



University of
Zurich^{UZH}

Design and Implementation of a Malware to Detect Containerized Sandboxing

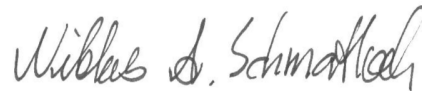
Niklas A. Schmatloch
Zürich, Schweiz
Student ID: 16-943-979

Supervisor: Jan von der Assen, Dr. Alberto Huertas
Date of Submission: August 1, 2024

Declaration of Independence

I hereby declare that I have composed this work independently and without the use of any aids other than those declared (including generative AI such as ChatGPT). I am aware that I take full responsibility for the scientific character of the submitted text myself, even if AI aids were used and declared (after written confirmation by the supervising professor). All passages taken verbatim or in sense from published or unpublished writings are identified as such. The work has not yet been submitted in the same or similar form or in excerpts as part of another examination.

Zürich, 30.07.2024



Signature of student

Abstract

Die Entwicklung von Schadprogrammen und das Absichern von Software-Systemen ist im ständigen Wettlauf gegeneinander. Während Entwickler stets bestrebt sind, Sicherheitslücken zu schliessen und Programmfehler zu beheben, entstehen fortlaufend neue Angriffswege, welche von Angreifern gesucht und ausgenutzt werden. Ein wichtiges Werkzeug zur Verteidigung ist die dynamische Analyse von aktiv genutzten Schadprogrammen, bei welcher das Programm in einer kontrollierten Umgebung ausgeführt und sein Verhalten beobachtet wird. Während Analyseprogramme traditionell virtuelle Maschinen einsetzen, um eine solche Umgebung zu schaffen, stellt sich mit dem Aufkommen von Containern die Frage, ob diese alternative Form der Virtualisierung sich ebenfalls dazu eignet. In diesem Analyseverfahren ist es unabdingbar, dass das Schadprogramm im Unwissen darüber bleibt, dass es analysiert wird. Ist es jedoch in der Lage, die Umgebung zu erkennen, hat es die Möglichkeit seine Funktion zu deaktivieren und sich so der Analyse zu entziehen.

Diese Arbeit nimmt die Position eines Schadprogrammentwicklers ein, der beabsichtigt seine Programm gegen diese Form der Analyse zu wappnen, um im Anschluss aufzuzeigen, wie ein Analyseprogramm unerkannt bleiben kann. Der Fokus liegt hierbei auf der gehärteten Container-Sandbox, *gVisor*.

The development of and defense against malware are in a constant race against each other. While developers constantly strive to close vulnerabilities and fix bugs, new ways of attacks are constantly emerging. An important tool for defense is the dynamic analysis of malware that is actively being used in the wild. In this analysis the malware is executed in a controlled environment to observe its behavior. While analysis tools traditionally use virtual machines to provide such an environment, the emergence of containers raises the question of whether this alternative form of virtualization is suitable for this purpose as well. In dynamic analysis, it is essential that the malicious program remains unaware that it is being analyzed. If it is, however, able to do so, it could deactivate its functionality in an attempt to hide its malicious nature and thus evade the analysis.

This work takes the position of a malware developer who intends to protect his program against this form of analysis, in order to subsequently show how an analysis program can remain undetected. The focus of the investigation is the hardened container sandbox, *gVisor*.

Acknowledgments

I would like to express my sincere gratitude to my supervisor, Jan von der Assen, for his guidance and support throughout the past six months. Our regular meetings were particularly helpful and the feedback was very valuable in finding the right direction for my research.

Contents

Declaration of Independence	i
Abstract	iii
Acknowledgments	v
1 Introduction	1
1.1 Motivation	1
1.2 Description of Work	2
2 Background	3
2.1 Types of Virtualization	3
2.1.1 Virtual Machines	3
2.1.2 Containers	4
2.1.3 gVisor	5
3 Related Work	7
3.1 Container-based Malware Analysis	7
3.2 Defensive Container Security	8
3.3 Virtualization Detection and Information Leaks	9

4	Architecture	13
4.1	VM Detection	13
4.1.1	BIOS Information	13
4.1.2	Kernel and Driver Logs in the Kernel Ring Buffer	16
4.2	Docker Containerization Detection	19
4.2.1	Intel RAPL	19
4.3	gVisor specific methods	20
4.3.1	Dockerenv and Dockerinit	21
4.3.2	Meminfo	21
4.3.3	Sysfs DMI Information	22
4.3.4	Uptime and Idle Time in the proc Filesystem	23
4.3.5	Unimplemented Syscalls	23
4.3.6	Modifying the Kernel Ring Buffer	25
5	Implementation	27
5.1	Choice of Language	28
5.2	Implementation Details of Strategies	28
5.2.1	BIOS Information	28
5.2.2	Kernel and Driver Logs in the Kernel Ring Buffer	29
5.2.3	Intel RAPL	30
5.2.4	Dockerenv	31
5.2.5	Meminfo	31
5.2.6	Sysfs DMI Information	32
5.2.7	Uptime and Idle Time in the proc Filesystem	32
5.2.8	Unimplemented Syscalls	33
5.2.9	Modifying the Kernel Ring Buffer	35

<i>CONTENTS</i>	ix
6 Evaluation	37
6.1 Setup of the Experiments	37
6.2 Performance of the Malware in Different Environments	38
6.3 Reliability and Mitigation of Individual Strategies	39
6.3.1 BIOS Information	39
6.3.2 Kernel and Driver Logs in the Kernel Ring Buffer	41
6.3.3 Intel RAPL	43
6.3.4 Dockerenv and Dockerinit	44
6.3.5 Meminfo	45
6.3.6 Sysfs DMI Information	46
6.3.7 Uptime and Idle Time in the proc Filesystem	47
6.3.8 Unimplemented Syscalls	49
6.3.9 Modifying the Kernel Ring Buffer	50
7 Summary and Conclusions	53
7.1 Future Work	53
Bibliography	55
Abbreviations	59
Glossary	61
List of Figures	61
List of Tables	63
A Installation Guidelines	67

Chapter 1

Introduction

The development of and defense against malware is an arms race. As malicious actors continuously attempt to find new vulnerabilities and attack vectors, developers strive to prevent this, by securing their software and systems.

1.1 Motivation

A crucial resource for developers aiming to secure programs, infrastructure, and software systems against attacks is knowledge about malware that is actively being used in the wild. Analyzing such malicious software provides insights into which vulnerabilities are presently being exploited and the attack vectors available to threat actors.

Gaining this knowledge about existing malware is done in different ways. While static analysis does not execute the malware, but rather attempts to reason about the program by, for example, reverse engineering the binary to human-readable code, dynamic analysis executes the malicious software in a controlled and isolated environment. This, on the one hand, prevents the malware to cause any harm on the host system it is being executed on, and on the other hand, allows for a more detailed analysis of its behavior. By monitoring its resource usage, like CPU, network, I/O, *etc.*, or its request to the kernel in the form of system calls, valuable information about the malware can be inferred [1].

Since the analysis of the malicious code helps finding and closing vulnerabilities that the malware exploits, it is contrary to the interest of the developer of the malware. Therefore, a developer may attempt to prevent this by programming the malware to detect such a controlled environment and, if detected, shut down its functionality in an attempt to hide its malicious nature and effectively evading the analysis. To achieve this, the malware may search for artifacts of a hypervisor or a virtual machine (VM), or use side-channels like timing differences of certain operations within the environment [2].

Traditionally virtual machines have been used to create secure environments for malware analysis. However, they come with certain downsides, particularly the computational cost.

Despite efforts of hardware manufactures to increase the efficiency of VMs by implementing hardware virtualization features in modern CPUs, and OS developers including kernel modules, like KVM, to provide these features to the virtual machines [3], there remains a significant overhead in terms of CPU and memory usage, as well as startup time.

With the emergence of more lightweight forms of virtualization, such as containers, replacing virtual machines in various areas, the question arises of whether this is a potential alternative in the field of malware analysis as well. Containers require less resources, as they use many isolation features directly provided by the kernel [4]. They also do not have any virtualized devices and consequently often approach near-native performance with a slim overhead. Yet, they are by their nature less isolated, as they share more components with their host system.

1.2 Description of Work

This work aims to provide a contribution to solving this question by focusing on the hardened container sandbox *gVisor*. Unlike most common forms of containerization, such as Docker, gVisor provides an own kernel running in user space to intercept system calls made by applications running within the container [5]. With this additional layer of isolation, gVisor is a suitable candidate to provide secure environments for dynamic malware analysis.

To approach this question, this thesis assumes the role of a malicious developer, who attempts to write a malware that avoids being analyzed in a gVisor-based malware analysis tool. First, in the fourth chapter, *Architecture*, detection mechanisms for more traditional forms of virtualization, like virtual machines and standard Docker containers, will be explored. Next, the design and architecture of gVisor will be studied in order to examine to which degree the found mechanisms are applicable to gVisor as well. Additionally, gVisor-specific characteristics that potentially allow it to be detected, will be discussed.

Based on these findings, a malware will be implemented and its design will be laid out in the fifth chapter, *Implementation*. Finally, in the sixth chapter, *Evaluation*, the malware will be tested in different environments to assess its effectiveness in an overview. Following thereafter, a more in-depth analysis of the individual detection mechanisms implemented by the malware, will allow for more insight on the theoretical reliability. Based on this analysis, mitigation techniques will be proposed and discussed for each mechanism. These techniques will be subdivided in ways to defend against the detection in the current state of gVisor (*i.e.*, configuration, preparation of the environment) and potential changes that gVisor or the Linux kernel would need to implement in order to achieve proper isolation.

Chapter 2

Background

This chapter introduces relevant technical terms and explains the essential concepts required to understand the following chapters.

2.1 Types of Virtualization

Virtualization is widely used across the IT industry; Especially in fields like cloud computing it has been ubiquitous. When used in reference to virtual machines or containers, the term *virtualization* describes the technique that allows the creation of multiple virtual computers on the same physical machine. Each virtual computer operates as if it were a stand-alone physical machine, capable of running programs isolated from processes running on the host machine or in other virtualized computers on the same host [6]. That is, a program running in an ideal virtualized environment, does not behave any different from being executed natively, nor is it able to detect that it is running virtualized.

While the form of virtualization that is commonly used in the industry has changed, *i.e.*, migrating from virtual machines to containers, its importance is not declining [7]. Below, the relevant types of virtualization are presented and explained briefly.

2.1.1 Virtual Machines

VMs provide virtualization by virtualizing actual hardware components, such as their CPU. By virtualizing these hardware devices, VMs are able to execute programs or run operating systems, that require a CPU architecture that is different from the host [5].

The virtualization is orchestrated by the *hypervisor*. The hypervisor, also called *Virtual Machine Monitor* (VMM), is a program that sits between the actual hardware and the guest system, as depicted in Figure 2.1 It's purpose is to take care of providing an interface to its guests resembling a real machine [5].

Virtual machines are categorized into two types. While in type II VMs, the hypervisor is a program running on the operating system of the host, in type I the hypervisor does not require a host OS at all and instead runs directly on the hardware [6].

Generally, virtual machines come with a high computational cost, as virtualizing individual hardware components adds a substantial overhead, *e.g.*, in terms of CPU and memory usage. While modern CPUs usually support hardware-assisted virtualization, *e.g.*, via Intel VT or AMD-V, running a program within a virtual machine still requires substantially more resources [8]. The advantage is, however, a strong isolation between the host and the guest.

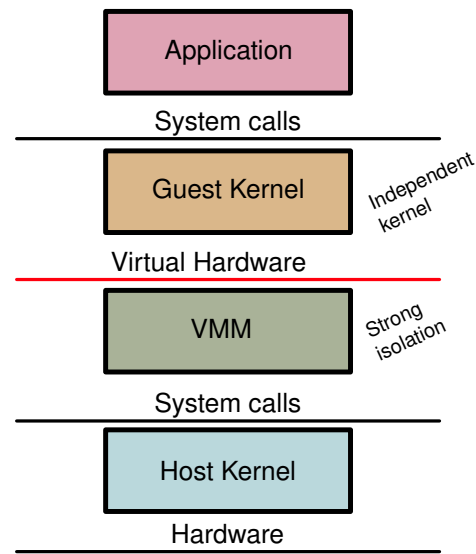


Figure 2.1: Machine-level Virtualization [5]

2.1.2 Containers

Containers attempt to provide a more lightweight form of virtualization, by virtualizing on an application-level, instead of a hardware-level. While VMs virtualize their hardware and run a complete, separate operating system with its own kernel, containers typically share the host’s kernel rather than having their own. In order to isolate processes from the host and from each other, they rely on isolation mechanisms implemented in the kernel of the host [7]. The isolation mechanisms that the Linux kernel provides which are taken advantage of by container technologies, like Docker, are explained in the following.

Control Groups The Linux kernel feature *control groups* (cgroups) allows processes to be assigned to groups, whose resources can be monitored and limited [9]. In cloud computing this feature can be used to charge customers on the basis of their actual resource usage instead of setting a fixed price per container [7]. Additionally, it can serve as precautionary measure to prevent a single container to disproportionately consume resources and consequently negatively affecting co-resident containers [10].

Namespaces The Linux kernel implements eight different namespaces. The purpose of each namespace is to isolate a process along a certain dimension.

- Control Group
- Mount
- PID
- User
- IPC
- Net
- Time
- UTS

More specifically, when two processes are member of different namespaces of the resource X , the interactions of each process with resource X are hidden from the other process [11].

For instance, the user namespace ensures isolation of the system users. In containers this can be used to make it seem as if, on the one hand, the process running in the container is running as the root user with user ID 0, and on the other hand, ensure that other system users present on the host or in different containers are invisible this process [7]. Similarly, the other namespaces isolate other resources, like mount points, process IDs, networks, *etc.*

Capabilities On certain operations, like accessing or modifying a file, or executing a program, the Linux kernel carries out permissions checks to determine whether the process requesting the operation is qualified to do so. In the traditional permission mechanism as inherited from UNIX, the kernel will distinguish between the two permission levels, *privileged* (*i.e.*, root user, superuser) and *unprivileged* (*i.e.*, regular users). The first will simply bypass any subsequent checks while the latter request will be evaluated based on its user ID and group ID, respectively the group and owner of the object of the request (*i.e.*, the file to be read, written or executed). The kernel feature *capabilities* adds a more fine-grained mechanism to assign and check permissions of operations, which can be enabled and disabled on a per-thread basis [12].

2.1.3 gVisor

Similar to regular Docker containers, gVisor uses the kernel isolation features to provide a virtualized environment. However, unlike Docker it does not rely on the kernel features alone, but additionally provides an application-level kernel, *Sentry*. While in VMs the guest kernel is running on virtualized hardware, *Sentry* runs on the host as an application in user space (See figure 2.2). Compared to VMs, this comes with a smaller overhead. When a process, running in a gVisor-backed container, makes a system call, *Sentry* will intercept it and respond to the caller instead of passing the it on to the host's kernel [5]. This provides an additional layer of isolation.

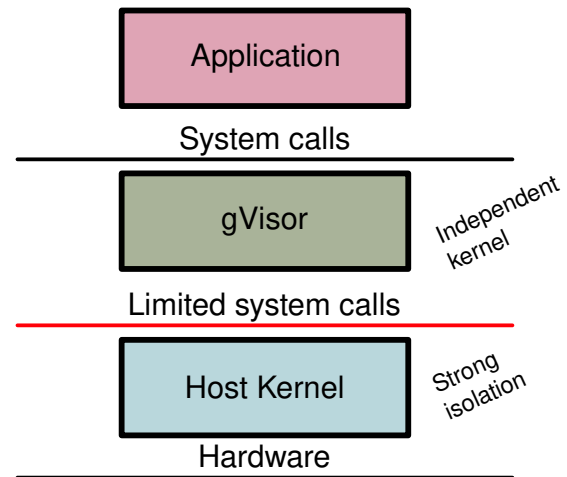


Figure 2.2: gVisor Sandboxing [5]

Since gVisor provides an OCI¹-compatible runtime `runc`, it can be used in combination with Docker to run containers based on regular Docker images. In this case `runc` will be used instead of Docker's `runc` as runtime, launching the container in gVisor's sandbox [5].

¹Open Container Initiative

Chapter 3

Related Work

This chapter provides an overview of related work, relevant for the objective of this thesis: the detection of container-based virtualization and its mitigation. While in many applications of containers, the detectability has no negative impact — and may even be desired — there are use-cases where hiding the presence of virtualization is crucial. To get a better understanding of such use-cases and assess their requirements and restrictions on possible defenses against detection, two container-based malware analysis systems are presented.

Next, works on defensive container security mechanisms will be examined to provide an overview of possible techniques to prevent information leaks, virtualization detection, and other attacks. This will include security mechanisms built into containerization software or the Linux kernel, past container runtime vulnerabilities, and methods for detecting attacks on containers.

Finally, the chapter will conclude with an exploration of methods for virtualization detection and information leaks. This section will focus more on the offensive view and will cover the detection of traditional forms of virtualization, *i.e.*, virtual machines, and methods used to gather information from containers for coordinating attacks. An overview of relevant papers is given in table 3.1.

3.1 Container-based Malware Analysis

In a recent work [13] created a malware analysis tool based on containers. For improved security, the hardened container sandbox, *gVisor*, was chosen to provide a secure isolated environment to perform dynamic malware analysis in. By monitoring both the resource usage of the container and the system calls made by processes the sandbox, and contrasting it with the behavior of an uninfected container with the same configuration, the dynamic analysis of malicious code is made possible. To facilitate the analysis, the metrics of the infected and the healthy container are depicted in real time on a local web interface. Through this interface the containers can be managed and controlled by sending commands to the containers while they are running. When a malicious program is to be

analyzed, its executable binary is downloaded from a malware database and started in the container.

Here, a reason to hide virtualization becomes evident. Were a malware to detect that it is running in SecBox's controlled environment, it could decide to disable its malicious function to avoid being analyzed. Without modifying the malware or preventing the detection of the environment, this would make the dynamic analysis impossible. Yet, since SecBox has a lot of control over the environment, it offers several opportunity to add mechanisms against detection.

On the one hand, the container environment can be modified. This can be done in two ways. Firstly, SecBox already offers the functionality to choose different container images for the execution of the malware. Adding custom images that extend the existing ones by applying mitigation techniques can therefore be done without much effort. Secondly, SecBox starts its gVisor-backed container through Docker via the Docker Python library, and thus, controls when the malware is executed and could possibly run commands to prepare the environment within the container before the malware is executed.

On the other hand, the runtime can be swapped out. In SecBox's code two different runtime configurations, one for the healthy and one for the infected are already present. While this currently serves the purpose of monitoring of the infected container, additional runtimes could be added or the existing one could be replaced by a patched version of `runc`, that was modified to counter certain mechanisms of detection.

In a different project, [14] made use of containerization to set up a honeypot, aimed at getting insights on container-specific attack vectors. In their experiments, [14] exposed containers to the internet, that were either susceptible to known vulnerabilities, or that were intentionally misconfigured. The containers were then kept running until an attacker identified the vulnerability and infected the container with malicious code. During the execution of the malware, the host would then collect logs about events in the guest container, which then were subsequently traced back to the activities in the individual container by utilizing OSQuery and eBPF. Since the monitoring of the malicious code is done on the host, instead of in the container, the risk of the malware tampering with the logs is averted. Yet, the honeypot still relies on the fact that the malware does not detect that it is running in the honeypot. While the detection of the honeypot by identifying the monitoring software is avoided by running it outside of the container, a malware could still detect the honeypot by detecting the containerized environment. If the malware is able to do so, it can prevent being found and analyzed. The experiments showed that vulnerable containers are quickly found by attackers and infected rapidly. On average, only five hours passed until a vulnerable container was infected, in one case just requiring a few minutes. This highlights the importance of investigating container security.

3.2 Defensive Container Security

Much research has been done on container security. To complement prior research that usually focused on vulnerabilities caused by container misconfiguration or Linux kernel

bugs, [15] surveyed 59 known CVEs caused by container runtime vulnerabilities. Of these 59 they focused on 28 for which exploits were publicly available. These were then studied and categorized to get a more high-level picture of what attacks are common.

They found that one of the most common attacks were aimed towards container escapes, in which an attacker attempts to break out of the isolated environment provided by the container: 13 out of the 28 vulnerabilities (46 %) enable this kind of attack. Their investigation of the cause that made the attacks possible showed that in most cases host information leaked into the guest container. This highlights the importance of proper isolation.

One method to defend against general information leaks in containers was proposed by [16]. While the vulnerabilities used by exploits investigated by [15] were caused by runtime vulnerabilities, [16] looked into avoiding information leaks into containers caused by sharing the host's kernel. By using a moving target defense (MTD) that deploys deceptive configurations, their proposed method hinders malicious actors to gather information without requiring regular applications to be modified. While this gives an insight on possibly information leakage channels, it is also worth investigating whether hardened runtimes with an own kernel, such as gVisor, are vulnerable too, and if the leakage channels can be isolated properly.

While [15] focused on runtime vulnerabilities, and [16] proposed a possible defense, [17] and [18] present techniques to detect the infection of containers. By using the relation of PIDs of processes on the host and within the container, [17] demonstrate a method to detect the occurrence of a container escape. Based on cAdvisor and Prometheus they built a monitoring system which automates this task. In a productive environment such an alert system could be used to control the damage a malware that succeeded in escaping the container can cause.

In contrast to [17] who focus on container escapes, [18] investigate how neural networks can be used to detect malicious workloads, such as coin miners or encrypting ransomware, within a container. In their research they extend the method of analyzing the distribution of system calls that applications within the container make, by including the arguments of the system calls. Their experiments, in which this analysis was used to distinguish containers with benign workloads from containers with malicious workloads, demonstrated that the inclusion of the arguments results in an improved effectiveness, specifically against mimicry attacks.

3.3 Virtualization Detection and Information Leaks

While containerization is generally considered to have weaker isolation, as it shares more components with the host, the more traditional form of virtualization, virtual machines, has been subject to many vulnerabilities as well. Even though, a stronger isolation is present, often the virtualization is not properly hidden from the applications running within it either. [2] describe four methods to detect whether a program runs in a virtual

machine based on the `cpuid` instructions, the kernel and driver logs, VM BIOS information, or the base load address of the global descriptor table. In an experiment on three industry-standard cloud providers, Amazon EC2, GCE, and Microsoft Azure, they found that all of them were vulnerable to all of their presented detection methods. While many of these techniques can potentially be mitigated through countermeasures, new methods such as using the GDT base load address can always arise.

Complementing this, [19] demonstrates how data from a timing analysis can be fused with data from a characteristic analysis. While [2] showed four detection methods based on a characteristic analysis, they outlined how most of them can be prevented by the VM's hypervisors. Combining this data with a timing analysis as demonstrated by [19] could allow for the detection of VMs in those cases in which the suggested countermeasures have been applied, while still not entailing the higher time cost of doing a timing analysis in the general case.

While [7] also investigate incomplete isolation of virtualization from the offensive view of a malware, it focuses on containers instead of VMs, and uses the information gained from the lack of isolation for coordination, rather than detection. Specifically, their investigation shows how improper isolation of Intel's RAPL, a technology allowing for monitoring power usage of physical hardware, can be used to detect co-residency in data centers which can effectively be used to coordinate a, *synergistic power attack*, in which a power spike is provoked, resulting in an outage.

To mitigate the information leakage channels, the paper suggests two options. Firstly, information that is not necessary for legitimate applications to run, could simply be hidden by restricting access, for example with AppArmor profiles. Secondly, for preventing the information leakage through Intel's RAPL, a power-based namespace is proposed which, similarly to other Linux namespaces, provides an identical RAPL interface to processes within the container but changes the provided information by calculating the share of the host's power usage that was caused by processes running within the container.

Table 3.1: Overview of Related Papers

Paper	Virtualization	Summary	Orientation	Attack	Implemented	View
[13]	gVisor	Malware analysis tool	Defensive	-	Dynamic analysis tool	Host
[14]	Docker	Container honeypot	Defensive	-	Container honeypot	Host
[16]	Container	Moving target defense against information leaks	Defensive	Information leaks	Configuration movement	Host
[18]	Container	NN analysis of syscalls to detect malicious containers	Defensive	Mimicry	NN analysis	Host
[15]	Container	Investigation of container security mechanisms and vulnerabilities	Both	Container escapes	-	-
[17]	Docker	Detecting container escapes using relation of PIDs	Defensive	Container escapes	Monitor and alert system for escapes	Host
[2]	VM	Virtualization detection using CPUID, syslog, DMI, GDT	Offensive	Detection	-	Guest
[19]	VM	Virtualization detection by fusion of characteristics and timing	Offensive	Detection	-	Guest
[7]	Docker, LXC	Coordination of a power attack exploiting incomplete kernel isolation	Offensive	Info leaks Power attack	Cross-validation tool for pseudo FS	Guest
This work	gVisor	gVisor detection exploiting incomplete isolation	Offensive	Detection	Malware Prototype	Guest

Chapter 4

Architecture

This chapter investigates gVisor’s architecture and behavior to discover distinctive characteristics that allow to detect its sandboxing. It will do so by first examining the detection methods discussed in the related work in the previous chapter that target virtual machines or standard Docker. In combination with the insight on gVisor’s architecture this chapter will determine whether gVisor is susceptible to these detection methods as well, and if not, whether other methods can be derived. Afterwards, this section will continue by investigating these derived methods and exploring entirely gVisor-specific methods that were not discussed in the related work.

In this chapter, unless specified otherwise, *Docker* will refer to running a container with the default runtime `runc` and default settings¹ while *gVisor* refers to using the `runcsc` runtime². By default, while this should not influence the results, the guest OS will be Debian Bookworm.

4.1 VM Detection

This section investigates the VM detection methods and whether they can be used or adapted to detect either regular Docker containers or gVisor’s sandboxed containerization.

4.1.1 BIOS Information

As [2] mention, one method to detect virtualization based on virtual machines is to investigate the information stored in the System Management BIOS (SMBIOS). The SMBIOS is a commonly used standard to provide system management information [20]. The `dmidecode` program can be used to access this information in guest operating systems. In the fields for “Manufacturer”, “Product name” and “Version” `dmidecode` returns information about the virtualized environment for many hypervisors. So, for instance, in the Google

¹`docker run --rm -it --entrypoint bash debian:bookworm`

²`docker run --rm -it --entrypoint bash --runtime=runcsc debian:bookworm`

Compute Engine `dmidecode` returned “Google” as manufacturer and “Google Compute Engine” as product name [2].

Docker To verify whether this approach can be used in a containerized environment, an initial experiment was conducted. When running `dmidecode` in Docker, it fails in the default configuration with the error:

```
Scanning /dev/mem for entry point.
/dev/mem: No such file or directory
```

This is because `dmidecode` first attempts to access the DMI table from `sysfs` and if this fails, it will attempt to read it from memory [21]. `dmidecode`'s first mechanism to retrieve BIOS information which functions through the `sysfs` filesystem, attempts to access the files `/sys/firmware/dmi/tables/smbios_entry_point` and `/sys/firmware/dmi/tables/DMI` and reads the BIOS information from there [22]. By default, Docker simply disables access to these files by not mounting the entire `/sys/firmware/dmi/` directory. No other isolation mechanisms are used here: By simply mounting the directory into the container with `-v /sys/firmware/dmi:/root/dmi` it will become available and readable³.

The (device) file `/dev/mem/` provides an interface to the main memory, allowing it to be accessed as if it were a regular file. The fallback mechanism of `dmidecode`, if the DMI table cannot be read from the `sysfs` files, attempts to read the BIOS information from this memory image [23].

In the default Docker configuration, there are two mechanisms preventing this attempted access. The first is that Docker does not mount the `/dev/mem` device of the host into the guest container. The second is Docker's use of the kernel feature *capabilities*. Different actions require processes to hold certain capabilities in order to be allowed by the kernel to perform them. The relevant capability in this case is `CAP_SYS_RAWIO`, which is required if a process wants to open `/dev/mem` which `dmidecode` attempts to do [12].

Without additional configuration, Docker disables all capabilities except a minimal set of necessary ones [24]:

- `CAP_CHOWN`
- `CAP_DAC_OVERRIDE`
- `CAP_FSETID`
- `CAP_FOWNER`
- `CAP_MKNOD`
- `CAP_NET_RAW`
- `CAP_SETGID`
- `CAP_SETUID`
- `CAP_SETFCAP`
- `CAP_SETPCAP`
- `CAP_NET_BIND_SERVICE`
- `CAP_SYS_CHROOT`
- `CAP_KILL`
- `CAP_AUDIT_WRITE`[25]

The first mechanism can be circumvented by mounting the `mem`-device into the container by adding the flag `--device /dev/mem:/dev/mem`. This takes the device at path `/dev/mem` and mounts it to the same path in the guest, making the device visible in the container (`ls /dev/mem` lists it). However, attempting to access it, will still fail:

³The directory was mounted outside of the container's `sysfs` filesystem to not cause conflicts with Docker's handling of pseudo filesystems

```
root@b4ee41e86f33:/# head --bytes=1 /dev/mem
head: cannot open '/dev/mem' for reading: Operation not permitted
```

This happens because of the second mechanism: When the process (`head`) in the unprivileged container requests from the host kernel to read from the device, it checks its capabilities and denies access because it does not hold the `CAP_SYS_RAWIO` capability. With the flag `--cap-add CAP_SYS_RAWIO` the container will receive this capability, and the device can be successfully accessed. Now `dmidecode` also succeeds in reading the DMI table, which is the host's real BIOS information. Its output is shown in listing 4.1.

Listing 4.1: Output of `dmidecode` in Docker After Granting Capability

```
# dmidecode 3.4
Scanning /dev/mem for entry point.
SMBIOS 3.0.0 present.

Handle 0x0001, DMI type 1, 27 bytes
System Information
    Manufacturer: Dell Inc.
    Product Name: Latitude 7280
    Version: Not Specified
    [...]

```

While the VM-detection method used `dmidecode` to read the BIOS information and evaluate its content to decide whether the current environment is a virtualized one (*i.e.*, when the BIOS information contains information about the hypervisor), this is not a feasible option in containers, as containers do not have separate BIOS from the host, unlike VMs. However, the investigation of how `dmidecode` works, presents two possible mechanisms to detect a containerized environment. While an unprivileged user or process cannot expect to run `dmidecode` or read `/dev/mem` or the respective files in `/sys/firmware/dmi/`, it can expect the respective files to be present. When they are not, this is an indicator of containerization:

- `/dev/mem`
- `/sys/firmware/dmi/tables/smbios_entry_point`
- `/sys/firmware/dmi/tables/DMI`

gVisor The same experiment was conducted on gVisor. Just as in standard Docker, the whole `/sys/firmware/dmi` directory is not mounted in the container but it also can be mounted with the `-v` option. By default gVisor also does not mount the device `/dev/mem`. Adding the flag `--device` also makes it visible but not accessible in gVisor. However, instead of failing with `Operation not permitted`, it fails with `No such device or address` and adding the `CAP_SYS_RAWIO` does not suffice to allow reading the device file.

In conclusion, the absence of each of the three paths that can be used to detect Docker containerization can also be used as an indicator for gVisor's sandboxing.

4.1.2 Kernel and Driver Logs in the Kernel Ring Buffer

Another method to detect VM-based virtualization demonstrated by [2] makes use of the `dmesg` command to display logs from the kernel and drivers stored in the kernel ring buffer. The `dmesg` command accesses the buffer and prints the messages. Virtual machines have virtual devices and therefore virtual devices drivers. These drivers usually print log messages containing the name of the device or a description of it, which frequently contains keywords such as *virtual*. Searching the logs for these keywords allows for detection of the virtual machine. In their research [2] found that the virtualization of all three public cloud providers that they examined can be detected using this method.

While containers do not have virtualized devices with own drivers, it is still worth investigating whether the kernel ring buffer access is properly isolated in Docker and gVisor. To get a better understanding about Docker's and gVisor's ring buffer isolation, this section investigates how the method that [2] chose (`dmesg`), works in detail and examines how Docker and gVisor respond to it.

Generally, `dmesg` is a tool to interact with the kernel ring buffer. It has three different methods to access the buffer: using the device file `/dev/kmsg`, using the `SYSLOG` system call or using a memory mapped file [26]⁴. In addition to the access method, `dmesg` has different *actions*. By default, `dmesg` accesses the kernel ring buffer using the `kmsg-device` and executes the action `SYSLOG_ACTION_READ_ALL` [26]⁵. This action does not modify or clear the buffer and simply returns all messages stored in it.

```

root@50161b19c59f:/# dmesg
[ 0.000000] Starting gVisor...
[ 0.294738] Mounting deweydecimalfs...
[ 0.571977] Segmenting fault lines...
[ 1.041326] Waiting for children...
[ 1.207888] Reticulating splines...
[ 1.703979] Accelerating teletypewriter to 9600 baud...
[ 1.961896] Letting the watchdogs out...
[ 2.363740] Granting licence to kill(2)...
[ 2.421607] Synthesizing system calls...
[ 2.774874] Checking naughty and nice process list...
[ 3.201215] Gathering forks...
[ 3.475301] Setting up VFS...
[ 3.629899] Setting up FUSE...
[ 3.824130] Ready!

```

Figure 4.1: Output of `dmesg` in gVisor

An initial experiment yielded that neither Docker nor gVisor mount the `/dev/kmsg/` device. While Docker prohibits `dmesg` from accessing the kernel ring buffer altogether, gVisor falls back to the `SYSLOG` system call and display a short log (Figure 4.1). The log in gVisor appears to be properly isolated as it neither completely prohibits access like Docker does, nor does it print any actual messages from the host's kernel ring buffer. However, the first line consistently indicates gVisor's presence.

⁴`sys-utils/dmesg.c` lines 174 - 178

⁵`sys-utils/dmesg.c` lines 1640 - 1648

Listing 4.2: gVisor Parsing Arguments and Actions of syslog System Call [27]

```

1 func Syslog(t *kernel.Task, sysno uintptr, args arch.SyscallArguments) (
2   uintptr, *kernel.SyscallControl, error) {
3   command := args[0].Int()
4   buf := args[1].Pointer()
5   size := int(args[2].Int())
6
7   switch command {
8   case _SYSLOG_ACTION_READ_ALL:
9       if size < 0 {
10          return 0, nil, linuxerr.EINVAL
11        }
12        if size > logBufLen {
13            size = logBufLen
14        }
15
16        log := t.Kernel().Syslog().Log()
17        if len(log) > size {
18            log = log[:size]
19        }
20
21        n, err := t.CopyOutBytes(buf, log)
22        return uintptr(n), nil, err
23   case _SYSLOG_ACTION_SIZE_BUFFER:
24       return logBufLen, nil, nil
25   default:
26       return 0, nil, linuxerr.ENOSYS
27 }

```

In its fallback method to access the kernel ring buffer, `dmesg` executes the `syslog` system call with the action `SYSLOG_ACTION_READ_ALL` given as argument. The source code of gVisor gives further insight on how gVisor's application level kernel *Sentry* handles the requests that `dmesg` makes. As seen in Listing 4.2, the only two actions of the `syslog` system call that are implemented are `SYSLOG_ACTION_READ_ALL` and `SYSLOG_ACTION_SIZE_BUFFER`. If any other action is specified, *Sentry* will return a `linuxerr.ENOSYS` indicating a missing implementation [27]⁶.

On the `SYSLOG_ACTION_READ_ALL` action that `dmesg` requests, gVisor returns the syslog of its kernel by calling `t.Kernel().Syslog().Log()`. This function returns the log if it was already initialized before or otherwise artificially generates a log [27]⁷.

The generation of the syslog messages (See Figure 4.2) simply chooses messages randomly from a hard-coded list of possible messages. The first and the last three messages are hard-coded (This is why the first line always indicates gVisor's sandboxing) and in-between ten random messages are chosen from the `allMessages` list seen in Figure 4.3. The timestamps are generated randomly as well. While the first message always has timestamp 0, the timestamps of each subsequent message increases continuously by a randomized amount [27]⁸.

⁶`pkg/sentry/syscalls/linux/sys_syslog.go` lines 35 - 61

⁷`pkg/sentry/kernel/syslog.go` lines 42 - 47

⁸`pkg/sentry/kernel/syslog.go` lines 49 - 118

```

selectMessage := func() string {
    i := rand.Intn(len(allMessages))
    m := allMessages[i]

    // Delete the selected message.
    allMessages[i] = allMessages[len(allMessages)-1]
    allMessages = allMessages[:len(allMessages)-1]

    return m
}

const format = "<6>[%11.6f] %s\n"

s.msg = append(s.msg, []byte(fmt.Sprintf(format, 0.0, "Starting gVisor..."))...)

time := 0.1
for i := 0; i < 10; i++ {
    time += rand.Float64() / 2
    s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, selectMessage()))...)
}

time += rand.Float64() / 2
s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Setting up VFS..."))...)
time += rand.Float64() / 2
s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Setting up FUSE..."))...)

time += rand.Float64() / 2
s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Ready!"))...)

// Return a copy.
o := make([]byte, len(s.msg))
copy(o, s.msg)
return o

```

Figure 4.2: Generation of an Artificial Syslog [27]

To summarize, similar to virtual machines, the kernel ring buffer can be used to detect gVisor’s sandboxing. Since the log does not contain actual log messages from virtual device drivers but simply a selection from known fake messages, some of which even with a hard-coded position, this detection mechanism is even more reliable. While the ring buffer is not accessible in Docker, the absence of the device `/dev/kmsg` is an indicator for containerization in both Docker and gVisor. Although gVisor’s syslog initially appeared to be isolated properly, a more in-depth investigation of the source code of `dmesg` and gVisor, revealed the following indicators that can be used to detect gVisor’s presence:

```

allMessages := []string{
    "Synthesizing system calls...",
    "Mounting deweydecimalfs...",
    "Moving files to filing cabinet...",
    "Digging up root...",
    "Constructing home...",
    "Segmenting fault lines...",
    "Creating bureaucratic processes...",
    "Searching for needles in stacks...",
    "Preparing for the zombie uprising...",
    "Feeding the init monster...",
    "Creating cloned children...",
    "Daemonizing children...",
    "Waiting for children...",
    "Gathering forks...",
    "Committing treasure map to memory...",
    "Reading process obituaries...",
    "Searching for socket adapter...",
    "Creating process schedule...",
    "Generating random numbers by fair dice roll...",
    "Rewriting operating system in Javascript...",
    "Reticulating splines...",
    "Consulting tar man page...",
    "Forking spaghetti code...",
    "Checking naughty and nice process list...",
    "Checking naughty and nice process list...", // Check it up to twice.
    "Granting licence to kill(2)...", // British spelling for British movie.
    "Letting the watchdogs out...",
    "Conjuring /dev/null black hole...",
    "Adversarially training Redcode AI...",
    "Singleplexing /dev/ptmx...",
    "Recruiting cron-ies...",
    "Verifying that no non-zero bytes made their way into /dev/zero...",
    "Accelerating teletypewriter to 9600 baud..."
}

```

Figure 4.3: Bucket of Syslog Messages gVisor Chooses from [27]

- Absence of `/dev/kmsg`
- 14 syslog messages in buffer
- All syslog messages are present in the `allMessages` list
- The hard-coded first and last three messages (First timestamp 0)
- Error indicating missing implementation (`ENOSYS`) on different syslog-actions (other than read-all or size)

Beyond the kernel ring buffer, the investigation into how gVisor's kernel, *Sentry*, handles the syslog system call, shows that several system calls are either not fully implemented or are implemented differently from native Linux. The discrepancy provides an opportunity for further detection methods, which will be investigated later in this chapter, in the sections about unimplemented system calls and about modifying the kernel ring buffer.

4.2 Docker Containerization Detection

This section investigates how methods for detecting standard Docker (`runc`), discussed in the previous chapter, behave in a gVisor-based sandboxed environment.

4.2.1 Intel RAPL

The Linux kernel allows monitoring and limiting power of individual devices of the host system through the *sysfs* pseudo-filesystem using the *Power Capping Framework*. The files to access this functionality are located under `/sys/devices/virtual/powercap/`. The kernel supports different mechanisms for power capping and each available one is assigned a directory within the `powercap` *sysfs*-directory [28]. One of the `powercap` mechanisms is *RAPL* (Running Average Power Limit) which is supported by many Intel CPUs since Sandybridge [7].

The pseudo-files of the Power Capping Framework are organized within the *sysfs* directory as follows:

```
/sys/devices/virtual/powercap/<control-type>/<power-zone>/<subzone>/
```

Here, *RAPL* is the `<control-type>` of the Power Capping Framework. Depending on the hardware of the device, different zones and subzones will be available within *RAPL*'s subdirectory. On a system that supports two *RAPL* zones the structure could look like this:

```
intel-rapl/ ..... Control-type, containing zones
├── intel-rapl:0/ ..... Zone 0 for CPU package 0
│   ├── intel-rapl:0:0/ ..... Subzone 0 for core-part
│   ├── intel-rapl:0:1/ ..... Subzone 1 for uncore-part
│   └── intel-rapl:0:2/ ..... Subzone 2 for dram
└── intel-rapl:1/ ..... Zone 1 for Psys
```

The files containing the relevant power information for the subzone i of zone j can be accessed via the path:

```
/sys/devices/virtual/powercap/intel-rapl/intel-rapl:i/intel-rapl:i:j/
```

Each subzone, as well as the zones themselves, contain information about the power usage. The *synergetic power attack*, proposed by [7], makes use of the file `energy_uj`, providing the total power usage of the zone, respectively the subzone, in microjoule [29]. Since gVisor computes its own uptime in `/proc/uptime` instead of using the host's uptime [27]⁹, this would allow comparing the uptime to the total energy used by the hardware components. A high energy usage while having a short uptime would indicate that a process is running in a virtualized environment.

Even though the isolation mechanism proposed by [7], the power namespace, has not been implemented in the Linux kernel, it was found in an experiment on a RAPL supporting machine, that both, Docker and gVisor, attempt to mitigate the information leakage by hiding the RAPL directories in the `sysfs`-filesystem altogether. While the proposed power namespace, which would, analogous to other Linux namespaces, simply return the (estimated) power used by processes within each container and thus provide information that resembles the information a process would expect to encounter on a native machine (accuracy), while at the same time not reveal any power information about the host's system (transparency), the hiding of the directories can be detected very easily. When these files are not available, the process can suspect that it is running virtualized.

In summary, the RAPL files located in `/sys/devices/virtual/powercap/intel-rapl/` are hidden in both Docker and gVisor. In Docker only the `intel-rapl` directory is hidden but its parent directory is present and gVisor hides the whole `powercap` directory. Both directories can potentially be missing on a machine that does not support RAPL. Therefore, the information about the presence of these directories needs to be combined with whether the CPU supports RAPL. On such a machine, both directories can be expected to be present, and their absence indicates the presence of gVisor. If it does not support RAPL, this is not a reliable indicator.

4.3 gVisor specific methods

The previous sections investigated VM and standard Docker detection methods from the related work in a gVisor environment. While some methods were found to be applicable in a similar form on gVisor, it demonstrated limitations for other methods. Oftentimes, the investigation on the reason why a method cannot be used on gVisor, revealed new mechanisms that could be used instead.

Although some findings from these investigations will be used in this section, the primary focus is to explore new detection methods by studying gVisor's design and behavior.

⁹`pkg/sentry/fsimpl/proc/tasks.go` lines 320 - 330

4.3.1 Dockerenv and Dockerinit

Docker creates the file `/.dockerenv` [30]¹⁰ and in previous versions used to create the file `/.dockerinit` in the filesystem of the container [31]. The `.dockerenv` exists as a remnant of the LXC built-in exec driver, which was replaced by libcontainer [32] and was used to store environment variables passed into the container, which were then read from there, within the container [33]¹¹. Since many applications, even Docker-components such as moby's libnetwork [34], relied on the presence of these files to detect that they are running in a container, `.dockerenv` was kept and applications using `.dockerinit` switched to `.dockerenv` for this use-case as well. Nowadays, `.dockerenv` is still created by Docker but does not contain any environment variables anymore [30]¹².

Since Docker mounts this file by default into images without providing an option to disable it, it can be serve as an indicator.

4.3.2 Meminfo

The proc-filesystem contains process- and system-information. The file `/proc/meminfo` gives information about the system memory, such as the total amount, free memory, swapped memory, *etc* [35].

An experiment was conducted on a native Linux machine, standard Docker and gVisor. While in Docker, all the information is accessible within the container, in gVisor only a small subset of the attributes is available. Missing attributes in gVisor, that are available on native and Docker are:

- SwapCached
- Zswap
- Zswapped
- KReclaimable
- Slab
- SReclaimable
- SUnreclaim
- KernelStack
- PageTables
- SecPageTables
- NFS_Unstable
- Bounce
- WritebackTmp
- CommitLimit
- Committed_AS
- VmallocTotal
- VmallocUsed
- VmallocChunk
- Percpu
- HardwareCorrupted
- AnonHugePages
- ShmemHugePages
- ShmemPmdMapped
- FileHugePages
- FilePmdMapped
- Unaccepted
- HugePages_Total
- HugePages_Free
- HugePages_Rsvd
- HugePages_Surp
- Hugepagesize
- Hugetlb
- DirectMap4k
- DirectMap2M
- DirectMap1G

An investigation into gVisor's source code revealed that the cause lies in its userspace kernel, *Sentry*. In the `Generate` function, which handles the generation of the content of the `meminfo` pseudofile, only a small subset of values is produced. Some of them are hard-coded to always return 0 kB, as highlighted in red in the list below [27]¹³:

¹⁰`daemon/initlayer/setup_unix.go` lines 23 - 68

¹¹`daemon/execdriver/lxc/init.go` lines 108 - 134

¹²`daemon/initlayer/setup_unix.go` lines 23 - 68

¹³`pkg/sentry/fsimpl/proc/tasks_files.go` lines 269 - 311

- | | | | |
|----------------|------------------|------------------|-------------|
| • MemFree | • Active | • Inactive(file) | • Dirty |
| • MemAvailable | • Inactive | • Unevictable | • Writeback |
| • Buffers | • Active(anon) | • Mlocked | • AnonPages |
| • Cached | • Inactive(anon) | • SwapTotal | • Mapped |
| • SwapCache | • Active(file) | • SwapFree | • Shmem |

On a native machine, most attributes are generally expected to be present. While not all attributes are always available (*e.g.*, if huge pages are disabled in the kernel options, the corresponding attributes will not show up), finding exactly these attributes with values of 0 kB is a distinctive characteristic of gVisor that can be used as an indicator for virtualization detection. Furthermore, a mistake in gVisor’s code causes the attribute that is called `SwapCached` in the Linux kernel [36]¹⁴ to be called `SwapCache` in gVisor [27]¹⁵, making it uniquely identifiable.

4.3.3 Sysfs DMI Information

In the previous section in this chapter about the VM detection method using `dmidecode`, it was found that absence of the paths used by `dmidecode`¹⁶ is an indicator of containerization, both Docker and gVisor.

While both sysfs files that `dmidecode` uses, require superuser privileges, the kernel also provides DMI information that is generally readable by regular users within the `/sys/devices/virtual/dmi/id` directory.

An experiment on a native Linux machine, Docker, and gVisor showed that the files are all also present in Docker. Some of the files (`board_serial`, `chassis_serial`, `product_serial`, `product_uuid`) are not readable by non-root users on a native machine but the Docker daemon which is running as root on the host, passes the files on into the container where they are then also readable by processes in the container. As these are serial numbers of hardware components of the host machine, this is an information leakage channel, specifically allowing for co-residency attacks, as this uniquely identifies the machine hosting the containers.

In gVisor containers, most of this information is isolated, making only the `product_name` available to processes within the container, which has the same content as on the host machine. Since the other information is isolated, not by providing non-identifying information in gVisor, but instead hiding it altogether, the absence of these files indicates gVisor’s sandboxing.

¹⁴`fs/proc/meminfo.c` line 65

¹⁵`pkg/sentry/fsimpl/proc/tasks_files.go` line 294

¹⁶`/sys/firmware/dmi/tables/smbios_entry_point` and `/sys/firmware/dmi/tables/DMI`

4.3.4 Uptime and Idle Time in the proc Filesystem

As found in the investigation of using Intel RAPL to detect virtualization, gVisor computes its own uptime. While in Docker, we get the host's uptime, gVisor returns the time since the container was started [27]¹⁷. In `/proc/uptime` the Linux kernel stores two numbers: the time since boot, and the combined idle time of all CPUs [35]. While gVisor correctly computes its own uptime, it is hardcoded to return 0 as combined idle time (See figure 4.4). While theoretically possible, this is extremely unlikely to be encountered on a native machine and can be used as an indicator of virtualization. The indicator can be made even stronger when comparing it to CPU load and CPU time of running processes.

```

320 var _dynamicinode = (*uptimeData)(nil)
321
322 // Generate implements vfs.DynamicBytesSource.Generate.
323 func (*uptimeData) Generate(ctx context.Context, buf *bytes.Buffer) error {
324     k := kernel.KernelFromContext(ctx)
325     now := time.NowFromContext(ctx)
326
327     // Pretend that we've spent zero time sleeping (second number).
328     fmt.Fprintf(buf, "%.2f 0.00\n", now.Sub(k.Timekeeper().BootTime()).Seconds())
329     return nil
330 }

```

Figure 4.4: gVisor's Computation of Uptime and Combined CPU Idle Time [27]

4.3.5 Unimplemented Syscalls

Unlike Docker, which does not have an own kernel and simply relies on the isolation mechanisms of the host's kernel, gVisor adds the component, *Sentry*, which is an application level kernel. When a process within the container attempts to make a system call, Sentry will intercept it and if supported, handles it itself and responds instead of passing it on to the host's kernel [5]. An overview of gVisor's components is given in Figure 4.5.

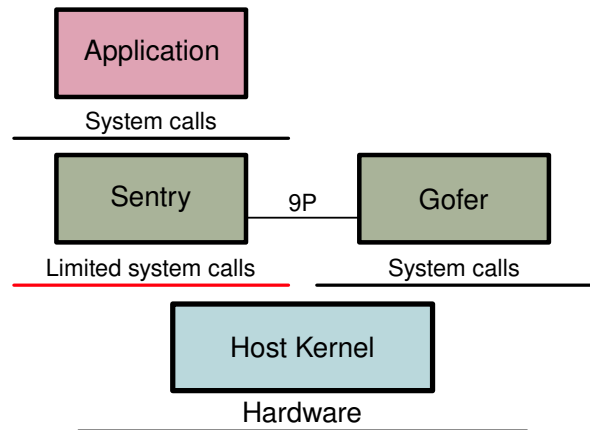


Figure 4.5: gVisor's Handling of System Calls [5]

This design potentially allows three forms of gVisor-specific virtualization detection mechanisms. Firstly, gVisor does not fully support all system calls. While it only attempts to implement an environment analogous to Linux 4.4, the current state is not fully equivalent [5]. Secondly, gVisor might already implement system calls which cannot be expected to be available on a machine running Linux 4.4. Consequently, comparing the reported kernel version with the availability of a system call which should not be available in Linux 4.4, may yield an indicator for gVisor's virtualization.

Thirdly, since system calls are intercepted by a user-level kernel, they are handled differently than on a native machine. This may lead to timing differences which would allow for side-channel virtualization detection.

¹⁷`pkg/sentry/fsimpl/proc/tasks_files.go` lines 320 - 330

This section will investigate the first category. In its application kernel, *Sentry*, gVisor distinguishes between three levels of support of system calls (`SupportUnimplemented`, `SupportPartial`, `SupportFull` [27]¹⁸) and implements them as seven different functions [27]¹⁹:

- `syscalls.Supported`
- `syscalls.SupportedPoint`
- `syscalls.PartiallySupported`
- `syscalls.PartiallySupportedPoint`
- `syscalls.Error`
- `syscalls.ErrorWithEvent`
- `syscalls.CapError`

All of them return a struct of type `Syscall`, which contains the necessary information for the syscall to be executed, including the name, the function that is called when the syscall is executed, the level of support, and a note [27]²⁰.

The system calls that are supported by gVisor have the struct field `SupportLevel` set to `SupportFull`. These are implemented with the functions `syscalls.Supported` and `syscalls.SupportedPoint`. The former will simply return a `Syscall` struct for the requested syscall, the latter adds a callback and then simply invokes the first function, which returns the struct.

The system calls of the second level `SupportPartial` are implemented with the functions `syscalls.PartiallySupported` or `syscalls.PartiallySupportedPoint`. Both behave similar to the corresponding ones of the `SupportFull` level, but are additionally annotated with information about which parts work or which are different or missing. For example, the partially supported system call `syslog` is annotated with “*Outputs a dummy message for security reasons*” [27]²¹. In the following chapter about modifying the kernel ring buffer it will become clear that these notes must be taken with a grain of salt as they do not necessarily contain all missing parts of the system call. In this case, the mentioned chapter will show, that the `syslog` system call does in fact output a dummy message for some arguments (actions) but the note does not reflect that the system call is unimplemented for most other actions.

Finally, the system calls that are not supported in gVisor are assigned the support level `SupportUnimplemented`. They are implemented with the functions `syscalls.Error`, `syscalls.ErrorWithEvent` or `syscalls.CapError`. The first simply returns a hard-coded error, the second also responds with a hard-coded error but also sends an event on the event-channel and the last checks a given capability and only if the process holds the capability, returns the error `ENOSYS`, revealing that the system call is not implemented. If the capability is not held, it instead responds with `EPERM` (Operation not permitted) [27]²².

This mechanism masks missing system calls in cases where the required capability is not held by the process and thus makes detecting the presence of gVisor more difficult.

¹⁸`pkg/sentry/kernel/syscalls.go` lines 55 - 67

¹⁹`pkg/sentry/syscalls/syscalls.go` lines 37 - 128

²⁰`pkg/sentry/kernel/syscalls.go` lines 70 - 86

²¹`pkg/sentry/syscalls/linux/linux64.go` line 550

²²`pkg/sentry/syscalls/syscalls.go` lines 74 - 128

4.3.6 Modifying the Kernel Ring Buffer

The investigation of the VM detection method using `dmesg` to access the kernel ring buffer showed that this method can be used in a similar way in gVisor. While this method uses the content of the messages that gVisor's kernel *Sentry* responds with when the `syslog` system call with the `SYSLOG_ACTION_READ_ALL` action as argument is executed, examining the source code of how this is handled, showed that this action and the `SYSLOG_ACTION_SIZE_BUFFER` action are the only ones that Sentry implements.

Attempting to call other actions on a native machine is expected to either work or result in a permission error (when running as non-privileged user). However, gVisor returns the `ENOSYS` error, indicating a missing implementation [27]²³.

For simplification, the action `SYSLOG_ACTION_CLEAR` is exemplarily chosen in this work. On successful execution it should clear the ring buffer. A simple experiment was conducted in gVisor and confirmed this detection mechanism, as attempting to clear the ring buffer failed with the message specifying that the implementation is missing.

²³`pkg/sentry/syscalls/linux/sys_syslog.go` lines 35 - 61

Chapter 5

Implementation

The implementation of the malware prototype combines all the detection mechanisms found in the previous chapter. The implementation stores each mechanism in an instance of a *strategy* struct. A strategy consists of a name, a description, and the function that will carry out the detection

```
type strategy struct {
    name string // Unique name, used for selecting individual strategies
    description string // Description of strategy
    score float64 // Achieved score, 0-100, 0 = not detected, 100 = certain detection
    weight float64 // How score is weighted, 0-1, 0 = ignored, 1 = fully weighted
    err error // Error that occurred during execution of strategy
    strat func() (float64, error) // Function attempting to detect gVisor, returns unweighted score
}
```

Figure 5.1: Strategy Definition

mechanism and return a score denoting the certainty this mechanism indicates virtualization. The reported score lies between 0, which indicates that the mechanism is certain that the malware is not running in gVisor, and 100, which will be returned when the indicators the mechanism checks are fully present. This score does not regard the reliability of the mechanism itself, it simply denotes to which extent the detection mechanism has found the characteristics it expects from gVisor in the current environment.

Therefore, for simple binary checks, it will always be either 0 or 100, and continuous mechanisms will always have at least one case where they return 0, or 100 respectively.

Since, however, the mechanisms vary in overall reliability, each *strategy* also includes a *weight*. This weight lies between 0, if a detection mechanisms is completely independent of the environment, and 1, indicating that the mechanism detects gVisor with maximal certainty. The final effective score of the *strategy* will be the product of this weight and the reported score of the detection mechanism.

By default, the malware iterates over all strategies and computes the average weighted score from the return values of the strategy-functions:

$$score = \frac{\sum_i (s_i \cdot w_i)}{\sum_i w_i}$$

Whenever a strategy returns with an error, its weight is set to 0 so that the score is ignored in the final score. If in the end the score is above a threshold, the malware categorizes the

result as having detected gVisor. Additionally, with the `-malicious` flag set, the malware will simulate malicious behaviour, i. e. a coin miner, when no virtualization is detected.

Alternatively, instead of using all strategies, specific strategies can be specified with the `-strat` option, passing a list of comma-separated strategies or a strategy-group. The available strategies and groups can be listed with `-list-strat`.

5.1 Choice of Language

The implementation is written in Go. Since malicious programs usually do not control the machine they run on, it is best to make as few assumptions as possible, *e.g.*, about available interpreters or shared libraries and the compatibility of their versions. Since Go is a compiled language, it does not require an interpreter, and because of its static linking, it produces a stand-alone executable, which does not require certain libraries or a runtime to be present on the host. Additionally, not only is this an implementation of a malware, but a malware that attempts to hide its malicious nature, when it is being analyzed, which gives another reason to prefer a compiled language, as the source code cannot simply be read. Finally, the environment the malware targets, gVisor, is also written in Go.

- No interpreter required on target machine
- No assumptions about available libraries or version conflicts due to static linking
- Code cannot simply be read
- Same language as target environment, gVisor

5.2 Implementation Details of Strategies

The following subsections match the sections of the previous chapter. For each detection mechanism found there, the corresponding subsection in this chapter will describe how the mechanism is implemented in the strategies of the malware. For each strategy the implementation will be explained and scores for the different cases will be assigned.

5.2.1 BIOS Information

As the investigation in chapter 4 showed, the VM detection method based on the BIOS information accessed by `dmidecode` cannot be used in the same way in gVisor. `dmidecode` first attempts to read the BIOS information from `sysfs` and if this is not possible, attempts to read it from the memory device file `/dev/mem`, both unavailable in gVisor.

Instead, the malware uses this as characteristic and implements the strategies `dmidecode_mem` and `dmidecode_sysfs`, one for each method of `dmidecode`.

The first simply checks for the presence of `/dev/mem`, returning 100 if it is absent (full detection), or 0 if present (no detection).

The second checks for the files `/sys/firmware/dmi/tables/smbios_entry_point` and `/sys/firmware/dmi/tables/DMI`. However, since in gVisor the expected case is that the whole `/sys/firmware/dmi/` is missing, only then 100 is returned. If some of the subdirectories, or even one of the two files exist, it will return lower, but non-zero scores. If both files are present, the malware categorizes this as native and thus returns 0.

5.2.2 Kernel and Driver Logs in the Kernel Ring Buffer

The VM-detection mechanism using the content of the kernel ring buffer can be used very similarly in gVisor. The investigation in chapter 4 demonstrated that in gVisor, a program can know even more about the characteristics of the ring buffer, and even about the content, than in a virtual machine. Therefore, instead of using `dmesg` to retrieve the syslog messages and grepping for keywords, like *virtual*, the malware implements four different strategies to evaluate the buffer.

1. `kernel_ring_buffer_messages_set`
2. `kernel_ring_buffer_messages_count`
3. `kernel_ring_buffer_messages_fixed`
4. `dev_kmsg`

By default, `dmesg` uses the `/dev/kmsg` device to access the kernel ring buffer. If this fails it uses the `syslog` syscall as a fallback. As the investigation showed, this device file is not available in gVisor, but the buffer still can be accessed with the fallback system call.

The `dev_kmsg` strategies is implemented as simple binary check, returning 0 if `/dev/kmsg` is present and 100 otherwise.

The remaining strategies all involve access to the kernel ring buffer in some form. To accomplish this, the malware implements shared helper functions for the three strategies: `get_syslog_size()` and `get_syslog_content()`. Both functions interact with the kernel ring buffer via the `syslog` system call, which is invoked through the `syscall` package from the Go standard library.

The first helper function executes the system call with the action `SIZE_BUFFER` to get the current size of the kernel ring buffer as return value. The second helper function returns the content of the current kernel ring buffer. Since the `syslog` system call with the action `READ_ALL` cannot simply return the entire log as return value, the malware first requests the size using the first helper method and allocates a local buffer of appropriate size. A pointer to this local buffer will then be passed as argument, together with its size and desired action (`READ_ALL`), when invoking the system call. The kernel then fills this local

buffer with the messages from its ring buffer. The local buffer, which is an array of bytes, is then converted to a list of strings, one message from the ring buffer per entry and returned.

As shown in the investigation, gVisor generates the messages in the ring buffer the first time it is requested. The first and last three messages are hard-coded and ten additional messages are randomly chosen from a list. The `kernel_ring_buffer_messages_set` strategy retrieves all messages from the kernel ring buffer and iterates over them to check whether they are messages that gVisor potentially can generate. Only if this is the case for 100 % of the messages in the current environment, the strategy will return a score of 100. If a small portion of other messages are present as well, it returns lower scores and 0 otherwise.

The second strategy `kernel_ring_buffer_messages_count` focuses solely on the number of messages rather than their content. Since gVisor hard-codes four specific messages and draws ten from its list at random, finding exactly 14 messages in the ring buffer strongly suggests the presence of gVisor. While this can potentially occur on native machines as well, production systems usually have a number of messages by a magnitude larger, making it very unlikely to be this amount exactly by coincidence. However, if the number deviates even slightly, it is most likely not a gVisor-based environment. Therefore, the strategy is binary and returns a score of 100 only if exactly 14 messages are present, and 0 otherwise.

Similar to the first strategy, the `kernel_ring_buffer_messages_fixed` strategy also examines the content of the messages. While the decision of the first strategy is based on what portion of the found messages are found within gVisor's set of possible messages, this strategy takes the position of the four hard-coded messages along with the initial timestamp into account. If all four messages are present in their expected positions and the timestamp matches, it returns a score of 100. Given that the first message is the strongest indicator (*Starting gVisor...*), if only the first message and timestamp match, the malware still estimates the environment to be gVisor-based with a high likelihood, therefore returning a score of 80.

5.2.3 Intel RAPL

As described in the corresponding section about RAPL in the previous chapter, the information about missing Powercap or RAPL sysfs-directories must be interpreted in combination with knowledge about RAPL support on the current machine.

To confront this problem, the strategy `intel_rapl` gathers CPU information by checking the vendor ID in the `/proc/cpuinfo` file. Since a *score* of 0 signifies certainty that the current environment is not gVisor-based, returning this score on non-Intel CPUs is not a sensible option. Instead, when the malware detects that the machine does not support RAPL, it returns an error. Whenever a strategy returns with a non-nil error, the main loop will assign a *weight* of 0 to that strategy. This ensures that this strategy will not affect the overall average score, and consequently, it will be ignored in the malware's decision.

If the malware, however, is running on an Intel machine, it will proceed to check the presence of the `powercap` and `intel-rapl` directories. Since gVisor hides not only the `intel-rapl` directory but the entire `/sys/devices/virtual/powercap/` parent directory, the malware will return a score of 100 only if it is in fact absent.

In Docker the `powercap` directory is present but the `intel-rapl` directory is absent. Although this case overall is more characteristic of a virtualized environment, it is untypical for gVisor-environments, which the malware focuses on. Hence, it will conservatively assign a rather low, but non-zero score of 40. Finally, on an Intel machine, the expected case for running natively is for both directories to be present. In this case the strategy returns a score of 0.

5.2.4 Dockerenv

This strategy `dockerenv` simply checks for the presence of the file `.dockerenv` in the container's root directory. In both Docker and gVisor it is expected to always be present. Thus, the decision of the strategy is binary, returning a score of 100 on existence, and 0 otherwise.

5.2.5 Meminfo

As described in chapter 4, the `/proc/meminfo` file allows processes to retrieve various information about the memory. In gVisor the file is present but many attributes are missing and only a subset is available, some of which are assigned a hard-coded value of 0 kB in gVisor's kernel *Sentry*. Furthermore, the attribute `SwapCached` was misspelled as `SwapCache`. Using these findings, the malware implements three strategies `meminfo_attributes`, `meminfo_zeros` and `meminfo_typo`.

Table 5.1: Expected Cases in `meminfo_attributes`

Characteristic	gVisor	Native
<code>attributes_gvisor</code> found	100 %	High
<code>attributes_missing</code> found	0	High

The first strategy contains two lists `attributes_gvisor` and `attributes_missing`. The former stores attributes that the investigation showed are generated in *Sentry*. The latter contains attributes commonly found on native machines that were found to be missing in gVisor. The expected case in gVisor and on a native machine is depicted in table 5.1. The absence of attributes of the first list strictly indicates that the environment is not gVisor-based, while their presence does not provide much information, as these attributes are commonly found on native machines as well. The strategy therefore starts by verifying that all of them are found. If just a single one is missing, a score of 0 is returned, which signifies certainty that the malware is not running on gVisor.

For the attributes of the second list, their presence is a strict indicator against gVisor, while the absence is a non-strict indicator for gVisor. While they could also be missing

on a native machine, it is unlikely to be exactly these attributes. Therefore, the strategy continues with a case distinction in which it assigns a score of 100 only if all of them are missing, and a rapidly decreasing score the more of them are found.

While the first strategy focuses solely on the presence and absence of attributes, the second strategy `meminfo_zeros` considers the known values. To achieve this, it contains a list of the attributes that the investigation of Sentry revealed to be hard-coded to 0 kB. Similar to the first strategy, it iterates through the lines of `/proc/meminfo` and parses the attribute name and value using a regular expression. If the list contains the found attribute, the malware will check if the value is 0 kB. Since gVisor should always generate this value for the given attributes, if only a single one has a different value, a score of 0 will be returned.

To handle cases of missing attributes, the strategy counts each attribute whose value was checked. While in general it is a sign against gVisor if some are missing, the score of this strategy should reflect the certainty of the environment being based on gVisor considering only the *values* and not the *presence* of attributes; this is already checked in the first strategy. Therefore, if not all values could be checked, the strategy becomes inapplicable and returns an error so that it will be assigned a weight of 0 in the main loop.

Finally, the third strategy `meminfo_typo` iterates over the `/proc/meminfo` file again to search for the `SwapCache` attribute. If it finds a line starting with `SwapCache:`, it returns a score of 100, and 0 otherwise.

5.2.6 Sysfs DMI Information

The findings from the corresponding section in the previous chapter are implemented in the strategy `available_dmi_info`. First, it checks for the availability of the `sysfs` directory containing the user-readable DMI information (`/sys/devices/virtual/dmi/id/`) and whether it contains the only file that is present in gVisor `product_name`. If both are present it proceeds to check the existence of the files commonly found in this directory.

The final decision of this strategy considers the percentage of the files found. Only if all but `product_name` are absent, a score of 100 is returned. For higher percentages the returned score decreases quickly.

5.2.7 Uptime and Idle Time in the proc Filesystem

As seen in the source code of gVisor's computation of the values in `/proc/uptime`, the application kernel, *Sentry*, computes an own uptime and hard-codes the combined idle time to 0. The latter is used by the strategy `proc_combined_idle_time` which reads the file and parses the second value. While the expected format is 0.00, to increase robustness the strategy converts the values to a float to compare it numerically (in case of 0 or 0.0, etc.). If the value matches, the function returns a score of 100. If the combined idle time is less than a tenth of a second, while suggesting it is not a gVisor environment, this is

still very unlikely to happen on a native machine. In this case the strategy returns a low but non-zero value of 10. On finding any higher number, a score of 0 is returned.

Failing to read the `/proc/uptime` file or being unable to parse the values due to an unexpected line format is never expected and the strategy should fail. Consequently, in this case the function passes the error on as return value to not be weighted in the overall score.

5.2.8 Unimplemented Syscalls

In the corresponding section of the architecture chapter it was shown that the way gVisor handles system calls enables three attack vectors. This strategy focuses on the first vector (the system calls that are unimplemented in gVisor) as they have the highest potential to allow gVisor to be detected.

Since gVisor's goal is to implement an environment resembling Linux 4.4 [5], all unsupported system calls that are also not supported by this Linux version are filtered out for the implementation of this detection strategy. The remaining system calls that gVisor's kernel *Sentry* does not implement [37] that are implemented in the Linux kernel 4.4 [38] are listed in below.

- | | | | |
|--------------------------------------|--------------------------------|---------------------------------|----------------------------------|
| • <code>setfsuid</code> | • <code>settimeofday</code> | • <code>remap_file_pages</code> | • <code>perf_event_open</code> |
| • <code>setfsgid</code> | • <code>swapon</code> | • <code>mq_timedsend</code> | • <code>fanotify_init</code> |
| • <code>uselib</code> | • <code>swapoff</code> | • <code>mq_timedreceive</code> | • <code>fanotify_mark</code> |
| • <code>personality</code> | • <code>reboot</code> | • <code>mq_notify</code> | • <code>name_to_handle_at</code> |
| • <code>ustat</code> | • <code>iopl</code> | • <code>mq_getsetattr</code> | • <code>open_by_handle_at</code> |
| • <code>sysfs</code> | • <code>ioperm</code> | • <code>kexec_load</code> | • <code>clock_adjtime</code> |
| • <code>sched_setparam</code> | • <code>init_module</code> | • <code>add_key</code> | • <code>kcmp</code> |
| • <code>sched_rr_get_interval</code> | • <code>delete_module</code> | • <code>request_key</code> | • <code>finit_module</code> |
| • <code>vhangup</code> | • <code>query_module</code> | • <code>ioprio_set</code> | • <code>sched_setattr</code> |
| • <code>modify_ldt</code> | • <code>quotactl</code> | • <code>ioprio_get</code> | • <code>sched_getattr</code> |
| • <code>sysctl</code> | • <code>set_thread_area</code> | • <code>migrate_pages</code> | • <code>kexec_file_load</code> |
| • <code>adjtimex</code> | • <code>get_thread_area</code> | • <code>vmsplice</code> | • <code>bpf</code> |
| • <code>acct</code> | • <code>lookup_dcookie</code> | • <code>move_pages</code> | • <code>userfaultfd</code> |

As seen in the corresponding section in the architecture chapter, we cannot simply expect unimplemented system calls to respond with `ENOSYS` (Function not implemented) when they are called, as gVisor attempts to mask missing implementation in several ways. Some simply return some other hard-coded error (mostly permission errors) and others require certain capabilities and only reveal the missing implementation when that capability is held by the process that called it.

To address this, an experiment was conducted in which the responses to all filtered unsupported system calls were gathered on a native Linux machine, both when running as a privileged and as an unprivileged user. These were then compared to the response gVisor gives and categorized into four potential classes, as depicted in table 5.2.

Table 5.2: Classification of Indicative Value of gVisor’s Unimplemented System Calls

Class	Description
Strong indicator	gVisor’s response is different in both cases
Difference to privileged native	gVisor’s response is different only to the privileged native execution
<i>Difference to unprivileged native</i>	<i>gVisor’s response is different only to the unprivileged native execution</i>
Weak indicator	gVisor’s response is the same as both native executions

In the experiment it was found that the third class is empty, so the strategy implements the remaining three classes. While the system calls whose responses are strong indicators (in the following referred to as *strong system calls*) can both indicate virtualization as well as indicate a native environment. This is because, when the malware receives a lot of responses to the strong system calls that are the responses that gVisor is expected to give, these responses are automatically an unexpected case for a native machine, and vice-versa. On the other hand, when all the responses to the weak system calls are the ones that the malware expects to receive from gVisor, it indicates neither that it is running on a native machine, nor that it is running in gVisor. In both environment, these responses are expected. In the case, however, that many of the weak system calls lead to unexpected responses, while it still is not an indicator for whether the malware is running native or virtualized, it is an indicator that it is not gVisor. This results in two possible options to handle the weak system calls. The first is to make the *score* rely solely on strong indicators and use the weak indicators not for the score (which represents to what degree the characteristics the detection mechanism checks were found) but instead make it influence the weight of the strategy which represents how reliable the strategy overall is. A high amount of unexpected responses then indicates that the malware is running in an environment where this method of checking the responses to system calls is not a reliable way to detect virtualization.

The second option is to use the weak system calls only as negative indicator. Here, the strong system calls are used to calculate the score, and unexpected results of weak system calls decrease the score thereafter.

While the first approach would be more appropriate for a malware that aims to generally detect any form of virtualization, the focus of this work is to reliably distinguish between native execution and gVisor’s sandboxing. Therefore, in the prototype the second approach is chosen. This design decision ensures that the weight can be read as “how reliable does the strategy detect gVisor overall”. As defined in the introduction, a low score does not necessarily indicate a native environment, but instead it indicates the certainty that it is not running in gVisor.

In conclusion, a response that is diverging from the expected response is always counted (for both, strong and weak system calls) to indicate that the environment is *not gVisor* (lowering the score), while a matching response is only counted

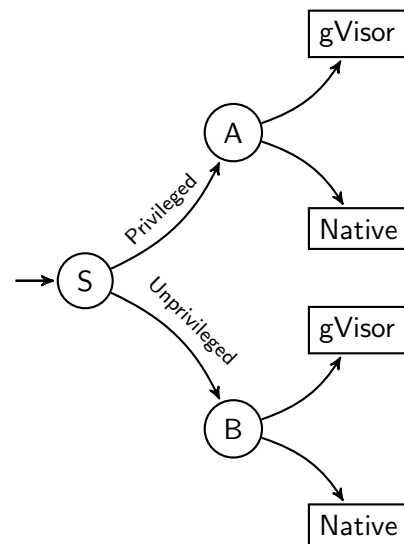


Figure 5.2: Decision Path for the Third Class of System Calls

only when it matches the expected response.

for strong system calls, where it indicates that the environment *is based on gVisor* (increasing the score).

The remaining class of system calls consists of those calls that give a different response than gVisor only if running on a privileged native machine. Since without further information, they can potentially be both, weak and strong, the malware further takes into account the user ID. If this ID is 0, it assumes that if running natively, it is running as a privileged user and assumes an unprivileged user otherwise.

The decision process of the malware is illustrated in Figure 5.2. With this additional information, the malware can distinguish whether it is at node *A* or *B*, and in both nodes, it has enough knowledge to interpret the responses to the system calls of the remaining class. In node *A*, all elements of the class become strong system calls, as not only do responses that deviate indicate that it is not gVisor (as weak system calls do) but additionally, matching responses are not expected in the native case (because in node *A*, the malware can assume the native process is privileged) and thus indicate the presence of gVisor. The last part, however, is not valid in point *B*, because in both possible cases, native and gVisor, the responses are expected to match. Since diverging responses still indicate that it is not gVisor, this class will be categorized as weak system calls, when the malware is at node *B*.

The malware implements each of the three classes as `map` which assign each system call in that class to the error gVisor is expected to return. Using Go's `syscall` package from the standard library, the malware iterates over each system call in the three maps and calls it. It then compares the returned error to the one that is expected and counts the ones that do not match. As described above, the malware then reduces the three classes to only two (weak and strong) based on the user ID and computes the percentage of matching responses for the two remaining classes:

$$\begin{aligned} \text{strong_percentage} &= \begin{cases} 1.0 - \left(\frac{\Delta\text{strong} + \Delta\text{priv}}{|\text{strong}| + |\text{priv}|} \right) & \text{if } uid = 0 \\ 1.0 - \left(\frac{\Delta\text{strong}}{|\text{strong}|} \right) & \text{if } uid \neq 0 \end{cases} \\ \text{weak_percentage} &= \begin{cases} 1.0 - \left(\frac{\Delta\text{weak}}{|\text{weak}|} \right) & \text{if } uid = 0 \\ 1.0 - \left(\frac{\Delta\text{priv} + \Delta\text{weak}}{|\text{priv}| + |\text{weak}|} \right) & \text{if } uid \neq 0 \end{cases} \end{aligned}$$

Here, *strong* and *weak* denote sets that represent the maps of strong and weak system calls in the malware. The *priv* denotes the set representing the third class of the system calls where the expected response is only different on a privileged native machine.

The variables prefixed with Δ represent the number of non-matching responses to the system calls in that class in the current environment.

Finally, the score is computed and returned:

$$\text{score} = \max(0, 100 \cdot (\text{strong_percentage} - (1.0 - \text{weak_percentage})))$$

5.2.9 Modifying the Kernel Ring Buffer

As seen in the source code of Sentry's handling of the `syslog` system call, only two actions are implemented. This is when `syslog` is called with either the argument `READ_ALL` or

`SIZE_BUFFER`. All other attempts to execute this system call will result in a `ENOSYS` error. The strategy `syslog_clear` makes use of this by choosing one alternative action from the many remaining available ones and attempts to execute it using the `syscall`-package from Go's standard library. The invocation of the `syscall` returns as third return value an error number. The malware evaluates this error and only if it equals `ENOSYS`, it returns a score of 100.

While any other action for the `syslog` system call can be chosen, the malware attempts w.l.o.g to invoke it with the action `SYSLOG_ACTION_CLEAR`. This action does not require an allocated buffer or length as argument and thus reduces error sources in the detection strategy.

Chapter 6

Evaluation

This chapter will evaluate how reliable the malware detects gVisor’s virtualization. First, the overall reliability will be investigated practically by conducting experiments in different environments. In each environment, the malware will be executed in a gVisor-backed Docker container using a helper script to compile an overview of the reported scores.

Afterwards, the reliability of the individual strategies will be discussed theoretically, by examining the characteristics of gVisor that enable it to be detected, to assess the likelihood of false negatives. In order to estimate the likelihood of false positives, the reliability to detect these characteristics in gVisor will be contrasted with the likelihood of encountering the same characteristics in a productive native environment.

Based on these discussions, this chapter will propose mitigation techniques for each strategy. It will do so by, on the one hand, showing preventative measurements that an application using gVisor could implement without having to modify gVisor (*e.g.*, configuration, workarounds), and on the other hand, by suggesting changes that could be made to gVisor or the Linux kernel. The proposed techniques will in turn be confirmed by experiments in hardened environments to which the techniques have been applied.

Since the focus of this thesis is the detection of gVisor’s sandboxing, false negatives are considered if the malware runs in a gVisor container but does not detect it. Failing to detect that it runs in a different form of virtualization, like a virtual machine or standard Docker, is not considered a false negative. Likewise, the evaluation will count as a false positive whenever the malware detects virtualization but runs not virtualized at all, on a native machine.

6.1 Setup of the Experiments

Listing 6.1: Dockerfile to Run the Malware in gVisor

```
1 FROM debian:bookworm
2
3 COPY prototype /opt/prototype
4 CMD /opt/prototype
```

Listing 6.2: Script to Build and Run the Malware and Container Image

```

1 #!/bin/sh
2
3 set -eu
4
5 go build .
6 docker build . -t prototype
7 docker run --runtime=runc prototype:latest

```

The container image is built using a Dockerfile (Listing 6.1) which simply takes the base image of Debian Bookworm and copies the compiled malware into the `/opt/` directory and starts it as default command. The Dockerfile is utilized by a helper script (Listing 6.2) which first compiles the malware, then builds the container image as defined by the Dockerfile and finally starts a container with that image specifying gVisor’s runtime (`--runtime runc`).

6.2 Performance of the Malware in Different Environments

Using the setup described in the previous section, the malware will be tested in the environments depicted in table 6.1.

Table 6.1: Environments the Malware was Tested in

ID	Environment	Host OS	Kernel (Host)	Host CPU
0	gVisor	Debian Bookworm	6.5.0	Intel i5-7300U
1	Native (Privileged)	Debian Bookworm	6.5.0	Intel i5-7300U
2	Native (Unprivileged)	Debian Bookworm	6.5.0	Intel i5-7300U
3	Native (Privileged)	Artix	6.9.7	Intel i5-8350U
4	Native (Unprivileged)	Artix	6.9.7	Intel i5-8350U
5	Native (Privileged)	Debian Stretch	4.9.0	AMD Athlon G. 3150U
6	Native (Unprivileged)	Debian Stretch	4.9.0	AMD Athlon G. 3150U

Table 6.2 shows the scores that the individual strategies of the malware achieved in the environments defined in table 6.1. The numbers in the title row reference the environments from table 6.1. In this row a green background signifies a gVisor-based environment, where the expected score is 100 and a red background signifies a native environment, in which the strategies are supposed to return a score of 0. The background of the data cells below is green to red based on the achieved score. If the strategy returns the correct score, the cell will have the same color as the title row.

As the results in table 6.2 show, the malware correctly decides whether gVisor is present in all cases. In all but one case, all applicable strategies give a perfect score, with the one case giving 0.34 instead of 0.00 out of 100.

Table 6.2: Overview of the Achieved Scores in Different Environments

Strategy	0	1	2	3	4	5	6
dmidecode_mem	100	0	0	0	0	0	0
dmidecode_sysfs	100	0	0	0	0	0	0
intel_rapl	100	0	0	0	0	-	-
dockerenv	100	0	0	0	0	0	0
meminfo_attributes	100	0	0	0	0	0	0
meminfo_zeros	100	0	0	0	0	0	0
meminfo_typo	100	0	0	0	0	0	0
available_dmi_info	100	0	0	0	0	0	0
proc_combined_idle_time	100	0	0	0	0	0	0
syscall_unimplemented	100	0	0	0	0	0	0.34
kernel_ring_buffer_messages_set	100	0	-	0	-	0	-
kernel_ring_buffer_messages_count	100	0	-	0	-	0	-
kernel_ring_buffer_messages_fixed	100	0	-	0	-	0	-
dev_kmsg	100	0	0	0	0	0	0
syslog_clear	100	0	-	0	-	0	-
Final weighted score	100	0	0	0	0	0	0.34

6.3 Reliability and Mitigation of Individual Strategies

In the following, the *reliability* sections describe the malware’s view, assessing its reliability in productive environments gVisor is commonly used in (as opposed to the resilience against specific countermeasures targeting the detection strategies). The specific countermeasures and their effectiveness are discussed in the *mitigation* sections.

6.3.1 BIOS Information

Reliability The first strategy, `dmidecode_mem`, uses the absence of the character device file `/dev/mem` as indicator for gVisor. While its equivalent for the kernel’s virtual memory `/dev/kmem` can be disabled by setting the kernel option `CONFIG_DEVMEM` to `n`, which several major Linux distributions do by default [39][40], disabling the file `/dev/mem` through the option `DEVMEM` [36]¹ is not commonly done on native machines, making false positives unlikely. False negatives are possible, if gVisor was configured to pass through the memory device file or if a file was specifically created at this location. Both options are not expected to be commonly found in gVisor use-cases.

The second strategy `dmidecode_sysfs` bases its decision on the `sysfs` files that `dmidecode` uses². On native machines they can be expected to be present, and a container would specifically need to be configured to provide these two files. Since DMI information is not required in most use-cases that containers are usually used for, both, false positives and false negatives are unlikely.

¹`drivers/char/Kconfig` lines 310 - 317

²`/sys/firmware/dmi/tables/smbios_entry_point` and `/sys/firmware/dmi/tables/DMI`

Mitigation Currently the strategy `dmidecode_mem` simply checks for the presence of the character device file `/dev/mem` but not the content. A simple way to mitigate this detection mechanism is to simply create a file in this location. In order to make the file type match the type `/dev/mem` has on regular systems, a character device file can be created using `mknod`.

Listing 6.3: Dockerfile Mitigating the `dmidecode_mem` Strategy

```

1 FROM debian:bookworm
2
3 COPY prototype /opt/prototype
4
5 CMD mknod /dev/mem c 1 1 && /opt/prototype

```

Executing the malware with this modified Dockerfile results in the strategy returning a score of 0. Consequently, the detection mechanism has been successfully prevented. To counter this mitigation technique an attacker would need to examine the content of the memory device file, which increases the required complexity of the malware. Defending against such an adapted strategy would, however, require proper isolation of the `/dev/mem` file. `gVisor` would need to provide this device file by simulating a physical memory that only contains data used by processes within the container. While `gVisor` has its own memory management, it does not have a physical memory, and accurately simulating one is not trivial and possibly susceptible to side-channel attacks using timing difference. `Sentry` would need to ensure that any data present in the simulated physical memory file is actually loaded in the host's physical memory to be able to respond in time. For memory not present in the simulated memory but loaded in the host's physical memory, delays would need to be simulated. Currently, `Sentry` has no simple way to determine which pages are loaded in the host's physical memory [41].

The second strategy `dmidecode_sysfs`, which currently simply checks for the existence of the two DMI files that `dmidecode` uses, can be mitigated by creating any file in this directory. Since the malware could simply counter this by attempting to decode the files (e.g. by replicating `dmidecode`), a more robust defense will be proposed.

As seen in the architecture chapter, the `sysfs` files, required by `dmidecode`, can be mounted with Docker's `-v` option. To validate that this is a possible option, `dmidecode` was installed in a `gVisor` container which was started with the flags shown in listing 6.4.

Listing 6.4: Mounting the Host's `sysfs` DMI Files

```

1 docker run --rm -it --entrypoint bash --runtime=runc -v /sys/firmware/
  dmi:/sys/firmware/dmi debian:bookworm

```

Running `dmidecode` in the container showed that it can now successfully retrieve the DMI information.

The helper script was modified (Listing 6.5) in order to validate that the strategy returns a score of 0 when this mitigation technique is applied, which was successful. Since the previous experiment showed that `dmidecode` is able to decode the DMI information in this configuration, it can be concluded that the proposed improved strategy, that would evaluate the files instead of simply checking for their existence, is mitigated with this technique as well.

Listing 6.5: Helper Script Mitigating the dmidecode_sysfs Strategy

```
1 #!/bin/sh
2
3 set -eu
4
5 go build ..
6 docker build . -t prototype_mitigation_dmidecode_sysfs
7 docker run --runtime=runc -v /sys/firmware/dmi:/sys/firmware/dmi
   prototype_mitigation_dmidecode_sysfs:latest
```

Mounting the host's DMI files is, however, only a proof of concept and is unsuitable for productive applications like a malware analysis tool, as it leaks the host's BIOS information into the container. Instead, the malware analysis tool could include mock DMI and `smbios_entry_point` files that were either copied from an unrelated machine or were artificially created to contain realistic data. To further increase resilience against this detection method, a set of DMI mock files could be included and selected randomly, or could be generated dynamically before the container is started.

6.3.2 Kernel and Driver Logs in the Kernel Ring Buffer

Reliability This section will describe the implementation of the three strategies that use the content of the kernel ring buffer:

1. `kernel_ring_buffer_messages_set`
2. `kernel_ring_buffer_messages_count`
3. `kernel_ring_buffer_messages_fixed`

False positives are very unlikely for all three. In the first strategy, all logged messages would need to appear in the list of the messages gVisor can generate. Since this list does not contain any messages that other programs would write to the syslog, no false positives are expected. In the second strategy, exactly 14 messages would need to be present. While this strategy is the most likely of the three to cause a false positive, production systems often have several hundreds of messages, making it improbable to match this number by coincidence. In the third strategy the first and last three messages would need to match gVisor's hard-coded syslog messages. Since the first message is "Starting gVisor...", it is practically impossible for even the first message to match, let alone all of them simultaneously.

Since for all possible syslogs that gVisor possibly can generate, all of the three strategy achieve a score of 100. For false negatives to occur the syslog would need to be changed. gVisor, however, only implements the actions `SIZE_BUFFER` and `READ_ALL` for the `SYSLOG` system call. Both of them are read-only operations.

Mitigation To defend against the detection strategies, the syslog needs to contain something different than all the possible syslog instances that gVisor possibly can generate.

Because the SYSLOG system call only implements read-only actions, it cannot be modified by a process running in the container, *i.e.*, before starting the malware. Consequently, the `Generate` function needs to be patched:

Listing 6.6: Patch to the Function Generating the Syslog

```
diff --git a/pkg/sentry/kernel/syslog.go b/pkg/sentry/kernel/syslog.go
index d03a14add..3975653b8 100644
--- a/pkg/sentry/kernel/syslog.go
+++ b/pkg/sentry/kernel/syslog.go
@@ -48,39 +48,32 @@ func (s *syslog) Log() []byte {

    // Not initialized, create message.
    allMessages := []string{
-   "Synthesizing system calls...",
-   "Mounting deweydecimalfs...",
-   "Moving files to filing cabinet...",

    [...]

-   "Recruiting cron-ies...",
-   "Verifying that no non-zero bytes made their way into /dev/zero...",
-   "Accelerating teletypewriter to 9600 baud...",
+   "Bluetooth: hci0: corrupted SCO packet",
+   "usb 1-9: USB disconnect, device number 5",
+   "usb 4-2: Manufacturer: Generic",
+   "intel_rapl_common: Found RAPL domain package",

    [...]

+   "input: PC Speaker as /devices/platform/pcspkr/input/input6",
+   "ACPI: button: Power Button [PWRF]",
+   "device-mapper: uevent: version 1.0.3",
+   "systemd[1]: Starting modprobe@fuse.service - Load Kernel Module fuse...",
    }

    selectMessage := func() string {
@@ -96,22 +89,16 @@ func (s *syslog) Log() []byte {

    const format = "<6>[%11.6f] %s\n"

-   s.msg = append(s.msg, []byte(fmt.Sprintf(format, 0.0, "Starting gVisor..."))...)
+   s.msg = append(s.msg, []byte(fmt.Sprintf(format, 0.0, "Linux version 4.4-amd64 (
    debian-kernel@lists.debian.org) (gcc-12 (Debian 12.2.0-14) 12.2.0, GNU ld (GNU
    Binutils for Debian) 2.40) #1 SMP PREEMPT_DYNAMIC Debian 6.1.94-1 (2024-06-21)"))...)

    time := 0.1
-   for i := 0; i < 10; i++ {
+   n := 5 + rand.Intn(len(allMessages) - 4)
+
+   for i := 0; i < n; i++ {
        time += rand.Float64() / 2
        s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, selectMessage()))...)
    }

-   time += rand.Float64() / 2
-   s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Setting up VFS..."))...)
-   time += rand.Float64() / 2
-   s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Setting up FUSE..."))...)
-
-   time += rand.Float64() / 2
-   s.msg = append(s.msg, []byte(fmt.Sprintf(format, time, "Ready!"))...)
-

    // Return a copy.
    o := make([]byte, len(s.msg))
    copy(o, s.msg)
}
```

In this proof of concept defense (Listing 6.6) three changes were made to the source of the gVisor version `release-20240715.0` (git tag). Firstly, the messages were replaced with

messages that can be found on real machines. Secondly, the hard-coded messages were removed, such that now only the first is hard-coded and its content was replaced with a message resembling a native machine more closely. Lastly, instead of generating the fixed amount of 14 messages, the number of messages is now randomized.

Listing 6.7 shows how the patch was applied to gVisor's source code, how gVisor was compiled thereafter and the patched runtime was installed to the system.

Afterwards, the malware was executed using the unmodified Dockerfile and helper script on the patched runtime. All of the three strategies now return a score of 0, verifying the proof of concept.

Listing 6.7: Application of the Patch and Compilation and Installation of gVisor

```
1 # Preparing source code
2 git clone https://github.com/google/gvisor.git
3 cd gvisor
4 git checkout release-20240715.0
5 git apply <patch-dir>/syslog.patch
6
7 # Building
8 mkdir -p bin
9 make copy TARGETS=runsc DESTINATION=bin/
10
11 # Backup old runsc and install patched version
12 sudo mv /usr/bin/runsc runsc_usr_bin.bak
13 sudo mv /usr/local/bin/runsc runsc_usr_local_bin.bak
14 sudo cp ./bin/runsc /usr/local/bin
```

A productive application, like a malware analysis tool, could build on this patch and replace the entire `selectMessage` function with a more sophisticated generation of the messages than simply drawing them from a predefined list.

6.3.3 Intel RAPL

Reliability The strategy `intel_rapl` that checks for the presence of relevant sysfs files and directories in combination with CPU information is less reliable than the previous strategies. On the one hand it fails on non-intel CPUs, giving no information at all, and on the other hand would also require a more sophisticated method of checking whether a CPU supports RAPL.

False negatives happen if the malware detects an Intel CPU but the RAPL powercap files do not exist in the sysfs directory. In its current state the malware simply checks the vendor in the CPU information and assumes RAPL compatibility when it is an Intel processor. To avoid false negatives on processors pre-Sandybridge, the model name would need to be taken into account in combination with information about which CPU models support RAPL. Beyond that, false negatives could also happen if the CPU supports RAPL but the distribution makes the information unavailable in the sysfs directories. This is possible using the `CONFIG_POWERCAP` or `CONFIG_INTEL_RAPL` kernel options [36]³. To

³`drivers/powercap/Kconfig` lines 17 - 36

estimate the likelihood of encountering an environment with such a kernel configuration, experiments were conducted on different machines with different distributions. In table 6.3 the **RAPL** column denotes whether the **CPU** supports RAPL. The **Sysfs Files** column then shows whether the **Distribution** using the specified **Kernel Release** made the RAPL information available via the sysfs file system in the experiments.

Table 6.3: Experiment: RAPL Support and Availability of Sysfs Files

CPU	Distribution	Kernel Release	RAPL	Sysfs Files
Intel i5-7300U	Debian Bookworm	6.5.0-44-generic	✓	✓
Intel i5-7300U	Fedora 40 Workstation	6.8.5-301.fc40.x86_64	✓	✓
Intel i5-7300U	Ubuntu 24.04 LTS	6.8.0-31-generic	✓	✓
Intel i5-7300U	Alpine 3.20.2	6.6.41-0-lts	✓	✓
Intel i5-8350U	Artix	6.9.7-artix1-1	✓	✓
Intel Xeon E5-2407	Debian Bookworm	6.1.0-22-amd64	✓	✓
AMD Athlon G. 3150U	Fedora 40 Workstation	6.9.9-200.fc40.x86_64	✓	✓
AMD Athlon G. 3150U	Debian Stretch	4.9.0-9-amd64	✓	✗
BCM2835	RPi OS 12	6.1.21-v7+	✗	✗

The experiments present two findings. Firstly, the proposed method of checking for RAPL support using a list of CPU models and whether they support RAPL, not only avoids false negatives on pre-Sandybridge Intel processors, but extends the range of machines the strategy is applicable on, because some non-Intel CPUs support RAPL as well. Secondly, it shows that none of the modern tested distributions prohibit access to RAPL via the sysfs files on RAPL-supporting CPUs, but very old distributions might not support it. This suggests reliability against false positives only on newer distributions.

Mitigation In its current version, only the files need to be present. gVisor, however, currently does not support mounting volumes to this directory with the `-v` option, nor can the files be created after the container was started, as the directory is mounted read-only.

In order to solve this problem, and to avoid the detection through the analysis of the content of the RAPL files (*e.g.*, checking that `energy_uj` increases), a power-based namespace could be added to the Linux kernel which estimates the share of the host's power usage caused by processes within the container, as proposed by [7].

6.3.4 Dockerenv and Dockerinit

Reliability When gVisor is used via Docker, no false negatives are expected to happen. Currently, Docker always creates the file, without providing an option to disable it. However, gVisor can also be used to run Kubernetes pods in its sandbox. Executing the malware in a gVisor-backed pod in Minikube 1.33.1 showed that the `.dockerenv` file is not created in Kubernetes, and consequently this strategy returns a wrong score. False positives, however, are not expected since native machines do not create the file.

Mitigation One way to mitigate this detection vector is to simply remove the `.dockerenv` file in the container before the malware is started. Although not possible during the build process of the container image, for malware analysis tools this can be a valid option, as they have control over when the malware is executed.

Listing 6.8: Dockerfile Mitigating the `dockerenv` Strategy

```
1 FROM debian:bookworm
2
3 COPY prototype /opt/prototype
4 CMD rm /.dockerenv && /opt/prototype
```

Caution, however, must be taken with unintended side-effects. For example `moby's enableBridgeNetFiltering` function depends on the file [30]⁴.

6.3.5 Meminfo

Reliability As the investigation of `gVisor's` source code in chapter 4 showed, three characteristics of `gVisor's` can be used to detect its presence. These were implemented in three strategies in the malware:

1. `meminfo_attributes`
2. `meminfo_zeros`
3. `meminfo_typo`

False negatives are expected in none of the three strategies, because the returned attributes and hard-coded 0 kB values are deterministically generated in `gVisor's` kernel.

For false positives to occur in the first strategy, a native machine would need to be running with a kernel that disabled exactly those features that produce the attributes that are missing in `gVisor`, which is unlikely to happen by coincidence.

In the strategy `meminfo_zeros` all the values that are hard-coded to be 0 kB in `gVisor` would need to be 0 on a native machine to cause a false positive. For some of the attributes that is a plausible scenario to encounter on a productive native machine, for example, a machine with no swap file or partition will have a value 0 kB in the three swap attributes. For others, like `Unevictable`, `Mlocked` or `Dirty`, the likelihood is lower, and is further reduced because they need to all be 0 simultaneously. Yet, it is the most likely strategy to cause a false positive of these three.

For the strategy `meminfo_typo`, false positives are not expected, since the Linux kernel does not generate an attribute called `SwapCache`.

⁴`moby/libnetwork/drivers/bridge/setup_bridgenetfiltering.go` lines 48 - 65 and 76 - 79

Mitigation The `/proc/meminfo` file cannot simply be written to, making it difficult to prevent this detection mechanism in the current state of gVisor. Consequently, gVisor's generation of the `meminfo` file would need to be patched.

The first strategy `meminfo_attribute` only relies on the set of `meminfo` attributes being available, a scenario which could possibly also happen on a native machine with very restricted kernel options. Implementing more of the features that are commonly present in Linux, for example huge pages, would reduce its distinctive character. The strategy `meminfo_zeros` uses the hard-coded 0 kB values. For some of the attributes this does not necessarily require mitigation, *i.e.*, the swap values. For others, like `Unevictable` and `Mlocked`, gVisor would need to be able to oversee which parts of the memory used by processes within the container have been locked to prevent it from being swapped [42]. Finally, the `meminfo_typo` can very easily be mitigated by correcting the name of the `SwapCache` attribute to `SwapCached` in the file `pkg/sentry/fsimpl/proc/tasks_files.go`

6.3.6 Sysfs DMI Information

Reliability The strategy `strat_available_dmi_info` depends on the sysfs files providing DMI information in the directory `/sys/devices/virtual/dmi/id`. To detect gVisor, it relies on the fact, that `product_name` is present while all other files in this directory are missing. Since gVisor has no functionality to create the other files, false negatives are not expected.

On Linux, on the other hand, these files are generally expected to be present, most of them even being readable by unprivileged users [21]. Therefore, no false positives should happen.

Mitigation In the current state of gVisor, the function that adds the specific inode for the `product_name` sysfs file, has no functionality to create further files [27]⁵. This, however, is necessary to defend against this detection mechanism, because the directory `/sys/devices/virtual/dmi/id/` is a read-only directory, making it impossible for a process within the container to create the files before the malware is executed.

As a proof of concept, gVisor's creation of the `id` sysfs directory was patched (Listing 6.9), adding the functionality to provide the `bios_date` file. In this proof of concept the bios date is set to the date the container was started.

Listing 6.9: Patch to the Function Adding the inodes in the id Sysfs Directory

```

1 diff --git a/pkg/sentry/fsimpl/sys/sys.go b/pkg/sentry/fsimpl/sys/sys.go
2 index 161e282d4..05d3875bc 100644
3 --- a/pkg/sentry/fsimpl/sys/sys.go
4 +++ b/pkg/sentry/fsimpl/sys/sys.go
5 @@ -21,6 +21,7 @@ import (
6     "os"
7     "path"
8     "strconv"
9 +    "time"
10
```

⁵`pkg/sentry/fsimpl/sys/sys.go` lines 85 - 200

```

11     "golang.org/x/sys/unix"
12     "gvisor.dev/gvisor/pkg/abi/linux"
13 @@ -176,6 +177,7 @@ func (fsType FilesystemType) GetFilesystem(ctx context.Context,
    vfsObj *vfs.Virt
14         "dmi": fs.NewDir(ctx, creds, defaultSysDirMode, map[string]kernfs.Inode{
15             "id": fs.NewDir(ctx, creds, defaultSysDirMode, map[string]kernfs.Inode{
16                 "product_name": fs.NewStaticFile(ctx, creds, defaultSysMode,
                    productName+"\n"),
17 +                 "bios_date": fs.NewStaticFile(ctx, creds, defaultSysMode, time.Now().
    Format("01/02/2006")+"\n"),
18             }},
19         }},
20     })

```

The patch (Listing 6.9) was then applied to the source code of gVisor at the git tag `release-20240715.0`. Afterwards, gVisor was compiled and the patched runtime was installed on the system, similar to the application of the previous patch (Listing 6.7), but replacing the patch name in line 5.

The malware was then run in a container backed by the patched runtime using the unmodified Dockerfile and helper script. The achieved score of 70 of the strategy in question, demonstrates that adding the creation of the remaining files will defend against this detection mechanism. Since most of the files contain very general information, gVisor could easily implement the generation of real-seeming information for the remaining files in that directory.

6.3.7 Uptime and Idle Time in the proc Filesystem

Reliability The strategy `proc_combined_idle_time` uses the fact that the combined idle time reported by gVisor in the file `/proc/uptime` is hard-coded to always be 0. Since it is hard-coded, no false negatives should occur. As the time is reported in seconds with two decimal places, for a false positive to occur, the combined idle time would need to be less than 10 ms on a native machine, which is unrealistic to encounter on most productive Linux machines.

Mitigation gVisor relies on the isolation mechanism `cgroups` to limit the allowed resources usage of a container [10]. To isolate the `/proc/uptime` file properly, Sentry could attempt to combine the assigned CPU limit with the actual CPU usage of the processes within the container to infer a reasonable idle time.

For applications which simply want to prevent the detection of gVisor but do not require a correct idle time, a defense mechanism is proposed by patching gVisor's kernel, *Sentry*: gVisor already computes a correct uptime and stores it in its kernel. When a process requests the combined idle time, *i.e.*, by accessing `/proc/uptime` the time between this request i and the previous request $i - 1$ is computed as Δt . For this time interval, the relative idle time pi will randomly be chosen between 0.5 and 1.0, signifying that the CPUs have been idling between 50 % and 100 % of the time. Additionally, the number of CPUs $nproc$ will be taken into account by taking the product of these three values, resulting in the combined idle time in the interval Δt . This will be added to the idle time

that was returned on the last request ci_{i-1} resulting in the new total combined idle time. This ensures that the value is increasing monotonically.

$$\begin{aligned} \Delta t &= t_i - t_{i-1} && \text{(Time interval since last request)} \\ pi &\leftarrow [0.5, 1.0] && \text{(Choose relative idle time in this interval)} \\ ci_i &= ci_{i-1} + (\Delta t \cdot pi \cdot nproc) && \text{(Update the combined idle time)} \end{aligned}$$

The Patch (Listing 6.10) shows how this was implemented in gVisor. First, gVisor's kernel in `pkg/sentry/kernel/kernel.go` was modified and two integer fields were added: The first `combinedIdleTimeLastRequest` stores the Unix time of the last request in milliseconds⁶. The second variable `combinedIdleTimeLastValue` stores the value that was returned on the last request. Additionally, the function `CombinedIdleTime` was added to the kernel. When called, it updates the combined idle time stored in the kernel as described in the equations above.

Finally, the `Generate` function in `pkg/sentry/fsimpl/proc/tasks_files.go` was modified by replacing the hard-coded 0 value with the correct retrieval of the idle time from the kernel.

Listing 6.10: Patching the Combined Idle Time in the proc Filesystem

```

1 diff --git a/pkg/sentry/fsimpl/proc/tasks_files.go b/pkg/sentry/fsimpl/proc/tasks_files.
  go
2 index 37bcff1f2..3e0b1c228 100644
3 --- a/pkg/sentry/fsimpl/proc/tasks_files.go
4 +++ b/pkg/sentry/fsimpl/proc/tasks_files.go
5 @@ -324,8 +324,7 @@ func (*uptimeData) Generate(ctx context.Context, buf *bytes.Buffer)
   error {
6     k := kernel.KernelFromContext(ctx)
7     now := time.NowFromContext(ctx)
8
9     // Pretend that we've spent zero time sleeping (second number).
10    fmt.Fprintf(buf, "%.2f 0.00\n", now.Sub(k.Timekeeper().BootTime()).Seconds())
11 +   fmt.Fprintf(buf, "%.2f %.2f\n", now.Sub(k.Timekeeper().BootTime()).Seconds(), float64
12     (k.CombinedIdleTime()) / 1000)
13     return nil
14 }
15 diff --git a/pkg/sentry/kernel/kernel.go b/pkg/sentry/kernel/kernel.go
16 index 230ed9742..13c9928c9 100644
17 --- a/pkg/sentry/kernel/kernel.go
18 +++ b/pkg/sentry/kernel/kernel.go
19 @@ -35,6 +35,8 @@ import (
20     "errors"
21     "fmt"
22     "io"
23 +   "math"
24 +   "math/rand"
25     "path/filepath"
26     "time"
27
28 @@ -153,6 +155,8 @@ type Kernel struct {
29     // See InitKernelArgs for the meaning of these fields.
30     featureSet          cpuid.FeatureSet
31     timekeeper          *Timekeeper
32 +   combinedIdleTimeLastRequest int64
33 +   combinedIdleTimeLastValue  int64

```

⁶Milliseconds since 1970-01-01

```

34     tasks                *TaskSet
35     rootUserNamespace    *auth.UserNamespace
36     rootNetworkNamespace *inet.Namespace
37 @@ -379,6 +383,11 @@ type InitKernelArgs struct {
38     // Timekeeper manages time for all tasks in the system.
39     Timekeeper *Timekeeper
40
41 + // Stores the time and returned value of the last access of the combined
42 + // idle time in milliseconds.
43 + CombinedIdleTimeLastRequest int64
44 + CombinedIdleTimeLastValue int64
45 +
46     // RootUserNamespace is the root user namespace.
47     RootUserNamespace *auth.UserNamespace
48
49 @@ -440,6 +449,10 @@ func (k *Kernel) Init(args InitKernelArgs) error {
50
51     k.featureSet = args.FeatureSet
52     k.timekeeper = args.Timekeeper
53 +
54 + k.combinedIdleTimeLastRequest = k.timekeeper.Now().UnixMilli()
55 + k.combinedIdleTimeLastValue = 0
56 +
57     k.tasks = newTaskSet(args.PIDNamespace)
58     k.rootUserNamespace = args.RootUserNamespace
59     k.rootUTSNamespace = args.RootUTSNamespace
60 @@ -1482,6 +1495,25 @@ func (k *Kernel) Timekeeper() *Timekeeper {
61     return k.timekeeper
62 }
63
64 +// Returns a forged combined idle time in milliseconds, internal variable is
65 +// updated on request
66 +func (k *Kernel) CombinedIdleTime() int64 {
67 + // Calculate time in seconds since the last request
68 + lastRequest := k.combinedIdleTimeLastRequest
69 + now := k.timekeeper.Now().UnixMilli()
70 + lastRequestDiff := now - lastRequest
71 +
72 + // Randomly decide how much of the interval since the last request was idle
73 + // (between 50% and 100%)
74 + idlePercentage := 0.5 + rand.Float64() * 0.5
75 +
76 + // Store time of this request and update idle time
77 + k.combinedIdleTimeLastRequest = now
78 + k.combinedIdleTimeLastValue += int64(math.Round(float64(lastRequestDiff) *
79 + idlePercentage * float64(k.ApplicationCores()))))
80 +
81 + return k.combinedIdleTimeLastValue
82 +}
83 // TaskSet returns the TaskSet.
84 func (k *Kernel) TaskSet() *TaskSet {
85     return k.tasks

```

The patch was applied to gVisor's source code at tag `release-20240715.0`, similar to the previous patch (Listing 6.7) but replacing the patch name in line 5. Executing the malware on the patched runtime by using the unmodified Dockerfile and helper script, showed that the strategy now returns a score of 0, demonstrating the effectiveness of the defense mechanism.

6.3.8 Unimplemented Syscalls

Reliability The system calls that the malware checks are all implemented with one of these functions in gVisor:

- `syscalls.Error`
- `syscalls.ErrorWithEvent`
- `syscalls.CapError`

While the first two always return the hard-coded error, specified in gVisor's source code, the last one depends on the environment. Here, the malware assumes that no special capabilities are granted to the container it is running in. If that is the case, however, the affected system calls will falsify the result. While the impact is small, since only the system calls implemented with `syscalls.CapError` are affected, and of those only exactly those whose required capabilities are granted, it is still possible to improve the malware in this regard. To improve the reliability in these scenarios, where non-default capabilities are granted, the malware could simply rely on the system calls that are implemented as `syscalls.Error` or `syscalls.ErrorWithEvent` in gVisor.

Another possibility, which does not reduce the amount of system calls that can be used for detection, is to implement a more fine-grained checking mechanism for system calls using `syscalls.CapError` which would check which capabilities it holds and determine with this information, what gVisor would respond. This only requires knowledge about gVisor's syscall-capability mapping, as each system call implemented like this, can only return `ENOSYS` or `EPERM` [27]⁷. For each system call this solely depends on a single capability which is hard-coded in `pkg/sentry/syscalls/linux/linux64.go`.

Mitigation Preventing this strategy is likely the most challenging of the presented detection mechanisms, as it would require gVisor to respond to every system call in the same way Linux 4.4 would. It is improbable that gVisor will implement all missing system calls in the near future, not least because Google has explicitly decided not to do so for some of them [43].

As an alternative to ensuring that these characteristics are removed by modifying gVisor, for many applications it might be more practical to reduce the amount of assumptions a malware can make in gVisor's environment. For example, an additional user could be created within gVisor and instead of executing processes as root, the regular user could be used. When the malware is not running with UID 0, it cannot expect as many system calls to work.

6.3.9 Modifying the Kernel Ring Buffer

Reliability Since gVisor is hard-coded to respond to all actions apart from `READ_ALL` and `SIZE_BUFFER` with the `ENOSYS` error, no false negatives are expected. Executing the `syslog` system call with the action `CLEAR` on native Linux should either successfully clear the ring buffer, or return a permission error. Thus, no false positives should happen either.

⁷`pkg/sentry/syscalls/syscalls.go` lines 111 - 128

Mitigation In order to prevent this detection mechanism, gVisor would need to implement the missing actions of the `syslog` system call. Sentry already keeps a log of messages that it can return on the `READ_ALL` buffer. While no quick workaround is possible to fix this, implementing the remaining actions is feasible.

Chapter 7

Summary and Conclusions

The investigation of the various ways to detect gVisor, shows that virtualization entails a large attack surface for detection. While the Linux kernel has many features to provide isolation, which is extended by gVisor's additional security features, such as intercepting system calls, perfectly isolating a container to make its detection impossible, is very challenging.

As the evaluation of the malware showed, in the current state, a malware that specifically targets gVisor in order to not be analyzed in a gVisor-based analysis tool, it is able to reliably and efficiently detect it and shut down its malicious behavior.

Yet, many detection mechanisms can be mitigated to such a degree which suffices for many use-cases. Specifically, in the case of container-based malware analysis tools or honeypots, it does not need to be *completely* impossible to detect its underlying virtualization. At the very least, when the detection requires such complexity or time, that the attempt to detect the virtualization can in itself be detected as malicious, the respective detection mechanism is sufficiently mitigated for many use-cases.

7.1 Future Work

In future work two things can be done. Firstly, since the focus of this work is mainly the detection, many of the mitigation techniques were shown simply as a proof of concept, some of which needing a complete implementation to be applicable in productive applications.

Secondly, the detection strategies investigated in this work use the characteristics of the virtualized environment for its detection. Since gVisor intercepts system calls, they are handled differently than on a native Linux machine. Future work could investigate whether this leads to distinctive timing differences which could be used as a side-channel for detecting the presence of gVisor's application-level kernel. Similar to the work of [19] on virtual machines, both sources of indicators could then potentially be fused to improve the reliability through the additional timing analysis, while maintaining the efficiency of the characteristics analysis in the general case.

Bibliography

- [1] U. Bayer, A. Moser, C. Kruegel, and E. Kirda, “Dynamic analysis of malicious code”, *Journal in Computer Virology*, vol. 2, no. 1, pp. 67–77, 2006. DOI: 10.1007/s11416-006-0012-2.
- [2] B. Asvija, R. Eswari, and M. B. Bijoy, “Virtualization detection strategies and their outcomes in public clouds”, in *2017 IEEE Asia Pacific Conference on Postgraduate Research in Microelectronics and Electronics (PrimeAsia)*, 2017, pp. 45–48. DOI: 10.1109/PRIMEASIA.2017.8280360.
- [3] *Kvm*, https://linux-kvm.org/page/Main_Page, 2024. (visited on 07/30/2024).
- [4] *Linux containers vs. docker: Which one should you use? | docker*, <https://www.docker.com/blog/lxc-vs-docker/>, 2024. (visited on 07/30/2024).
- [5] *What is gvisor? - gvisor*, <https://gvisor.dev/docs/>, 2024. (visited on 07/01/2024).
- [6] K. Sunitha, “A survey on securing the virtual machines in cloud computing”, *International Journal of Innovative Science, Engineering & Technology*, vol. 1, no. 4, 2014.
- [7] X. Gao, Z. Gu, M. Kayaalp, D. Pendarakis, and H. Wang, “Containerleaks: Emerging security threats of information leakages in container clouds”, in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017, pp. 237–248. DOI: 10.1109/DSN.2017.49.
- [8] X. Tao, L. Wang, Z. Xu, and R. Xie, “Detection of hardware-assisted virtualization based on low-level feature”, in *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, 2022, pp. 914–919. DOI: 10.1109/CSCWD54268.2022.9776255.
- [9] *Cgroups(7) — manpages — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/manpages/cgroups.7.en.html>, 2024. (visited on 07/29/2024).
- [10] *Security model - gvisor*, https://gvisor.dev/docs/architecture_guide/security/, 2024. (visited on 07/27/2024).
- [11] *Namespaces(7) — manpages — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/manpages/namespaces.7.en.html>, 2024. (visited on 07/29/2024).
- [12] *Capabilities(7) — manpages — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/manpages/capabilities.7.en.html>, 2024. (visited on 05/25/2024).

- [13] J. v. d. Assen, A. H. Celdrán, A. Zermin, R. Mogenicato, G. Bovet, and B. Stiller, “Secbox: A lightweight container-based sandbox for dynamic malware analysis”, in *NOMS 2023-2023 IEEE/IFIP Network Operations and Management Symposium*, 2023, pp. 1–3. DOI: 10.1109/NOMS56928.2023.10154293.
- [14] Y. C. Jadhav, A. Sable, M. Suresh, and M. K. Hanawal, “Securing containers: Honey-pots for analysing container attacks”, in *2023 15th International Conference on COMmunication Systems & NETworkS (COMSNETS)*, 2023, pp. 225–227. DOI: 10.1109/COMSNETS56262.2023.10041276.
- [15] M. Reeves, D. J. Tian, A. Bianchi, and Z. B. Celik, “Towards improving container security by preventing runtime escapes”, in *2021 IEEE Secure Development Conference (SecDev)*, 2021, pp. 38–46. DOI: 10.1109/SecDev51306.2021.00022.
- [16] T. Kong, L. Wang, D. Ma, K. Chen, Z. Xu, and Y. Lu, “Configrand: A moving target defense framework against the shared kernel information leakages for container-based cloud”, in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, 2020, pp. 794–801. DOI: 10.1109/HPCC-SmartCity-DSS50907.2020.00104.
- [17] K. Wu, D. Yang, X. Gao, W. Yang, M. Li, and D. Wang, “Process based container escape monitoring and resource isolation scheme”, in *2022 13th International Conference on Information and Communication Systems (ICICS)*, 2022, pp. 54–58. DOI: 10.1109/ICICS55353.2022.9811204.
- [18] H. Gantikow, T. Zöhner, and C. Reich, “Container anomaly detection using neural networks analyzing system calls”, in *2020 28th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2020, pp. 408–412. DOI: 10.1109/PDP50117.2020.00069.
- [19] J.-B. Wang, Y.-F. Lian, and K. Chen, “Virtualization detection based on data fusion”, in *2012 International Conference on Computer Science and Information Processing (CSIP)*, 2012, pp. 393–396. DOI: 10.1109/CSIP.2012.6308876.
- [20] *System management bios*, <https://www.dmtf.org/standards/smbios>, 2024. (visited on 05/25/2024).
- [21] *Dmidecode(8) — dmidecode — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/dmidecode/dmidecode.8.en.html>, 2024. (visited on 05/25/2024).
- [22] J. Delvare, *Dmidecode - source (commit 190a23e)*, <https://git.savannah.nongnu.org/git/dmidecode.git>, 2024. (visited on 06/24/2024).
- [23] *Mem(4) — manpages — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/manpages/mem.4.en.html>, 2024. (visited on 05/25/2024).
- [24] *Docker security | docker docs*, <https://docs.docker.com/engine/security/>, 2024. (visited on 05/25/2024).
- [25] *Moby/oci/caps/defaults.go at master - moby/moby*, <https://github.com/moby/moby/blob/master/oci/caps/defaults.go#L6-L19>, 2024. (visited on 05/25/2024).

- [26] *Util-linux source (commit e0c4173)*, [git://git.kernel.org/pub/scm/utils/util-linux/util-linux.git](https://git.kernel.org/pub/scm/utils/util-linux/util-linux.git), 2024. (visited on 07/09/2024).
- [27] *Gvisor source (commit 66630c9)*, <https://github.com/google/gvisor.git>, 2024. (visited on 06/26/2024).
- [28] *Power capping framework — the linux kernel documentation*, <https://www.kernel.org/doc/html/latest/power/powercap/powercap.html>, 2024. (visited on 06/11/2024).
- [29] *[patch 0/1] rapl (running average power limit) driver*, <https://lore.kernel.org/all/1364940936-20846-1-git-send-email-jacob.jun.pan@linux.intel.com/>, 2013. (visited on 06/11/2024).
- [30] *Moby source (commit ff1e2c0)*, <https://github.com/moby/moby.git>, 2024. (visited on 06/25/2024).
- [31] *Remove dockerinit once and for all by cyphar - pull request #19490 - moby/moby*, <https://github.com/moby/moby/pull/19490>, 2016. (visited on 06/25/2024).
- [32] *Deprecated engine features | docker docs - lxc built-in exec driver*, <https://docs.docker.com/engine/deprecated/#lxc-built-in-exec-driver>, 2024. (visited on 06/25/2024).
- [33] *Moby source (release v1.8.0)*, <https://github.com/moby/moby.git>, 2015. (visited on 06/25/2024).
- [34] *Libnetwork/drivers/bridge/setup_bridgenetfiltering.go at 9fb7ba8fa01f0ddce99713799e760223c842fb4b · moby/libnetwork*, https://github.com/moby/libnetwork/blob/9fb7ba8fa01f0ddce99713799e760223c842fb4b/drivers/bridge/setup_bridgenetfiltering.go#L159, 2015. (visited on 06/25/2024).
- [35] *The /proc filesystem — the linux kernel documentation*, <https://www.kernel.org/doc/html/latest/filesystems/proc.html>, 2024. (visited on 06/11/2024).
- [36] *Linux source (commit 66ebddf)*, <https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/>, 2024. (visited on 07/23/2024).
- [37] *Linux/amd64 - gvisor*, https://gvisor.dev/docs/user_guide/compatibility/linux/amd64/, 2024. (visited on 07/01/2024).
- [38] *Syscalls(2) — manpages-dev — debian bookworm — debian manpages*, <https://manpages.debian.org/bookworm/manpages-dev/syscalls.2.en.html>, 2024. (visited on 07/01/2024).
- [39] *Security/features - ubuntu wiki*, <https://wiki.ubuntu.com/Security/Features>, 2024. (visited on 07/23/2024).
- [40] *Security features matrix - fedora project wiki*, https://fedoraproject.org/wiki/Security_Features_Matrix, 2021. (visited on 07/23/2024).
- [41] *Resource model - gvisor*, https://gvisor.dev/docs/architecture_guide/resources/, 2024. (visited on 07/23/2024).
- [42] *Mlock(2) — manpages-dev — debian bullseye — debian manpages*, <https://manpages.debian.org/bullseye/manpages-dev/mlock.2.en.html>, 2024. (visited on 07/27/2024).

- [43] *Support setfsuid, setfsgid · issue #260 · google/gvisor*, <https://github.com/google/gvisor/issues/260>, 2019. (visited on 07/28/2024).

Abbreviations

BIOS	Basic Input Output System
CGroup	Control Group
CPU	Central Processing Unit
CVE	Common Vulnerabilities and Exposures
FS	File System
GCE	Google Compute Engine
GDT	Global Descriptor Table
I/O	Input / Output
LXC	Linux Containers
NN	Neural Network
OS	Operating System
SMBIOS	System Management BIOS
UID	User Identifier
VMM	Virtual Machine Monitor
VM	Virtual Machine
eBPF	Extended Berkeley Packet Filter

Glossary

Containerization Containerization is a form of virtualization that uses features built into the host's kernel to achieve its isolation and generally does not use virtualized hardware.

Isolation Isolation refers to the mechanism of hiding the presence of processes and their actions from each other. A resource is properly isolated, if, instead of hiding the resource altogether, an identical interface is provided but only provides the part of the data that belongs to the same domain.

Virtualization Virtualization allows the creation of multiple virtual computers on the same physical host.

List of Figures

2.1	Machine-level Virtualization [5]	4
2.2	gVisor Sandboxing [5]	5
4.1	Output of dmesg in gVisor	16
4.2	Generation of an Artificial Syslog [27]	18
4.3	Bucket of Syslog Messages gVisor Chooses from [27]	18
4.4	gVisor's Computation of Uptime and Combined CPU Idle Time [27]	23
4.5	gVisor's Handling of System Calls [5]	23
5.1	Strategy Definition	27
5.2	Decision Path for the Third Class of System Calls	34

List of Tables

3.1	Overview of Related Papers	11
5.1	Expected Cases in meminfo_attributes	31
5.2	Classification of Indicative Value of gVisor's Unimplemented System Calls	34
6.1	Environments the Malware was Tested in	38
6.2	Overview of the Achieved Scores in Different Environments	39
6.3	Experiment: RAPL Support and Availability of Sysfs Files	44

Appendix A

Installation Guidelines

Detailed setup instructions are provided in the `README.md`.