 22/03/2019).

[35] Andrew S Tanenbaum and Herbert Bos. *Modern operating systems*. Pearson, 2015.

[36] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai and Onur Mutlu. 'Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors'. In: *ACM SIGARCH Computer Architecture News*. Vol. 42. 3. IEEE Press. 2014, pp. 361–372.

[37] Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz and Yuval Yarom. 'Spectre Attacks: Exploiting Speculative Execution'. In: *arXiv:1801.01203 [cs]* (Jan. 2018). arXiv: 1801.01203. URL: http://arxiv.org/abs/1801.01203 (visited on 16/03/2019).

[38] *SYSENTER - OSDev Wiki*. URL: https://wiki.osdev.org/SYSENTER (visited on 09/03/2019).

[39] H.J. Lu, Michael Matz, Milind Girkar, Jan Hubicka, Andreas Jaeger and Mark Mitchell. *System V Application Binary Interface AMD64 Architecture Processor Supplement (With LP64 and ILP32 Programming Models)*. English. Jan. 2018. URL: https://github.com/hjl-tools/x86-psABI/wiki/X86-psABI (visited on 03/09/2019).

[40] Jim Keniston, Ananth Mavinakayanahalli, Prasanna Panchamukhi and Vara Prasad. 'Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps'. In: *Proceedings of the 2007 Linux symposium*. 2007, pp. 215–224.

[41] Jim Keniston, Prasanna S Panchamukhi and Masami Hiramatsu. *Kernel Probes (Kprobes)*. URL: https://www.kernel.org/doc/Documentation/kprobes.txt (visited on 15/03/2019).

[42] *utrace (Linus Torvalds; Theodore Ts'o)*. URL: https://yarchive.net/comp/linux/utrace.html (visited on 15/03/2019).

[43] Frank Ch Eigler and Red Hat. 'Problem solving with systemtap'. In: *Proc. of the Ottawa Linux Symposium*. 2006, pp. 261–268.

[44] *strace*. URL: https://strace.io/ (visited on 15/03/2019).

[45] TIS Committee. *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification*. English. May 1995. URL: http://refspecs.linuxfoundation.org/elf/elf.pdf (visited on 03/10/2019).

[46] M. Witt, C. Jansen, D. Krefting and A. Streit. 'Fine-Grained Supervision and Restriction of Biomedical Applications in Linux Containers'. In: *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. May 2017, pp. 813–822. DOI: 10.1109/CCGRID.2017.53.

[47] Guido Noordende, Ádám Balogh, Rutger Hofman, Frances Brazier and Andrew Tanenbaum. 'A Secure Jailing System for Confining Untrusted Applications.' In: Jan. 2007, pp. 414–423.

[48] Tal Garfinkel et al. 'Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools.' In: *NDSS*. Vol. 3. 2003, pp. 163–176.

[49] K. Jain and R. Sekar. 'User-Level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement'. In: *In Proc. Network and Distributed Systems Security Symposium*. 1999.

[50] Matt Noonan, Alexey Loginov and David Cok. 'Polymorphic Type Inference for Machine Code'. In: *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '16. New York, NY, USA: ACM, 2016, pp. 27–41. ISBN: 978-1-4503-4261-2. DOI: 10.1145/2908080.2908119. URL: http://doi.acm.org/10.1145/2908080.2908119.

[51] Randall Hyde. *The Art of Assembly Language*. San Francisco, CA, USA: No Starch Press, 2003. ISBN: 978-1-886411-97-5.

[52] Eui Chul Richard Shin, Dawn Song and Reza Moazzezi. 'Recognizing Functions in Binaries with Neural Networks'. en. In: 2015, pp. 611–626. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/shin (visited on 29/03/2019).

[53] M. Buinevich, K. Izrailov and A. Vladyko. 'Testing of utilities for finding vulnerabilities in the machine code of telecommunication devices'. In: *2017 19th International Conference on Advanced Communication Technology (ICACT)*. Feb. 2017, pp. 408–414. DOI: 10.23919/ICACT.2017.7890122.

[54] Alfred V Aho. *Compilers: principles, techniques and tools (for Anna University), 2/e*. Pearson Education India, 2003.

[55] *IDA Pro: About*. URL: https://www.hex-rays.com/products/ida/index.shtml (visited on 26/03/2019).

[56] *OllyDbg v1.10*. URL: http://www.ollydbg.de/ (visited on 26/03/2019).

[57] *GDB: The GNU Project Debugger*. URL: https://www.gnu.org/software/gdb/ (visited on 26/03/2019).

[58] Jordan Wiens // Vector 35 Inc. *Binary Ninja > home*. en. URL: https://binary.ninja/ (visited on 26/03/2019).

[59] *radare*. URL: https://www.radare.org/r/index.html (visited on 26/03/2019).

[60] *Panopticon*. en. URL: https://gitlab.com/p8n (visited on 26/03/2019).

[61] William Landi. 'Undecidability of Static Analysis'. In: *ACM Lett. Program. Lang. Syst.* 1.4 (Dec. 1992), pp. 323–337. ISSN: 1057-4514. DOI: 10.1145/161494.161501. URL: http://doi.acm.org/10.1145/161494.161501 (visited on 30/03/2019).

[62] V. D'Silva, D. Kroening and G. Weissenbacher. 'A Survey of Automated Techniques for Formal Software Verification'. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 27.7 (July 2008), pp. 1165–1178. ISSN: 0278-0070. DOI: 10.1109/TCAD.2008.923410.

[63] M. V. Emmerik and T. Waddington. 'Using a decompiler for real-world source recovery'. In: *11th Working Conference on Reverse Engineering*. Nov. 2004, pp. 27–36. DOI: 10.1109/WCRE.2004.42.

[64] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner and David Brumley. '{BYTEWEIGHT}: Learning to Recognize Functions in Binary Code'. en. In: 2014, pp. 845–860. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/bao (visited on 29/03/2019).

[65] Junghee Lim and Thomas Reps. 'TSL: A System for Generating Abstract Interpreters and Its Application to Machine-Code Analysis'. In: *ACM Trans. Program. Lang. Syst.* 35.1 (Apr. 2013), 4:1–4:59. ISSN: 0164-0925. DOI: 10.1145/2450136.2450139. URL: http://doi.acm.org/10.1145/2450136.2450139 (visited on 30/03/2019).

[66] David Brumley, Ivan Jager, Thanassis Avgerinos and Edward J. Schwartz. 'BAP: A Binary Analysis Platform'. en. In: *Computer Aided Verification*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011, pp. 463–469. ISBN: 978-3-642-22110-1.

[67] Alan Mycroft. 'Type-Based Decompilation (or Program Reconstruction via Type Reconstruction)'. en. In: *Programming Languages and Systems*. Ed. by S. Doaitse Swierstra. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1999, pp. 208–223. ISBN: 978-3-540-49099-9.

[68] K. Troshina, Y. Derevenets and A. Chernov. 'Reconstruction of Composite Types for Decompilation'. In: *2010 10th IEEE Working Conference on Source Code Analysis and Manipulation*. Sept. 2010, pp. 179–188. DOI: 10.1109/SCAM.2010.24.

[69] K. Dolgova and A. Chernov. 'Automatic Type Reconstruction in Disassembled C Programs'. In: *2008 15th Working Conference on Reverse Engineering*. Oct. 2008, pp. 202–206. DOI: 10.1109/WCRE.2008.20.

[70] Michael James Van Emmerik. *Static single assignment for decompilation*. University of Queensland, 2007.

[71] JongHyup Lee, Thanassis Avgerinos and David Brumley. 'TIE: Principled Reverse Engineering of Types in Binary Programs'. en. In: (Feb. 2011). DOI: 10.1184/R1/6469466.v1. URL: https://kilthub.cmu.edu/articles/TIE_Principled_Reverse_Engineering_of_Types_in_Binary_Programs/6469466 (visited on 29/03/2019).

[72] Safa Omri, Pascal Montag and Carsten Sinz. 'Static Analysis and Code Complexity Metrics as Early Indicators of Software Defects'. en. In: *Journal of Software Engineering and Applications* 11 (Mar. 2018), p. 153. DOI: 10.4236/jsea.2018.114010. URL: https://www.scirp.org/journal/PaperInformation.aspx?PaperID=83690&#abstract (visited on 29/03/2019).

[73] I. Popov. 'Malware detection using machine learning based on word2vec embeddings of machine code instructions'. In: *2017 Siberian Symposium on Data Science and Engineering (SSDSE)*. Apr. 2017, pp. 1–4. DOI: 10.1109/SSDSE.2017.8071952.

[74] Mihai Christodorescu and Somesh Jha. *Static Analysis of Executables to Detect Malicious Patterns*. en. Tech. rep. WISCONSIN UNIV-MADISON DEPT OF COMPUTER SCIENCES, Jan. 2006. URL: https://apps.dtic.mil/docs/citations/ADA449067 (visited on 30/03/2019).

[75] James R Bell. 'Threaded code'. In: *Communications of the ACM* 16.6 (1973), pp. 370–372.

[76] Steven S. Muchnick and Neil Jones. *Program Flow Analysis: Theory and Applications*. Jan. 1981. ISBN: 978-0-13-729681-1.

[77] Johannes Kinder and Helmut Veith. 'Jakstab: A Static Analysis Platform for Binaries'. en. In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 423–427. ISBN: 978-3-540-70545-1.

[78] S. Rapps and E. J. Weyuker. 'Selecting Software Test Data Using Data Flow Information'. In: *IEEE Transactions on Software Engineering* SE-11.4 (Apr. 1985), pp. 367–375. ISSN: 0098-5589. DOI: 10.1109/TSE.1985.232226.

[79] Ingmar Stein and Florian Martin. 'Analysis of path exclusion at the machine code level'. In: *7th International Workshop on Worst-Case Execution Time Analysis (WCET'07)*. Ed. by Christine Rochange. Vol. 6. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2007. ISBN: 978-3-939897-05-7. DOI: 10.4230/OASIcs.WCET.2007.1196. URL: http://drops.dagstuhl.de/opus/volltexte/2007/1196 (visited on 27/03/2019).

[80] Benedikt Huber, Daniel Prokesch and Peter Puschner. 'Combined WCET Analysis of Bitcode and Machine Code Using Control-flow Relation Graphs'. In: *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*. LCTES '13. event-place: Seattle, Washington, USA. New York, NY, USA: ACM, 2013, pp. 163–172. ISBN: 978-1-4503-2085-6. DOI: 10.1145/2491899.

2465567. URL: http://doi.acm.org/10.1145/2491899.2465567 (visited on 27/03/2019).

[81] Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler and Mathias Payer. 'Control-Flow Integrity: Precision, Security, and Performance'. In: *ACM Comput. Surv.* 50.1 (Apr. 2017), 16:1–16:33. ISSN: 0360-0300. DOI: 10.1145/3054924. URL: http://doi.acm.org/10.1145/3054924 (visited on 31/03/2019).

[82] Leonardo de Moura and Nikolaj Bjørner. 'Z3: An Efficient SMT Solver'. en. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[83] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma and Roberto Sebastiani. 'The mathsat5 smt solver'. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2013, pp. 93–107.

[84] Bruno Dutertre and Leonardo De Moura. 'The yices smt solver'. In: *Tool paper at http://yices. csl. sri. com/tool-paper. pdf* 2.2 (2006), pp. 1–2.

[85] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio and Roberto Sebastiani. 'The mathsat 4 smt solver'. In: *International Conference on Computer Aided Verification*. Springer. 2008, pp. 299–303.

[86] Markus Mohnen. 'A Graph-Free Approach to Data-Flow Analysis'. In: *Proceedings of the 11th International Conference on Compiler Construction*. CC '02. London, UK, UK: Springer-Verlag, 2002, pp. 46–61. ISBN: 3-540-43369-4. URL: http://dl.acm.org/citation.cfm?id=647478.727795.

[87] Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen and Zhendong Su. 'A Survey on Data-Flow Testing'. In: *ACM Comput. Surv.* 50.1 (Mar. 2017), 5:1–5:35. ISSN: 0360-0300. DOI: 10.1145/3020266. URL: http://doi.acm.org/10.1145/3020266 (visited on 15/03/2019).

[88] Evelyn Duesterwald, Rajiv Gupta and Mary Lou Soffa. 'A demand-driven analyzer for data flow testing at the integration level'. In: *Proceedings of the 18th international conference on Software engineering*. IEEE Computer Society. 1996, pp. 575–584.

[89] David L. Bird and Carlos Urias Munoz. 'Automatic generation of random self-checking test cases'. In: *IBM systems journal* 22.3 (1983), pp. 229–245.

[90] Nicos Malevris and Derek F Yates. 'The collateral coverage of data flow criteria when branch testing'. In: *Information and Software Technology* 48.8 (2006), pp. 676–686.

[91] Cristian Cadar and Koushik Sen. 'Symbolic execution for software testing: three decades later.' In: *Commun. ACM* 56.2 (2013), pp. 82–90.

[92] Patrick Cousot and Radhia Cousot. 'Systematic Design of Program Analysis Frameworks'. In: *Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. POPL '79. San Antonio, Texas: ACM, 1979, pp. 269–282. DOI: 10.1145/567752.567778. URL: http://doi.acm.org/10.1145/567752.567778.

[93] Flemming Nielson and N Jones. 'Abstract interpretation: a semantics-based tool for program analysis'. In: *Handbook of logic in computer science* 4 (1994), pp. 527–636.

[94] Laurent Mauborgne and Xavier Rival. 'Trace Partitioning in Abstract Interpretation Based Static Analyzers'. In: *Programming Languages and Systems*. Ed. by Mooly Sagiv. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 5–20. ISBN: 978-3-540-31987-0.

[95] K. Muthukumar and M. Hermenegildo. 'Compile-time derivation of variable dependency using abstract interpretation'. In: *The Journal of Logic Programming* 13.2 (1992), pp. 315–347. ISSN: 0743-1066. DOI: https://doi.org/10.1016/0743-1066(92)90035-2. URL: http://www.sciencedirect.com/science/article/pii/0743106692900352.

[96] Patrice Godefroid, Nils Klarlund and Koushik Sen. 'DART: directed automated random testing'. In: *ACM Sigplan Notices*. Vol. 40. 6. ACM. 2005, pp. 213–223.

[97] Ella Bounimova, Patrice Godefroid and David Molnar. 'Billions and billions of constraints: Whitebox fuzz testing in production'. In: *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press. 2013, pp. 122–131.

[98] Peter Boonstoppel, Cristian Cadar and Dawson Engler. 'RWset: Attacking path explosion in constraint-based test generation'. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2008, pp. 351–366.

[99] Vitaly Chipounov, Vlad Georgescu, Cristian Zamfir and George Candea. 'Selective symbolic execution'. In: *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*. CONF. 2009.

[100] *Manticore: Symbolic execution for humans*. en. Apr. 2017. URL: https://blog.trailofbits.com/2017/04/27/manticore-symbolic-execution-for-humans/ (visited on 09/05/2019).

[101] Yan Shoshitaishvili et al. 'SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis'. In: *IEEE Symposium on Security and Privacy*. 2016.

[102] Glenford J Myers, Corey Sandler and Tom Badgett. *The art of software testing*. John Wiley & Sons, 2011.

[103] Unicorn. *Unicorn – The ultimate CPU emulator*. URL: http://www.unicorn-engine.org/ (visited on 13/03/2019).

[104] W. E. Wong, Y. Qi, L. Zhao and K. Cai. 'Effective Fault Localization using Code Coverage'. In: *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*. Vol. 1. July 2007, pp. 449–456. DOI: 10.1109/COMPSAC.2007.109.

[105] Mustafa M. Tikir and Jeffrey K. Hollingsworth. 'Efficient Instrumentation for Code Coverage Testing'. In: *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA '02. event-place: Roma, Italy. New York, NY, USA: ACM, 2002, pp. 86–96. ISBN: 978-1-58113-562-6. DOI: 10.1145/566172.566186. URL: http://doi.acm.org/10.1145/566172.566186 (visited on 30/03/2019).

[106] Boyuan Chen, Jian Song, Peng Xu, Xing Hu and Zhen Ming (Jack) Jiang. 'An Automated Approach to Estimating Code Coverage Measures via Execution Logs'. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. event-place: Montpellier, France. New York, NY, USA: ACM, 2018, pp. 305–316. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238214. URL: http://doi.acm.org/10.1145/3238147.3238214 (visited on 30/03/2019).

[107] Michael Hicks. *What is memory safety? - The PL Enthusiast*. en-US. July 2014. URL: http://www.pl-enthusiast.net/2014/07/21/memory-safety/ (visited on 30/03/2019).

[108] Wookhyun Han, Byunggill Joe, Byoungyoung Lee, Chengyu Song and Insik Shin. 'Enhancing memory error detection for large-scale applications and fuzz testing'. In: *Network and Distributed System Security Symposium (NDSS)*. 2018.

[109] Emery D Berger and Benjamin G Zorn. 'Probabilistic memory safety for unsafe languages'. In: *Acm sigplan 2006 conference on programming language design and implementation (pldi 2006). Ottawa, Canada: Diehard*. 2006.

[110] Nicholas Nethercote and Julian Seward. 'Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation'. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. event-place: San Diego, California, USA. New York, NY, USA: ACM, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: http://doi.acm.org/10.1145/1250734.1250746 (visited on 30/03/2019).

[111] Konstantin Serebryany, Derek Bruening, Alexander Potapenko and Dmitriy Vyukov. 'AddressSanitizer: A fast address sanity checker'. In: *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*. 2012, pp. 309–318.

[112] Juan Caballero, Gustavo Grieco, Mark Marron and Antonio Nappa. 'Undangle: Early Detection of Dangling Pointers in Use-after-free and Double-free Vulnerabilities'. In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. event-place: Minneapolis, MN, USA. New York, NY, USA: ACM, 2012, pp. 133–143. ISBN: 978-1-4503-1454-1. DOI: 10.1145/2338965.2336769. URL: http://doi.acm.org/10.1145/2338965.2336769 (visited on 30/03/2019).

[113] Will Drewry and Tavis Ormandy. 'Flayer: exposing application internals'. In: (2007).

[114] Byoungyoung Lee, Chengyu Song, Yeongjin Jang, Tielei Wang, Taesoo Kim, Long Lu and Wenke Lee. 'Preventing Use-after-free with Dangling Pointers Nullification.' In: *NDSS*. 2015.

[115] Josselin Feist, Laurent Mounier and Marie-Laure Potet. 'Statically detecting use after free on binary code'. en. In: *Journal of Computer Virology and Hacking Techniques* 10.3 (Aug. 2014), pp. 211–217. ISSN: 2263-8733. DOI: 10.1007/s11416-014-0203-1. URL: https://doi.org/10.1007/s11416-014-0203-1 (visited on 30/03/2019).

[116] Omer Tripp, Pietro Ferrara and Marco Pistoia. 'Hybrid Security Analysis of Web JavaScript Code via Dynamic Partial Evaluation'. In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. event-place: San Jose, CA, USA. New York, NY, USA: ACM, 2014, pp. 49–59. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2610385. URL: http://doi.acm.org/10.1145/2610384.2610385 (visited on 31/03/2019).

[117] Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot and Salvatore Guarnieri. 'Andromeda: Accurate and Scalable Security Analysis of Web Applications'. en. In: *Fundamental Approaches to Software Engineering*. Ed. by Vittorio Cortellessa and Dániel Varró. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2013, pp. 210–225. ISBN: 978-3-642-37057-1.

[118] Jonas Magazinius, Alejandro Russo and Andrei Sabelfeld. 'On-the-fly inlining of dynamic security monitors'. In: *Computers & Security*. IFIP/SEC 2010 "Security & Privacy - Silver Linings in the Cloud" 31.7 (Oct. 2012), pp. 827–843. ISSN: 0167-4048. DOI: 10.1016/j.cose.2011.10.002. URL: http://www.sciencedirect.com/science/article/pii/S0167404811001180 (visited on 31/03/2019).

[119] A. Russo and A. Sabelfeld. 'Dynamic vs. Static Flow-Sensitive Security Analysis'. In: *2010 23rd IEEE Computer Security Foundations Symposium*. July 2010, pp. 186–199. DOI: 10.1109/CSF.2010.20.

[120] H. S Venter and J. H. P Eloff. 'A taxonomy for information security technologies'. In: *Computers & Security* 22.4 (May 2003), pp. 299–307. ISSN: 0167-4048. DOI: 10.1016/S0167-4048(03)00406-1. URL: http://www.sciencedirect.com/science/article/pii/S0167404803004061 (visited on 16/03/2019).

[121] K. Keahey, K. Doering and I. Foster. 'From sandbox to playground: dynamic virtual environments in the grid'. In: *Fifth IEEE/ACM International Workshop on Grid Computing*. Nov. 2004, pp. 34–42. DOI: 10.1109/GRID.2004.32.

[122] Filippo Valsorda. *Searchable Linux Syscall Table for x86 and x86_64 | PyTux*. URL: https://filippo.io/linux-syscall-table/ (visited on 16/03/2019).

[123] Mohamed Sabt, Mohammed Achemlal and Abdelmadjid Bouabdallah. 'The Dual-Execution-Environment Approach: Analysis and Comparative Evaluation'. en. In: *ICT Systems Security and Privacy Protection*. Ed. by Hannes Federrath and Dieter Gollmann. IFIP Advances in Information and Communication Technology. Springer International Publishing, 2015, pp. 557–570. ISBN: 978-3-319-18467-8.

[124] admin. *Intel® Software Guard Extensions SDK*. en. URL: https://software.intel.com/en-us/sgx-sdk (visited on 01/04/2019).

[125] B. Ngabonziza, D. Martin, A. Bailey, H. Cho and S. Martin. 'TrustZone Explained: Architectural Features and Use Cases'. In: *2016 IEEE 2nd International Conference on Collaboration and Internet Computing (CIC)*. Nov. 2016, pp. 445–451. DOI: 10.1109/CIC.2016.065.

[126] F. Schuster, M. Costa, C. Fournet, C. Gkantsidis, M. Peinado, G. Mainar-Ruiz and M. Russinovich. 'VC3: Trustworthy Data Analytics in the Cloud Using SGX'. In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 38–54. DOI: 10.1109/SP.2015.10.

[127] Stefan Brenner, Colin Wulf, David Goltzsche, Nico Weichbrodt, Matthias Lorenz, Christof Fetzer, Peter Pietzuch and Rüdiger Kapitza. 'SecureKeeper: Confidential ZooKeeper Using Intel SGX'. In: *Proceedings of the 17th International Middleware Conference*. Middleware '16. event-place: Trento, Italy. New York, NY, USA: ACM, 2016, 14:1–14:13. ISBN: 978-1-4503-4300-8. DOI: 10.1145/2988336.2988350. URL: http://doi.acm.org/10.1145/2988336.2988350 (visited on 01/04/2019).

[128] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell and Vikram Adve. 'Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation'. In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '15. event-place: Istanbul, Turkey. New York, NY, USA: ACM, 2015, pp. 191–206. ISBN: 978-1-4503-2835-7. DOI: 10.1145/2694344.2694386. URL: http://doi.acm.org/10.1145/2694344.2694386 (visited on 01/04/2019).

[129] Ragunathan Rajkumar, Kanaka Juvva, Anastasio Molano and Shuichi Oikawa. 'Resource kernels: a resource-centric approach to real-time and multimedia systems'. In: *Multimedia Computing and Networking 1998*. Vol. 3310. International Society for Optics and Photonics, Dec. 1997, pp. 150–165. DOI: 10.1117/12.298417. URL: https://www.spiedigitallibrary.org/conference-proceedings-of-spie/3310/0000/Resource-kernels--a-resource-centric-approach-to-real-time/10.1117/12.298417.short (visited on 02/04/2019).

[130] Fangzhe Chang, Ayal Itzkovitz and Vijay Karamcheti. 'User-level Resource-constrained Sandboxing'. In: *Proceedings of the 4th Conference on USENIX Windows Systems Symposium - Volume 4*. WSS'00. event-place: Seattle, Washington. Berkeley,

CA, USA: USENIX Association, 2000, pp. 3–3. URL: http://dl.acm.org/citation.cfm?id=1267102.1267105 (visited on 02/04/2019).

[131] Liberios Vokorokos, Anton Baláž and Branislav Madoš. 'Application security through sandbox virtualization'. In: *Acta Polytechnica Hungarica* 12.1 (2015), pp. 83–101.

[132] Gerald J Popek and Robert P Goldberg. 'Formal requirements for virtualizable third generation architectures'. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[133] L. Turnbull and J. Shropshire. 'Breakpoints: An analysis of potential hypervisor attack vectors'. In: *2013 Proceedings of IEEE Southeastcon*. Apr. 2013, pp. 1–6. DOI: 10.1109/SECON.2013.6567516.

[134] Jakub Szefer, Eric Keller, Ruby B Lee and Jennifer Rexford. 'Eliminating the hypervisor attack surface for a more secure cloud'. In: *Proceedings of the 18th ACM conference on Computer and communications security*. ACM. 2011, pp. 401–412.

[135] admin. *Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. en. URL: https://software.intel.com/en-us/download/intel-64-and-ia-32-architectures-sdm-combined-volumes-1-2a-2b-2c-2d-3a-3b-3c-3d-and-4 (visited on 06/04/2019).

[136] Rashid Tahir, Matthew Caesar, Ali Raza, Mazhar Naqvi and Fareed Zaffar. 'An Anomaly Detection Fabric for Clouds Based on Collaborative VM Communities'. In: *Proceedings of the 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Press. 2017, pp. 431–441.

[137] Bin Fan, Dave G. Andersen, Michael Kaminsky and Michael D. Mitzenmacher. 'Cuckoo Filter: Practically Better Than Bloom'. In: *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: ACM, 2014, pp. 75–88. ISBN: 978-1-4503-3279-8. DOI: 10.1145/2674005.2674994. URL: http://doi.acm.org/10.1145/2674005.2674994.

[138] and, H. Chen and B. Zang. 'Defending against VM rollback attack'. In: *IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN 2012)*. June 2012, pp. 1–5. DOI: 10.1109/DSNW.2012.6264690.

[139] Fengzhe Zhang, Jin Chen, Haibo Chen and Binyu Zang. 'CloudVisor: retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization'. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*. ACM. 2011, pp. 203–216.

[140] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula and N. Fullagar. 'Native Client: A Sandbox for Portable, Untrusted x86 Native Code'. In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 79–93. DOI: 10.1109/SP.2009.25.

[141] Mathias Payer and Thomas R Gross. 'Fine-grained user-space security through virtualization'. In: *ACM SIGPLAN Notices* 46.7 (2011), pp. 157–168.

[142] Dionysus Blazakis. 'The apple sandbox'. In: *Arlington, VA, January* (2011).

[143] Zhenkai Liang, Weiqing Sun, V. N. Venkatakrishnan and R. Sekar. 'Alcatraz: An Isolated Environment for Experimenting with Untrusted Software'. In: *ACM Trans. Inf. Syst. Secur.* 12.3 (Jan. 2009), 14:1–14:37. ISSN: 1094-9224. DOI: 10.1145/1455526.1455527. URL: http://doi.acm.org/10.1145/1455526.1455527 (visited on 18/03/2019).

[144] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter and Emmett Witchel. 'Ryoan: A distributed sandbox for untrusted computation on secret data'. In: *ACM Transactions on Computer Systems (TOCS)* 35.4 (2018), p. 13.

[145] Jonathan Corbet. 'Seccomp and sandboxing'. In: *LWN. net, May* 25 (2009).

[146] Robert NM Watson, Jonathan Anderson, Ben Laurie and Kris Kennaway. 'Capsicum: Practical Capabilities for UNIX.' In: *USENIX Security Symposium*. Vol. 46. 2010, p. 2.

[147] Yanlin Li, Jonathan McCune, James Newsome, Adrian Perrig, Brandon Baker and Will Drewry. 'Minibox: A two-way sandbox for x86 native code'. In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 409–420.

[148] Z. C. Schreuders, C. Payne and T. McGill. 'Techniques for Automating Policy Specification for Application-oriented Access Controls'. In: *2011 Sixth International Conference on Availability, Reliability and Security*. Aug. 2011, pp. 266–271. DOI: 10.1109/ARES.2011.47.

[149] Elena Reshetova, Janne Karhunen, Thomas Nyman and N. Asokan. 'Security of OS-level virtualization technologies: Technical report'. In: *arXiv:1407.4245 [cs]* (July 2014). arXiv: 1407.4245. URL: http://arxiv.org/abs/1407.4245 (visited on 17/03/2019).

[150] Niels Provos. 'Improving Host Security with System Call Policies.' In: *USENIX Security Symposium*. 2003, pp. 257–272.

[151] Stefan Schumacher. 'Systemaufrufe mit Systrace steuern'. German. In: *UpTimes* 4 (Dec. 2007), 12
bibrangessep —
bibrangessep 19. ISSN: 1860-7683. URL: http://kaishakunin.com/publ/guug-uptimes-systrace.pdf (visited on 07/11/2009).

[152] Aleksey Kurchuk and Angelos D. Keromytis. 'Recursive Sandboxes: Extending Systrace to Empower Applications'. en. In: *Security and Protection in Information Processing Systems*. Ed. by Yves Deswarte, Frédéric Cuppens, Sushil Jajodia and Lingyu Wang. IFIP — The International Federation for Information Processing. Springer US, 2004, pp. 473–487. ISBN: 978-1-4020-8143-9.

[153] *OpenBSD: Innovations*. URL: https://www.openbsd.org/innovations.html (visited on 17/03/2019).

[154] *Systrace - Interactive Policy Generation for System Calls*. URL: http://www.citi.umich.edu/u/provos/systrace/ (visited on 17/03/2019).

[155] Robert NM Watson. 'Exploiting Concurrency Vulnerabilities in System Call Wrappers.' In: *WOOT* 7 (2007), pp. 1–8.

[156] Jake Edge. *A seccomp overview*. Tech. rep. LWN.net, Sept. 2009. URL: https://lwn.net/Articles/656307/.

[157] James Morris, Stephen Smalley and Greg Kroah-Hartman. 'Linux security modules: General security support for the linux kernel'. In: *USENIX Security Symposium*. ACM Berkeley, CA. 2002, pp. 17–31.

[158] Chris Wright, Crispin Cowan, James Morris, Stephen Smalley and Greg Kroah-Hartman. 'Linux security module framework'. In: *Ottawa Linux Symposium*. Vol. 8032. 2002, pp. 6–16.

[159] Stephen Smalley, Chris Vance and Wayne Salamon. 'Implementing SELinux as a Linux security module'. In: *NAI Labs Report* 1.43 (2001), p. 139.

[160] Stephen Smalley. 'Configuring the SELinux policy'. In: *NAI Labs Rep #02-007* (2002), pp. 1–35.

[161] *4.13. Multi-Level Security (MLS) - Red Hat Customer Portal*. URL: https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/selinux_users_and_administrators_guide/mls (visited on 18/03/2019).

[162] Yuichi Nakamura. 'SELinux & AppArmor-comparison of secure oses'. In: *Hitachi Software Engineering* (2007).

[163] Sergey Bratus, Michael E. Locasto, Boris Otto, Rebecca Shapiro, Sean W. Smith and Gabriel Weaver. *Beyond SELinux: the Case for Behavior-Based Policy and Trust Languages*. en. Monograph. Aug. 2011. URL: https://www.alexandria.unisg.ch/183170/ (visited on 17/03/2019).

[164] P. Amthor, W. E. Kühnhauser and A. Pölck. 'Model-based safety analysis of SELinux security policies'. In: *2011 5th International Conference on Network and System Security*. Sept. 2011, pp. 208–215. DOI: 10.1109/ICNSS.2011.6060002.

[165] Said Marouf, Doan Minh Phuong and Mohamed Shehab. 'A Learning-based Approach for SELinux Policy Optimization with Type Mining'. In: *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*. CSIIRW '10. event-place: Oak Ridge, Tennessee, USA. New York, NY, USA: ACM, 2010, 70:1–70:4. ISBN: 978-1-4503-0017-9. DOI: 10.1145/1852666.1852746. URL: http://doi.acm.org/10.1145/1852666.1852746 (visited on 18/03/2019).

[166] James Morris. 'Have you driven an SELinux lately'. In: *Linux Symposium Proceedings*. 2008.

[167] Björn Vogel and Bernd Steinke. 'Using SELinux Security Enforcement in Linux-based Embedded Devices'. In: *Proceedings of the 1st International Conference on MOBILe Wireless MiddleWARE, Operating Systems, and Applications*. MOBILWARE '08. event-place: Innsbruck, Austria. ICST, Brussels, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2007, 15:1–15:5. ISBN: 978-1-59593-984-5. URL: http://dl.acm.org/citation.cfm?id=1361492.1361511 (visited on 18/03/2019).

[168] *Home · Wiki · AppArmor / apparmor*. en. URL: https://gitlab.com/apparmor/apparmor/wikis/home (visited on 18/03/2019).

[169] Z. Cliffe Schreuders, Tanya McGill and Christian Payne. 'Empowering End Users to Confine Their Own Applications: The Results of a Usability Study Comparing SELinux, AppArmor, and FBAC-LSM'. In: *ACM Trans. Inf. Syst. Secur.* 14.2 (Sept. 2011), 19:1–19:28. ISSN: 1094-9224. DOI: 10.1145/2019599.2019604. URL: http://doi.acm.org/10.1145/2019599.2019604 (visited on 17/03/2019).

[170] Tarek Helmy, Ismail Keshta and Abdallah Rashed. 'Performance Evaluation of System Resources Utilization with Sandboxing Applications'. en. In: *Information Science and Applications*. Ed. by Kuinam J. Kim. Lecture Notes in Electrical Engineering. Springer Berlin Heidelberg, 2015, pp. 475–482. ISBN: 978-3-662-46578-3.

[171] Paul B Menage. 'Adding generic process containers to the linux kernel'. In: *Proceedings of the Linux symposium*. Vol. 2. Citeseer. 2007, pp. 45–57.

[172] Eric W Biederman and Linux Networx. 'Multiple instances of the global linux namespaces'. In: *Proceedings of the Linux Symposium*. Vol. 1. Citeseer. 2006, pp. 101–112.

[173] Rami Rosen. 'Resource management: Linux kernel namespaces and cgroups'. In: *Haifux, May* 186 (2013).

[174] *Linux Containers*. URL: https://linuxcontainers.org/ (visited on 21/03/2019).

[175] *Linux Containers - LXD - Introduction*. URL: https://linuxcontainers.org/lxd/introduction/ (visited on 21/03/2019).

[176] A. M. Joy. 'Performance comparison between Linux containers and virtual machines'. In: *2015 International Conference on Advances in Computer Engineering and Applications*. Mar. 2015, pp. 342–346. DOI: 10.1109/ICACEA.2015.7164727.

[177] D. Bernstein. 'Containers and Cloud: From LXC to Docker to Kubernetes'. In: *IEEE Cloud Computing* 1.3 (Sept. 2014), pp. 81–84. ISSN: 2325-6095. DOI: 10.1109/MCC.2014.51.

[178] W. Felter, A. Ferreira, R. Rajamony and J. Rubio. 'An updated performance comparison of virtual machines and Linux containers'. In: *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. Mar. 2015, pp. 171–172. DOI: 10.1109/ISPASS.2015.7095802.

[179] Thanh Bui. 'Analysis of Docker Security'. In: *arXiv:1501.02967 [cs]* (Jan. 2015). arXiv: 1501.02967. URL: http://arxiv.org/abs/1501.02967 (visited on 17/03/2019).

[180] *Docker Compose*. en. Mar. 2019. URL: https://docs.docker.com/compose/ (visited on 21/03/2019).

[181] *Swarm mode overview*. en. Mar. 2019. URL: https://docs.docker.com/engine/swarm/ (visited on 21/03/2019).

[182] Rui Shu, Xiaohui Gu and William Enck. 'A Study of Security Vulnerabilities on Docker Hub'. In: *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. CODASPY '17. event-place: Scottsdale, Arizona, USA. New York, NY, USA: ACM, 2017, pp. 269–280. ISBN: 978-1-4503-4523-1. DOI: 10.1145/3029806.3029832. URL: http://doi.acm.org/10.1145/3029806.3029832 (visited on 21/03/2019).

[183] T. Combe, A. Martin and R. Di Pietro. 'To Docker or Not to Docker: A Security Perspective'. In: *IEEE Cloud Computing* 3.5 (Sept. 2016), pp. 54–62. ISSN: 2325-6095. DOI: 10.1109/MCC.2016.100.

[184] Michael Witt, Christoph Jansen, Dagmar Krefting and Achim Streit. 'Sandboxing of biomedical applications in Linux containers based on system call evaluation'. de. In: *Concurrency and computation* 30.12 (2018). ISSN: 1532-0626. DOI: 10.1002/cpe.4484. URL: https://publikationen.bibliothek.kit.edu/1000085776 (visited on 21/03/2019).

[185] *Docker - Post-installation steps for Linux*. en. Mar. 2019. URL: https://docs.docker.com/install/linux/linux-postinstall/ (visited on 21/03/2019).

[186] *Rocket - CoreOS*. en. URL: https://coreos.com/rkt/ (visited on 21/03/2019).

[187] *Apache Mesos*. URL: http://mesos.apache.org/ (visited on 21/03/2019).

[188] *Open source container-based virtualization for Linux*. en. URL: https://openvz.org/ (visited on 21/03/2019).

[189] *containerd*. en-us. URL: https://containerd.io/ (visited on 21/03/2019).

[190] Alessandro Di Federico, Amat Cama, Yan Shoshitaishvili, Christopher Kruegel and Giovanni Vigna. 'How the {ELF} Ruined Christmas'. en. In: 2015, pp. 643–658. ISBN: 978-1-931971-23-2. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/di-frederico (visited on 09/09/2019).

[191] Gil Dabah. *gdabah/distorm*. original-date: 2015-03-20T23:14:30Z. Jan. 2021. URL: https://github.com/gdabah/distorm (visited on 06/01/2021).

[192] *inotify(7) - Linux manual page*. URL: http://man7.org/linux/man-pages/man7/inotify.7.html (visited on 12/05/2019).

[193] Edward W. Felten and Michael A. Schneider. 'Timing attacks on Web privacy'. In: *Proceedings of the 7th ACM conference on Computer and Communications Security*. CCS '00. New York, NY, USA: Association for Computing Machinery, Nov. 2000, pp. 25–32. ISBN: 978-1-58113-203-8. DOI: 10.1145/352600.352606. URL: https://doi.org/10.1145/352600.352606 (visited on 09/04/2021).

[194] B. Liang, W. You, L. Liu, W. Shi and M. Heiderich. 'Scriptless Timing Attacks on Web Browser Privacy'. In: *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. ISSN: 2158-3927. June 2014, pp. 112–123. DOI: 10.1109/DSN.2014.93.

[195] *coreutils - Debian Wiki*. URL: https://wiki.debian.org/coreutils (visited on 31/10/2019).

[196] *NTP - Debian Wiki*. URL: https://wiki.debian.org/NTP (visited on 25/04/2021).

[197] *OVAL - Open Vulnerability and Assessment Language*. URL: https://oval.mitre.org/ (visited on 25/05/2019).

[198] *Debian OVAL repository*. URL: https://www.debian.org/security/oval/ (visited on 25/05/2019).

[199] *CVE - Common Vulnerabilities and Exposures (CVE)*. URL: https://cve.mitre.org/ (visited on 31/10/2019).

[200] *VirusShare.com*. URL: https://virusshare.com/ (visited on 26/05/2019).

[201] *Virusshare - BossaBot*. URL: https://www.virustotal.com/gui/file/2252064b8cf60a7208f63852e21906ad338ce289b1f5cbed0bcb146353e98643/detection (visited on 12/12/2020).

[202] *Virusshare - Android.Trojan.FakeJobOffer*. URL: https://www.virustotal.com/gui/file/bf65c7de0523ebcbcd90a02acad31a813f0348e8326114095f8fadc746990edc/details (visited on 12/12/2020).

[203] *Virusshare - ELF/Xbash.A, Trojan.Linux.Agent.ZX*. URL: https://www.virustotal.com/de/file/0b9c54692d25f68ede1de47d4206ec3cd2e5836e368794eccb3daa632334c641/analysis/1540319986/ (visited on 12/12/2020).

[204] *Virusshare - Linux.DDOS.Flood.S, HEUR:Trojan-DDoS.Linux.Znaich.a*. URL: https://www.virustotal.com/file/1eb72c76f79fa01ce39198c91af5c7a4e36897e9a9a8f5d29ca68ba7371a2361/analysis/ (visited on 12/12/2020).

[205] *Virusshare - ELF:MrBlack-A [Trj], Linux/Flooder*. URL: https://www.virustotal.com/file/8d8aa7c1d3e9293292d599c06ae8c636e519bcebefa83bc074ca937a3af84aa0/analysis/ (visited on 12/12/2020).

[206] *Virusshare - ELF:BitCoinMiner-CW [PUP], not-a-virus:HEUR:RiskTool.Linux.BitCoinMiner.ao*. URL: https://www.virustotal.com/file/f90bada1b2562c3b7727fee57bd7edf453883219a2ee45923abbd539708236b0/analysis/1528105104/ (visited on 12/12/2020).

[207] *Virusshare - Trojan.Linux.ChinaZ.B, LINUX/Zanich.gckjw, TrojanDDoS.Linux.at*. URL: https://www.virustotal.com/file/605d85c54a2537e61a5642b0b3deadbd5f0ffbadea8346acf5b95311f7b8c862/analysis/1505374557/ (visited on 12/12/2020).

[208] *Virusshare - HEUR:Backdoor.Linux.Gafgyt.y, Linux.BackDoor.Fgt.92*. URL: https://www.virustotal.com/file/90c7dd9818c16515c9ced7a3e9ab580b1815c779045e1b6db618173547cd6c34/analysis/ (visited on 12/12/2020).

[209] *Virusshare - Suspicious_GEN.F47V0424, HEUR:Trojan-DDoS.Linux.Znaich.a, Linux.Trojan.Agent.CWKW4E*. URL: http://www.virustotal.com/file/c453e0d47de8106884381fcc0db2bf7927f714fc480fe31356809fff629c8a33/analysis/ (visited on 12/12/2020).

[210] *Virusshare - Trojan.Elf64.Gafgyt.ffaucp, Linux.Backdoor.Gafgyt.Pezz, TROJ_GEN.F04JC00G218*. URL: https://www.virustotal.com/file/ef6359c3d23d22e47ce400d0dc1101d037bbf79ccf92d43ae6dbc8e9c1aa4d54/analysis/1540309823/ (visited on 12/12/2020).

[211] *Virusshare - ELF:BitCoinMiner-AT [PUP]*. URL: https://www.virustotal.com/de/file/265e47865fb549fedc1f8c08f1625e5d9ab36c934dfd55db4705460d9a3d8725/analysis/1527136624/ (visited on 12/12/2020).

[212] *Virusshare - Trojan.Linux.Lady Linux/Ddosagent.8444416*. URL: https://www.virustotal.com/file/9ad4559180670c8d60d4036a865a30b41b5d81b51c4df281168cb6af69618405/analysis/ (visited on 12/12/2020).

[213] *Virusshare - Trojan/Linux.Agent.fr*. URL: https://www.virustotal.com/de/file/dbc380cbfb1536dfb24ef460ce18bccdae549b4585ba713b5228c23924385e54/analysis/1535971847/ (visited on 12/12/2020).

[214] *Virusshare - ELF/SpyLock.A*. URL: http://www.virustotal.com/file/2f7af825a9d3b817b881e6e7f6e0cd8bb21c888e778cbad29a54b77cb58d7368/analysis/ (visited on 12/12/2020).

[215] *Virusshare - LINUX/Dldr.Agent.eaklt*. URL: https://www.virustotal.com/de/file/03179a152a0ee80814ec62c91f8e7f0d0d6902bf190d2af1ecb6f17a0c0b8095/analysis/1538567947/ (visited on 12/12/2020).

[216] *Virusshare - Python.BackDoor.16 TROJ_GEN.F04JHooD717*. URL: https://www.virustotal.com/file/33fbb32d76b40c28b661899396041a516e14fbc4de921b0c94a5a86c170cca5c/analysis/ (visited on 12/12/2020).

[217] *Virusshare - Unix.Malware.GoScanSSH-6464971-0*. URL: https://www.virustotal.com/file/157942e817f4b619aa0f5445ccdab220e9d2548307c85cee3e8700f220cac999/analysis/1538688284/ (visited on 12/12/2020).

[218] *Virusshare - Linux.Siggen.1211 LINUX/CoinMiner.jpldu*. URL: https://www.virustotal.com/file/704643f3fd11cda1d52260285bf2a03bccafe59cfba4466427646c1baf93881e/analysis/1541489785/ (visited on 12/12/2020).

[219] *Virusshare - RiskWare[RiskTool]/Linux.BitCoinMiner.b, Unix.Tool.Miner-6414491-0*. URL: https://www.virustotal.com/file/bd8b120207f78de558397be228bbb6308aa0ee5223f3a657d1043a405fac00f2/analysis/1532467913/ (visited on 12/12/2020).

[220] *Virusshare - Gen:Variant.Backdoor.Linux.Gafgyt.1 Linux/DDoS-BI*. URL: https://www.virustotal.com/file/515def7780826cfa412037a04164998d58f3782845be100b34898341e422894e/analysis/ (visited on 12/12/2020).

[221] *Virusshare - Gen:Variant.Backdoor.Linux.Gafgyt.1, Linux_BASHLITE.SMD*. URL: https://www.virustotal.com/file/4fc78c42d119dfba03b5c73ddd00bb274ba341459748fcf9435f01b3012c01db/analysis/ (visited on 12/12/2020).

[222] *Virusshare - Gen:Variant.Backdoor.Linux.Tsunami.1, ELF_KAITEN.SM*. URL: https://www.virustotal.com/file/628bfa22c06a90c2a8b95167effdc47d43168f4e24d62c84e8238e81e6a0af8a/analysis/ (visited on 12/12/2020).

[223] *Virusshare - Linux.BackDoor.Fgt.374, LINUX/Gafgyt.lpmcu*. URL: https://www.virustotal.com/file/e1d92e9f8c983b930a849a969176bd5d59e52b00beb9375362e37712d50e5948/analysis/ (visited on 12/12/2020).

[224] *Virusshare - Linux.Lightaidra, Linux/Fgt.BP*. URL: https://www.virustotal.com/file/e937ab0efa1689af665f7addbb0882ad6c237f7c17afdbc4c303fdc73ad07102/analysis/ (visited on 12/12/2020).

[225] *Virusshare - Linux/Shellcode.CR, TROJ_GEN.F04JCooCF18, Trojan.Linux.ShellConn.b*. URL: https://www.virustotal.com/file/9eed60f1e055548052ed56b826cc573301841d2a5aac39d0c1708ce5c5c72fb9/analysis/1537924477/ (visited on 12/12/2020).

[226] *Virusshare - Gen:Variant.Backdoor.Linux.Gafgyt.1, Linux/DDoS-BI*. URL: https://www.virustotal.com/file/8a07f79e27191752adee6514c2fc62579e43618abcd2284cb4af1cbeb8416455/analysis/1541814412/ (visited on 12/12/2020).

[227] *Virusshare - ELF/Mirai.OX!tr, Linux.Trojan.Agent.DJG5DB*. URL: https://www.virustotal.com/file/4525c719ae32de9b389b445162e6cf55fe109e8174f106d6ea03d9cae30cd218/analysis/1549095506/ (visited on 12/12/2020).

[228] *Virusshare - Backdoor.Linux.Tsunami.A, ELF_KAITEN.SM*. URL: https://www.virustotal.com/file/883016830413f90bdf1cf7174e7fa70b716245a49884314e28e39c1528d4b144/analysis/1505376300/ (visited on 12/12/2020).

[229] *Virusshare - Linux.Trojan.IptabLex, HEUR:Trojan-DDoS.Linux.Sotdas.a*. URL: http://www.virustotal.com/file/8c5fe9f1df9a8b0fb9583851250bdbc28a308fdb02830921e3db3930d0d33d6b/analysis/ (visited on 12/12/2020).

[230] *Virusshare - Linux.BackDoor.Fgt.1596, Unix.Malware.Agent-6719240-0*. URL: https://www.virustotal.com/file/bb0514adb9e2d44dd43866c8c953c79b90101c5de850d9aac6af693364fc3e43/analysis/1540079536/ (visited on 12/12/2020).

[231] *Virusshare - DDoS:Linux/Lightaidra!rfn, ELF/Trojan.ULBE-8, Unix.Worm.Hakai-6654627-3*. URL: https://www.virustotal.com/file/4c6f0b44c426cd3c698fa9e17c3529ae5eeb9ce66724241093bfaaefb63148b8/analysis/1541727918/ (visited on 12/12/2020).

[232] *Virusshare - HEUR:HackTool.Linux.Agent.p*. URL: https://www.virustotal.com/file/69ee8bd4b2a27665de6309adb4ea35c6c1c3af0b9fc7cbe858faf1c70c6e5152/analysis/1511213658/ (visited on 12/12/2020).

[233] *Virusshare - HEUR:Backdoor.Linux.Gafgyt.y, Unix.Trojan.Mirai-5607483-0*. URL: https://www.virustotal.com/file/f2aa467b5a4d8d0bd025f67b7a9759b8c7ec34d9a7060dff55b1c8cf41e5e7f7/analysis/ (visited on 12/12/2020).

[234] *Virusshare - Gen:Variant.Backdoor.Linux.Gafgyt.1, Trojan.Linux.Gafgyt.taa, Trojan.Elf64.Gafgyt.enxptw*. URL: https://www.virustotal.com/file/7ca9ae5c391ab5545969210bbdef202fd0c9bee30f071c37b750d52f9d3dc439/analysis/1530550748/ (visited on 12/12/2020).

[235] *Virusshare - Trojan:Win32/Skeeyah.A!rfn, Unix.Malware.Agent-6727418-0, ELF:DDoS-Y [Trj]*. URL: https://www.virustotal.com/file/6584fbaccf655cf758475f790e9fcf6e101ed8d6e00c1d797606badaee1515e1/analysis/1541989046/ (visited on 12/12/2020).

[236] *Virusshare - Worm.Linux.ADS, Linux/Goscan.4390176, Trojan.Linux.GoScanSSH*. URL: https://www.virustotal.com/file/0159c232e9bdd983f8280211c6a4b23a83d735dabc768022876b44dbbf17c482/analysis/1544317482/ (visited on 12/12/2020).

[237] *Virusshare - ELF:Gafgyt-BA [Trj], Linux.BackDoor.Fgt.256*. URL: https://www.virustotal.com/file/ff4eda30af624be39af3b198ae7173062c76c03627df740d9f6f354ecd6fe414/analysis/ (visited on 12/12/2020).

[238] *Virusshare - Linux.Trojan.Agent.ROUCFX, TROJ_GEN.F04JCooFO18, Backdoor.Gafgyt.Linux.30247*. URL: https://www.virustotal.com/file/fc00e21b1ab4bc8697b8866e178a4706d84fffc9b9eb981743a2e70b12fa2f0d/analysis/1535934842/ (visited on 12/12/2020).

[239] *Virusshare - Possible_BASHLITE.SMLBO5, Trojan.Backdoor.Linux.Gafgyt.1, LINUX/Gafgyt.ghjtv*. URL: https://www.virustotal.com/file/a72a344b85912634922db04931b57ad6aad61ac07cd62ac3a124c61f22e75860/analysis/1541901773/ (visited on 12/12/2020).

[240] *Virusshare - GrayWare/Win32.Presenoker, LINUX/CoinMiner.pdiup, ELF:Agent-RQ [PUP]*. URL: https://www.virustotal.com/file/d1248ba0438f0bf0513a48bddec48afd0a2a160fd7ee51bc84eb6380fc8d1d30/analysis/1542187565/ (visited on 12/12/2020).

[241] *Virusshare - Linux.Backdoor.Kaiten, Backdoor.Tsunami.Linux.2250*. URL: https://www.virustotal.com/file/1e82b23bd76ab11a968c004b79bcca6182b7faa4d692443bb810181aa5b783fc/analysis/1530550810/ (visited on 12/12/2020).

[242] *Virusshare - HEUR:Backdoor.Linux.Agent.p, Downloader.OpenConnection.JS.104223, Linux.Trojan63e*. URL: http://www.virustotal.com/file/88227bc0fd067cf1918a9dddc051b9b9b2b32079f91d22d562b2c23a7770480b/analysis/ (visited on 12/12/2020).

[243] *Virusshare - Malware 3fcmri5a94pbr, LINUX/CoinMiner.cupjq*. URL: https://www.virustotal.com/file/f7cbe3aef850dffecbac6344f2b49d0a5d23aa9b84421a6f2144c1777f0cf00a/analysis/1549021858/ (visited on 12/12/2020).

[244] *Virusshare - Gen:Variant.Backdoor.Linux.Gafgyt.1, Backdoor.Gafgyt.Linux.33946*. URL: https://www.virustotal.com/file/158e91b4516d11c215a35475068307e40c9f4fc77e4ee2bfc17ce7374544c682/analysis/1534450558/ (visited on 12/12/2020).

[245] *Virusshare - Linux.DDOS.Flood.B, HEUR:Trojan-DDoS.Linux.DnsAmp.a*. URL: https://www.virustotal.com/file/20867fc3201c2e7948946cc93e00821186ce5e15ab45ce9dde0d36e2afdf8b21/analysis/ (visited on 12/12/2020).

[246] *How to break out of a chroot() jail*. URL: http://www.ouah.org/chroot-break.html (visited on 12/12/2020).

[247] *CVE-2000-1219*. URL: https://gcc.gnu.org/ml/gcc-bugs/2002-05/msg00198.html (visited on 12/12/2020).

[248] *CVE-2018-20376*. URL: https://lists.nongnu.org/archive/html/tinycc-devel/2018-12/msg00013.html (visited on 12/12/2020).

[249] *CVE-2014-9938*. URL: https://lists.nongnu.org/archive/html/tinycc-devel/2018-12/msg00013.html (visited on 12/12/2020).

[250] *CVE 2019-6974*. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-6974 (visited on 12/12/2020).

[251] *CVE 2018-0500*. URL: https://nvd.nist.gov/vuln/detail/CVE-2018-0500 (visited on 12/12/2020).

[252] *CVE-2017-1000366*. URL: https://nvd.nist.gov/vuln/detail/CVE-2017-1000366 (visited on 12/12/2020).

[253] *CVE-2016-5195*. URL: https://nvd.nist.gov/vuln/detail/CVE-2016-5195 (visited on 12/12/2020).

[254] *Map Address oxo to application*. URL: https://www.exploit-db.com/exploits/46502 (visited on 12/12/2020).

[255] *CVE-2019-7665*. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-7665 (visited on 12/12/2020).

[256] *CVE-2019-7663*. URL: https://nvd.nist.gov/vuln/detail/CVE-2019-7663 (visited on 12/12/2020).

DECLARATION

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Alle sinngemäß und wörtlich übernommenen Textstellen aus fremden Quellen wurden kenntlich gemacht.
Ich versichere weiterhin, dass diese Arbeit bisher weder veröffentlicht oder einer anderen Einrichtung/Kommission zur Prüfung vorgelegt wurde.

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This thesis was not previously presented to another examination board and has not been published.

Berlin, May 2021

Michael Witt